Database - Bug #7252

Use direct-access to support for-each queries without WHERE and BY clauses

04/04/2023 09:57 AM - Alexandru Lungu

Status:	WIP	**Start date:**	
Priority:	Normal	**Due date:**	
Assignee:	Radu Apetrii	**% Done:**	90%
Category:		**Estimated time:**	0.00 hour
Target version:			
billable:	No	**case_num:**	
vendor_id:	GCD	**version:**	

Description

History

#1 - 04/04/2023 10:11 AM - Alexandru Lungu

- Subject changed from Use direct-access to support for-each queries without WHRE and BY clauses to Use direct-access to support for-each queries without WHERE and BY clauses

4GL code:

```
FOR EACH tt:
```

or

```
DEFINE QUERY q FOR tt. OPEN QUERY q FOR EACH tt.
```

SQL:

```
select tt.* from tt where tt._multiplex = ? order by tt._multiplex asc, tt.recid asc lazy
```

We successfully compressed the SQL (using * wildcard) and enforced a lazy traversal (using lazy keyword). This only happens with H2 JDBC. However, we also have a parallel implementation for direct-access. We can extend the current API with something like getAllLazy(String tableName, long multiplex, String indexName, boolean reversed) which will retrieve a lazy index cursor on the specified _multiplex from a specific index (only for AdaptiveQuery). Note that the ResultSet is FWD-H2 lazy, so it does the whole invalidation job itself.

The same functionality can be extended to cover FIND (FIRST|LAST) tt (without WHERE and BY). More general, this functionality can be extended for cases in which the WHERE clause is based on a index look-up (a conjunction of conditionals over an index's fields).

In an application I tested, ~20% of the SQL queries executed against H2 were from a simple FOR EACH tt. I think it is best to start off with a better understanding how common is this case against multiple customer applications and what is the total time spent vs a simple direct-access call. In the past, direct-access could optimize thousands of FIND queries for some extra milliseconds. I don't know what to expect from optimizing FOR-EACH this time.

#2 - 04/19/2023 06:41 AM - Alexandru Lungu

- Status changed from New to WIP

- Assignee set to Radu Apetrii

#3 - 04/20/2023 07:35 AM - Alexandru Lungu

Radu, can you please make a statistic in a dummy Java project for:

- 10,000 JDBC statements doing select tt.* from tt where tt._multiplex = [...] order by tt._multiplex asc, tt.recid asc lazy.
- 10,000 prepared JDBC statements doing select tt.* from tt where tt._multiplex = ? order by tt._multiplex asc, tt.recid asc lazy { 1: [...] }
- 10,000 direct-access calls similar to getAllLazy, something like table = findTable("tt") > tt = findIndex(table, "recid asc") > cursor = getCursor(idx, multiplex) > makeResultSet(cursor). This is a high-level pseudo-code here; you will need something to retrieve the result-set accordingly. This same implementation you can reuse in your FWD direct-access driver. Note that I already have an implementation in FWDDirectAccessDriver.getUniqueRow that finds an index for the provided columns in an arbitrary order; you may need to change ti to cover the order and asc/desc.

Adjust the SQLs and number of executions accordingly. Populate the tables accordingly (maybe 100 records for 10 multiplex values?). Report back with the time differences between these interfaces.
You may do this test in several phases:

- Always presume that the query uses the unique index (recid). Presume that there is always only one multiplex.
- Use the index look-up by order by (I think it is already implemented in H2 as Select.getSortIndex; you can extract this code somewhere else). Presume that there is always one multiplex.
- Full-test: index look-up and multiple multiplex values.

This way we can compare how certain logic impacts the implementation like what is slower: to identify the multiplex, to parse the order-by, to find the table, to construct a cursor, to parse a statement?

