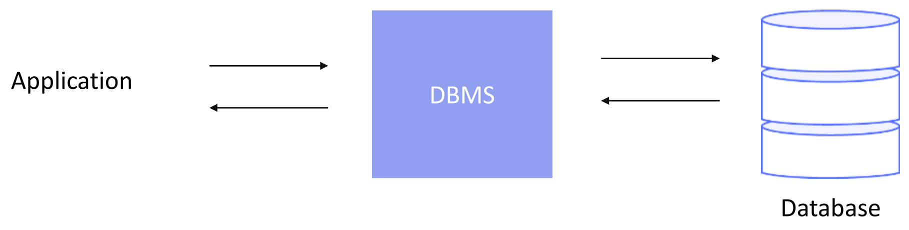
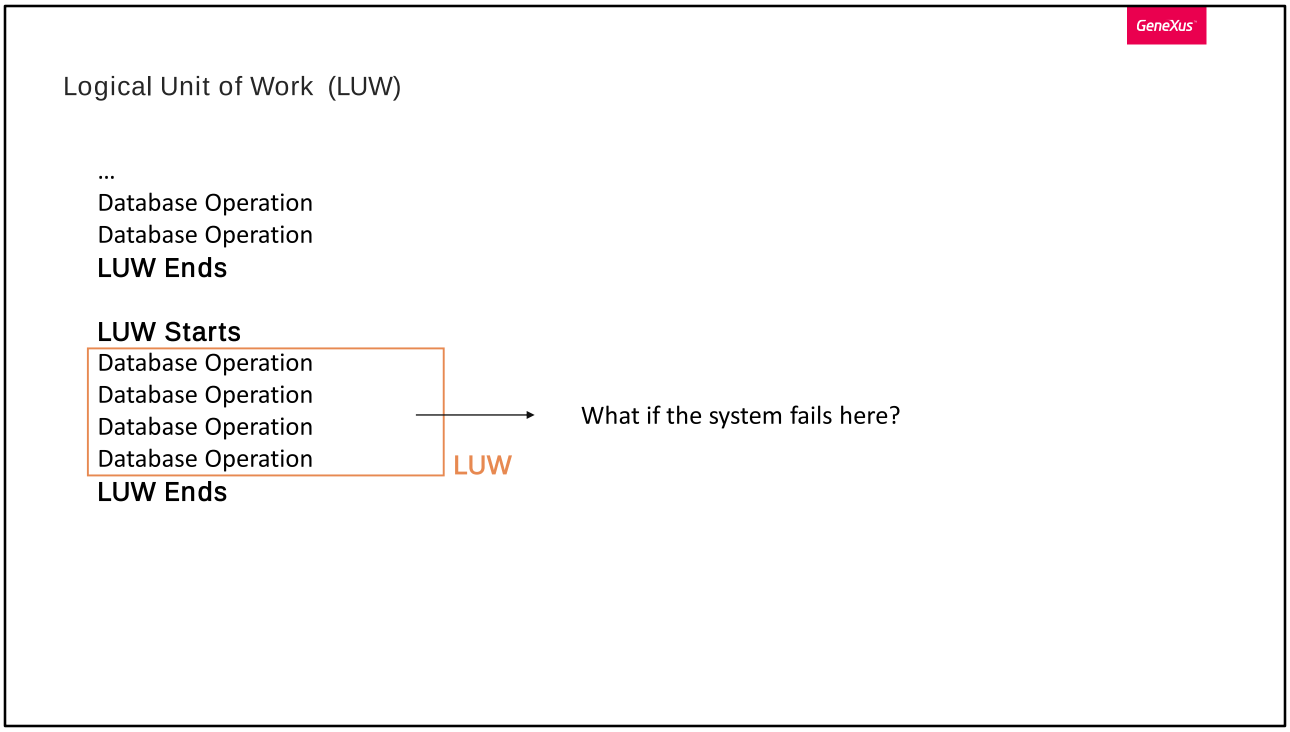
# Transactional Integrity

## Database Management System



Many database management systems (DBMSs) have failure recovery systems that leave the database in a consistent state when unforeseen events such as power outages or system failures occur.



## Logical Unit of Work (LUW)

…
Database Operation
Database Operation
**LUW Ends**

**LUW Starts**
Database Operation
Database Operation
Database Operation
Database Operation
**LUW Ends**

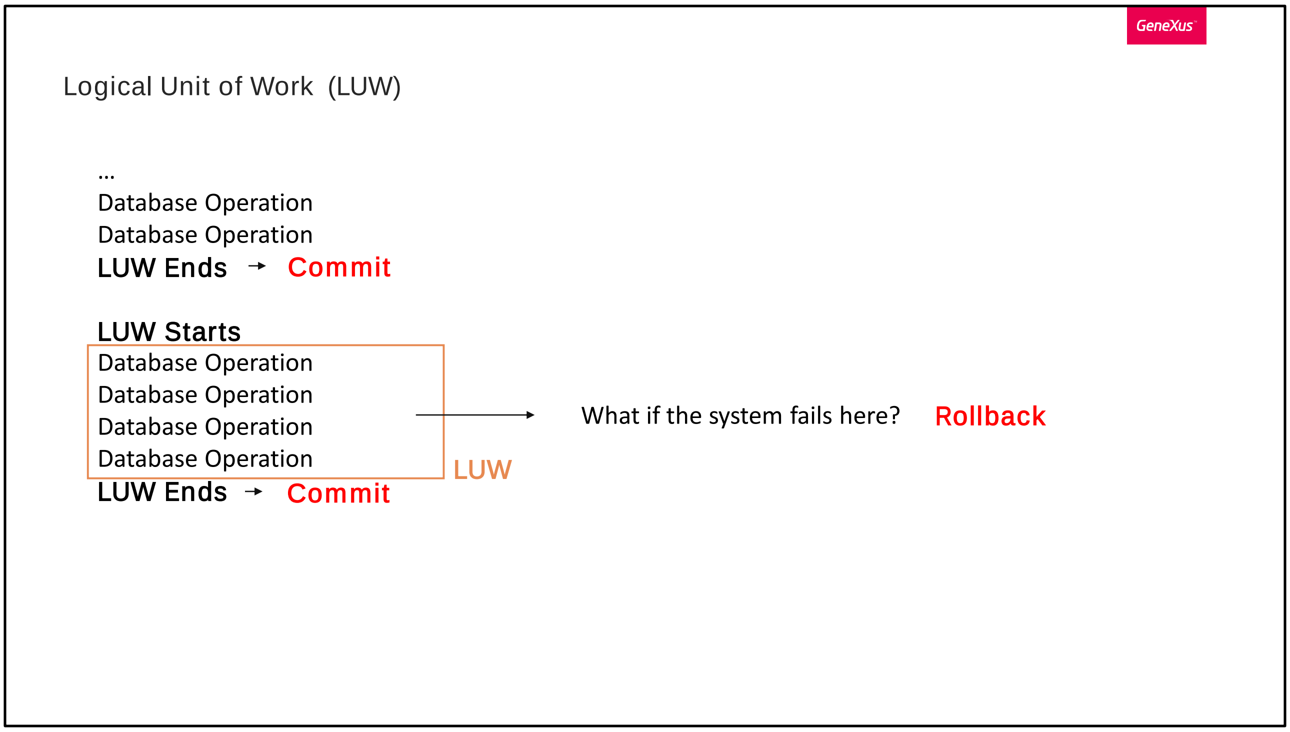
LUW
What if the system fails here?
GeneXus

