Database update

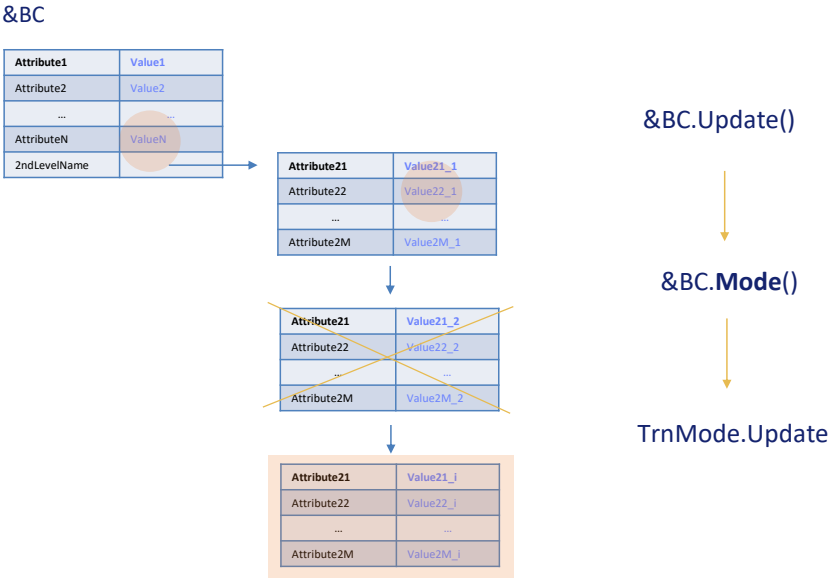
Update with Business component. Behind the scenes

GeneXus™

Transaction

Attribute1*
Attribute2
…
AttributeN

2ndLevelName
{
Attribute21*
Attribute22
…
Attribute2M
}

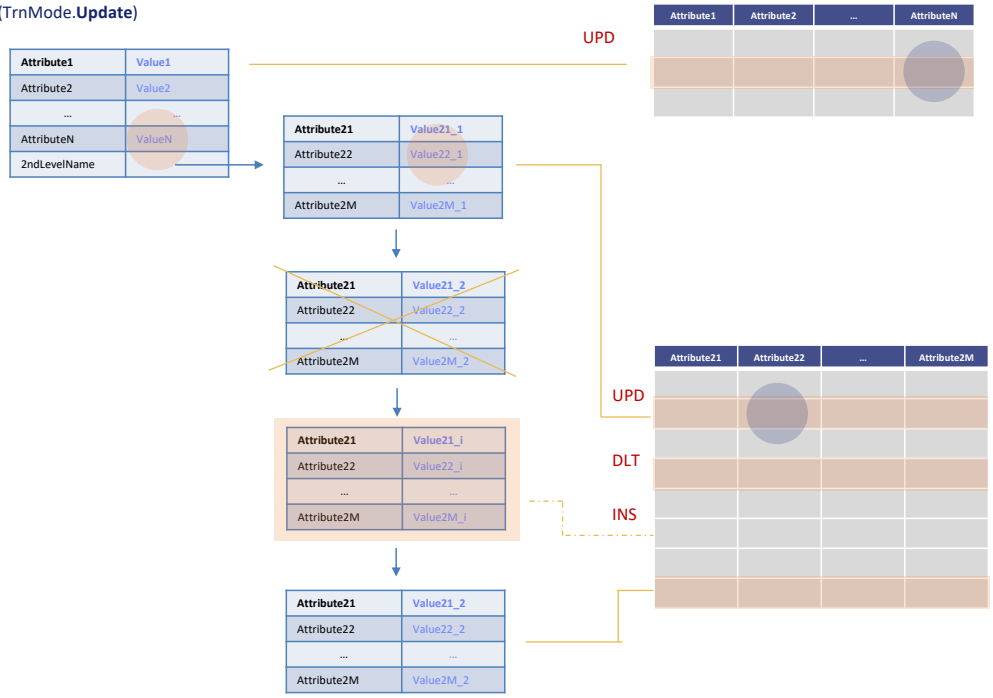


Let's suppose that we have a two-level transaction with the Business Component property turned on, and a variable of that data type.

We had previously seen that what will be done internally when the Update method is executed depends on the variable's mode.

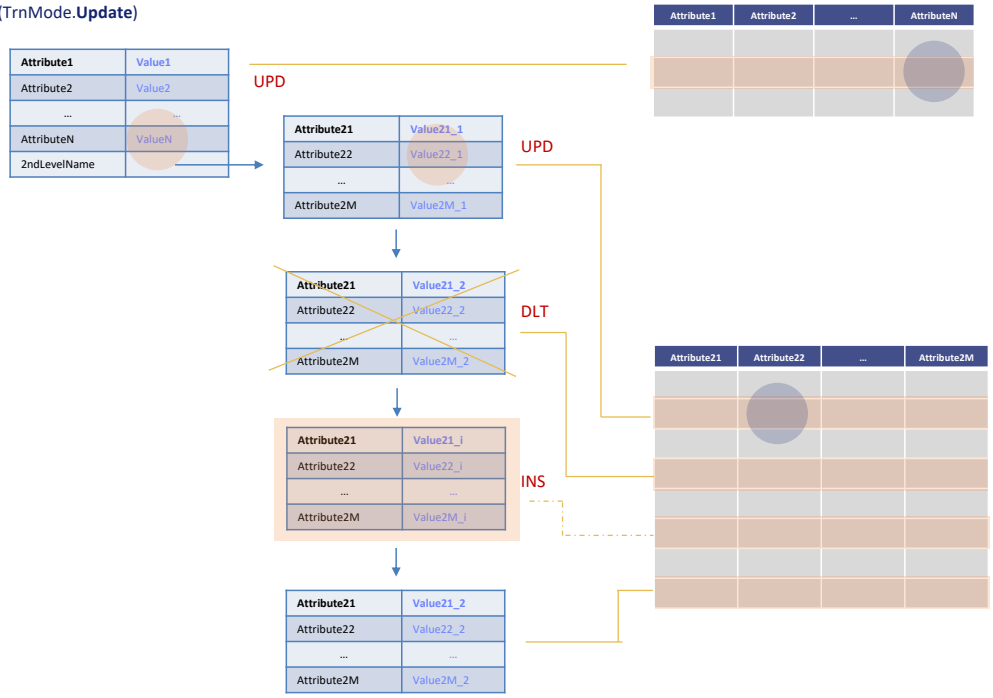
If the variable were in Update mode, its header would be updated regardless of whether any of its properties— elements, that is, attributes—had been modified; if something had been done on the line collection, this would be reflected in the database records. We will go into a bit more detail here.

&BC (TrnMode.**Update**)

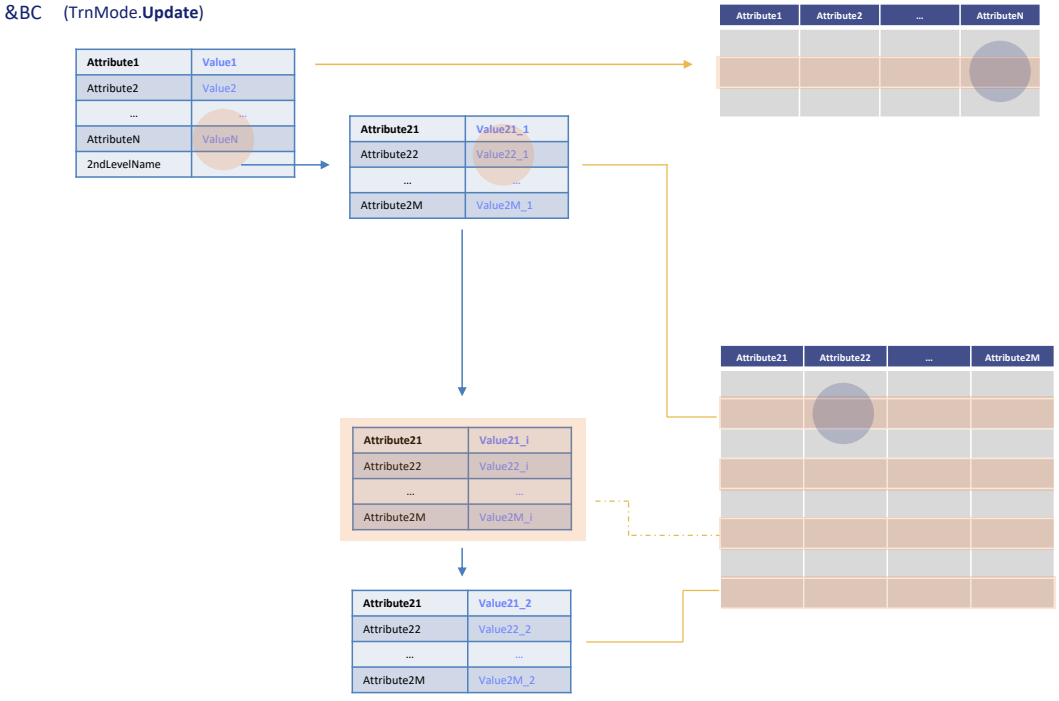


How does it really work in the background? Does it go to the database looking for differences between the existing records and the data contained in the BC at the time of the Update?