#4 - 05/02/2023 04:02 AM - Radu Apetrii

After doing some testing, I managed to form the following table:

Scenario	Normal statements	Direct-access
Unique index (recid)	3584	2337
Index look-up	4147	2551
Full-test	4338	2626

Notes:

- Time is measured in milliseconds.
- There were 1 million queries over a temp-table that had 100 records and 10 multiplex values.
- There is no Prepared Statements column because in that case, the statement does not get parsed every iteration, it is reused. Hence, Prepared Statements will have its own table (below).
- In the case of Prepared Statements, it's not really accurate to assume that the statement gets parsed just one time (out of a million queries), but neither it is to assume that it gets parsed every time. I'm not sure how to balance this out.

For Prepared Statements (where the query gets parsed only the first time), the results look like:

Scenario	First iteration	From second iteration forwards
Unique index (recid)	0.26	371

Index look-up	0.29	382
Full-test	0.32	396

What to take from this test:

- **Parsing a statement is definitely time-consuming.** This can be seen not only in normal statement vs direct-access comparison, but also in the difference between prepared statements: the first iteration that included the parsing vs the other 999,999 iterations (which contain a query that executes on average in 0.0003 ms).
- Getting the right index that matches the ORDER BY takes some time, especially if no suitable one is found. Some work regarding this can be found on #7066.
- The time spent for identifying the multiplex seems to be a minor issue (if it can be considered an issue at all).

I'll move forward with the implementation for direct-access unless I find a better solution for testing PreparedStatements in the meantime.

#5 - 05/08/2023 06:31 AM - Radu Apetrii

I've committed to 7252a, rev.14562 and 7252a_h2, rev.18 a few changes that allow the use of direct-accessing when retrieving all the records from a temporary table.

This currently works for AdaptiveQueries that are "simple", meaning that:

- They have exactly one component (are single-table).
- They are executed over a temporary database.
- They do not contain a WHERE clause (apart from the _multiplex one).

Also, I've left two TODOs which will be taken care of as soon as the changes from #7066 will be considered OK.

The changes were tested with a large customer application, with which I got the following results:

- 16.38% of the AdaptiveQueries that were single-table and executed on a temporary database were solved with direct-access. This percentage is very similar to Alex's one from #7252-1.
- The cache that was implemented to store Command s got a 77.57% hit ratio. For two Command s to be considered the same they need to have the same session, table, multiplex and direction (ascending or descending).

Regarding the previous comment with PreparedStatements, **I've created a table to show the comparison of the direct-access method with caching vs the normal prepared statements**.
Notes:

- PS stands for PreparedStatements and DA stands for DirectAccess.
- Time is measured in milliseconds.
- There were 1 million queries over a temp-table that had 100 records and 10 multiplex values.

Scenario	PS -> First iteration	PS -> From second iteration forwards	DA -> First iteration	DA -> From second iteration forwards
Unique index (recid)	0.26	371	1.5	278
Full-test	0.32	396	1.6	288

#6 - 05/08/2023 10:08 AM - Alexandru Lungu

Radu Apetrii wrote:

> Also, I've left two TODOs which will be taken care of as soon as the changes from #7066 will be considered OK.

Are these functionally dependent in any way? Can we integrate #7252 without the #7066 changes and still have a good performance increase? I would like to split #7066 effort in smaller pieces and integrate sequentially.

> **The changes were tested with a large customer application, with which I got the following results**:
>
> - 16.38% of the AdaptiveQueries that were single-table and executed on a temporary database were solved with direct-access. This percentage is very similar to Alex's one from #7252-1.
> - The cache that was implemented to store Command s got a 77.57% hit ratio. For two Command s to be considered the same they need to have the same session, table, multiplex and direction (ascending or descending).

77.57% is a decent cache ration, but I think psCache is doing better. Can we eliminate the multiplex from the key, so we cache the commands independently from the multiplex? Afterwards, we can just execute the same command with different multiplex values. I expect something like ~90% from such cache.

> Notes:
>
> - PS stands for PreparedStatements and DA stands for DirectAccess.

First PS iteration is after preparing I guess, while the first iteration from DA does a cache miss. The results are as expected (-25% performance improvement) - maybe we can achieve more.