Database management systems (DBMSs) that provide transactional integrity allow establishing logical units of work (LUW).
Logical Units of Work correspond to nothing less than the concept of database “transactions.”

Basically, a LUW is a set of database operations, all or none of which must be executed. That is to say, it is not possible that within a LUW some operations are executed and others are not. This ensures the integrity of the database.

In the example, you can see a LUW consisting of four operations on the database, which could be insertions, updates or deletions. Let's suppose that the first two were performed successfully, but before executing the third one, the system fails. Since the LUW was not completed, the two operations that were completed will have to be undone. Otherwise, as logical units of work define the consistent states of the database at a logical level, the database would be inconsistent.

In this case, the DBMS performs a so-called rollback to recover, keeping the last consistent state of the database.



## Logical Unit of Work (LUW)

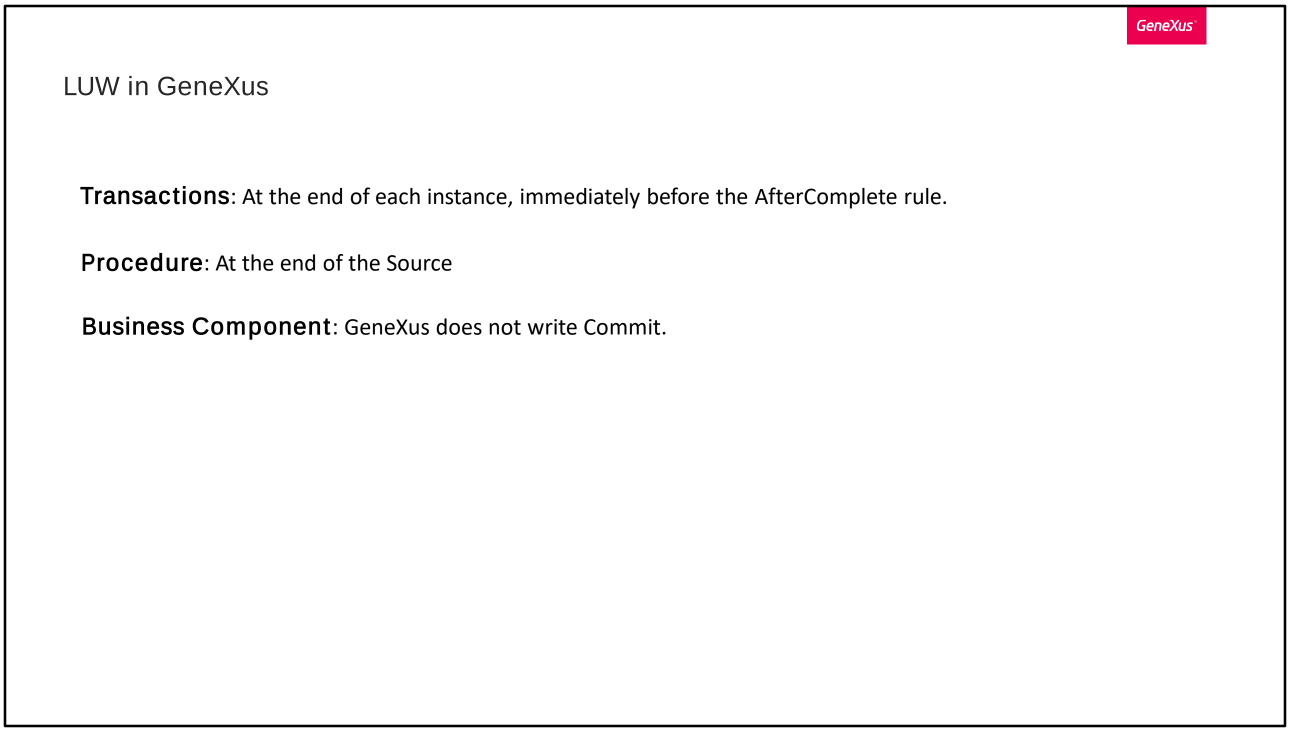
...
Database Operation
Database Operation
**LUW Ends** → **Commit**

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

The Commit command determines the end of a LUW.
Thus, a LUW is defined by the operations performed between two commits.

If the system fails where it is indicated, the two operations performed after the last Commit (which are the operations pending Commit) are undone by the automatic Rollback performed by the DBMS when it recovers from the failure.

That is to say, since not all the operations that make up the LUW have been performed, those that have been partially performed are undone or reversed.



## LUW in GeneXus

**Transactions**: At the end of each instance, immediately before the AfterComplete rule.

**Procedure**: At the end of the Source

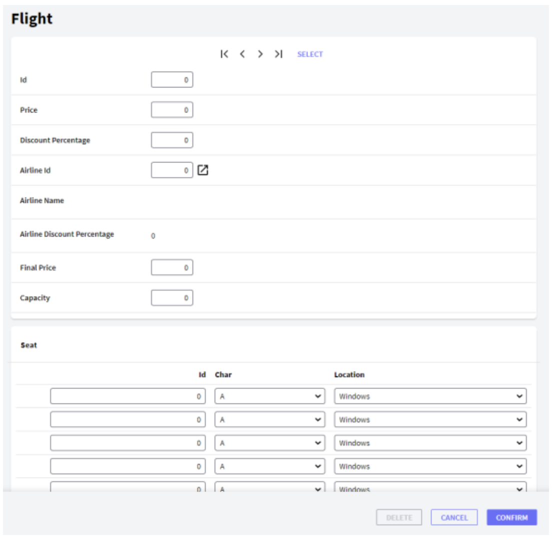
**Business Component**: GeneXus does not write Commit.

