Integridad Transaccional

IT

¿Qué es el concepto: integridad transaccional?

- Un conjunto de actualizaciones a la base de datos tiene **integridad transaccional** cuando en caso de una finalización "anormal", la base de datos permanece en estado consistente.

Muchos manejadores de bases de datos (DBMSs) cuentan con sistemas de recuperación ante fallos, que permiten dejar la base de datos en estado consistente cuando ocurren imprevistos tales como apagones o caídas del sistema.

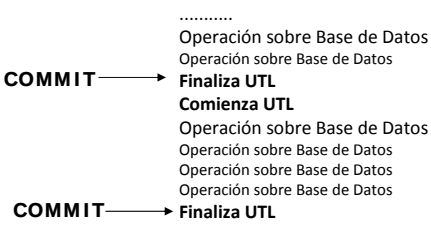
¿Qué es el concepto: unidad de trabajo lógica (UTL)?

- Una **unidad de trabajo lógica (UTL)** es un conjunto de operaciones a la base de datos, **que deben ejecutarse o bien todas o bien ninguna de ellas**.

Los manejadores de bases de datos (DBMSs) que ofrecen integridad transaccional permiten establecer unidades de trabajo lógicas (UTLs), que corresponden ni más ni menos que al concepto de transacciones de base de datos.

¿Qué es efectuar COMMIT?

- El comando COMMIT permite especificar que cierto conjunto de operaciones realizadas sobre una base de datos, ha culminado de efectuarse correctamente:

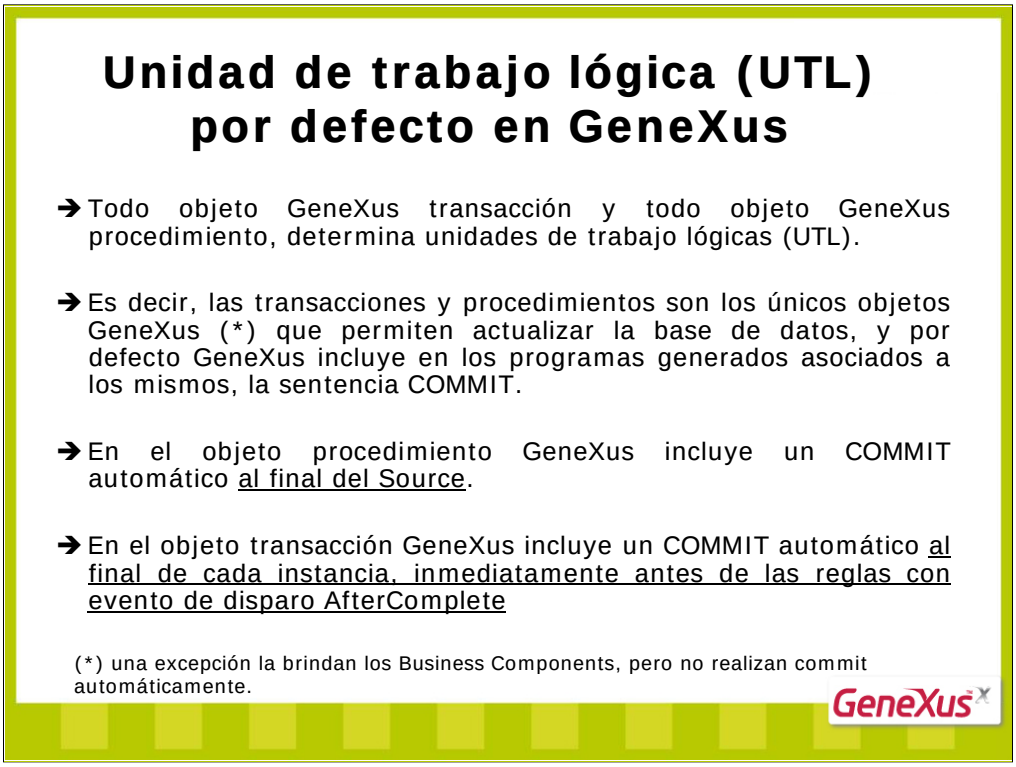


- De modo que efectuar COMMIT en una base de datos, significa que se da por finalizada una unidad de trabajo lógica (UTL).

Podemos ver que una unidad de trabajo lógica (UTL) queda definida por el conjunto de operaciones entre un par de Commits.