Review for **rev.14562 and 7252a_h2, rev.18**:

- In FWD, I see you use String fullIndexName = null and send it like this to the driver. Shouldn't we use a non-null value here? String fullIndexName = canFindIndex ? fullIndexNameBuilder.toString() : null; is commented out.
- I am a bit afraid of !canFindIndex && !sortClause.contains("recid"). Not finding an index should always be a problem. I agree that in FWD you can have an adaptive query over the primary index (recid) and this is not found by IndexHelper. However, !sortClause.contains("recid") doesn't look very safe. recid can be part of a field/table name, or maybe occur in some other way. I recommend implementing indexHelper.isRecidSort to do a safe check for tt.recid asc or tt.recid desc.
- You already have the session extracted, don't call buffer.getSession(false) again at the end.
- I feel like we should have a standard FWD way of naming the indexes, not by custom composing fullIndexNameBuilder. Can you check if such composing method exists? Changing how index names are build may break your implementation.
- In FWD-H2, you only retrieve the record by RECID in ascending order. I expected adding the proper order by order to match the index here. However, I see your point of using USE_INDEX from #7066.
- Extract session.getDatabase in a separate variable.

In order to test this properly, we need a functional version. I will do an extensive testing ASAP with 7066a_h2 (alone) and if everything is OK, I will merge in in trunk. After rebase, you can go ahead with the proper implementation here. Please take a look in 7066a_h2, to ensure everything is "locked and loaded".

#7 - 05/09/2023 04:45 AM - Radu Apetrii

Alexandru Lungu wrote:

> Radu Apetrii wrote:
>
> > Also, I've left two TODOs which will be taken care of as soon as the changes from #7066 will be considered OK.
>
> Are these functionally dependent in any way? Can we integrate #7252 without the #7066 changes and still have a good performance increase? I would like to split #7066 effort in smaller pieces and integrate sequentially.

Assuming that the commented code remains commented, the changes are totally independent and can be integrated separately. The current state of the code implies that no index will be forced/suggested/given to H2, it will let H2 select an index by itself.

> Can we eliminate the multiplex from the key, so we cache the commands independently from the multiplex? Afterwards, we can just execute the same command with different multiplex values. I expect something like ~90% from such cache.

This is similar to my original idea, to create some sort of a template Command and to just set the values for the given parameters. I believe this could be done. I will think of this more thoroughly and do it.

- In FWD, I see you use String fullIndexName = null and send it like this to the driver. Shouldn't we use a non-null value here? String fullIndexName = canFindIndex ? fullIndexNameBuilder.toString() : null; is commented out.

That other part is commented out because in the current state of FWD-H2, the index is selected automatically. After the #7066 changes make it through, we can simply pass the index name to H2 and it will select it (by force). But for the moment, there is no need to pass an index name since all the records can be retrieved without specifying one.
The commented code is for the near future. String fullIndexName = null is used only in this moment to pass as a parameter. **Instead of removing the parameter from multiple places and bringing it back when #7066 changes will appear, I have chosen to pass a null parameter for now.**

Regarding the other points from the review, I will apply them and get back with updates.
If you believe that the things said here need to be refactored, let me know and I will do the changes.

#8 - 05/11/2023 09:20 AM - Alexandru Lungu

- % Done changed from 0 to 60

Ok, I guess we can go ahead with integrating it without the fullIndexName (for now). However, I am concerned of the following aspect:

- You mention that H2 will chose an index. This means that the records will be returned in an arbitrary order. We already have an index/order-by clause that we don't use right now.
- Please consider that ORDER BY is not by default (_multiplex, recid); it is mentioned in the order by clause of the adaptive query, which usually resembles an index.
- If we want to integrate it as it is (which is fine to me), you should set the proper order by. This also means extending your cache key.