Transactions and procedures are the GeneXus objects created to update the information in the database. For this reason, GeneXus writes the Commit command when it generates the programs in the language that has been defined. Where?

In the transaction object: at the end of each instance, immediately before the rules with AfterComplete trigger event (i.e. after the header and lines have been handled).
In the procedure object: at the end of the Source.
Business Components, which are created from transactions, do not include Commit because they can be used in any object, and it will be up to the developer to determine where to commit.

We will see this next.

## Transaction and Automatic Commit



| | |
|---|---|
| Id | 0 |
| Price | 0 |
| Discount Percentage | 0 |
| Airline Id | 0 |
| Airline Name | |
| Airline Discount Percentage | 0 |
| Final Price | 0 |
| Capacity | 0 |

Seat

| Id | Char | Location |
|---|---|---|
| 0 | A | Windows |
| 0 | A | Windows |
| 0 | A | Windows |
| 0 | A | Windows |
| 0 | A | Windows |

**Header**

BeforeValidate
**VALIDATION**
AfterValidate / BeforeInsert – BeforeUpdate - BeforeDelete
**RECORDING**
AfterInsert - AfterUpdate - AfterDelete

For each line

**VALIDATION**
AfterValidate / BeforeInsert – BeforeUpdate - BeforeDelete
**RECORDING**
AfterInsert - AfterUpdate - AfterDelete
**END ITERATION LEVEL 2**
AfterLevel Level attLevel2 / BeforeComplete
**COMMIT**
AfterComplete

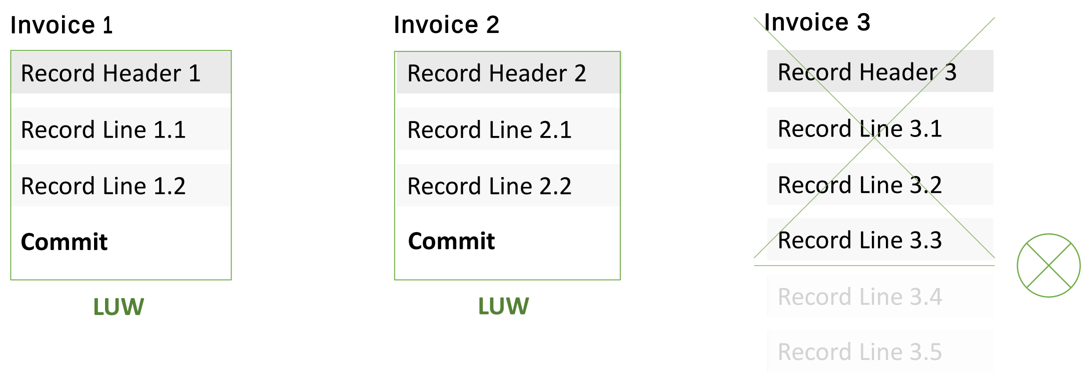
The user adjusts the header and lines and presses "Confirm." On the server, the rules and formulas are executed according to the evaluation tree for the first level and then the rules conditioned to the BeforeValidate event are triggered. After the header information is validated, the rules conditioned to the AfterValidate events and, depending on the mode, those conditioned to BeforeInsert, BeforeUpdate or BeforeDelete are triggered. After that, the header is saved and the rules that were conditioned to AfterInsert, AfterUpdate or AfterDelete are triggered, depending on the mode.

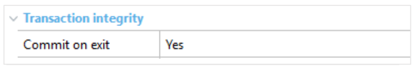
Then, for each line:

The rules are executed according to the evaluation tree.
Rules with a BeforeValidate trigger event are executed at the line level.
The line is validated (it is considered satisfactory).
Rules with AfterValidate trigger event or, depending on the mode, BeforeInsert, BeforeUpdate or BeforeDelete are executed.
The record corresponding to the line is inserted/modified/deleted in the database.
Rules with AfterInsert, AfterUpdate or AfterDelete trigger event are executed, depending on the mode of the line.

After completing the last line, the rules that have AfterLevel of a second-level attribute as a triggering event are executed.
If there is another parallel level, the same is repeated for that other level.
When the last level is completed, the rules conditioned to the BeforeComplete event are triggered. After this, GeneXus places the Commit command automatically. Therefore, the Commit is executed. That is to say, the header and line information is committed. Next, the rules that were conditioned to the AfterComplete event are triggered.

## Transaction and Automatic Commit




Transaction integrity

| | |
|---|---|
| Commit on exit | Yes |

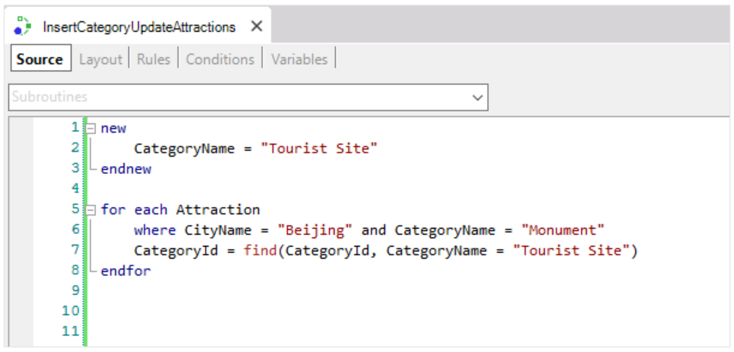
Suppose the user wants to enter 3 invoices into the system. He or she enters the first one, then the second one, and when he confirms the third one, let's suppose that when the program is processing the third line, after having saved it, the system crashes and the database has to be started again. What state will the database be in?

Since the DBMS will perform a rollback it will undo all operations that have not been committed. In this case, the records corresponding to the header and the three lines of invoice 3 will be deleted.

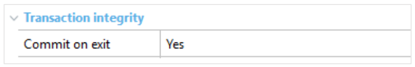
Note that if the automatic commit of the Invoice transaction had been disabled (property “Commit on exit” = “No”) none of the inserted records (neither those of invoice 1 nor those of invoice 2, and certainly not those of invoice 3) will remain in the database.  In this case, all the operations will make up a LUW, but if the default value for the Commit on Exit property is “Yes,” each header with its lines will make up a different LUW.

For transactions, setting this property to Yes is recommended.

## Procedure and Automatic Commit



```
new
    CategoryName = "Tourist Site"
endnew

for each Attraction
    where CityName = "Beijing" and CategoryName = "Monument"
    CategoryId = find(CategoryId, CategoryName = "Tourist Site")
endfor
```



| | |
|---|---|
| Commit on exit | Yes |

In every procedure that accesses the database, GeneXus will automatically add a Commit (unless otherwise indicated through the Commit on Exit property).