¿Qué es efectuar ROLLBACK?

- Hacer **ROLLBACK (vuelta a atrás)** provoca que se deshagan todas las operaciones efectuadas en la base de datos que no hayan quedado con COMMIT.
- Esto se resuelve deshaciendo todas las operaciones posteriores al último COMMIT.



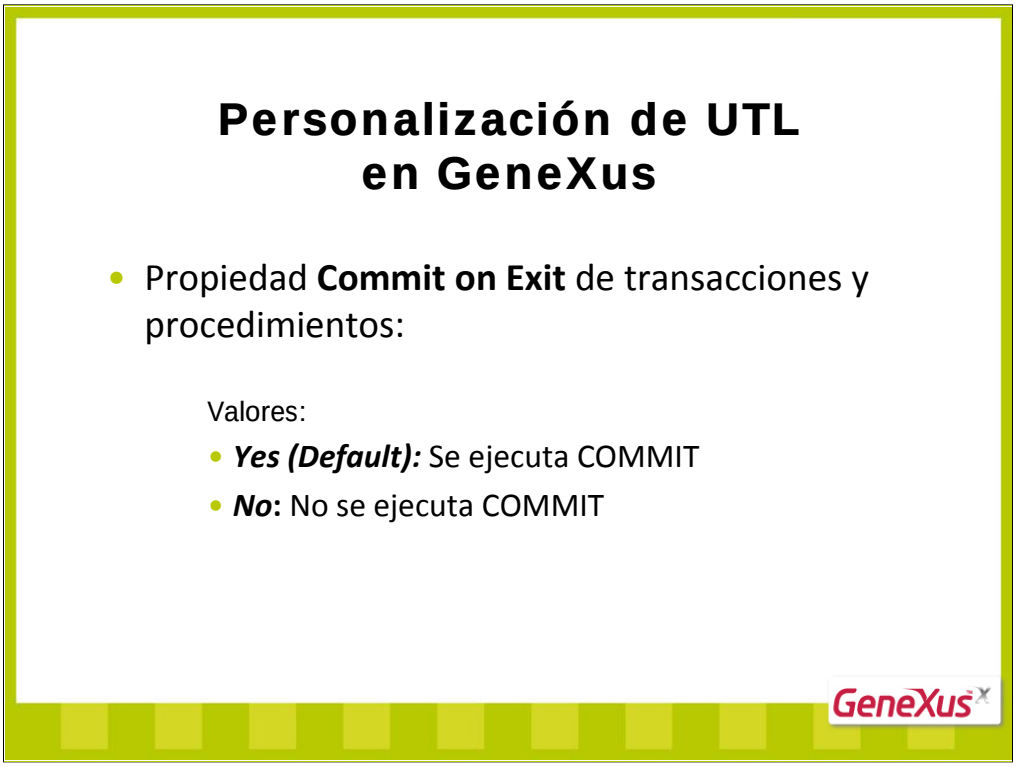
Es importante aclarar que GeneXus incluye la sentencia COMMIT en los programas generados asociados a transacciones y procedimientos, sólo en ambientes de trabajo Cliente/Servidor (incluyendo, por tanto, los ambientes Web). El motivo de esto es que en ambientes Cliente/Servidor existe un DBMS que asegura la integridad transaccional, por lo tanto GeneXus efectúa la tarea de definir las unidades de trabajo lógicas (UTLs).

¿Dónde incluye GeneXus COMMIT exactamente?

En cada procedimiento: al final del programa fuente.