&BC (TrnMode.**Update**)



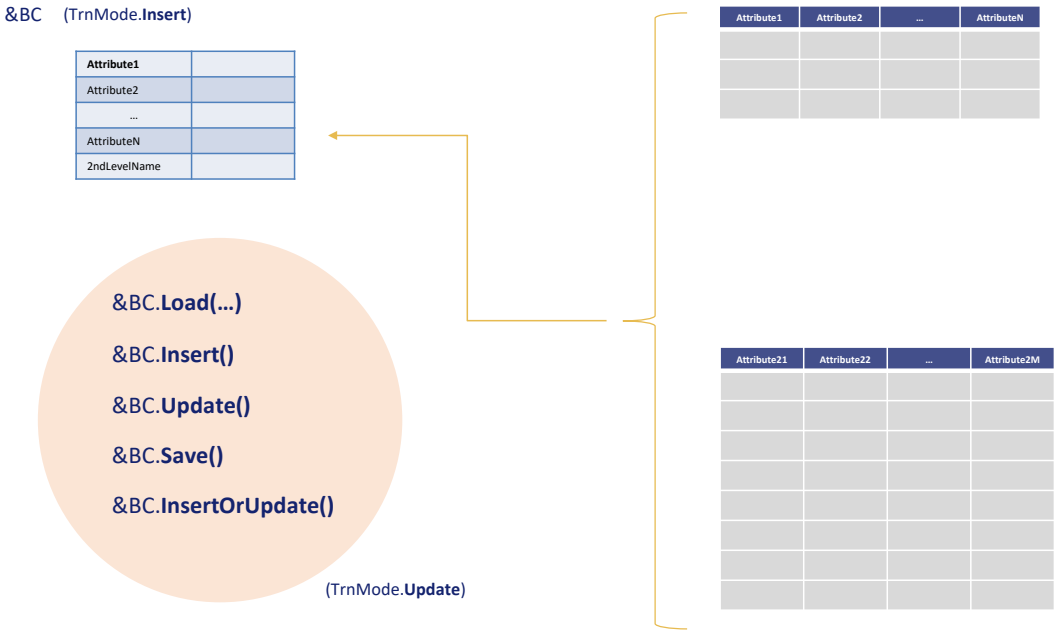
Or is everything that was performed on the BC kept in memory so that these operations are now passed on to the database, without checking exactly the consistency between memory and database?



If the answer were the first one, the record in the header table would have to be accessed and all the fields compared, to check if any of them changed and thus update the particular field or fields that have changed in the record. Or directly overwrite the record so as to skip the comparison.

And then access all the records corresponding to the rows, and if any of them are not in the BC collection, delete them from the database. And then run through the collection and for each item check if the record exists, and if so check if anything has changed so as to modify it or overwrite it directly. If nothing has changed in this one, we could leave it as it is. And we would be left with this item that doesn't have a record in the database, so it is inserted.

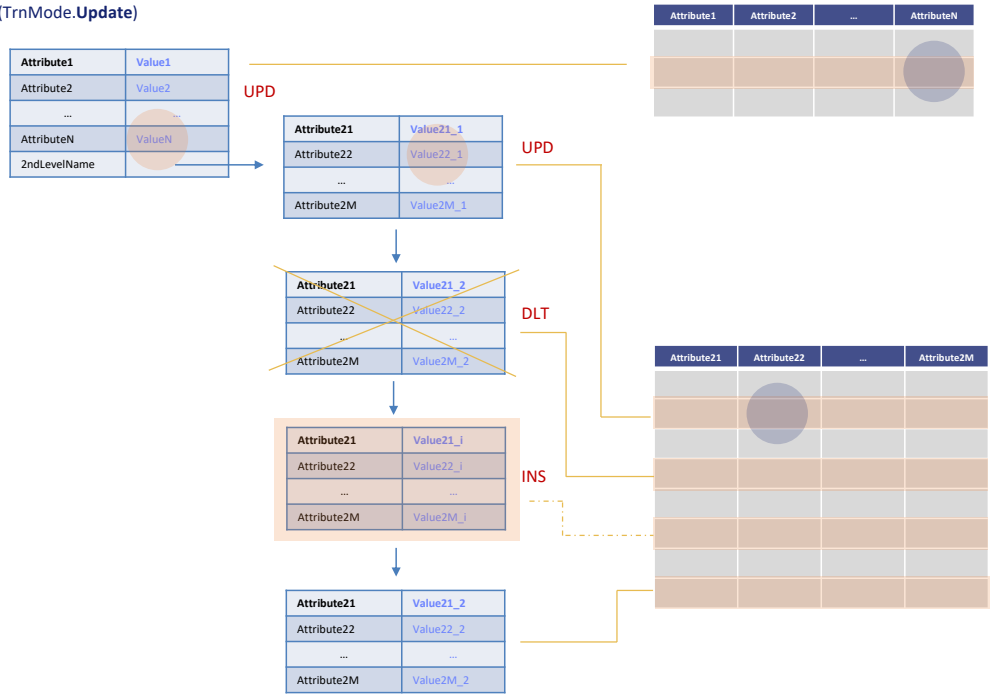
Doing this whole comparison would be very burdensome. It will not be done.



What GeneXus actually does is the latter. At the beginning of the execution of the object where every BC variable is found, it starts in Insert mode because when it is declared, empty memory space is already reserved for it.

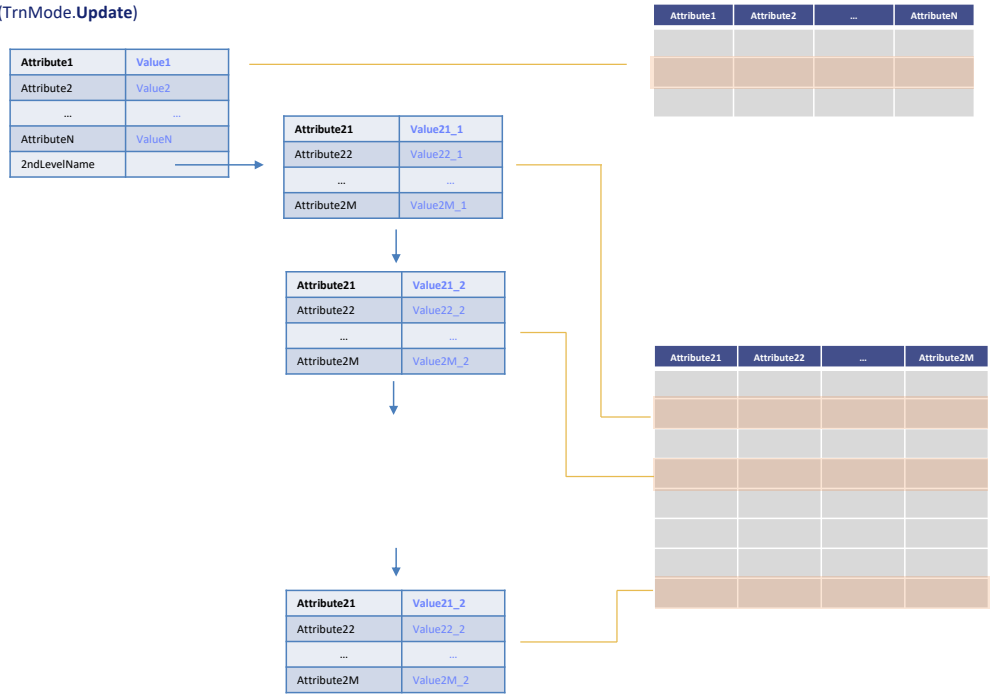
For a BC variable to be in Update mode, it must have been loaded from the database, either with Load, or with a previous (successful) Insert, Update, Save, or InsertOrUpdate operation. Only then it is in Update mode.

&BC (TrnMode.**Update**)

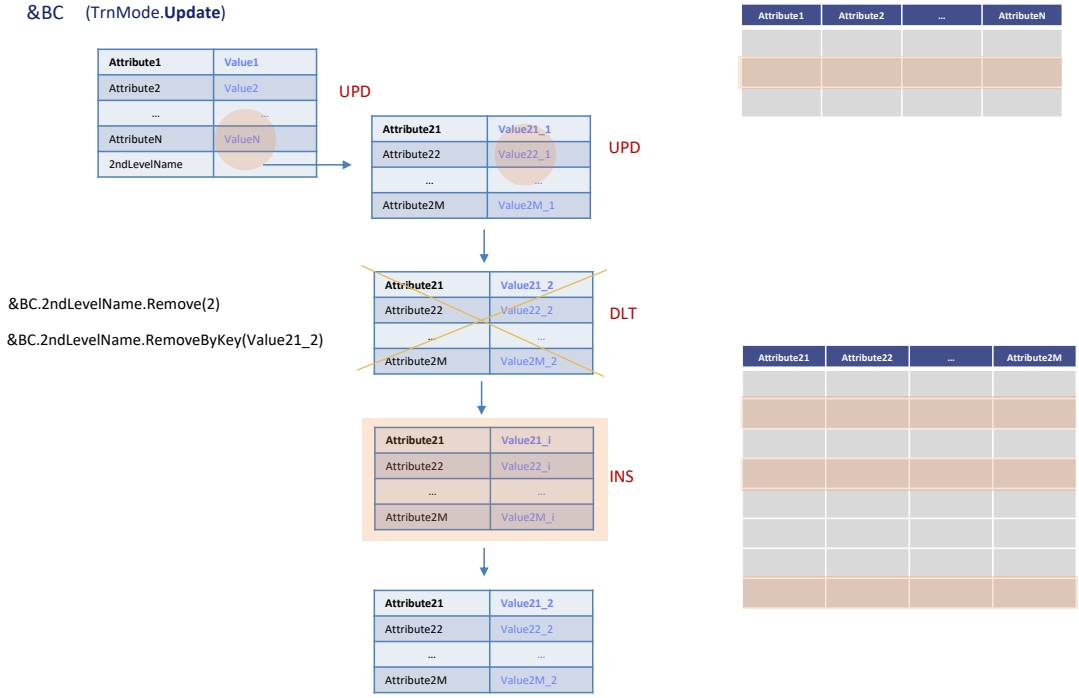


Therefore, since the last operation that left the variable loaded, when handling it by code, it will record in its hidden information what is being done with the header and the items: whether to update them, mark them for deletion, or insert them, so as to execute these operations later.