Since procedures are used for other things besides updating the database (for example, to list information or perform calculations or simply query the database) GeneXus will automatically insert the Commit command in the generated program if it understands that the procedure is trying to update the database. Otherwise, it doesn't add it, regardless of the value of the Commit on Exit property.

In this case, where the New command is being used to insert a category and the For Each is being used to update the CategoryId attribute of the table associated with the Attraction transaction, since the Commit on Exit property is set to Yes, GeneXus will automatically write the Commit in the generated program, at the end of the code.  This means that if after having inserted the new category in the CATEGORY table, and when modifying the third Beijing monument by changing the category to the new one, the system fails, none of the previous changes (neither the new category, nor the changes in the two previous monuments) will remain in the database. All the operations of this procedure will be part of the same LUW. Where does this LUW start?

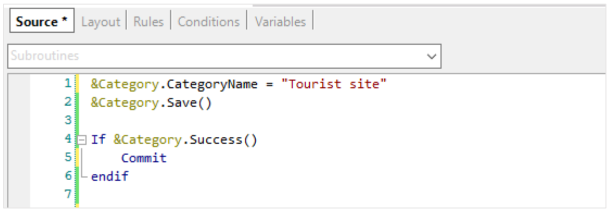
It will depend on when the last Commit was made. If a Commit was made after the last operation on the database before invoking this procedure, then the LUW starts with the New of this procedure. Otherwise, this entire code will be part of a LUW that started earlier. Where? At the point immediately following the previous Commit.

Where does the LUW end? If the Commit on Exit property is set to “Yes,” it finishes at the end of the procedure. Otherwise, it ends where the next Commit is found (in the one that called this procedure, we will have to see what follows its invocation).

GeneXus

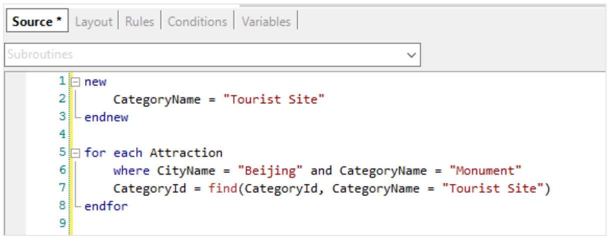
## Procedure and Explicit Commit

### Explicit Commit



```
&Category.CategoryName = "Tourist site"
&Category.Save()

If &Category.Success()
    Commit
endif

```

### Automatic Commit



```
new
    CategoryName = "Tourist Site"
endnew

for each Attraction
    where CityName = "Beijing" and CategoryName = "Monument"
    CategoryId = find(CategoryId, CategoryName = "Tourist Site")
endfor

```

New

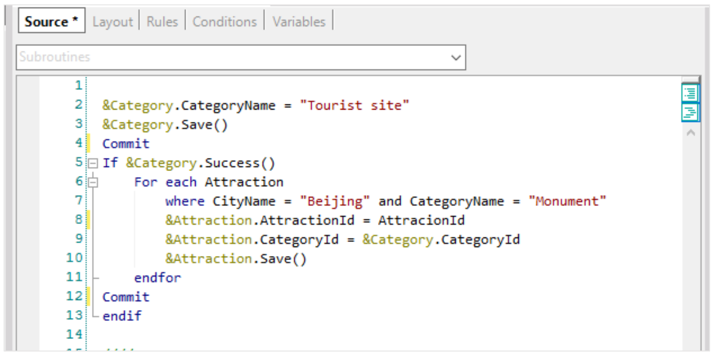
For each that updates

Delete

GeneXus realizes that it has to access the database within a procedure when using New, For Each to update or delete. In these cases, it adds the implicit Commit. Otherwise, it doesn't understand that there is a database access, and that's why the following is done in a procedure. Or even if instead of .Save() we do .Insert(), .Update() or .InsertOrUpdate(). It doesn't add the Commit. GeneXus doesn't realize that we want to do an Insert on the database, even using the Insert method, so we have to write the Commit command explicitly.

As we have just said, if within the procedure code there is a New, For Each that updates or Delete, in any of these cases it would not be necessary to explicitly write the Commit. If we only have this code in our procedure, we will have to add it explicitly.

## Customizing LUWs


Source * | Layout | Rules | Conditions | Variables

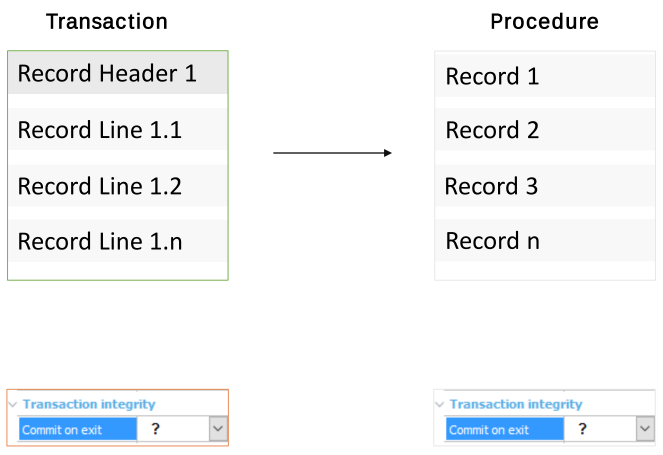
Subroutines

```
&Category.CategoryName = "Tourist site"
&Category.Save()
Commit
If &Category.Success()
    For each Attraction
        where CityName = "Beijing" and CategoryName = "Monument"
        &Attraction.AttractionId = AttracionId
        &Attraction.CategoryId = &Category.CategoryId
        &Attraction.Save()
    endfor
Commit
endif
```

In the example, we saw GeneXus automatically placed a Commit at the end. But if we wanted the recording of the category to be part of a LUW and the recordings of the attractions to be part of another one, so that if the system failed before finishing the modification of the attractions, the category was entered but the attractions were not, how would we do it?

We would use a Commit. We will have to write a Commit after having inserted the category, and another one after having inserted all the attractions. The latter can be avoided if we have the Commit on Exit property enabled. Anyway, it seems a good practice to write it explicitly, in case more operations are added later to the Source of the procedure, which should be in another LUW.