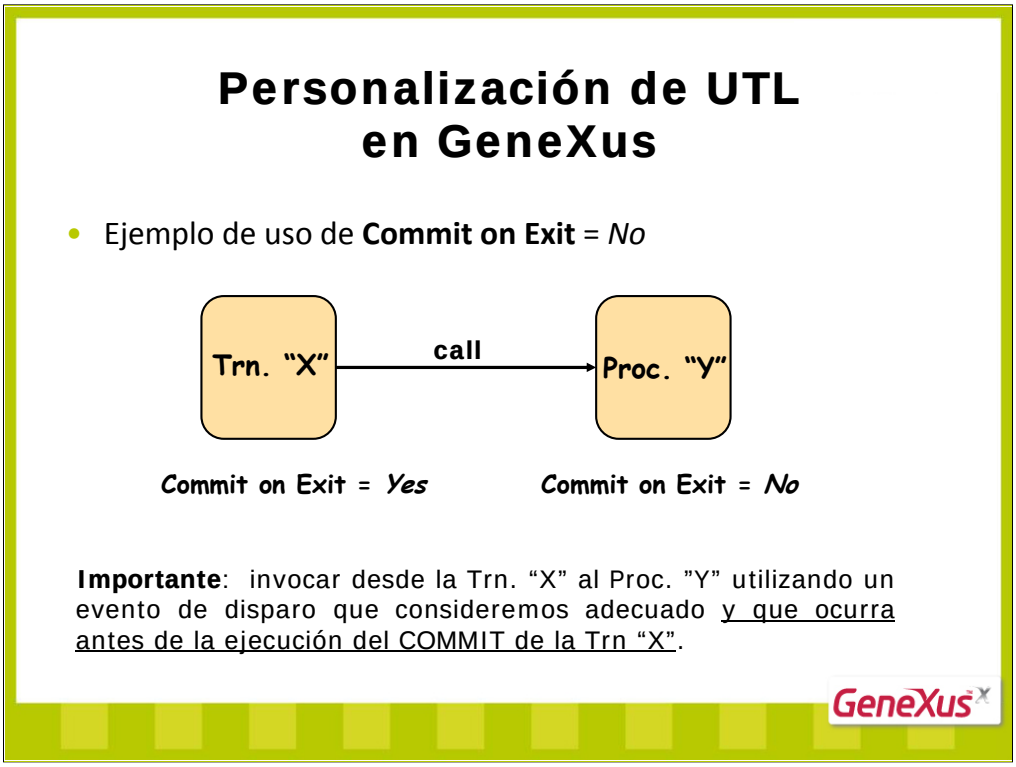
En cada transacción: inmediatamente antes de las reglas con evento de disparo AfterComplete (E inmediatamente después de las BeforeComplete). Es decir, que por cada iteración completa que se efectúe en tiempo de ejecución por medio de la transacción, habrá un COMMIT, justo antes de las reglas con evento de disparo AfterComplete.

Nota: El tipo de datos Business Component que veremos más adelante permite actualizar la base de datos desde cualquier objeto GeneXus, pero también como veremos, no realiza automáticamente un COMMIT.



GeneXus ofrece una propiedad a nivel de cada objeto transacción y procedimiento, para definir si se desea que su programa generado efectúe COMMIT, o no. El nombre de la propiedad es **Commit on Exit** y su valor por defecto es Yes (por eso, toda transacción y procedimiento por defecto efectúa COMMIT).

Si se desea que cierta transacción o procedimiento no tenga en su programa generado COMMIT, bastará con cambiar el valor de la propiedad **Commit on Exit** a No.



¿Por qué motivo se puede necesitar no efectuar COMMIT en una transacción o procedimiento?

Para personalizar una unidad de trabajo lógica (UTL). Es decir, podemos necesitar ampliar una unidad de trabajo lógica (UTL) para que varias transacciones[1] y/o procedimientos, conformen una única unidad de trabajo lógica (UTL).

Ejemplo (mostrado arriba):

La transacción "X" invoca al procedimiento "Y", y se desea que ambos objetos conformen una única UTL. La transacción actualiza ciertos registros, y el procedimiento otros, y se desea que ese conjunto total de operaciones conforme una única UTL (para asegurarnos de que si ocurre una falla, quede efectuado el conjunto completo de actualizaciones a la base de datos, o nada).

Para lograrlo podemos eliminar el COMMIT del procedimiento y dejar que se realice en la transacción (al retornar del procedimiento a la transacción, para que se ejecute al final de todas las operaciones); de modo que configuraríamos la propiedad **Commit on Exit** del procedimiento con valor: **No** y dejaríamos la propiedad **Commit on Exit** de la transacción con el valor por defecto: **Yes**. Pero además de esto, es fundamental que la invocación al procedimiento se realice antes de que se ejecute el COMMIT en la transacción (ya que la idea es que ambos objetos conformen una única UTL, y para ello el COMMIT debe efectuarse en la transacción al retornar del procedimiento); así que la invocación al procedimiento deberá definirse en la transacción, con un evento de disparo que ocurra antes de la ejecución del COMMIT (dependiendo de si la transacción es de un nivel o más, y de los requerimientos, podría servir AfterInsert por ejemplo, AfterUpdate, o AfterLevel Level *Atributo del 2do nivel*, o BeforeComplete, pero no AfterComplete).