&BC (TrnMode.**Update**)



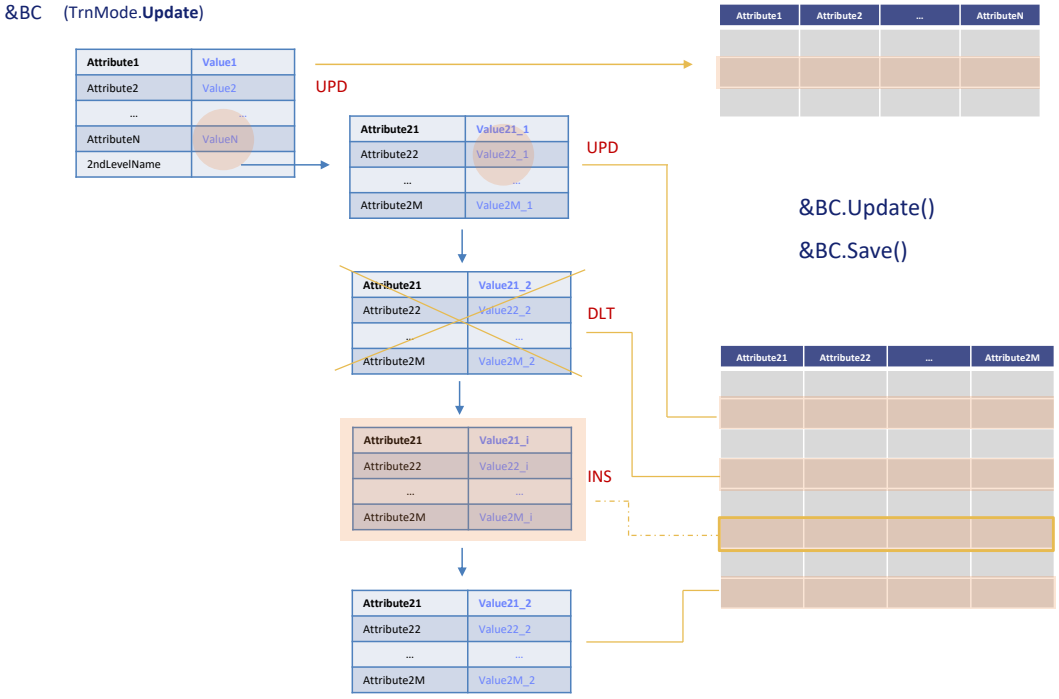
Let's slow down. Suppose that a Load of the &BC variable was made, leaving it in Update mode and with this information (which matches that of the database).



Next, we manipulate the variable by code and do, for example, the following:

- Change this header value,
- This one of this item,
- Delete this other item (either with the Remove method of the collection, asking to delete the second item, or the RemoveByKey, to which we must pass the identifier). When doing this the item will not be actually deleted from the collection, but **marked** to be deleted, so it will still be there (that is to say, it will be a logical deletion,
- Add this other one, and
- Leave the last one as it was.

Then the elements of the BC are marked in this way: header and first item to be updated; second item marked to be deleted; third item marked to be inserted; and the last one wouldn't need to have any mark because nothing was done with it. We'll return to this later.

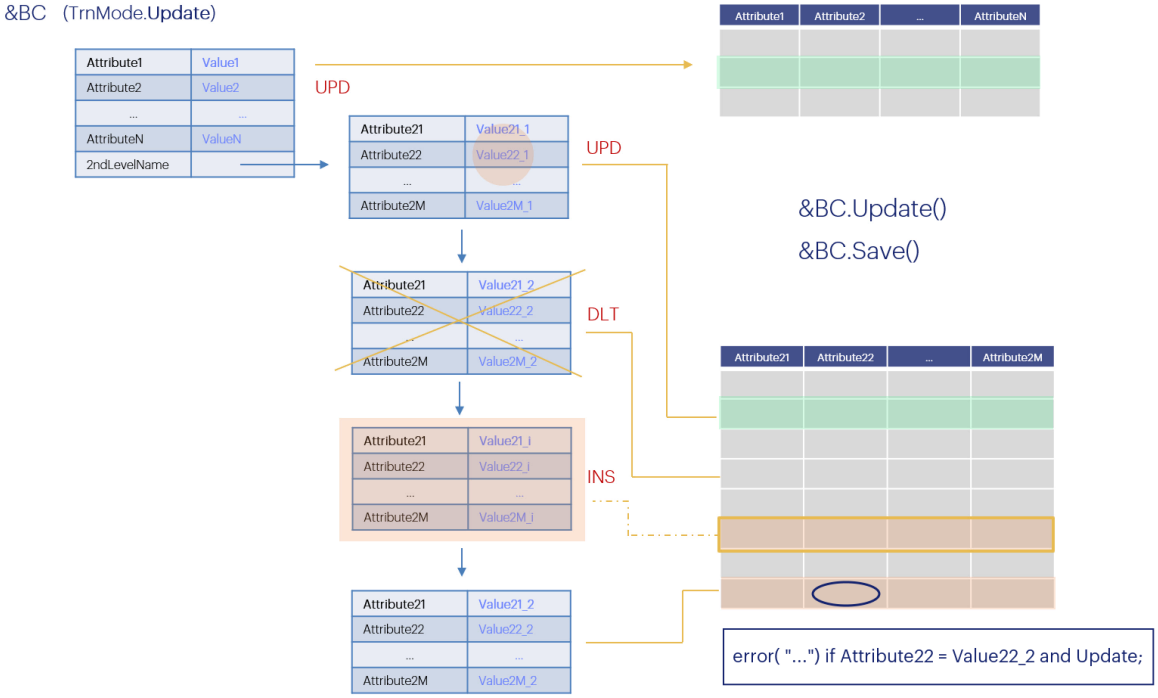


We give the order to Update (or Save, which in this case is the same because the variable is in Update mode).

It will try to overwrite the header, regardless if anything has changed in it (for which, clearly, it must first check that there is a record in the table with that primary key; otherwise, we already know that it will fail by PrimaryKeyNotFound). This is the same thing that the transaction does when you click on Confirm in Update mode. It will update the header in the database even if the user has not modified any of its fields.

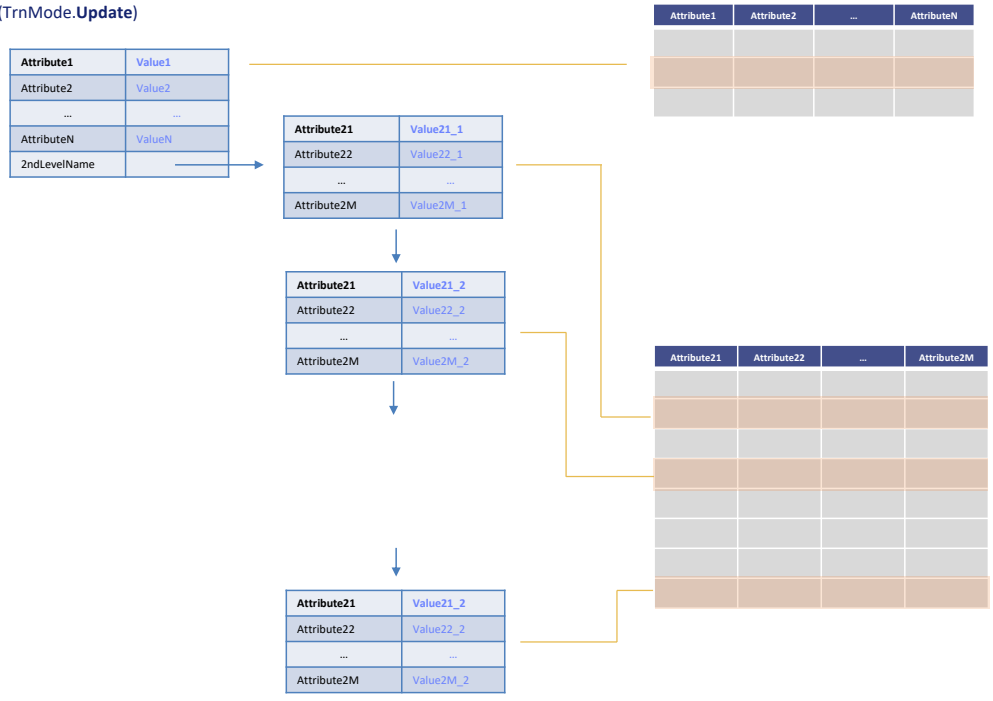
And then, for each item in the collection:

- If it is marked as update, the entire record is overwritten,
- If it is marked as deleted, the record is searched for and deleted from the table,
- If it is marked as new in the collection, it is inserted into the table,
- And if nothing was done with it... well, here there is a difference with the transaction: it is still overwritten. It is as if it remained in Update mode.