## Customizing LUWs



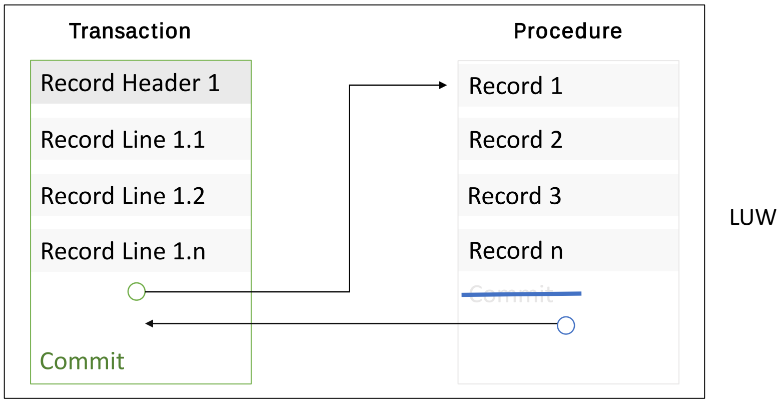
Transaction integrity
Commit on exit ?

Transaction integrity
Commit on exit ?

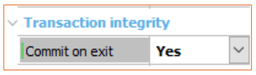
If we need to invoke from a transaction "A" a procedure "B" that performs operations on the database, so that the updates of the transaction header record and all the lines, as well as all the records of the procedure, make up a single LUW (and therefore, if the system crashes before completing everything all the changes are undone), what should we program?

There are several ways to do this.

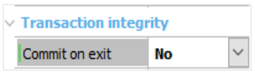
## Customizing LUWs



Procedure (parm1, parmN) on BeforeComplete;



Commit on exit: Yes



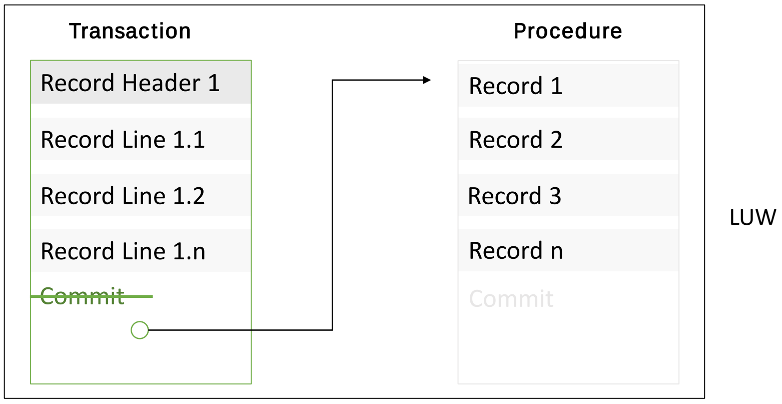
Commit on exit: No

One option is to follow these steps:

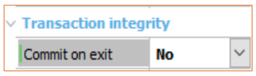
1. Invoke the procedure at some point PRIOR to the automatic Commit. For example, "on BeforeComplete."
2. Disable the automatic Commit of the procedure.

In this way, the procedure will be invoked after all the records have been saved (the one corresponding to the header and those corresponding to the lines), having already triggered all the rules conditioned to AfterLevel events. In other words, the procedure will be called right before the Commit. The procedure will perform all its database record updates, and since we have disabled its Commit, if the developer did not explicitly include this command within the code, the LUW will not be closed. Once the execution of the procedure code is finished, it returns to the caller, to the statement following the invocation. This is where the Commit will be found.

## Customizing LUWs

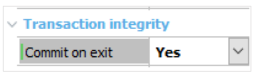


Procedure (parm1, parmN) on AfterComplete;



Transaction integrity

| | |
|---|---|
| Commit on exit | No |



Transaction integrity

| | |
|---|---|
| Commit on exit | Yes |

Another option is to follow these steps:

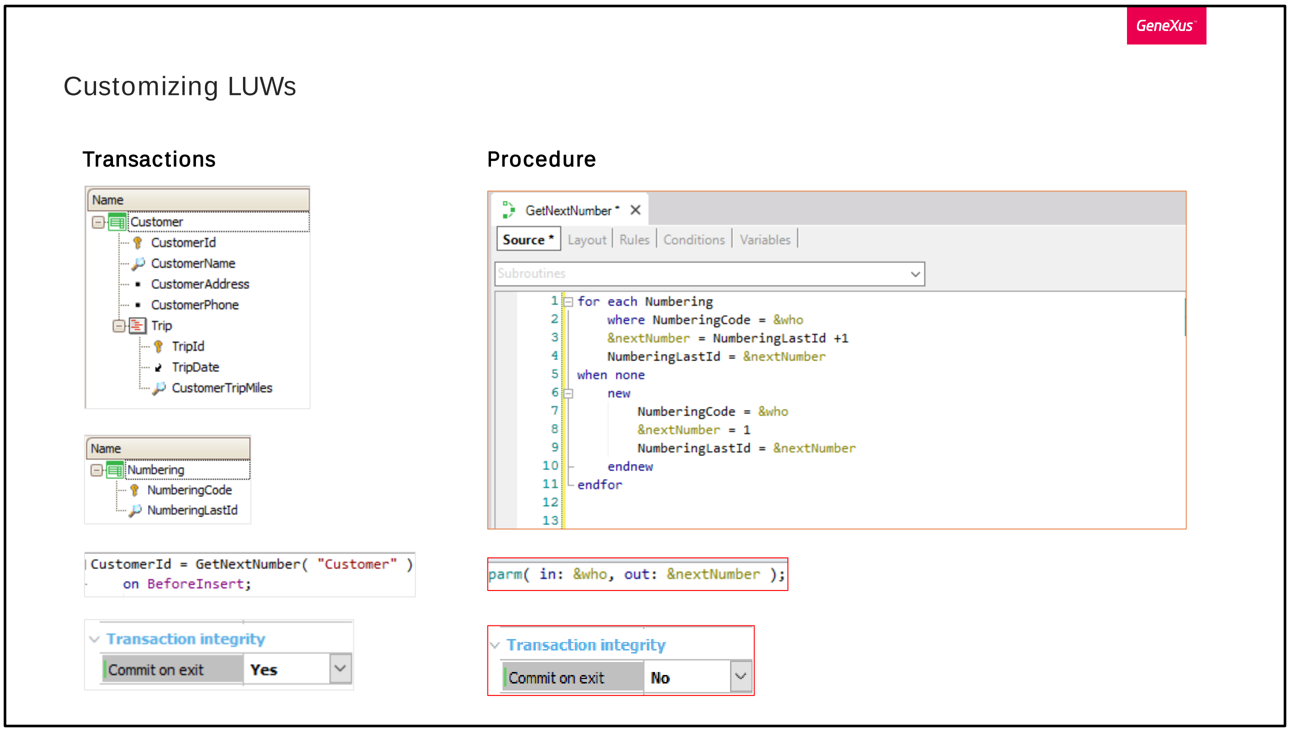
1. It doesn't matter when the procedure is invoked, as long as it is after all the records (header and lines) of the transaction have been saved, even after where the Commit would go: For example, on AfterComplete.

