Database - Feature #4018

possibly move temp-table support into persistent database

04/01/2019 10:25 AM - Eric Faulhaber

Status:	New	**Start date:**	
Priority:	Normal	**Due date:**	
Assignee:		**% Done:**	0%
Category:		**Estimated time:**	0.00 hour
Target version:			
billable:	No	**version:**	
vendor_id:	GCD		

Description

History

#1 - 04/01/2019 10:55 AM - Eric Faulhaber

In the early days of FWD, temp-table support was implemented in the persistent database (e.g., PostgreSQL). To avoid the overhead of moving data over the network stack and potentially across the wire (depending on the deployment model), at some point we re-implemented temp-tables within an embedded, in-memory H2 database. This improved throughput w.r.t. I/O (reads and write), at the expense of disabling server-side joins and sub-selects between temp-tables and persistent tables.

The purpose of this task is first to explore whether there are use cases where performance actually would be better if that support were moved back into the persistent database. The assumption is that such use cases would have low to moderate I/O requirements, plus the requirement to join temp-tables with persistent tables, such that a server-side join would result in a more performant implementation for those use cases than the current model.

Currently, idioms such as:

```
FOR EACH some-temp-table ...,
    EACH some-persistent-table ...
```

and

```
FIND FIRST some-persistent-table
     WHERE CAN-FIND (FIRST some-temp-table ...) ...
```

cannot be optimized to use server-side joins or sub-selects, respectively. This is due to the fact that temporary tables and persistent tables are implemented in physically separate databases. Any join must take place on the database client (i.e., the FWD application server). Thus, conversion rules create sub-optimal Java query constructs from these idioms, such as CompoundQuery (in the first case) or client-side WHERE clause expressions (in the second case). Each of these must bring records back from both databases and join them on the FWD server. In most cases, far more records will be returned to the FWD server to perform this join (which is a form of filtering) than would be the case for a server-side join.

At least for the query (read) aspect of these idioms, we would want the temporary table and the persistent table to exist in the same database. However, this relocation must be balanced against the cost of the insert/update (write) aspect, which must first populate the temporary table(s) with the necessary data. If a particular scenario must write a lot of data to a temporary table before any query activity takes place, the potential performance improvement of the query may easily be outweighed by the increased cost of moving data over the network and into a temporary table in the primary database.

So, we cannot simply move all temp-tables back into the persistent database, or we will just be back to the original problem which led us to implement the embedded H2 database in the first place. This must be done selectively, to find the right balance which provides optimal performance. This suggests the decision probably needs to be made at runtime, possibly with input from information gathered during conversion.

Another issue to be resolved is, for those scenarios where this relocation makes sense from a performance standpoint, how do we determine which persistent database is the correct target for the temporary table, in a system with multiple persistent databases? Again, this would have to be determined either through conversion hints (if the converted code makes this decision statically); or through some runtime heuristics and possibly the use of statistics collection; or some combination of both.

#2 - 04/01/2019 11:02 AM - Eric Faulhaber

We already do a form of optimization for CompoundQuery, where we attempt to determine whether a multi-table query which was converted as a client-side (i.e., FWD server) join can be optimized to a database server-side join. Currently, the rules engine which attempts this optimization aborts when it encounters a temp-table joined to a persistent table, since they are in physically separate databases. However, can we collect information from this attempt, such that the next time this temp-table is used in this scenario, it is created in the persistent database, such that the server-side join is possible?

In other words, can we enhance and leverage the existing compound query optimizer to gather intelligence which can help with this implementation?