From my POV, the last points to fix here are:

- Properly set the ORDER BY clause. Maybe you can send the fullIndexName and resolve it in H2 to an order-by clause (instead of USE-INDEX clause).
- Make the command re-executable with parameterized multiplex
- Other review aspects, especially !canFindIndex && !sortClause.contains("recid")

#9 - 05/15/2023 07:07 AM - Radu Apetrii

I've committed to 7252a, rev. 14563 and to 7252a_h2, rev. 19 some changes that target the remaining points from the latest posts:

- Instead of manually computing fullIndexName, I've called a method that does that. In addition, the comment that I left in the code for that index name composing has been completely removed, since it is not needed anymore.
- Regarding the previous bullet, I got rid of the !sortClause.contains("recid") condition since that function does the checking for me.
- I removed the _multiplex value from the command cache key and made it work as a parameter. **This change resulted in a very decent cache hit ratio increase, from 77.57% to 97.76% on a large customer application I tested.**
- I changed the ORDER BY clause so that it matches the given index's columns.
    - You already have the session extracted, don't call buffer.getSession(false) again at the end.
    - Extract session.getDatabase in a separate variable.
- Also solved these two issues.

Taking into account how much easier was to create the index's name with this function, I definitely need to integrate this idea in #7066. I will do this as quickly as possible.

#10 - 05/15/2023 10:06 AM - Alexandru Lungu

I have mixed feelings about the cache on the second thought.

My concern is that the cache is unbound, so we may end up with thousand of entries cached there without a proper expiry policy (like a LRU). We can

pin >100MB with such cache if we are not careful.
On the other hand, the cached values are Command instances, which are stored by a PreparedStatement as well. However, for PreparedStatement we already have psCache which does a decent job (with expiry).

I feel like we can go a step further here and try to "dissolve" the Select, if that makes sense. My point is that we are already facing a very simple query (select all based on a provided index, without filter except multiplex). Maybe we can do a SelectAll class which is very straight-forward - provide a result-set which lazily extracts the result from an index. Using the whole Command API (with expressions, tables, order by, etc.) seems like an over-kill now.

I know we already had a discussion on this, but I feel we are trying to reimplement the wheel now: caching some already parsed commands to allow re-execution (sounds like psCache, but without query strings).

Your solution is still faster due to less query strings being generated and faster cache look-up, but I still think we can do better. In my mind, we can do something like the following. Of course, this is very high-level.

```
Table table = findTable(tableName);
Index idx = table.findIndex(idxName);
IndexCursor cursor = idx.getLazyCursor(multiplex);
return new ResultSet(cursor);
```

Note that for tt.* (as you generate), H2 will actually generate an ExpressionColumn for each field. In tables with 300 fields, this is not optimal. This happens in query strings, but we can avoid in direct-access.
The planning can be also avoided. Try to check if you can minimally replicate what you need in a SelectAll. One single TableFiler with no special preprocessing for group by / order by / joins, etc.

If you see any obvious impediments for approaching this, please let me know.

#11 - 01/09/2024 08:03 AM - Alexandru Lungu

Radu, lets see how we can "resurrect" this task. I think the initial statistic of **lots** of SELECT statements that select all data is still valid. Also, the initial implementation you had with direct-access seemed to provide some decent improvements. I still am keen on having something like #7252-10, so either bring back to life 7252a, cherry-pick what you need and eventually create a more up-to-date branch.

First and foremost:

- See how many such queries are on POC. I just want to confirm my initial findings of ~20% of the queries being of type "select all records".

Later on:

- Detect as soon as possible if we have a "select everything" kind of query from an AdaptiveQuery.
- If we do, detect the index we want. We can extract its name and pass it to the direct-access late on **OR** simply extract the fields and use a routine like the one in getUniqueRow (which searches the suitable index for the fields provided). I would opt for the FWD-side implementation using IndexHelper
- Pass to the direct-access the table name, multiplex and index.

- Implement the prototype from #7252-10 as simple as possible to have some initial metrics on performance.

#12 - 01/15/2024 08:37 AM - Radu Apetrii

Alexandru Lungu wrote:

> First and foremost:
> See how many such queries are on POC. I just want to confirm my initial findings of ~20% of the queries being of type "select all records".