2. As long as the automatic Commit of the transaction is disabled and the automatic Commit of the procedure is kept.

By invoking the procedure at this point, we are sure that all header and line records will have been recorded. Then the procedure will perform its database record updates and commit, thus committing all records (its own and those of the transaction).

These are only two of the many possible options. The selected one will depend on the logic you are trying to implement (usually you will be concerned about the time the procedure is invoked).

## Customizing LUWs



**Transactions**

Customer
- CustomerId
- CustomerName
- CustomerAddress
- CustomerPhone
- Trip
  - TripId
  - TripDate
  - CustomerTripMiles

Numbering
- NumberingCode
- NumberingLastId

```
CustomerId = GetNextNumber( "Customer" )
    on BeforeInsert;
```

Transaction integrity

| | |
|---|---|
| Commit on exit | Yes |

**Procedure**

GetNextNumber

```
for each Numbering
    where NumberingCode = &who
    &nextNumber = NumberingLastId +1
    NumberingLastId = &nextNumber
when none
    new
        NumberingCode = &who
        &nextNumber = 1
        NumberingLastId = &nextNumber
    endnew
endfor
```

```
parm( in: &who, out: &nextNumber );
```

Transaction integrity

| | |
|---|---|
| Commit on exit | No |

There is a Customer transaction that records customers and their booked trips.

Suppose we don't want to use the database auto-numbering strategy, but rather we want to have our own internal table in the database that records the last number given to each entity to number its identifier.

To this end, we create a Numbering transaction whose NumberingCode identifier attribute records the name of the entity in question (for example, “Customer,” “Trip,” “Invoice,” “Category,” “Attraction,” etc.) and its NumberingLastId attribute, the last number given for that entity.

Next, we will program a procedure, GetNextNumber, which will obtain the next number to be assigned to the identifier of the calling entity.

We will examine the code in a moment.

For example, from the Customer transaction, when the user wants to enter a new customer, he will leave the CustomerId field empty on the screen and when he leaves the field, moving to the next field, CustomerName, the transaction will know that he is trying to enter a new record (it will be in “INS,” insert mode).

Since the CustomerId attribute is not auto-numbered, we will have to invoke the GetNextNumber procedure to obtain the number to assign to CustomerId.
We must send who we are as a parameter to the procedure, that is to say, in this case, “Customer” so that the procedure looks for the last number given to a “customer” in the Numbering table.

Let's see in detail what the created procedure does.
First of all, in the rules we declare two parameters, one input and one output parameter. The input parameter will be used to know what we want to number, such as a customer, a trip, an invoice, etc. This variable will be of character type.
And as output we declare the NextNumber variable, of numeric type, which will return the value we are interested in.

Going back to the source, the table of the Numbering transaction will be run through. With Where we indicate that we want to retrieve the record in which NumberingCode has the same value as the variable that we sent in a parameter, in this case Customers.
If it finds it, one will be added to the NumberingLastId attribute, and that value will be assigned to the nextNumber variable. Then, to the NumberingLastId attribute we assign the value of nextNumber.

If no record is found in which NumberingCode has the value of the variable we sent to it, we will create a record in the database using the New command. Here NumberingCode will have the value of the variable sent in a parameter, for example, Invoice. Next, we assign to the nextNumber variable the value one, and to the NumberingLastId attribute we assign the value of nextNumber, because as it is a new record in this table we will start with this value.

If we invoked this procedure using the assignment rule without a trigger event:
CustomerId = GetNextNumber( “Customer” ) if Insert; the procedure will be triggered:

1. Once as soon as the user leaves the CustomerId field empty on the screen and leaves the field.
2. A second time when the user confirms, and the rules are triggered again in order, on the server.

Let's imagine that the last customer ID is 5.

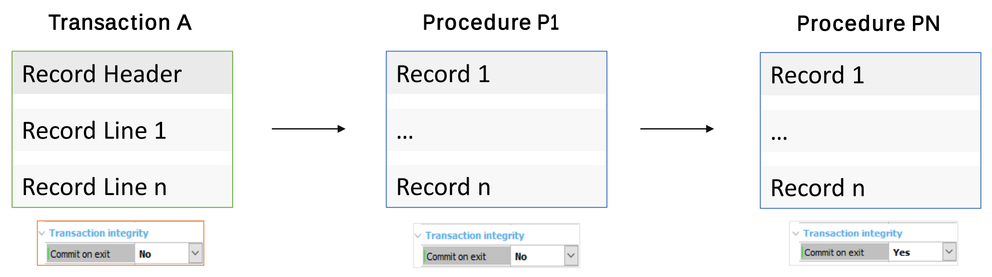
The procedure will be executed by updating the corresponding Numbering table record to 6 and immediately displaying the number 6 to the user on the screen. But what happens if the user changes his mind and never presses Confirm, but cancels? Number 6 will have been lost.

Even if the user does confirm, the procedure will be run again, and the number assigned to the customer will be 7, so the number 6 will also be lost.

How do we solve this situation? By conditioning the invocation of the procedure to a trigger event (in this way the assignment rule will not be triggered on the client, while the user is working on the screen). When is the right time? The last possible moment is BeforeInsert of the header. Why? Because after that moment, the value we assign to any of its attributes will have no effect, since the record will have already been saved.

The GetNextNumber procedure modifies a record in the Numbering table, or inserts one if it didn't exist. By default, it will place an automatic Commit at the end. In this way, if after returning to the transaction, suppose that the customer is inserted in its table, and when the third trip is being inserted, the system fails, the records associated with the customer will not be recorded when the system is restored, but the number will be lost, because it has already been committed. For all the operations performed through a transaction and procedure to remain within the same LUW, in this case it will be enough to not Commit on Exit in the procedure, and that the Commit of everything is done by the transaction.

