Database - Feature #4369

implement stateless FWD server clustering

10/24/2019 02:03 PM - Greg Shah

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

Related issues:

Related to Database - Feature #1879: push lock manager back into database	**New**
Related to Runtime Infrastructure - Feature #5170: add support for cloud-base...	**New**

History

#1 - 10/24/2019 02:53 PM - Greg Shah

The core idea here is that if customers have written their application to be stateless in the 4GL, then today they can implement multiple appservers/PASOE instances and take advantage of horizontal scaling of the appserver tier while SHARING access to one or more DATABASE instances. We have work to do in enabling to this approach.

- Customers wanting to move to this mode will need to ensure their application is truly stateless. If not, then it will not be able to work in a clustered environment. This limitation would also exist in the 4GL.
- All FWD servers would be required to use the same converted application jar(s).

#3 - 10/24/2019 03:06 PM - Greg Shah

- Related to Feature #1879: push lock manager back into database added

#4 - 10/24/2019 03:09 PM - Greg Shah

After a discussion with Eric, we have this list of items that are persistence related.

- dirty share database
  - This is a quirk of the 4GL implementation where uncommitted index updates in one session are visible to other sessions. It has multiple implications that can be seen from 4GL code.
  - queries in one session can traverse uncommitted records from another session
    - Our idea at this time is that we WILL NOT support this in stateless clustering mode.
    - This is part of the changes that would be needed to make an application stateless.
  - unique constraint violation that is detected early in the 4GL because the index changes are visible across sessions
    - This part needs support.
    - For performance reasons, this is currently being rewritten to use maps.
- persistence global event queue
  - Used to notify other sessions when there are index changes.
  - This is a different aspect to the same "uncommitted index updates" problem but it also can manifest in cases where there are committed changes.
  - It is used to shift record-oriented loops (e.g. FOR EACH) from adaptive into dynamic mode.
  - This will be needed.
- identity pool
  - We no longer use recycled primary keys but we do have a small batch of pre-fetched valid IDs which are specific to the server.
  - This was done for performance instead of querying the sequence for each ID.
  - This has to be made safe.
- locking
  - The current locking approach is all in-memory.

    - We must rework lock manager, _lock metadata and the lock built-in functions.
    - In #1879 we have a potential path to moving this back into the database, which would naturally resolve this issue.
    - Another approach would be to implement a clustered version of the lock manager.
- metadata usage that is done in-memory using H2
    - _connect
    - edits to _file, _field (reads would naturally be server independent but updates must be shared)
    - check each metadata table to see which ones have editable state which must be shared
- caches
    - Several persistence data structures/classes/instances are cached for performance or memory reasons today.
    - Some of these might be safe for leaving as a per-server cache with each server having a separate (unsync'd) cache.
    - Each of the caches (dynamic temp-table definitions, queries...) will need to be inspected to determine the right approach.

#5 - 10/24/2019 03:14 PM - Greg Shah

For data structures which cannot be backed by the database AND which must be shared across JVM instances in the FWD cluster, we will need to implement a facility that handles this synchronization of state in real-time. A possible solution is Hazelcast which is an Apache 2.0 licensed in-memory data grid technology designed for exactly this use-case.

#6 - 10/24/2019 03:17 PM - Greg Shah

One thing to consider is that we don't want these server instances to be hard to configure. At one time we considered a kind of directory master/directory slave approach where each server in a cluster has a read-only copy of a common multi-server directory and one server was the master where changes could be made.

We still may want to go this way. The actual changes per-server are minimal (cert/private key, IP addresses). Most of the directory can and should be shared/common.

#7 - 10/24/2019 03:19 PM - Greg Shah

At this time I am assuming the Admin Console is not changing, but we must at least make it work with the directory approach. In a perfect world, the Admin Console would be able to manage the entire cluster. This may not happen in our first pass.

#8 - 10/24/2019 03:20 PM - Greg Shah

Constantin/Ovidiu: Can you think of any other persistence or even non-persistence state which we need to share across JVMs? Also, please share any other thoughts, comments or questions.

#9 - 10/24/2019 03:50 PM - Eric Faulhaber

We also should consider the potential duplication of dynamic proxies and other dynamically generated interfaces and classes across server instances. This is touched upon with the mention of caches above and it may be enough to just look at this as a caching issue. However, our use of custom classloaders and dynamically generated classes may also complicate the clustering design. ATM, I can't specifically put my finger on a problem here, but I suspect we will hit something related to this...

#10 - 10/24/2019 04:08 PM - Constantin Asofiei

Most (if not all) of the non-persistence runtime is context-local. One part I see that may require improvements (if we decide so) is the global publish/subscribe, to work across all servers in the cluster. A review would be needed, but otherwise I don't recall non-persistence state being shared across contexts.

