Database - Feature #7459

Implement BucketIndex in FWD-H2 to support multiplex

06/22/2023 09:06 AM - Alexandru Lungu

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

Description

History

#1 - 06/22/2023 09:28 AM - Alexandru Lungu

- Tracker changed from Bug to Feature

This task is inside my head for a really long time. I had an attempt at some point to implement it, but I didn't had a proper plan back then.

This feature is about having a custom type of index that allows multiple fields. The syntax is highly experimental, but I think is kind of suggestive:

```
create index idx1 on tt (f1, f2, recid) group by (_multiplex)
```

With such index, I want to achieve hashing on group by fields and b-tree on each bucket. This is similar to separating the multiplex across multiple tables, but AFAIK, #6713 proved that having multiple tables is an overhead for small scoped multiplex. This is because one should create / drop a table (and its indexes) to represent some records in a multiplex. There are lots of cases where a multiplex has < 5 records, so generating / parsing / executing the SQL for CREATE TABLE, CREATE INDEX and DROP were too big comparing to the actual work done on the tables. The only upside was that DROP was faster than DELETE; however, parsing / executing the CREATE was slow.

I don't have an idea how we can do a prerequisite profiling on this, to understand how a BucketIndex can help. However, I think such index can help a lot with direct-access, as we can clear multiplex buckets very fast, comparing to DELETE FROM tt WHERE _multiplex = ?.

As for implementation details, I always found BucketIndex quite accessible as it aggregates a HashIndex and a TreeIndex. I suppose something like HashMap<Value, Index> is the core structure here. I don't think we should bother implementing multi-field group by as I don't see a use of it now.

There may be some overhead in the planning part as the plan with group by index doesn't look trivial. I guess it may relate to the classic TreeIndex plan, but give reasonable score only when the grouped field is selected and is used with =. As for the ORDER BY, FWD always tried to fit the _multiplex in the ORDER BY just to match the index. However, there is no functional reasoning behind. Thus, we should change FWD to avoid injecting _multiplex in the ORDER BY anymore.

The final goal here is:

- almost no overhead in creating a new multiplex: map.put(<new-multiplex-value>, new TreeIndex())
- most probable there won't be any performance increase in queries, inserts, deletes and updates. With our current TreeIndex, there are only some integer comparisons to resolve the "bucket" - multiplex sub-tree.
- this whole work will reflect in the BATCH DELETE operations. These will be really fast and won't need any index traversal.

I will put a little more thought in this, but I would like to know your input here. The effort in FWD-H2 is considerable, but yet not impossible. In FWD, we should do some careful performance / functional testing. Anyway, I will go ahead understanding what is the total time spent in DELETE tt WHERE _multiplex = ?.

#2 - 07/03/2023 06:42 AM - Alexandru Lungu

- % Done changed from 0 to 20

- Assignee set to Alexandru Lungu

- Status changed from New to WIP

I've done some basic implementation for this just to test out the hypothesis. I have:

```
conn.createStatement().execute("create local temporary table tt(_multiplex int, recid int);");
conn.createStatement().execute("create index idx1 on tt (recid) group by (_multiplex)");
```

and

```
conn.createStatement().execute("create local temporary table tt(_multiplex int, recid int);");
conn.createStatement().execute("create index idx1 on tt (_multiplex, recid)");
```

I've got the following with 100 multiplex of 1000 records each. I agree that this is unrealistic, but allows me to compute the times properly:

```
Bucket insert: 53.67ms
Bucket query: 2.27ms
Bucket update: 58.21ms
Bucket delete direct-access: 6.85ms
Bucket thrash: 10.06ms

Tree insert: 53.29ms
Tree query: 3.66ms
Tree update: 73.86ms
Tree delete: 37.31ms
Tree trash: 8.65ms
```

Tests:

- insert: 1000 records at 100 multiplex value - one single prepared statement.
- query: retrieve first 3 records at each multiplex 1.000 times
- update: overwrite all recid of first 10 entries at each multiplex + overwrite each multiplex value
- delete: delete all records from a multiplex
- thrash: insert 1 record, deletes all records - repeated 10.000 times

The tests are executed on a warm database. As expected, inserts and queries are similarly fast (hash vs several tree comparisons). Updates are just a bit faster on bucket index - quite strange. For bulk delete, bucket index is really fast. Note that this is a case where only one index is defined. If there are several, the optimization degree is even larger. The only constraint I had here is that the scan index should be consistent, so I still have to iterate all deleted rows. On the other side, buckets are cleared almost instantly, comparing to tree indexes that need rebalance after each node removed.

The biggest concern now is that "thrash" tests are always worse on buckets, due to the need of constantly initializing and dropping hash buckets.

The implementation is highly experimental, but it is good news that this out-performs tree indexes. I will do some final touches to the implementation an commit. Afterward, I will try to integrate it in FWD. For this, I will need to:

- rewrite queries that force multiplex into the ORDER BY - hopefully I won't see any performance drop at this point
- if this works, I will change the SQL multiplex pruning into direct-access pruning - I expect to see some improvement here.

From my profiling, there are lots of such multiplex pruning. One call is not expensive, but there are tens of thousands in a POC.