Handling Commit and Isolation levels

GeneXus™

The GeneXus objects of Transaction and Procedure type offer the property Commit on Exit that can take the value Yes or No. In this way, it is decided whether the generated programs will execute an automatic Commit.

Commit on Exit Property

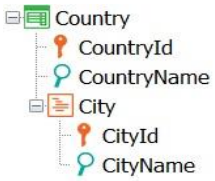
Commit on Exit = Yes

Procedures

```
For each Country
     Where CountryId = 2
     CountryName = "New country name"
endfor
```

An automatic Commit is included at the end of the source in the generated programs.

Transactions



An automatic Commit is included in the generated programs, after a modification in the database.

By default, GeneXus includes a Commit statement in the generated programs associated with objects of Transaction and Procedure type.

- In Procedures, an automatic Commit is included at the end of the Source in the generated programs.
- In Transactions, an automatic Commit is included in the generated programs, after a modification is made in the database. That is, after inserting, modifying or deleting data, and immediately **before** executing the code associated with the rules conditioned to the AfterComplete triggering moment and the code associated with the AfterTrn event.

For example, if there is a Country transaction, with City as the second level, and 5 countries are inserted through its form, the Commit will be executed 5 times, after saving the information of each country and its cities, but before the execution of the rules conditioned to the AfterComplete moment.

Commit on Exit Property

What are the possible reasons for not executing a Commit in a Transaction or Procedure?

Logical Unit of Work (LUW)

...
Database Operation
Database Operation
LUW Ends → **Commit**

LUW Starts
Database Operation
Database Operation
Database Operation
Database Operation
LUW Ends → **Commit**

→ What if the system fails here? **Rollback**

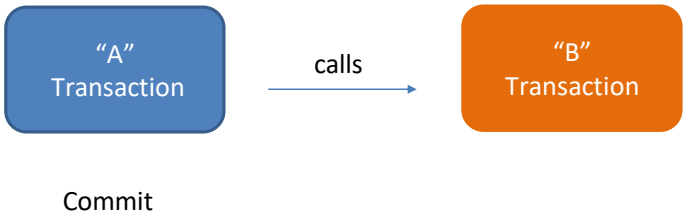
LUW

What may be the reasons for not executing a Commit in a transaction or procedure?

To customize a Logical Unit of Work (LUW)

This means that it is necessary to expand a LUW, so that, for example, several procedures, or a transaction with one or more procedures, form a logical unit of work, and it is necessary for a Commit to cover all of them.

Restrictions



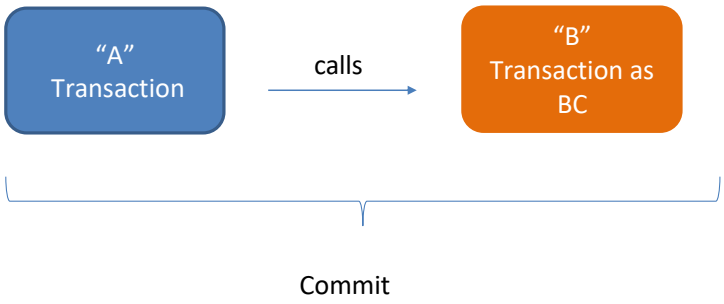
Commit

Now let's look at some important restrictions in relation to this issue

In web environments, each transaction can only commit its own set of operations performed on the database, and the procedures invoked by it, but not the operations performed by another transaction.

That is, if a transaction calls another transaction, the Commit executed by one transaction does not apply to the records inserted, modified or deleted by the other. Therefore, two different transactions cannot be included in the same LUW.

Restrictions



The Commit on Exit property will be ignored in transactions used as Business Components.

If you need to execute operations through two different transactions, and you want to form a single LUW between them, the solution is to execute them using the Business Component concept, including, then, the Commit command after executing the operations associated with both transactions.

It should be noted that the Commit on exit property will be ignored in transactions used as Business Components.

This means that, although the transaction has the Commit on exit property set to Yes, if the transaction is used as a Business Component, the commit will not be executed automatically, and it will be necessary to declare the commit command explicitly.

The reason for this behavior is to allow specifying logical units of work between multiple transactions, including the commit command where necessary.

Commit on Exit property - Examples

1

Commit on Exit = Yes


```
For each Country
     Where CountryId = 2
     CountryName = "New country name"
endfor
```

The Commit on Exit property will be considered and GeneXus will add the Commit automatically.

We will look at four very specific examples and analyze the behavior:

Let's look at the first example:

Suppose there is a Country transaction and the source of the procedure shown. The procedure has the Commit on Exit property set to Yes.

Since the For each performs an update on the database, GeneXus will add the commit automatically and the country with value CountryId = 2 will be updated with its new name.

Commit on Exit property - Examples

2

Business Component = Yes

Commit on Exit = Yes

```
&Country.CountryName = "Brazil"
&Country.Insert()
```

The Commit on Exit property will be ignored and the country will not be committed.

