Database - Feature #4681

prepared statement caching/pooling

06/16/2020 12:56 PM - Eric Faulhaber

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

Description

Related issues:

Related to Database - Feature #4011: database/persistence layer performance i...	**Closed**

History

#1 - 06/16/2020 12:58 PM - Eric Faulhaber

This issue is to discuss the implementation of PreparedStatement caching/pooling as a potential performance optimization.

#2 - 06/16/2020 01:39 PM - Ovidiu Maxiniuc

My concern is related to multiple session close (they have a short life), which will invalidate the cached statements associated with respective connection. I am not saying that we will not benefit from this, but I estimate the gain will be marginal, when the advantage of using an already ready statement is combined with the extra management and occupied resources while item are cached.

In majority of cases if not all of them, we use prepared statements like this:

```
    Connection conn = session.getConnection();
    checkClosed(conn);
    stmt = conn.prepareStatement(sql);
    setParameters(stmt);
    ResultSet resultSet = stmt.executeQuery();
    hydrate(resultSet);
    stmt.close();
```

There are about 20-30 places where something similar is done and I guess we should create a manager to cover then all. Then, to cache the statements we need to rewrite the above code like:

```
    PreparedStatementManager.get().getPreparedStatement(session.getConnection(), sql);
    setParameters(stmt);
    ResultSet resultSet = stmt.executeQuery();
    hydrate(resultSet);
```

The manager will probably store the statement double mapped: by their sql, and by connection. The getPreparedStatement() will make sure the connection is valid, and if new, a new entry will be added. We will also need a connection close callback which will be called when a session is closed. Here is a draft of such class:

```
    private static Map<Connection, Map<String, PreparedStatement>> data = new HashMap<>();

    public PreparedStatement getPreparedStatement(Connection conn, String sql)
    throws SQLException
    {
       if (conn.isClosed())
       {
          connectionClosed(conn);
          return null; // or throw something
       }
```

```
      Map<String, PreparedStatement> connMap = data.computeIfAbsent(conn, k -> new HashMap<>());
      return connMap.computeIfAbsent(sql, k -> conn.prepareStatement(sql));
   }

   public void connectionClosed(Connection conn)
   throws SQLException
   {
      Map<String, PreparedStatement> map = data.remove(conn);
      if (map == null)
      {
         return;
      }
      for (Map.Entry<String, PreparedStatement> stringPreparedStatementEntry : map.entrySet())
      {
         stringPreparedStatementEntry.getValue().close();
      }
   }
```

Of course, the naive map can be replaced with a more intelligent cache we already have, but when a connection is dropped because of cache constraints (time or space), the associated statements should be closed using connectionClosed() method.

Some synchronization is necessary if we use static data as above, or make the instance context local. I estimate a slightly better performance this way (static).

#3 - 06/16/2020 04:36 PM - Eric Faulhaber

I think we can simplify this idea a bit. I don't see the value in making a static cache. A PreparedStatement instance can only work with the connection that created it, so if we were to create a statement cache, it makes sense for it to live close to its connection and for it to be cleared/discarded when that connection is closed (or returned to the pool). As such, I think it makes sense to store it within the Session object itself, since the Session object's lifespan also is tied to that of the connection. As long as you have a reference to the current Session instance when you need to prepare/re-use a statement, this avoids any need for synchronization or context-local lookup. It also avoids an extra layer of lookup by connection, since the relationship between statement cache and connection becomes implicit.

However, many production-ready connection pools, including c3p0, offer statement pooling as a feature. I think it makes the most sense to investigate this first, before rolling our own solution. Currently, our temporary table data source provider does not use c3p0 as a backing data source, but we should implement this, instead of the current DriverManager.getConnection approach, which is very slow when using many, short-lived sessions.

Although I already have this c3p0 setting in the directory for use with 4011a:

```
<node class="integer" name="maxStatementsPerConnection">
  <node-attribute name="value" value="100"/>
</node>
```

...I'm not sure that is quite enough to leverage this feature properly. There are some things to figure out and some benchmark testing to be done to make sure we are actually gaining the advantage of the pooling. At minimum, we have to actually be using c3p0 behind the temp table provider, but I think there may also be some tuning we need to do. For instance, we currently set prepareThreshold to 1 in directory.xml, but it looks like I didn't carry over support for this JDBC parameter/setting to 4011a, so it is being set to whatever the default value is (3?). Furthermore, the c3p0 documentation and an independent article suggests that the statement pooling could actually be detrimental to performance in some cases and should be disabled.

See:

- https://www.mchange.com/projects/c3p0/#configuring_statement_pooling
- https://stackoverflow.com/questions/2920740/should-i-activate-c3p0-statement-pooling (a little old, but probably still relevant)
- https://developer.bring.com/blog/c3po-issue-in-booking/

#4 - 06/16/2020 04:37 PM - Eric Faulhaber

- Related to Feature #4011: database/persistence layer performance improvements added