Database - Feature #4996

Integrate hash indexes in H2 temp-tables

11/04/2020 12:43 PM - Adrian Lungu

Status:	WIP	**Start date:**	
Priority:	Normal	**Due date:**	
Assignee:	Adrian Lungu	**% Done:**	20%
Category:		**Estimated time:**	0.00 hour
Target version:			
billable:	No	**vendor_id:**	GCD

Description

History

#1 - 11/06/2020 04:31 AM - Adrian Lungu

The goal of this task is to explore hash index potential uses for the H2 temp-tables. For example, this can boost the queries which use the _multiplex field (which are a lot). All these queries do interrogate _multiplex for equality, so the hash index may fit perfectly.

However, H2's means of using hash indexes are trivial. Presuming we want to add something like create hash index idx_tt1_multiplex_hash on tt1 (_multiplex);. This has several implications:

- In H2, if the where clause contains a _multiplex equality, than this index with have a very low cost. This means that this index will always be used.
- In case the field is not unique (which is happens as _multiplex is not unique), the NonUniqueHashIndex implementation is used. This kind of implementation stores the rows based on the _multiplex value. Basically, there is no hash; the bucket identifier is the _multiplex value. This index can be dangerous, as it will only do a "preselection" of the rows having a specific _multiplex and nothing more.
- The hash index works with only one field according to the NonUniqueHashIndex implementation.
- Based on the fact that this index will always be chosen in the query plan and the hash index only preselects rows based on the _multiplex value, all our queries will lack index optimization (on all fields excepting _multiplex).
- The insert works relatively fast for hash indexes. However, the remove is done linear; this means that the bucket is entirely traversed in order to find the row to delete. On a generated testcase inserting ten million rows and removing them afterwards results in 1.7s (add time) and 25.3s (remove time) only on the hash index. I tried to optimize this with an eventual "bulkRemove", but no significant change in more complex scenarios. Reimplementing NonUniqueHashIndex such that each bucket is a HashSet or LinkedHashSet results in a huge lack of performance. This is because a cursor on this index will have to work with an Interator and the next operation is costly (even on streams). Note that "cursor traversal" is a way more encountered operation than "remove from index". Finally, the last resort was to store the buckets as sets and the cursor to be computed over an ArrayList of rows. Unfortunately, this increases the index find operation time from 1.2s to 34.1s due to the copy operation from set to array.

It starts to be hard to find value in hash indexes, unless they are created for unique fields (or almost unique), which isn't the case for _multiplex. Going to do some slim investigation for hash indexes in the context of the recid field.

#2 - 11/11/2020 03:01 PM - Eric Faulhaber

Adrian Lungu wrote:

> In H2, if the where clause contains a _multiplex equality, than this index with have a very low cost. This means that this index will always be used.

If I understand your statement correctly, I do not think we want this.

When possible, we want H2 to use an index which does not require that it performs a sort of the initial result set after record retrieval. If the hash index on _multiplex is always chosen, we will **always** end up performing a sort of the initially fetched records to match the query's ORDER BY clause. In the majority of cases, the ORDER BY clause we use will have been generated to match a legacy index, and the results must be returned in this order to match legacy behavior. We may get a performance boost in the initial fetch, but I'm concerned that we will always give that improvement back (and probably more), if we have to then sort the records to match the ORDER BY clause.

Ideally, the initial fetch will be done using an index which matches the ORDER BY clause, so no explicit sort will be necessary. However, this cannot be guaranteed, since AFAIK, H2 also will consider the components of the WHERE clause, in combination with its selectivity statistics, when choosing a query plan. If we game the planner by creating a _multiplex index which is likely always to be chosen, it seems we will never take advantage of the situation where an explicit sort is not needed.

If I have misunderstood your comment, please clarify.

#3 - 11/11/2020 05:19 PM - Eric Faulhaber

Sorry, I re-read your post and it seems I had misread your conclusion on my first pass. Nevertheless, I expect we would have a similar issue with always using an index on the recid field, in terms of the sort cost (unless you had something else in mind...).

#4 - 11/11/2020 09:05 PM - Ovidiu Maxiniuc

IMHO, we can not benefit much from the hash indexes. The way I understand the H2 documentation (H2 Performance / In-Memory (Hash) Indexes), these can be used only if the key is unique. They *"only supports direct lookup (WHERE ID = ?)"* and are activated when using *"CREATE UNIQUE HASH INDEX and CREATE TABLE ...(ID INT PRIMARY KEY HASH,...)"* constructs. As Adrian mentioned, our _multiplex is not unique, it is shared among all records in 'same' TEMP-TABLE.