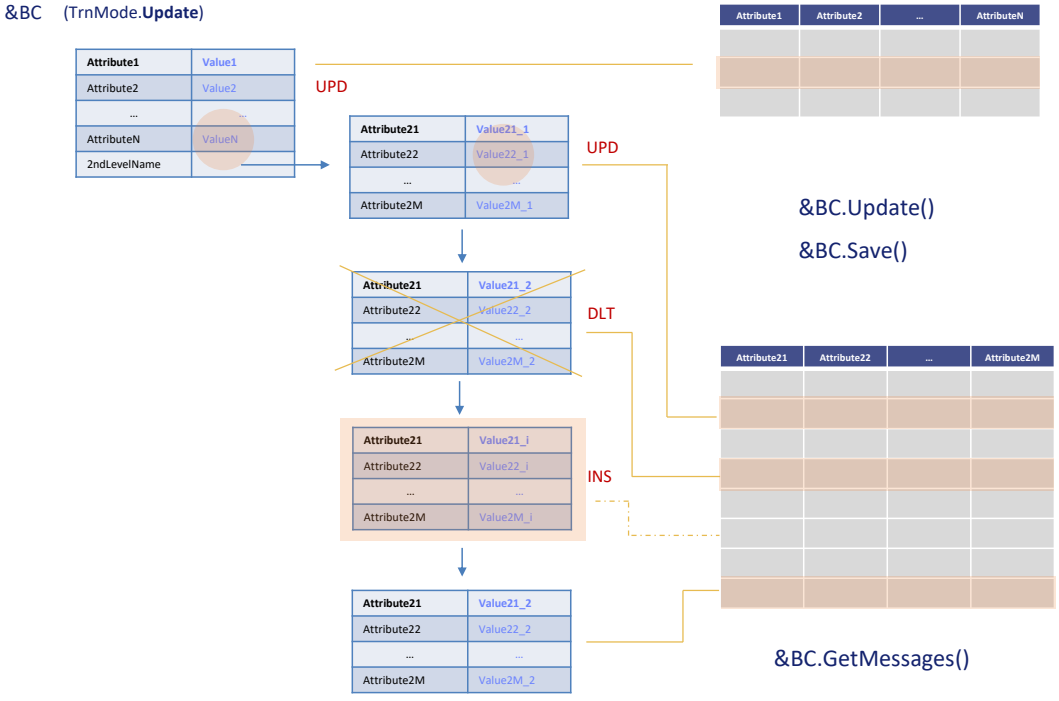


A word of caution: at the time this video was recorded, it was being reviewed whether to change this behavior so that the business component works in the same way as the transaction. In the transaction, if a line is in Update mode and nothing is done to it, that line is not processed at all. Therefore, not even rules are evaluated for it. However, for the BC, since the line is processed, because it is updated anyway, its rules will be evaluated and triggered. It is important to be aware of this.

&BC (TrnMode.**Update**)

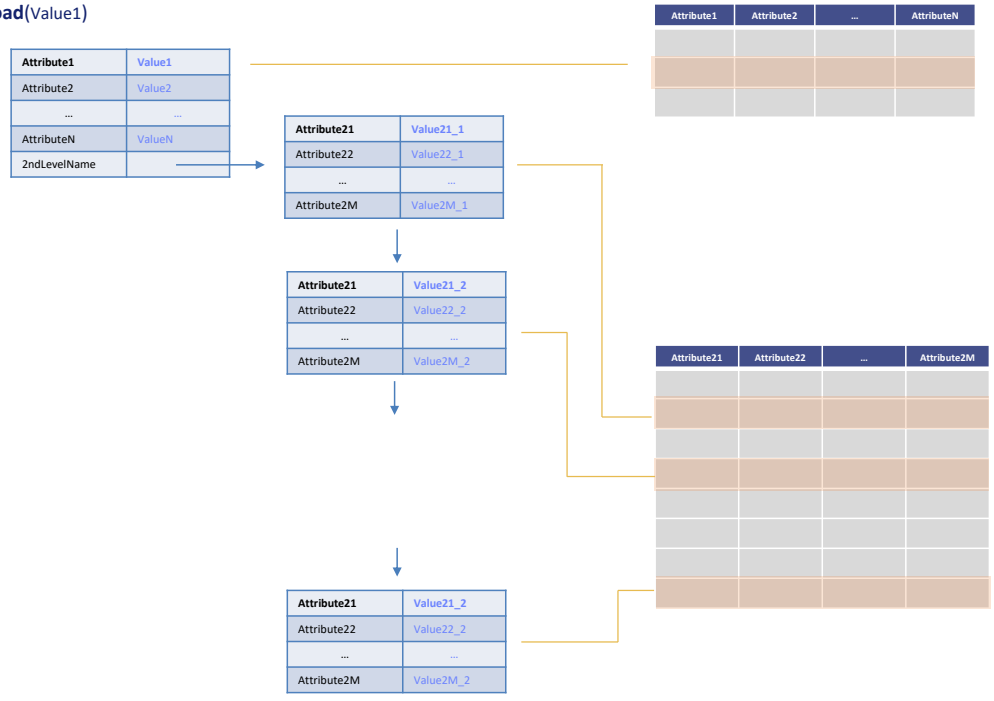


Let's resume our work. Of course, this solution is based on the premise that the contents of the BC when it was loaded...

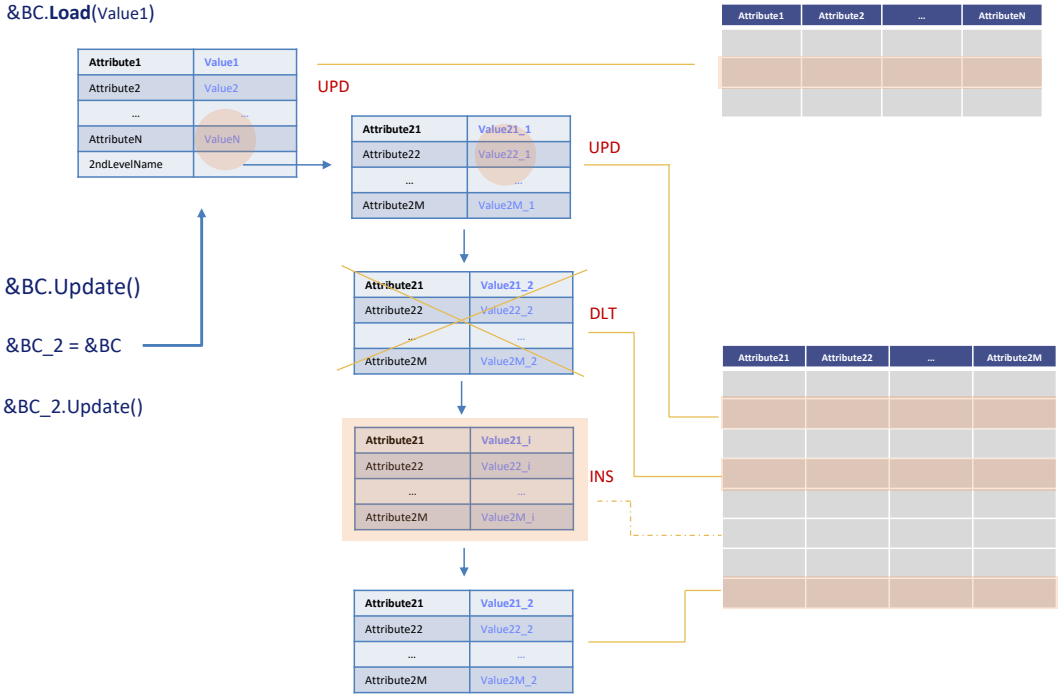


...before making the changes by code, match the database state immediately before the Update or Save operation was executed. Keep this in mind, because if someone changed the state of the database for these records in the meantime, the result will depend on those changes, so it is a good idea to review the messages generated after the Update or Save.

&BC.**Load**(Value1)



And what would happen if, for example, we loaded the &BC variable from the database first?

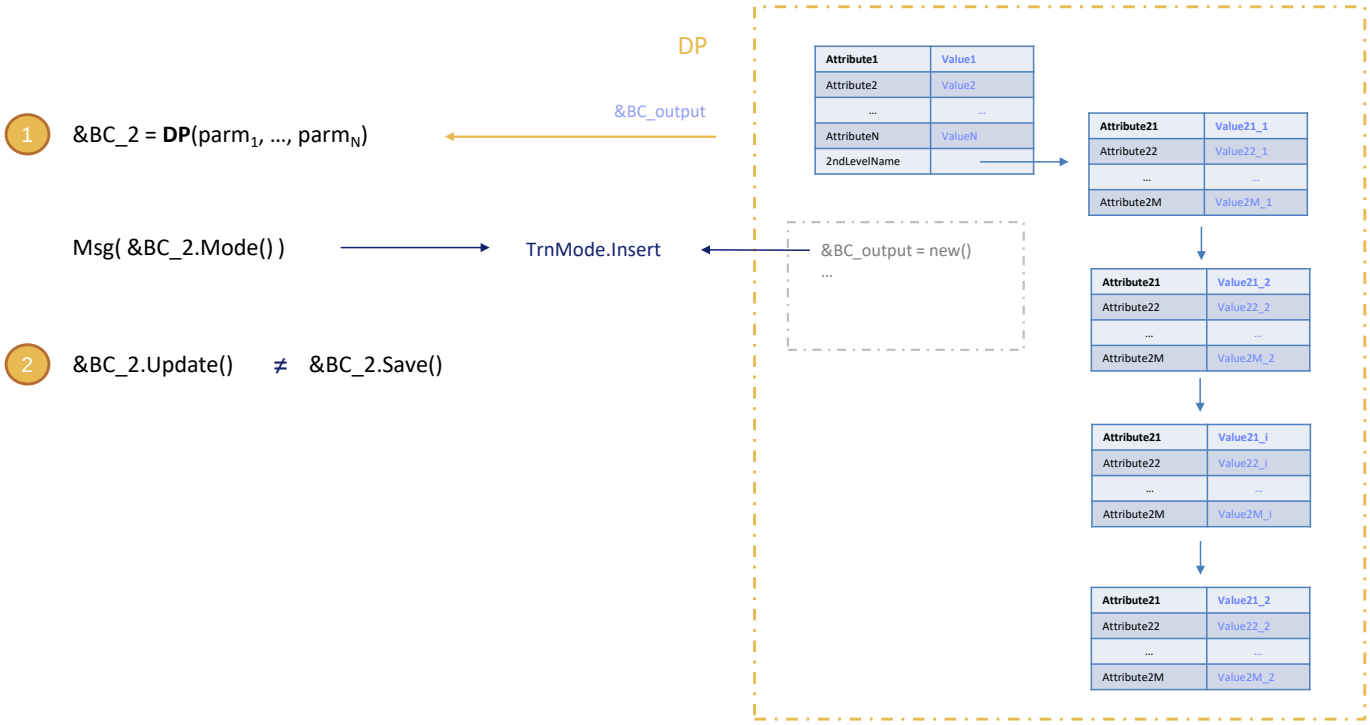


... And we handled it as we did here, and instead of doing an Update or Save, we assigned it to another BC variable, for example, to one to which we had applied a new before the assignment? And then we applied the Update to this new variable.