Eric, you mention dynamic proxies and other dynamically-generated Java bytecode - why do you think this matters? Do you think these kind of instances will be used for sending state to other servers in the cluster (like for _connect or other meta tables)?

An alternative for the meta tables would be to store it in a physical database, to which all servers in the cluster connect (maybe the same database as the app's db?). Can a user explicitly lock a record in a meta table, to edit it? If so, then it might make sense to treat these tables the same way as the user-defined tables.

And another note: I think the in-server appserver agents would be a good addition to this (I know we plan to add this).

#11 - 10/25/2019 12:30 PM - Ovidiu Maxiniuc

First, pleat let me know if I got this tight.
For me, running in a stateless mode means that the clients connect and execute independent (quick) request on one of the available server instances. This means some kind of authentication is performed at each connect. Of course, the 4GL programmer can implement a stateful application using database to store and eventual context there, but the idea is that at the next request the context is available on any of the configured servers.

How about the persistent procedures?
As I see this, they are allowed as long they are unloaded by the end of the request. Leaving them alive will break the state constraint. But, after a deeper though, they should be unloaded as when the client request ends, its context (where the persistent procedures are held) are discarded.

Can the clients execute parallel requests on different servers?

In the end, I see a rather blurred line between current FWD and the stateless support. So maybe I should better think like of what are we missing from supporting running parallel applications correctly on same backing database?

#12 - 10/25/2019 12:44 PM - Ovidiu Maxiniuc

Constantin Asofiei wrote:

> An alternative for the meta tables would be to store it in a physical database, to which all servers in the cluster connect (maybe the same database as the app's db?). Can a user explicitly lock a record in a meta table, to edit it? If so, then it might make sense to treat these tables the same way as the user-defined tables.

Yes, this is the right way to do it, I guess. The metadata are of two kind:

- persistent. They are user-editable. This is the case of _user table. To create a new account you can write a record directly to this table. This is why we have already decided to keep this particular meta-table with the rest of the database and allow editing and locking on it;
- VST (Virtual System Tables). They are readable, but not writeable in 4GL. They (usually) contain statistical information on the system/database and which are updated automatically, event-based. Probably they are reset with the 4GL/database server. Since the data should be accessible on all FWD servers it makes sense to store these tables on same database. However, because of the volatile content, they might be implemented as global temporary tables.

#13 - 10/25/2019 01:02 PM - Constantin Asofiei

Ovidiu Maxiniuc wrote:

> How about the persistent procedures?
> As I see this, they are allowed as long they are unloaded by the end of the request. Leaving them alive will break the state constraint. But, after a deeper though, they should be unloaded as when the client request ends, its context (where the persistent procedures are held) are discarded.

The business logic can decide to cache some persistent procedures, which is OK, as long as there is no request-specific data there. So, a request can be something like this:

- the user accesses a login procedure, which checks some credentials and returns a token
- on each subsequent requests, the token is passed back, so the appserver knows which app user is executing
- on a certain request, the appserver can decide to i.e. initialize some classes or other kind of persistent programs, and cache these in global vars or other mechanisms. Again, this is OK, as long as the app doesn't cache request-specific state.

Also, this cache is dependent on the appserver mode: for example, in STATELESS, the agent doesn't have its context reset, so any global-shared temp-tables or vars, or OO static fields, will remain there.

#14 - 10/25/2019 01:19 PM - Greg Shah

> I think the in-server appserver agents would be a good addition to this (I know we plan to add this).

I think so too. You'll see a task for that shortly. :)

#15 - 10/25/2019 01:22 PM - Greg Shah

> For me, running in a stateless mode means that the clients connect and execute independent (quick) request on one of the available server instances. This means some kind of authentication is performed at each connect. Of course, the 4GL programmer can implement a stateful application using database to store and eventual context there, but the idea is that at the next request the context is available on any of the configured servers.

Correct.

> Can the clients execute parallel requests on different servers?

Yes.

> In the end, I see a rather blurred line between current FWD and the stateless support. So maybe I should better think like of what are we missing from supporting running parallel applications correctly on same backing database?

Yes, this is a reasonable way to think about it.

#16 - 03/02/2021 10:11 AM - Greg Shah

- Related to Feature #5170: add support for cloud-based load balancing and WAF added

#17 - 10/03/2021 06:21 PM - Greg Shah

Another open source framework to consider is MicroStream. It can be used to replicate object graphs across multiple JVMs.

#19 - 05/17/2022 10:51 AM - Greg Shah

Evidently REDIS also has an object synchronization capability. We should evaluate this in addition to the other frameworks. Please note that one dbig advantage of REDIS is that Amazon provides a managed REDIS service so there may be some operational benefits there whereas Hazlecast would have to be application managed.