It looks to me the HASH indexes are a fast solution for a rather slim set of queries, similar to what we do in "find-by-rowid" queries. This is because the rowid of a record is indeed, unique, and these queries will match the direct lookup predicate. Maybe it is worth investigating this direction (although this is a quite rare usecase)?

#5 - 11/12/2020 04:24 AM - Adrian Lungu

#4996-4 is quite on spot. I may add that H2 allows hash indexes for fields which are not unique, but the queried records will be scanned (not indexed) and eventually reordered (#4996-2). This is why my conclusion in #4996-1 is to use hash indexes only for unique field; or almost unique, meaning that

there is a small number of potentially identical fields so the scan and reorder won't be such an issue.

As Ovidiu mentioned, for now hash indexes can help in "find-by-rowid" queries, as we can add a hash index for recid (which is unique). However, we need to check if such queries are enough to justify the insert time overhead.

What I was hoping in the first place was to have an index which could rapidly resolve the _multiplex = ? where clause and afterwards find a suitable index which will filter the records (and eventually have them ordered). This sounds more like a "compound index" (a hash index in which the buckets are tree indexed). However, this is far from H2's standard index capabilities, but may fit FWD queries needs.

#6 - 11/12/2020 12:42 PM - Ovidiu Maxiniuc

Adrian Lungu wrote in #4996-1:

- The hash index works with only one field according to the NonUniqueHashIndex implementation.

This is a big constraint, I think. According to H2 manual, it will silently ignore the HASH option for indexes defined with multiple components. If we intend to use them with temp-tables, that component will be _multiplex, as it is mandatory for TEMP-TABLES. Which means H2 will be able to quickly isolate the requested temp-table records, but the rest of the query predicate will be processed *client-side* if the HASH index is available and H2 chooses a query plan based on it.

Ovidiu Maxiniuc wrote in #4996-4:

> It looks to me the HASH indexes are a fast solution for a rather slim set of queries, similar to what we do in "find-by-rowid" queries. This is because the rowid of a record is indeed, unique, and these queries will match the direct lookup predicate. Maybe it is worth investigating this direction (although this is a quite rare usecase)?

Further thinking on the subject: if we notice multiple "find-by-rowid" queries for a specified table we can intentionally create an additional HASH index for temp-tables (without _multiplex component as the rowid key is unique). In this case, H2 will probably select it based on the way we generate this kind of SQLs. If the "find-by-rowid" query is performed on a permanent table, the index will be added only for H2 dialect, if it is found in configuration file.

#7 - 11/12/2020 02:24 PM - Greg Shah

Could a view be useful here? I know much depends on how the H2 implements views and how the query optimizer handles them. Since views are usually most performant when they are used to filter things, I wonder if views could let the hash index be used for the view part and then the subsequent "normal" business index could be used for the actual query. This would mean that the indexes would need to be duplicated on the view itself, if it works.

#8 - 11/12/2020 02:51 PM - Eric Faulhaber

Adrian Lungu wrote:

> As Ovidiu mentioned, for now hash indexes can help in "find-by-rowid" queries, as we can add a hash index for recid (which is unique). However, we need to check if such queries are enough to justify the insert time overhead.

What is the benefit of adding a second index on the primary key? AFAIK, recid already has a dedicated, unique index by virtue of being the primary key. Wouldn't H2 already use the fastest type of index (presumably a hash index) for this purpose?

#9 - 11/13/2020 04:01 AM - Adrian Lungu

Not really. H2 prefers a more "general use" index for the primary key - a tree index. This is because it prefers fast range queries rather than super-fast equality queries. However, there is this syntax: primary key hash recid in order to state that the index associated with the primary key is hash. **Unless we ever use range queries on recid**, we can move our tree-indexed primary keys to hash indexes. This might make all operations faster I guess (insert, select, remove), as no tree look-ups are made anymore.

#10 - 11/20/2020 08:25 AM - Adrian Lungu

- % Done changed from 0 to 20

I tried out changing tree indexed primary keys to hash indexed primary keys (recid). The modification doesn't change the performance so much on a real customer application. The test had 2.3 million SQL statements, from which ~6500 were of type recid = ? (resolved using a hash index) and ~3200 were of type recid > ? (but were resolved using a tree index). The test was run in 4.6 seconds for both configurations. I suppose that the temp-tables in my testcase did not contained enough data to justify the hash index.

This type of indexes were useful in the #4055-54 approach, but in the meantime #4055-70 doesn't need the hash index optimization anymore. Given the not so encouraging results, I think we can delay the integration of hash indexes until a more powerful use case can be found. Maybe other fields can benefit from hash indexes (apart from recid?).