We could believe that here things become more complex, because the &BC_2 variable will be in Insert mode and not Update, because of the new. And we could also be led to believe that the recording of the operations performed on the items will have been lost. But none of this will be true.

Although the &BC_2 variable was left in Insert mode after this assignment, this other one will do away with that. In fact, it will no longer point to the new memory space, but will point to that of the &BC variable. The assignment does not make a copy of what has been assigned, but points to exactly the same place, as a pointer. For this reason, it will assume both the mode of the &BC variable and the recording of operations. And when the Update request arrives, it will be the same as having done the Update of the &BC variable.

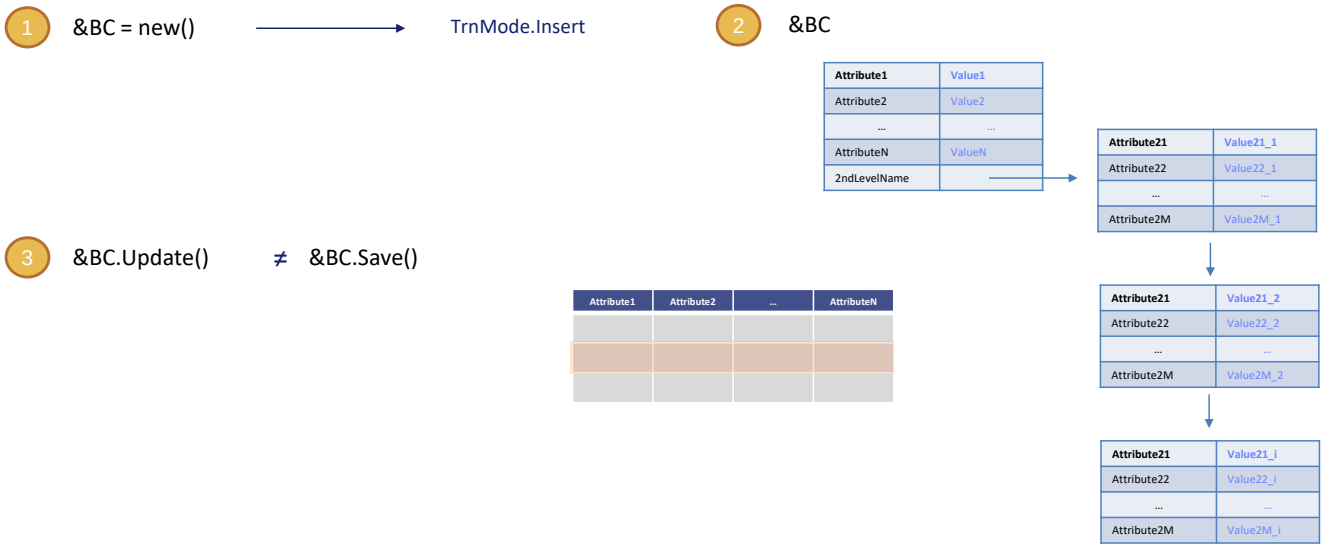
If the mode of &BC_2 at this point is Insert, in this other one it will be the mode of &BC, which in our example was Update.



Why doesn't it work like this when we assign the result of a Data Provider to a BC variable?

If we ask here for the mode of the &BC_2 variable, we will be surprised to find that it is Insert. This is because the first thing that the Data Provider does internally is a new, so the variable that will be returned will be, inevitably, in Insert mode.

Therefore, this case is equivalent to the one that we haven't analyzed here yet but in the videos of the Advanced course: what happens when the variable is in Insert mode and an Update operation is requested. Here it is not the same if the operation requested is the Update operation and not the Save operation.



Let's think about Save. Suppose we first request memory, so the variable is in Insert mode; then we assign values to the header properties and add 3 lines, and execute the Save. It will try to insert the BC into the database, which can only be successful if the values indicated in the BC for the primary key do not correspond to an existing record in the database table for the header. Remember that the Save method will always try to do the operation that corresponds to the mode in which the variable is in. In this case, if the record existed, the Save() would fail.

On the other hand, what will happen with the Update when the mode is Insert? In this case, since the variable is in Insert mode, it can be assumed that it was not loaded from the database. In other words, it is blind to the database record that the developer is assuming exists (it is assumed to exist, clearly, because otherwise an Update would not be explicitly requested).

In short, the developer loaded the variable with values that don't need to have been extracted from the database. This means, logically, that the Load operation was not performed, because in that case the variable would be in Update mode if that Load had been successful.
So it can be assumed that for the primary key values were assigned that the developer assumes correspond to an existing record, because if this is not the case, the Update operation will fail.

How does the Update work here?

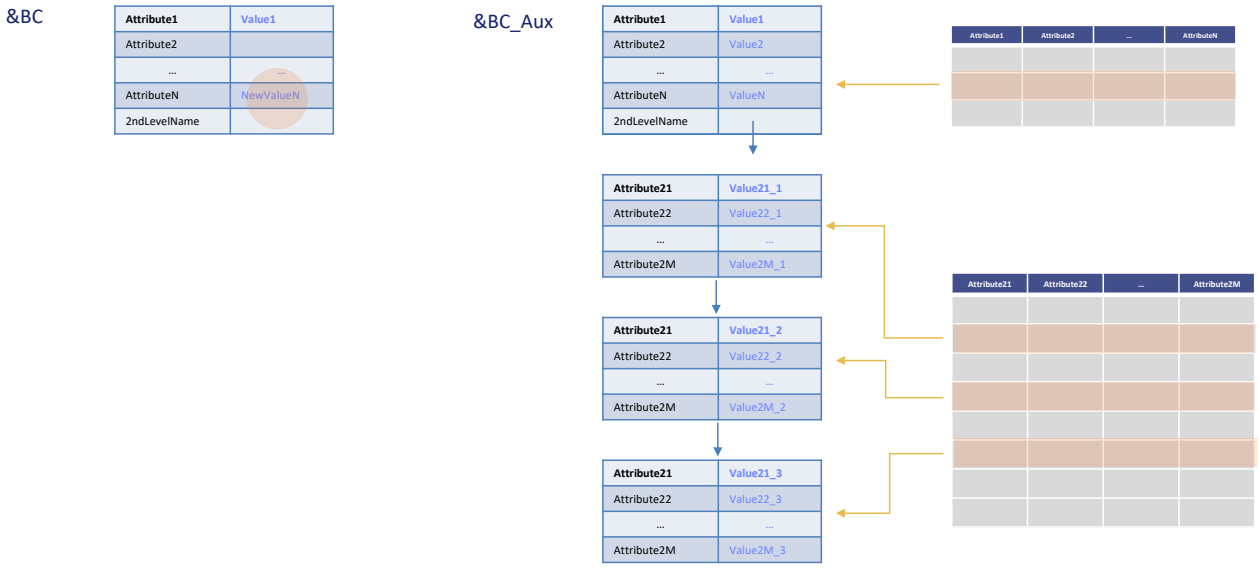
&BC

Attribute1	
Attribute2	
...	...
AttributeN	
2ndLevelName	

Attribute1	Attribute2	...	AttributeN

Attribute21	Attribute22	...	Attribute2M

Let's suppose that this is the variable on which we did a new or that we use for the first time in the code, so it will be in Insert mode.



&BC.Update()

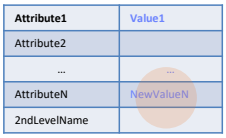
Then we only assign a value to the property that corresponds to the primary key, in order to identify the database record that will correspond to the header. And we give a value to the property that we want to modify, the one that corresponds to this attribute. All the other properties are left unchanged; that is to say, they remain empty.

Then we handle the collection of lines—we will see what we can do—and execute the Update method.

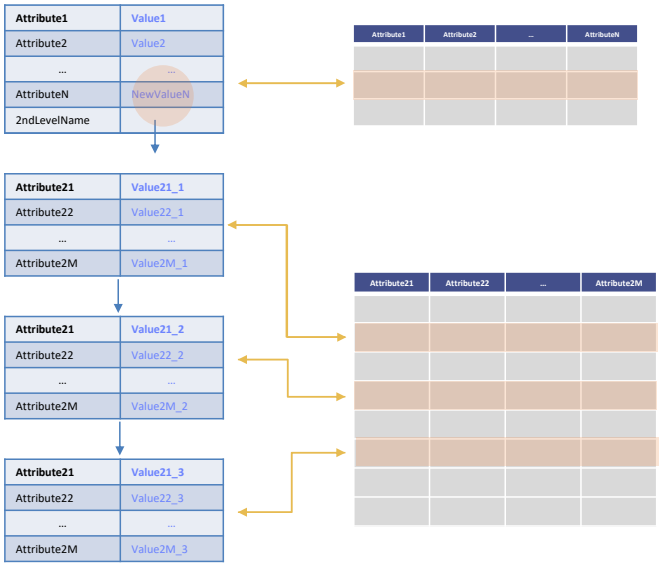
Internally, the method detects that the variable is in Insert mode, so it will create an auxiliary variable on which it will apply a Load with the values of the primary key of the developer's variable. Therefore, if the record exists, it will load in the auxiliary variable exactly the same information from the database for that identifier.

What does it do next? In the example where only one property of the header was modified...