No existe una única solución para personalizar una UTL. Lo fundamental es analizar cuál objeto puede hacer COMMIT (pudiendo haber más de una posibilidad) y una vez que se decida cuál objeto efectuará COMMIT, las invocaciones que se requieran hacer, deberán efectuarse en momentos adecuados, considerando si ya se efectuó el COMMIT o no.

---

[1] En ambiente Web existe una importante restricción a este respecto: si desde una transacción se invoca a otra, el Commit que realice una no aplica sobre los registros ingresados/modificados/eliminados por la otra. Es decir, el Commit de cada transacción solo tiene "visibilidad" sobre los registros operados por esa transacción, y no por la otra, por lo que dos transacciones distintas no pueden quedar incluidas en una misma UTL. No puede realizarse personalización en este caso.

Por ejemplo, para que la transacción y procedimiento vistos conformen una única UTL, podríamos haber optado también por la alternativa de que no efectúe COMMIT la transacción (Commit on Exit = No), sino que lo haga el procedimiento al final de todo; y de hacerlo así, no sería un error –como sí lo sería en la solución anterior- invocar al procedimiento utilizando el evento de disparo AfterComplete, porque la transacción no hará COMMIT, sino que lo hará el procedimiento.

Concluyendo, es cuestión de decidir cuál objeto hará COMMIT y que las invocaciones que se deban hacer, se hagan en momentos adecuados, para que la UTL personalizada quede bien definida.

Otro ejemplo:

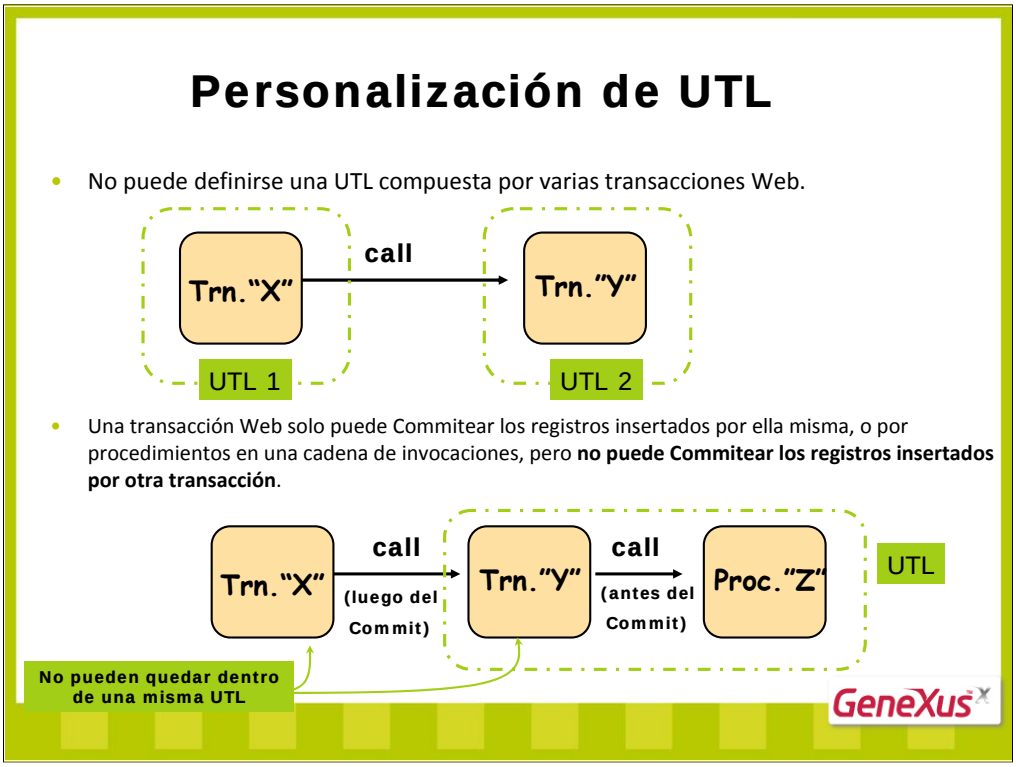
Sea la transacción “Invoice” estudiada hasta el momento. Supongamos que no modificamos el valor predeterminado de la propiedad Commit on Exit.

