ANE-25725 Set database levels back automatically in a code rollback.

When a new version of AN is run for the first time, it executes a series of custom updates, pushing forward system.databaselevel. Similarly, there are "background DB updates" which update systeminfo. background\_db\_level.

If there is a rollback, the databaselevel will already have been advanced to the level required by the new version; similarly the background\_db\_level, while it may not have completed, might be advanced.

When the new version is redeployed, this means that these custom updates will not be re-run. So any new data entered in the database during the rollback period which might require a custom update, won't get the update.

The manual procedure to work around this is to set the two levels back to the maximum level for the lower version, so when the higher version is redeployed, they'll run again. (This does require as a design point that custom updates be designed to be safely rerunnable).

In this ticket, we'll make the servlet do this itself. If it finds the database level deployed is higher than the maximum level it knows about, it will set it back. Then no special manual work is required in a rollback.

As part of this work, a long comment was created at the top of the DBUpdateScript class, which is good background on the operation of custom updates, not all of which might be known by all developers and testers:

/\*Start Comment\*/

CUSTOM UPDATES:

This class is for "custom DB updates", which are changes to the data

(not the schema) required for a new version. Upsizer will perform all the schema updates, based on the ERD XML; however, any new columns will just get standard defaults -- "" for char, 0 for numeric, and 12/30/1899 for dates.

Custom updates are required in cases such as:

-- A new column is initialized with a non-default value, generally based on other data.

-- Records are being written to a new table, based on existing data.

-- A data repair is being done.

The code requires that custom updates for the current version be completed in order for the servlet to finish initializing. This allows the servlet to assume that all the data is updated to the current level.

To prevent two servlets from executing the same custom update at the same time, the code in SystemInfo acquires a SystemLock on the pseudo-table "SystemInfoRaiseDBLevel". As a result, all servlets for the org are blocked in initialization while one servlet is performing the custom updates.

The operation of the update process is driven by the column system.databaselevel. Each custom update is given a database level, starting at 100 (under 100 were the Safari updates prior to Safari and Activenet being forked). The update code goes through each update in sequence, executing that update if the current databaselevel is lower than the level of the update, then setting the system.databaselevel to the level just executed.

Currently, the entire update process operates on a single dbc, without any commits, which does ensure that each custom update is all-or-nothing. However, interim commits on each custom update would probably be a better approach.

At the very end of raiseDBLevel(), the code checks that it has reached a configured level required for the code to work:SystemInfo.required\_database\_level. If this isn't met,

the code throws an exception and init() fails.

When we branch the code and begin work on a new major release, we leave a gap of 10 to the first custom update of the next major release. This allows space for one or more database updates in maintenance releases on the branch, such as for data repair.

Note that when any custom updates are done in the branched code, it is necessary to analyze the custom update to determine if it is also necessary to do the same operation at a different database level in the trunk. Generally, if all orgs are running the branch, this isn't necessary, as they'll all get the custom update on the branch before they're upgraded to the trunk. However, if some orgs are already running the trunk code as part of UAT, it may be necessary to clone the same custom update code to a higher level.

ROLLBACKS:

Special considerations for custom updates apply in order to facilitate code rollbacks. Let's say we are running the servlet at version V1, at databaselevel DL1, then update it to version V2. The servlet init will perform custom updates to bring the databaselevel to DL2. Then let's say there's a problem with V2, and we have to re-install V1. Without doing anything else, the database level would stay at DL2.This means that if new records were now entered into V1, which should have gotten a custom update into V2, they won't be updated.

So to make rollbacks work, the code in DBUpdateScript (and similarly in RaiseDBLevelThread) set the database level back to their own maximum databaselevel (e.g., DL1), so the custom updates from DL1 to DL2 will all be rerun when V3 is later installed.

This leads to one critical requirement of all custom updates, which must be ensured through code review: Custom Updates \*must\* be safely rerunnable! This can generally be ensured fairly easily through design; e.g, with SQL updates:

-- If a new column is being updated from a default value, add a check in a where clause saying "where it's still at the default value"

-- If new records are being inserted, check that the records aren't already there.

ASYNCHRONOUS UPDATES:

If an update is expected to take a long time, it should be implemented as an asynchronous update, which are processed on a background thread in the class RaiseDBLevelThread. However, this adds the additional complexity of designing the code to work correctly with both old and new format data, so is rarely used for updates. Most custom updates are implemented with SQL which will run fairly quickly. On the other hand, if the update requires Java code to enumerate through records, a background update may be required.

/\*End Comment\*/