&BC



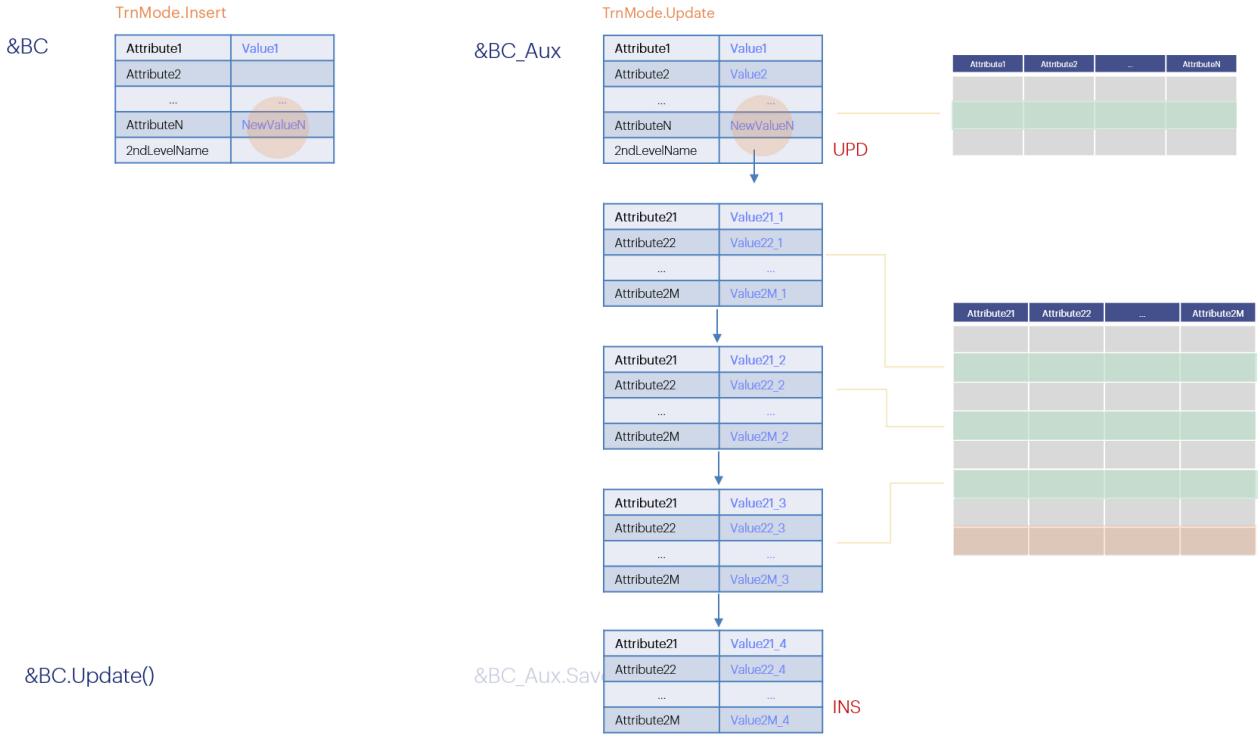
&BC_Aux



&BC.Update()

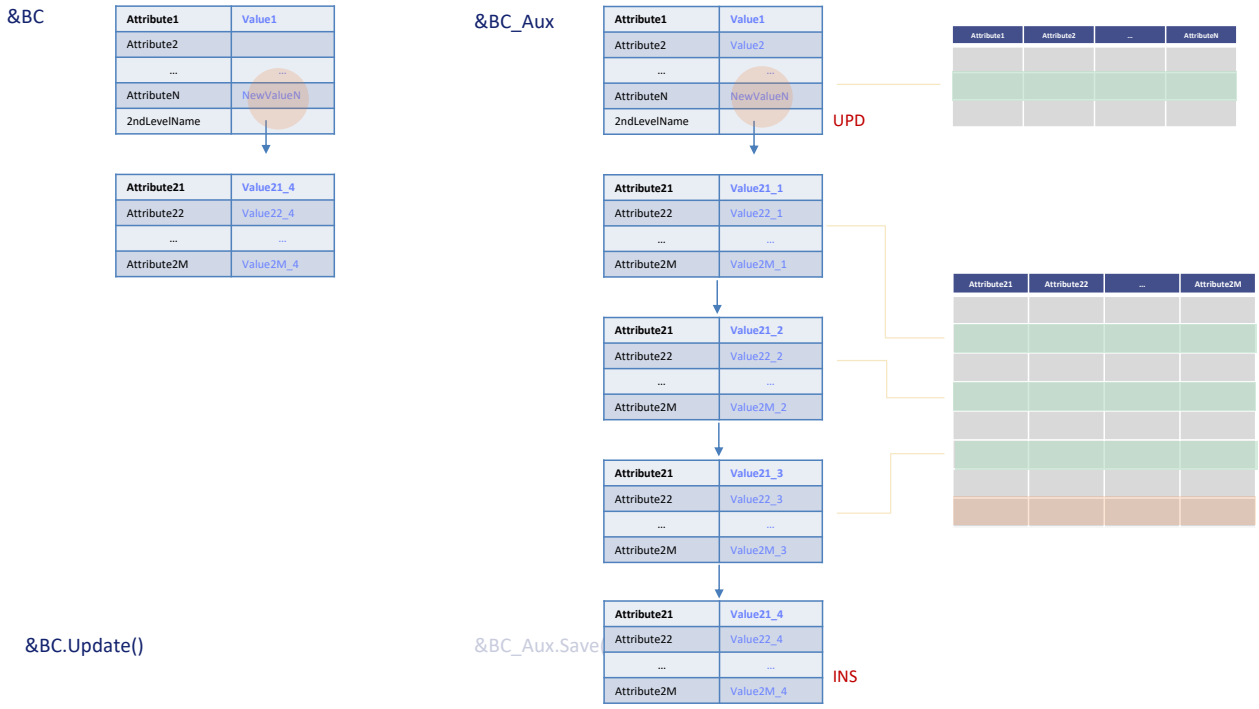
&BC_Aux.Save()

...it copies that value for the same property of the auxiliary variable, overwriting it. And on the auxiliary BC variable it executes a Save, which is the same that we analyzed at the beginning of the video, since the auxiliary variable is in Update mode.

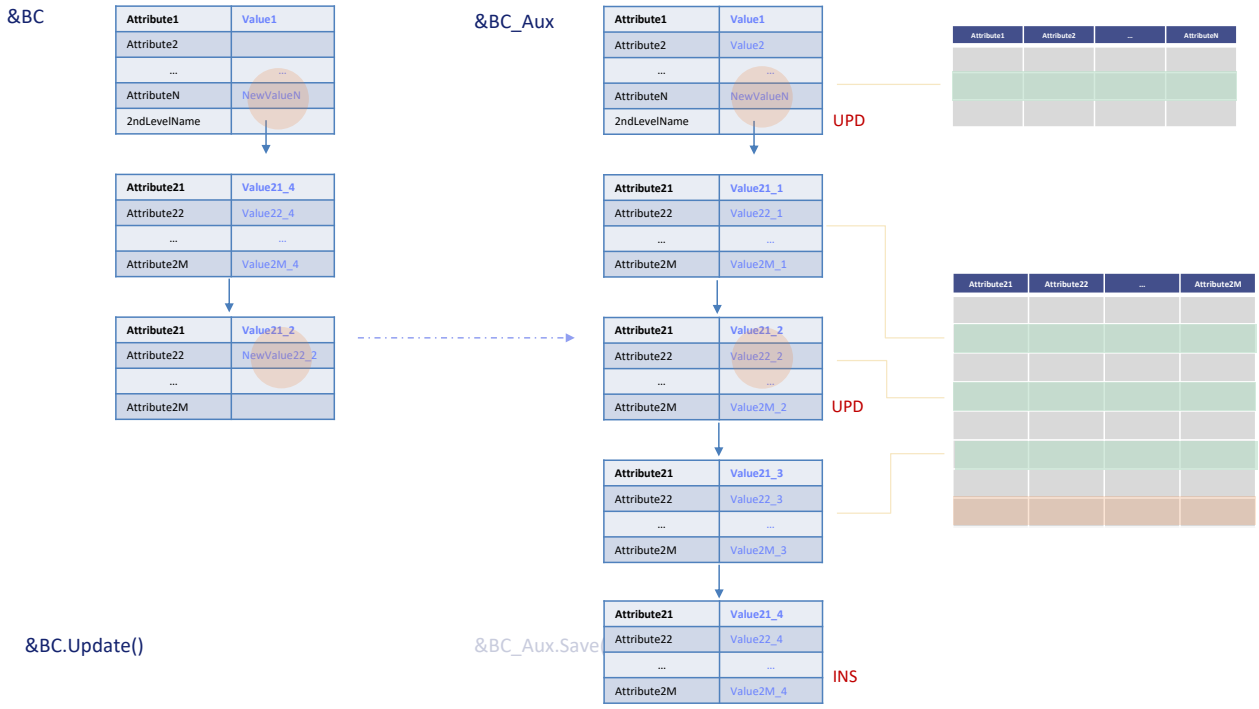


Now let's think about the lines and give the complete explanation.

Let's suppose that what we wanted, in addition to modifying this attribute, was to add a new record to the second level table. To get this done on the auxiliary variable that will have an impact on the Insert of this record in the database...