Now that POC is usable again, I've gathered some statistics on this. From my findings, there were roughly ~33.5% queries of type "select all records", which is a bit more than your initial gatherings. I'll move on with the other steps.

#13 - 01/16/2024 04:46 AM - Alexandru Lungu

There are a lot of H2 queries that were replaced with direct-access. Is not that the "select all records" increased, but the total number of SQL queries decreased.

#14 - 01/19/2024 09:02 AM - Radu Apetrii

I've created branches 7252b and 7252b_h2 for the experiment with the SelectAll class.

#15 - 01/23/2024 10:03 AM - Radu Apetrii

- Status changed from WIP to Review

- % Done changed from 60 to 100

I've committed to both 7252b and 7252b_h2 the following things:

- 7252b **rev. 14933** - Added attempt to solve a Select All type of AdaptiveQuery with DirectAccess. In here, FWD checks if such a query exists and can be resolved with DirectAccess, passing the arguments mentioned in #7252-11.
- 7252b_h2 **rev. 40** - Added fix for the TestLazy tests that were not working.
- 7252b_h2 **rev. 41** - Added fix for the TestSoftUniqueValidate tests that were not working.
- 7252b_h2 **rev. 42** - Added SelectAll class which uses LazyResult to iterate the records from a given table with respect to the multiplex given.

In terms of tests, I've only done a few, but they all passed. While waiting for a review, I'll run the POC regression and performance tests.

#16 - 01/25/2024 05:10 AM - Radu Apetrii

- Status changed from Review to WIP

- % Done changed from 100 to 90

I'll move this back to WIP because I encounter a NPE in DataSetSDOHelper.getDataObjects in a test. I'm looking into it.

#17 - 02/05/2024 11:02 PM - Radu Apetrii

I've finally managed to isolate that NPE issue I encountered. Long story short (I say "short" because I had a really detailed post written earlier, but before I had the chance to submit it, my computer completely froze and I was unable to salvage anything), the root of the problem is the process of extracting the index name for passing it to DirectAccess. I'll explain in an example below:

- Assume there is a temporary table tt with two fields: f1 as int, f2 as int. I'll exclude multiplex and recid from further points as they do not affect the result at all.
- There are two records created for tt: (1, 2) and (2, 1).
- There are two indexes for tt:
  - idx1 on column f1 asc which is also the primary index. Knowing that the index is a TreeIndex, the structure would look like:
    - Record (1, 2) is the root.
    - Record (2, 1) represents the right child of the root.
  - idx2 on column f2 asc. Structure of this index:
    - Record (2, 1) is the root.
    - Record (1, 2) represents the right child of the root.

Now, if I were to execute a query such as

```
for each tt by tt.f2:
    message tt.f1 tt.f2.
    leave.
end.
```

This would print 1 2 (which is wrong) because DirectAccess receives the name of the **primary** index instead of orienting itself based on the sort clause too. If one was to iterate the first record of the correct index, idx2, then the result would be 2 1. Possible fixes:

- Disable DirectAccess with SelectAll for queries that contain an order by. I personally **dislike** this approach as I think we would neglect a potential significant performance boost. I noticed that quite a lot of "select all" queries contain an order by clause.
- Detect the correct index from the start and perform such queries with SelectAll. There is also a benefit from the fact that SelectAll uses LazyResult which means that it is sensitive to changes such as deletes and inserts. The problem with this approach is that I do not know how to get the SQL name of the index directly from classes like AdaptiveQuery, if this is even possible at all.
- If a sort clause is identified, instead of passing the index name to H2, pass a parameter that tells H2 to do the index retrieving itself. This can definitely be done, I'm just not sure how straight-forward it would be to implement.

Additional notes:

- In AdaptiveQuery I have access to the TemporaryBuffer in use, IndexHelper, and the ORDER BY clause.
- The current implementation uses ((TemporaryBuffer) buffer).getImplicitSqlIndexName to retrieve the SQL index name, but this works only for getting the primary index name.
- I've tried doing indexHelper.getIndexForSort(buffer, sortClause), but it looks like the name retrieved represents the P2J index name, not the SQL one.