Let's see the following:

Consider the same Country transaction set as Business Component, and the source of the procedure shown:

The CountryId attribute is autonumbered, and the procedure has the Commit on Exit property set to Yes.

What will be the behavior?

Since the procedure only performs operations with the Business Component, even if the procedure has the Commit on Exit property set to Yes, it will be ignored and the country will not be committed. To achieve this, it is necessary to add the Commit command explicitly.

Commit on Exit property - Examples

3

Business Component = Yes

Commit on Exit = Yes

```
&Country.CountryName = "Brazil"
&Country.Insert()

For each Country
     Where CountryId = 2
     CountryName = "New country name"
endfor
```

The Commit on Exit property will be considered and both the BC and For each operations will be committed.

Let's move on to the next example:

Again, let's suppose that the same Country transaction has been set as Business Component, and the source of the procedure is as shown:

The CountryId attribute is autonumbered, and the procedure has the Commit on Exit property set to Yes.

In this example, since there are operations with a Business Component in the source of a procedure, and there is also a For each that performs updates on the database, then the Yes value of the Commit on Exit property is considered and both the operations with the Business Component and the operations of the For each are committed.

Therefore, the country with name Brazil will be inserted and also the country with value CountryId = 2 will change its name.

Commit on Exit property - Examples

4

Business Component = Yes

Commit on Exit = Yes

```
For each Country
     Where CountryId < 3
     &Country.Load(CountryId)
     &Country.CountryName = "New country name"
     &Country.Update()
endfor
```

The Commit on Exit property will be ignored.

Let's see the last example:

Again, we consider the same Country transaction set as Business Component, and the procedure source shown:

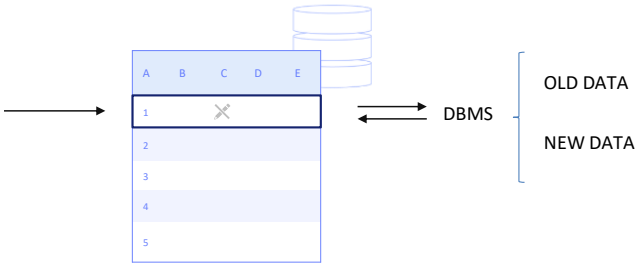
The CountryId attribute is autonumbered, and the procedure has the Commit on Exit property set to Yes.

In this case, the value Yes of the Commit on Exit property will also be ignored, because an update is attempted through the Business component and not directly through the For each. If this Business Component is not present, the For each itself does not perform any action.
Therefore, it will be necessary to add the Commit command explicitly.

So, after seeing these examples, as a general rule we can say that an object with the Commit on Exit property set to Yes will Commit on completion, as long as this object performs an update on the database not only through a Business Component.

Isolation levels

Read Only



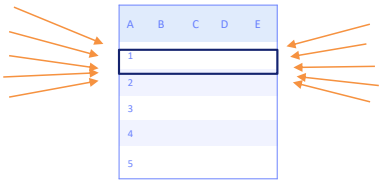
OK. Let's see something else:

Although we will not focus on studying concurrency control here, it should be mentioned that when several users perform operations on the database, if the information to be read is locked by another writing program, the values displayed will depend on the DBMS. The DBMS will then decide whether to display the old or the new value of the data being queried.

Specifying the **isolation level** affects read and concurrency control.

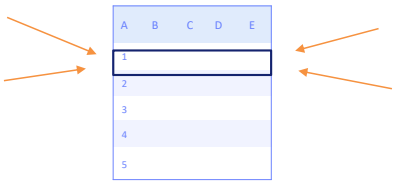
Isolation levels

Low isolation level



Increases simultaneous access effects

High isolation level



Reduces simultaneous access effects

In database systems, isolation determines how the integrity of database transactions is visible to other users and systems.

- A lower isolation level increases the ability of many users to access the same data at the same time, but it also increases the number of concurrency effects, such as dirty reads or lost updates that users may encounter. The possibility of deadlock is also increased, which requires careful analysis and programming techniques to avoid.