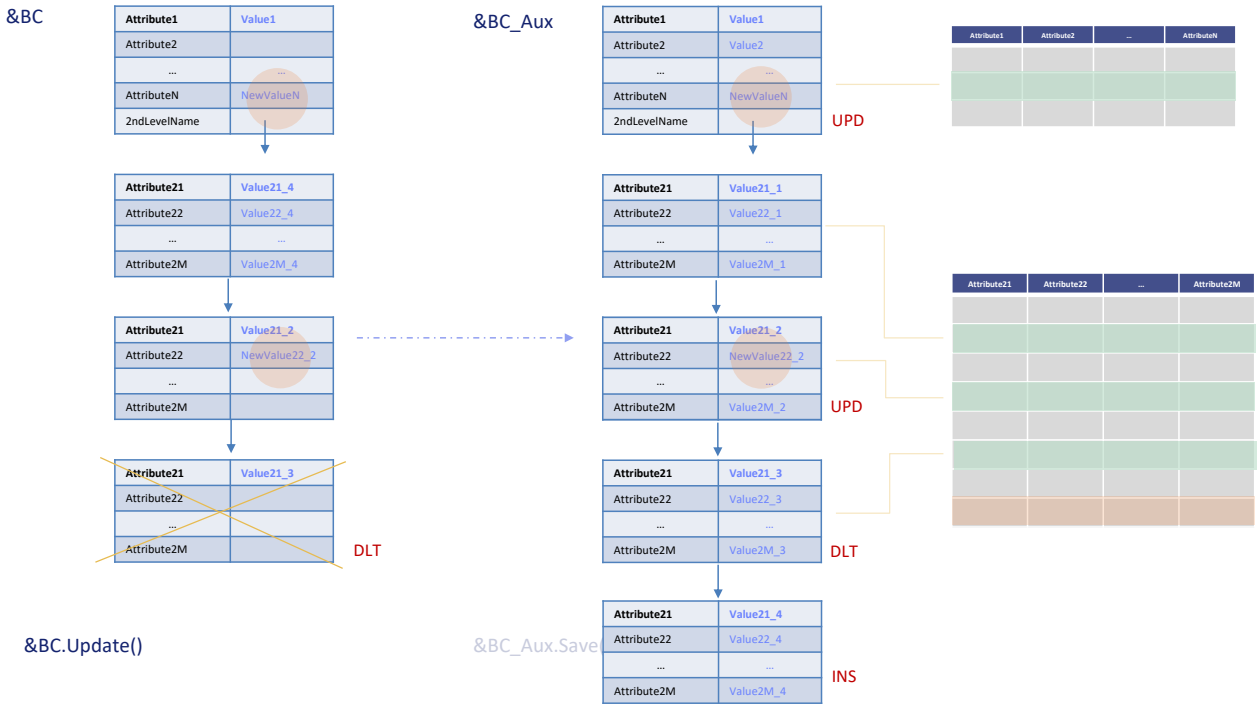


...we add an item to the collection of our variable, and assign all its values to it. What will happen in the background is that when this item is evaluated to determine what to do with it on the auxiliary variable, the GetByKey method will search on the line collection for one with this value. If it is not found, then it is added.



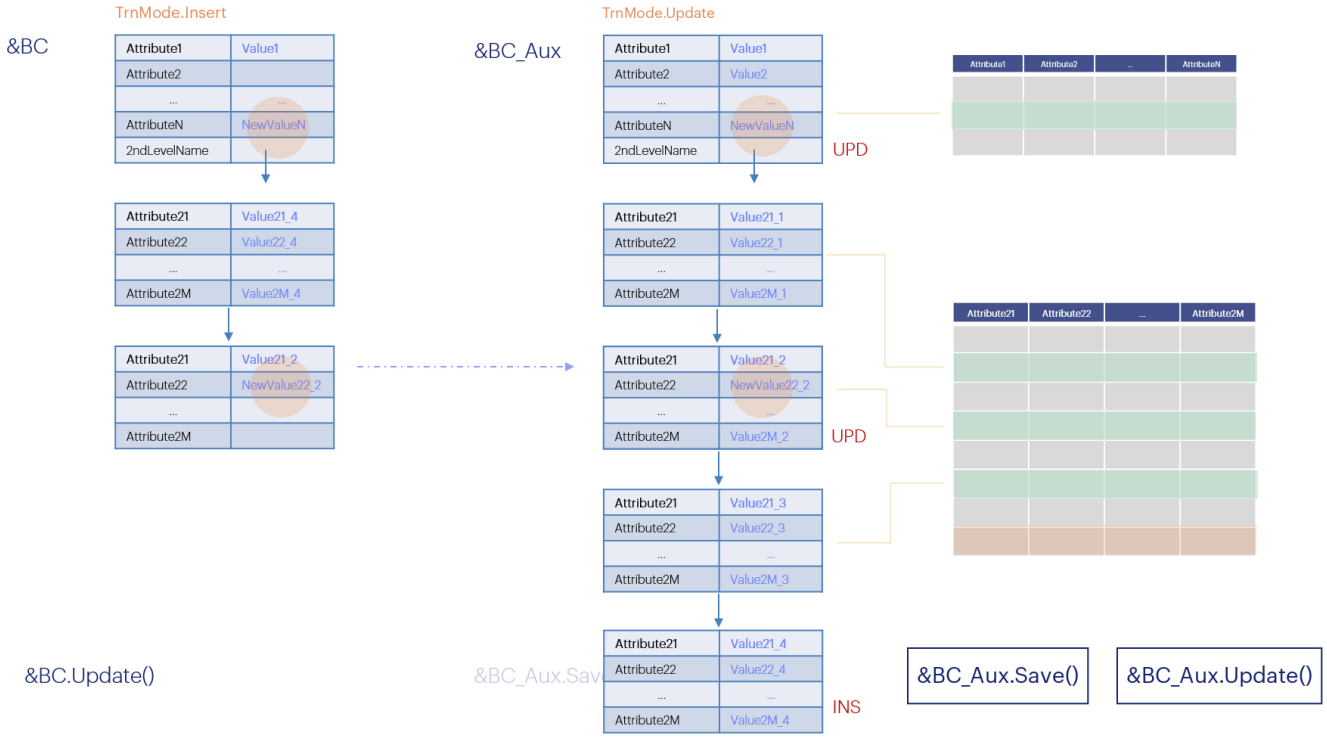
Next let's suppose we want to update an existing record— for example, this value—so we add another item but where we only assign a value to the identifier and to the property corresponding to the attribute we want to modify. We only indicate the new value. All other properties are left unchanged; that is, empty.

When analyzing what to do with this item on the auxiliary variable, a GetByKey is run, and since an item is found, only what was changed—that is, this property—is copied.



Finally, let's suppose that we want to delete a record, so we could assume that it is enough to add another item, only assigning a value to the property that identifies it, and then execute a Remove or RemoveByKey **of that item**, so that it is marked for deletion.

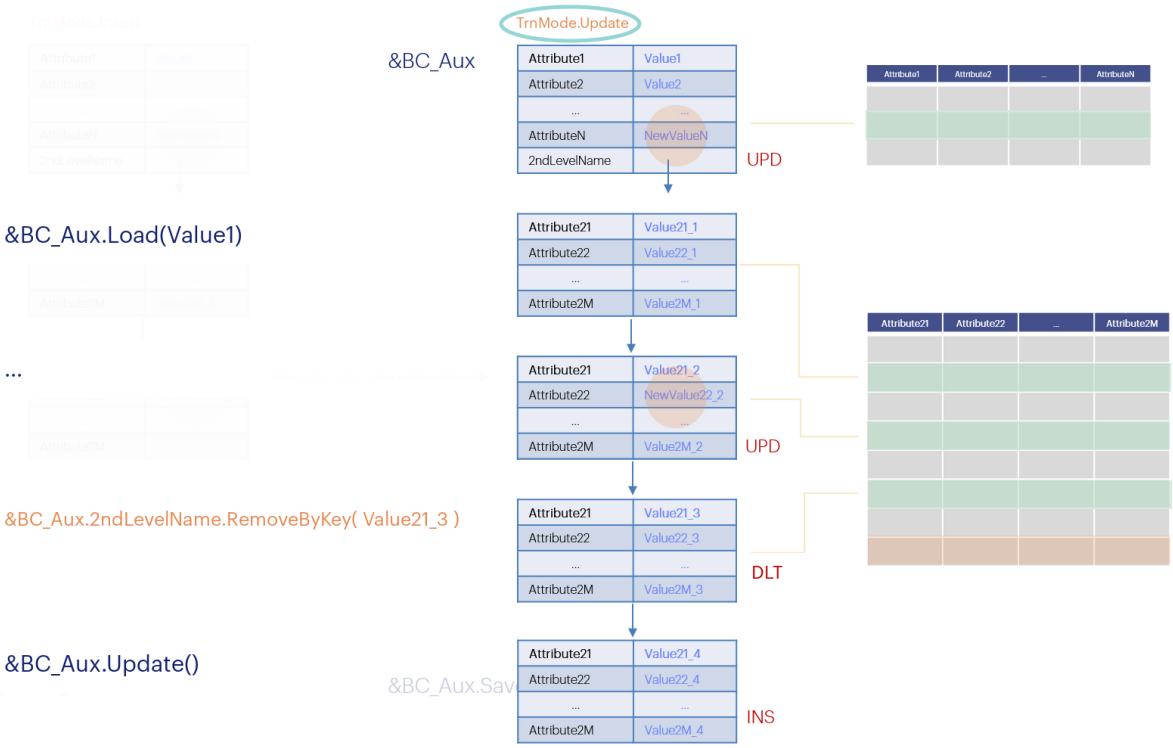
However, it will not work like that. If the BC variable is in Insert mode, when the item is removed from the collection it is not marked for deletion, but is directly deleted. This means that we **cannot delete lines** with BC variables in **Insert mode**. They must be in Update mode to be able to do this, because only in that mode the item is marked for deletion.