If more information is needed, I will gladly provide feedback.

#18 - 02/06/2024 07:26 AM - Radu Apetrii

For reference, if one were to assume that an index called idx2 would be created for table tt, then the following would happen:

- By using indexHelper.getIndexForSort(buffer, sortClause), the retrieved name would be idx2. This is the P2J index name which doesn't help with the situation.
- The actual SQL index name from the table in H2 is IDX__TT17_IDX2__1. This is what I want to have in AdaptiveQuery to assure a smooth and error-free DirectAccess process.

Is there a way to retrieve the SQL index name? Note that I work with a non-primary index, thus buffer.getImplicitSqlIndexName() is of no use.
If the response is negative, then I will try letting H2 know that the index should be searched and it is not already given.

#19 - 02/06/2024 08:49 AM - Greg Shah

For static queries, we know the 4GL name of the selected index at conversion time. We can easily pass that to the query constructor (we may already do that for OPEN QUERY cases). If we store mapping information, then the converted index names (FWD and SQL) could be easily found.

It seems better to "remember" this information than to calculate it at runtime.

#20 - 02/07/2024 11:53 AM - Radu Apetrii

Greg Shah wrote:

> It seems better to "remember" this information than to calculate it at runtime.

I'll add this idea to the current solution and then do some performance testing, now that I got them working.

#21 - 02/07/2024 07:19 PM - Radu Apetrii

I've committed to:

- **7252b rev. 14934** a change for better handling of index choosing in AdaptiveQuery.getAllLazyDirectAccess. This does not contain Greg's suggestion, as I wanted to create a checkpoint before integrating those changes.
- **7252b_h2 rev. 43** a change in SelectAll that targets the handling of the cases in which a cursor resets or it is placed before/after the results.

The tests that I have done so far provide correct results, but I am yet to do performance testing.

#22 - 02/08/2024 05:09 AM - Radu Apetrii

Alex/Greg: Are there any other ways of generating a "Select all" type of **static** query that turns into an AdaptiveQuery, other than for each tt [...] and open query q for each tt [...]?

For open query cases Greg was right in saying that FWD stores the index-information. I've added the same operation for for each cases, and I'm wondering if there are other ones which I should investigate.

#23 - 02/08/2024 05:24 AM - Alexandru Lungu

You can have compound queries. A component can be instantiated as an AdaptiveQuery. See for each tt, each tt2, as it will generate 2 AdaptiveQuery instances. The more "dangerous" code is when optimized for each tt, each tt2 where tt.id = tt2.fk as this will generate a multi-table AdaptiveQuery.

#24 - 02/08/2024 06:35 AM - Radu Apetrii

Alexandru Lungu wrote:

> You can have compound queries. A component can be instantiated as an AdaptiveQuery. See for each tt, each tt2, as it will generate 2 AdaptiveQuery instances.

I will create a test with this and check if the process is correct.

> The more "dangerous" code is when optimized for each tt, each tt2 where tt.id = tt2.fk as this will generate a multi-table AdaptiveQuery.

This should not be a problem as AdaptiveQuery.getAllLazyDirectAccess only works with queries that have a single component.

#25 - 02/08/2024 06:42 AM - Alexandru Lungu

Radu, if the solution is stable. Lets get some profiling results asap.
Please go in-depth with some JMX to check the performance difference in the areas you altered the code. First and foremost, we need to ensure the effort here is driven by big potentials.

#26 - 02/08/2024 09:01 AM - Radu Apetrii

In terms of JMX, results look promising. Out of a million of "Select all" type of queries, the DirectAccess method performs ~**29.89%** faster than the current implementation. Note that the times were measured just for those type of queries, not for the whole application, nor for all of the executed queries.

Next steps:

- Commit Greg's idea. I have implemented and tested it, but not on a large application.
- Do a full round of profiling. This might take a bit since I need to do a reconversion of POC to get the full effect of the changes.