## Customizing LUWs



Transaction A: Commit on exit: No

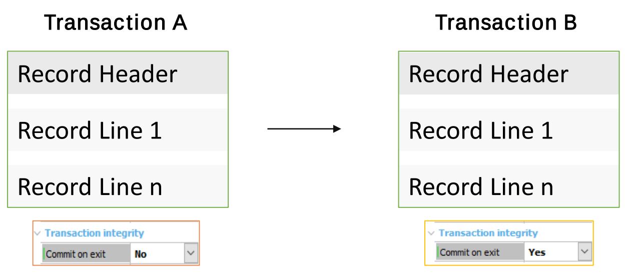
Procedure P1: Commit on exit: No

Procedure PN: Commit on exit: Yes

A single LUW cannot be made up between transactions.

If from a transaction a procedure is invoked that invokes another procedure, all with Commit disabled except the last one (or its Commit can also be disabled and have the transaction Commit when the control is returned) it will commit the records of the transaction and of all the procs.

## Customizing LUWs

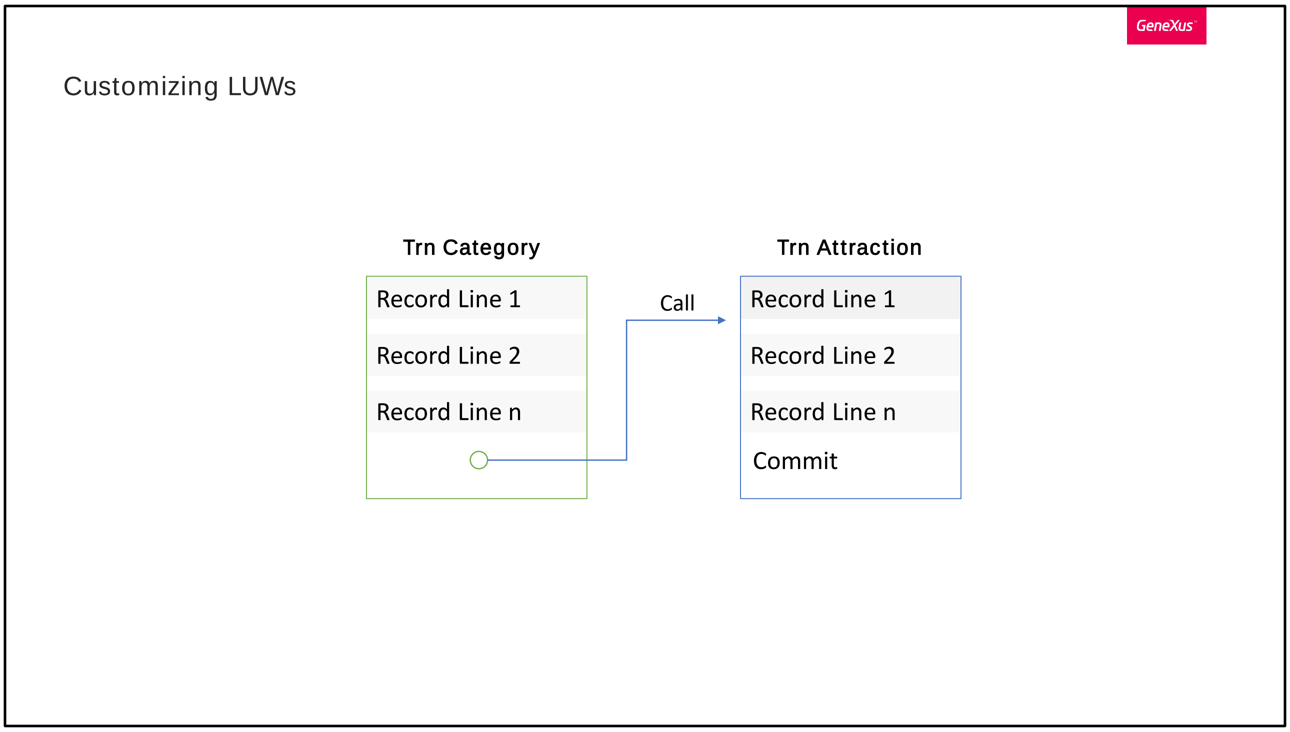


Commit on exit: No

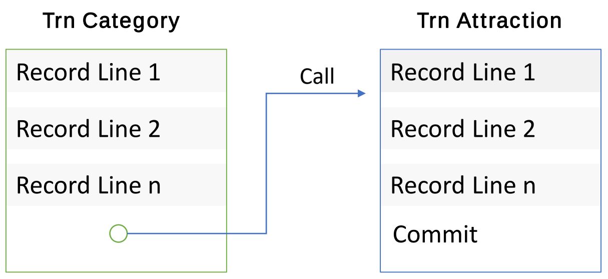
Commit on exit: Yes

However, if transaction (“A”) calls another transaction (“B”), the commit of the second one (“B”) will NOT commit the records handled by the first one (“A”). They are independent Commit operations. So if we disable the commit of “A” and call “B”, which does Commit, the records of “A” will not be committed by anyone!

So how do we get the header and lines of transaction “A” to be entered, transaction “B” to be called to enter somewhat related information, and all these operations to make up a single LUW?



## Customizing LUWs



From Work With Categories we want that by pressing Insert the user is offered the possibility to enter a new category and right afterwards an attraction of that category. Because we do not want to leave categories inserted without related attractions.

To do so, from the Category transaction it will be enough to invoke the transaction Attraction on AfterInsert (so that the CategoryId already has the correct auto-numbered value, given by the database), and to declare in Attraction that it will receive a parameter.

This way, when we enter a Category, it will take us to the Attraction transaction to enter an attraction. The category that we have just sent in a parameter will have been selected.

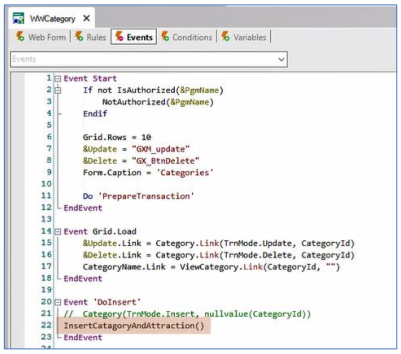
But...what happens if the system fails immediately after Attraction has made its commit? Will this Commit have also committed the Category record that has already been inserted? The answer is No, because of what we saw before.

We need the insertion of a category and then of an attraction to make up the same LUW, and the final Commit to include both records. How do we go about it?

We will look at two of the many options available.