Supongamos ahora que el usuario ejecuta la transacción, ingresando la factura 1 con todas sus líneas. Luego pasa a ingresar la factura 2 y cuando está ingresando la 3era. línea de la misma, ocurre un apagón. Al recuperarse la energía y reiniciarse la ejecución, ¿qué registros habrán quedado grabados en las tablas y cuáles se habrán perdido?

La factura 1 íntegra estará grabada (cabezal y sus líneas). ¿Por qué? Pues porque al terminar de ingresarla y pasar a ingresar la factura 2, se efectuó un Commit. La factura 2 con los registros que se habían grabado hasta el momento de la falla de energía, se habrá perdido. ¿Por qué? Pues porque la transacción realiza el rollback de todo lo que se hubiere efectuado luego del último Commit. El cabezal de la factura 2 y las 2 líneas que se habían ingresado no estaban “commiteadas” aún.

Observar entonces que el Commit no es por transacción entera (es decir, todas las iteraciones del cabezal y sus líneas) sino por cada instancia de cabezal + líneas.

Si el Commit se realizara una única vez antes de cerrar la transacción, entonces si se hubieran ingresado 29 facturas y a la trigésima se cayera el sistema, se perderían las 29 facturas anteriores (se desharía todo, ya que aún no se habría alcanzado el Commit). Esto no es así, y si ocurriera una caída del sistema a la trigésima factura ingresada, las 29 anteriores quedarán grabadas (no así la trigésima).



En ambiente Web los registros "visibles" para ser commiteados por una transacción son los actualizados por la propia transacción, y por los procedimientos que ésta invoque antes de su Commit, pero no los de otra transacción.

Cada transacción trabaja, así, sobre UTLs distintas.

Es por ello que en el primer ejemplo presentado arriba, donde la transacción "X" llama a la transacción "Y" luego de haber insertado un registro, aunque la transacción "Y" realice un Commit al final de que cabezal y líneas sean ingresados, este Commit no valdrá sobre el registro que había sido ingresado previamente por la transacción "X". Este registro quedará "perdido", sin Commit.

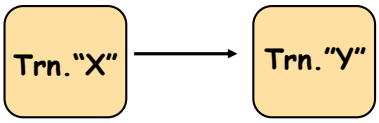
Por la forma de trabajo en Internet, las transacciones Web "viven" solamente el tiempo entre que el usuario de un navegador selecciona el link o presiona un botón y la nueva página es mostrada. Toda modificación a la base de datos que se haga durante la "vida" de la transacción debe ser confirmada o eliminada antes de que la Transacción Web termine su ejecución y retorne la página resultante.

Como consecuencia, una Transacción Web inicia una UTL (unidad de trabajo lógica) al comenzar a ejecutar y la cierra (ya sea por COMMIT o ROLLBACK) antes de terminar. No puede formar parte de otra UTL. Si un programa llama a una Transacción Web, ésta iniciará otra (nueva) UTL.

En cambio no sucede lo mismo con los procedimientos. En el segundo ejemplo mostrado arriba, vemos que podemos formar una UTL que engloba a la transacción "Y" y al procedimiento "Z"… sin embargo no podemos incluir a la transacción "X" en la misma UTL.

Personalización de UTL

- Si deseamos que las inserciones mediante dos transacciones distintas conformen una única UTL:



Tenemos una solución: utilizar Business Components y el comando Commit al terminar de insertar mediante las variables Business Components los registros asociados a ambas transacciones (se verá más adelante).

Si se necesita que las operaciones de dos o más transacciones (con o sin procedimientos incluidos) conformen una misma UTL, se pueden emular las transacciones con Web panels y Business Components y utilizar el comando Commit.

Dejamos aquí anotado simplemente el tema, para volver a él luego de estudiados los Business Components, donde nos será posible comprender esta solución.

Comandos COMMIT y ROLLBACK de GeneXus

- GeneXus ofrece los comandos: COMMIT y ROLLBACK
- Se pueden incluir en Procedimientos, Web Panels, así como en combinación con Business Components.
- Ejemplo (usuario final decide si ejecutar Commit o Rollback): Se invoca desde una transacción a varios procedimientos consecutivos, se les configura a todos ellos la propiedad **Commit on exit = No…** y en el último procedimiento se le pregunta al usuario si confirma; dependiendo de la respuesta del usuario, habrá que ejecutar el comando COMMIT o ROLLBACK