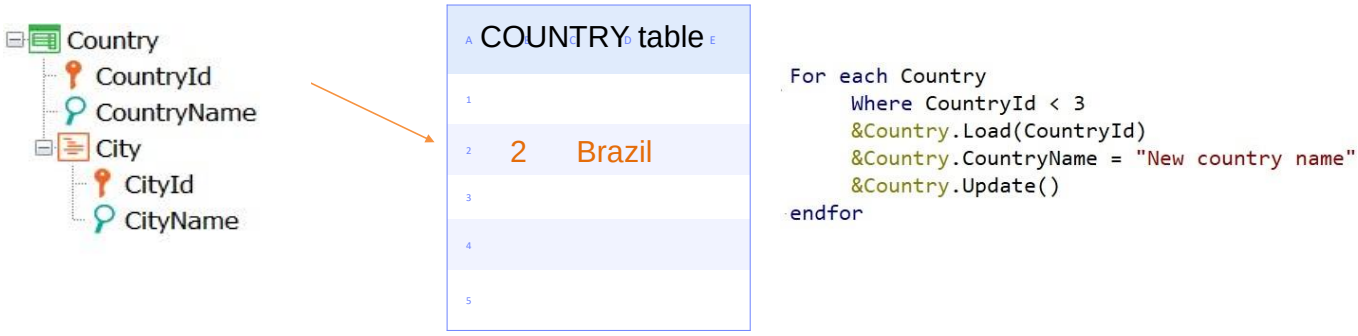
- Conversely, a higher isolation level reduces the types of concurrency effects that users may encounter, but requires more system resources and increases the chances that one database transaction will block another. The DBMS usually acquires locks on data that can result in a loss of concurrency, or implements multiversion concurrency control. This requires adding logic for the application to work correctly.

Isolation levels

Isolation levels are as follows:

- **Serializable**: This is the highest isolation level. It requires read and write locks (acquired on selected data) to be released at the end of the transaction. When using non-lock based concurrency control, if the system detects a write collision among several concurrent transactions, only one of them is allowed to commit.
- **Repeatable read:** In this isolation level, a lock-based concurrency control DBMS implementation keeps read and write locks until the end of the transaction. However, range locks are not managed, so phantom reads can occur.
- **Read committed:** This isolation level ensures that any data read is committed at the moment it is read. It simply prevents the reader from seeing any intermediate, uncommitted, 'dirty' reads.
- **Read uncommitted:** This is the lowest isolation level. Dirty reads are allowed, so one database transaction may see changes made by other transactions that have not been committed yet.

Isolation levels - Example



If Isolation level is Read Committed – CountryName = “Brazil”

If Isolation level is Read Uncommitted – CountryName = “New country name”

DIRTY READ

Let's look at an example to graph these concepts:

Consider the COUNTRY table.

A user accesses record 2 and obtains Brazil as the value of the CountryName attribute.

Then this user executes code like the one displayed.

What should we take into account? Because it is a Business Component and the commit has not been explicitly written in the database, the name “Brazil” is physically maintained until a Commit is actually executed.

Before a commit is executed, another user enters to read record 2 and then will get different values depending on the isolation level.

- If the isolation level is Read Committed, the user gets the value “Brazil” for the CountryName attribute.
- If the isolation level is Read Uncommitted, the user gets “New Country Name” as the CountryName attribute value.

So:

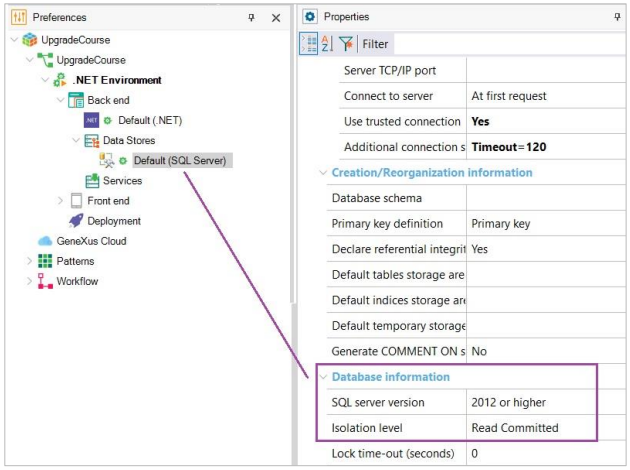
Read Uncommitted is a low isolation level, while Read Committed is a high level.

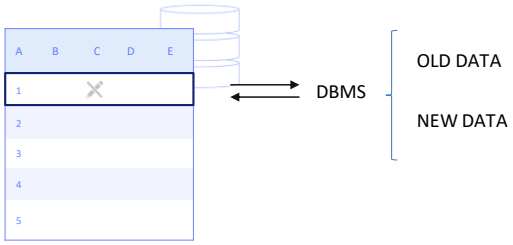
The concept of “Dirty read” is associated with the Read Uncommitted level, which refers to the possibility of reading data that has not been committed yet.

As for the Serializable and Repeatable Read levels, they are even more restrictive since they add

a lock; that is, the new user will not even be able to access record 2 because it will be locked.

GeneXus – Isolation Level property





In GeneXus, the Isolation Level property, at the Data Store level, allows specifying the isolation level of the changes made by the programs.

The available values are as follows:

- **Read Committed:** Programs do not see the changes made by other users until a Commit is executed. This is the default value taken by this property.
- **Read Uncommitted** In this case, the programs see the changes made by other users even though the Commit has not been executed yet.

As we have seen, specifying the isolation level affects data read and concurrency control.

training.genexus.com
wiki.genexus.com
training.genexus.com/certifications