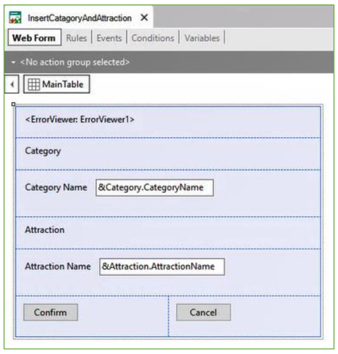
## Customizing LUWs

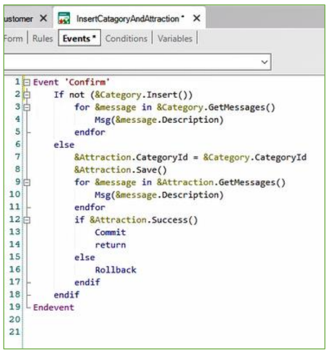
**WorkWithCategoy**



```
Event Start
    If not IsAuthorized(&PgmName)
        NotAuthorized(&PgmName)
    Endif

    Grid.Rows = 10
    &Update = "GXM_update"
    &Delete = "GX_BtnDelete"
    Form.Caption = 'Categories'

    Do 'PrepareTransaction'
EndEvent

Event Grid.Load
    &Update.Link = Category.Link(TrnMode.Update, CategoryId)
    &Delete.Link = Category.Link(TrnMode.Delete, CategoryId)
    CategoryName.Link = ViewCategory.Link(CategoryId, "")
EndEvent

Event 'DoInsert'
//  Category(TrnMode.Insert, nullvalue(CategoryId))
InsertCatagoryAndAttraction()
EndEvent
```

**InsertCategoryAndAttraction**


InsertCatagoryAndAttraction
Web Form | Rules | Events | Conditions | Variables
<No action group selected>
MainTable
<ErrorViewer: ErrorViewer1>
Category
Category Name
&Category.CategoryName
Attraction
Attraction Name
&Attraction.AttractionName
Confirm
Cancel


ustomer
InsertCatagoryAndAttraction *
Form | Rules | Events * | Conditions | Variables

```
Event 'Confirm'
    If not (&Category.Insert())
        for &message in &Category.GetMessages()
            Msg(&message.Description)
        endfor
    else
        &Attraction.CategoryId = &Category.CategoryId
        &Attraction.Save()
        for &message in &Attraction.GetMessages()
            Msg(&message.Description)
        endfor
        if &Attraction.Success()
            Commit
            return
        else
            Rollback
        endif
    endif
Endevent
```

One solution will be for the Insert button of the "Work With" to invoke a web panel, which is an object with an interface where developers can freely program behavior.

There we insert two variables &category and &attraction in its form. They will be Business Components of Category and Attraction, respectively. We only need the user to insert the name of the category he wants to create, and the name of the attraction of that category. Doing so and pressing Confirm will trigger the associated event, which we have programmed as shown in the event screen.

For this example, note that the CountryId and AttractionId attributes, of the Country and Attraction transactions respectively, have the auto-number property set to True.

First, we try to Insert the BC of Category. If it fails (for example if the user left the category name empty on the screen and the transaction has an error rule to prevent that) we show in the ErrorViewer control the messages generated by the BC.
In that example the error message is triggered twice, once on the client side when we leave the field empty. And another time when we confirm, on the server side. Both cases yield the same result. We call this type of

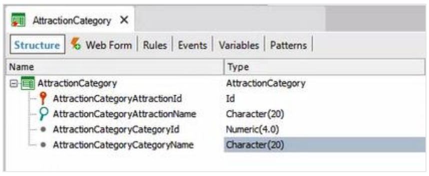
rules idempotent.

If we enter all the data correctly, then to the BC variable of Attraction we assign as category the one we’ve just inserted and try to Save. We show the messages generated and then, if the operation was successful, we Commit, after which both records will be committed.

If the operation failed, then note that we have the Rollback command to undo the insertion of the Category record that had been successfully inserted.

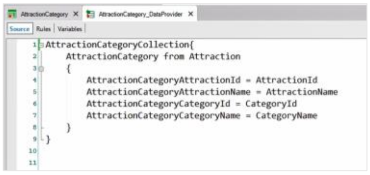
## Customizing LUWs

**Trn AttractionCategory**



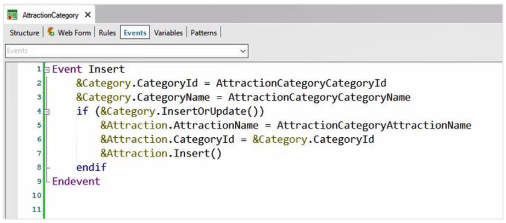
| Name | Type |
|---|---|
| AttractionCategory | AttractionCategory |
| AttractionCategoryAttractionId | Id |
| AttractionCategoryAttractionName | Character(20) |
| AttractionCategoryCategoryId | Numeric(4.0) |
| AttractionCategoryCategoryName | Character(20) |

Data Provider



```
AttractionCategoryCollection{
    AttractionCategory from Attraction
    {
        AttractionCategoryAttractionId = AttractionId
        AttractionCategoryAttractionName = AttractionName
        AttractionCategoryCategoryId = CategoryId
        AttractionCategoryCategoryName = CategoryName
    }
}
```

Insert Event



```
Event Insert
    &Category.CategoryId = AttractionCategoryCategoryId
    &Category.CategoryName = AttractionCategoryCategoryName
    if (&Category.InsertOrUpdate())
        &Attraction.AttractionName = AttractionCategoryAttractionName
        &Attraction.CategoryId = &Category.CategoryId
        &Attraction.Insert()
    endif
Endevent
```

A second alternative to solve the same requirement is to use a dynamic transaction instead of the web panel.

Remember that dynamic transactions are those that do not generate a physical table; the queried data is obtained at runtime.
They are defined by configuring the following transaction properties. Data Provider set to True, and Used To set to Retrieve Data.
By setting the Data Provider to True, a Data Provider object will be automatically created. And by setting Used To to Retrieve Data, GeneXus will understand that it will not have to create the table associated with the transaction, because the Data Provider will declare where to obtain the data from.

In the example we create the dynamic transaction AttractionCategory, whose information will be taken from the attraction table, knowing that each attraction has a category. It will be like an attraction view.

When the user wants to insert an attraction in this transaction, he will be allowed to enter the attraction data as well as the category data. In the Insert event we use &category business component of Category, and &attraction business component of Attraction and copy to its members the values that the user specified in the attributes of the dynamic

transaction, through its form.
If the category doesn't exist, it is created before inserting the attraction. Otherwise, its name can be updated. Next, the attraction is inserted with the category.
If we leave the Commit on Exit property of the transaction with its default value, Yes, after the Insert event is executed, since it is a single level transaction, the Commit will be made, which will have an effect on the two records inserted through the business components.

training.genexus.com
wiki.genexus.com