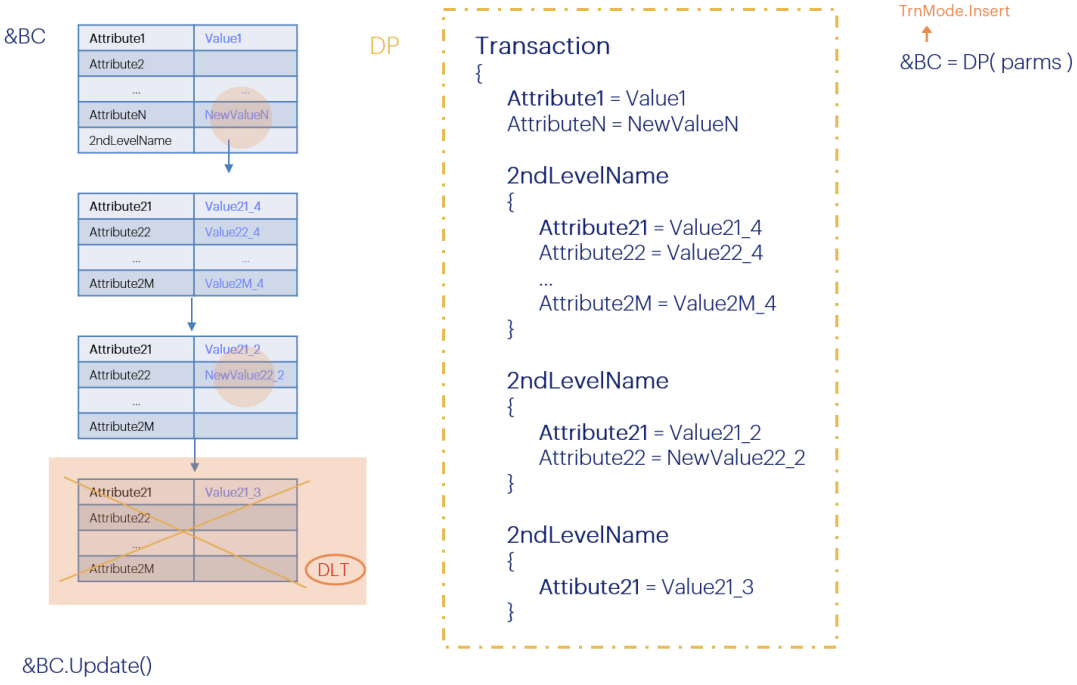
In short, a variable is used in Insert mode only to modify and/or add lines, but not to delete.

And what the developer does on this new variable is only indicate the values to be modified from the header, and for each line to be modified, its identifier and value or values to be modified; and for a new line, provide all its values.

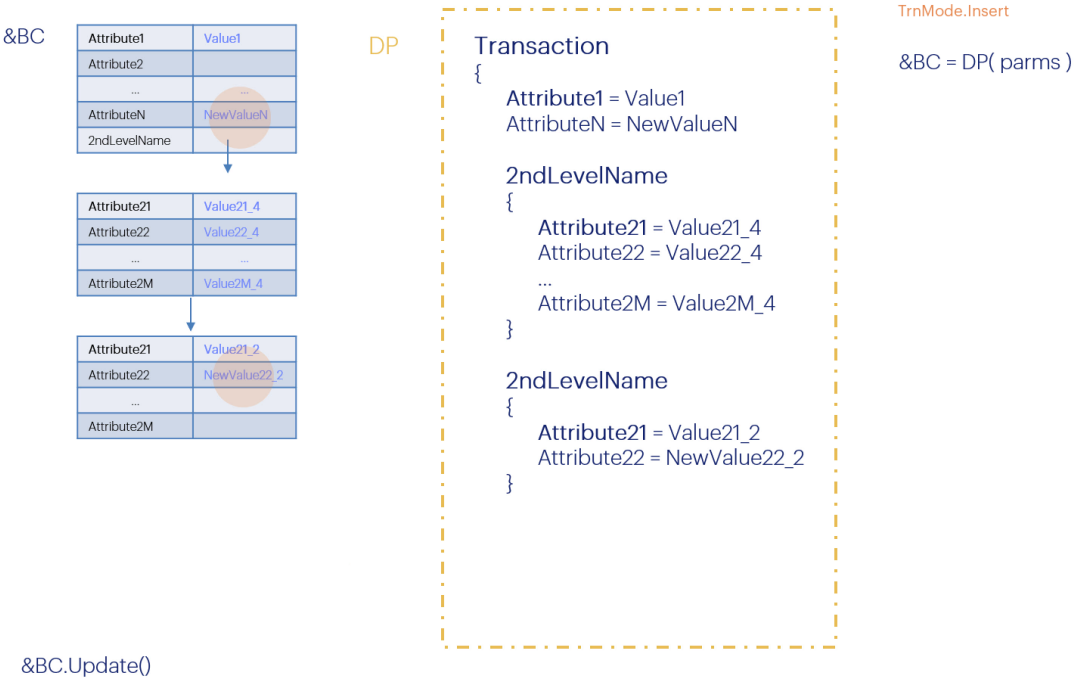
This will have the consequences that we have just analyzed on the auxiliary variable, on which the Save() will then be performed, which in this case does match the Update.



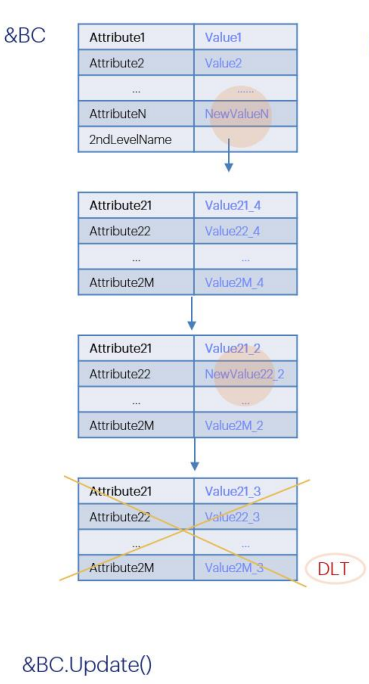
From what we have seen, if we want not only to modify data but also to delete lines, it will be convenient to make an explicit Load so that the variable is loaded from the database and there perform all the operations, including the deletion of items.



However, if we are going to use a Data Provider to load the &BC variable, since in this case we know that it will inevitably be in Insert mode, how can we delete this line?



To achieve what we want, in this case we should not add the item to the variable, clearly, but we should only make the changes related to modifications and new items...



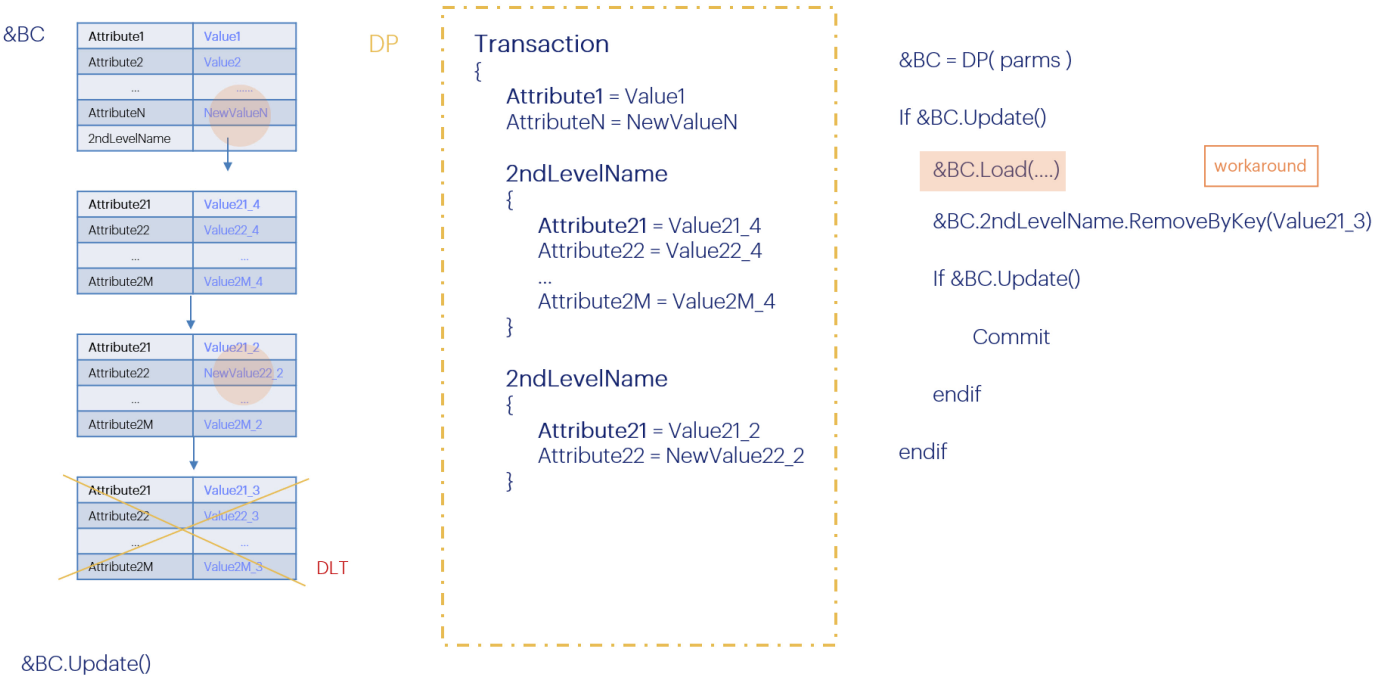
DP

```
Transaction
{
    Attribute1 = Value1
    AttributeN = NewValueN

    2ndLevelName
    {
        Attribute21 = Value21_4
        Attribute22 = Value22_4
        ...
        Attribute2M = Value2M_4
    }

    2ndLevelName
    {
        Attribute21 = Value21_2
        Attribute22 = NewValue22_2
    }
```

```
&BC = DP( parms )

If &BC.Update()

    &BC.2ndLevelName.RemoveByKey(Value21_3)

    If &BC.Update()

        Commit

    endif

endif
```

&BC.Update()

...execute the Update and then, if it is successful, since the variable will be in Update mode, we can add the Remove or RemoveByKey because after the Update we know that the variable will be loaded with all the information of the database, including the line that we want to delete.
Then the RemoveByKey will mark it to be deleted in the next Update, which is what we will have to execute for this action to be performed. Next, we will be able to commit.



However, in GeneXus 17 there is a bug so if the &BC variable is in Insert mode when the Update is made, even if it is successful, the content of the variable is not updated. It is left in the database as is and therefore the line that we want to delete is loaded. It will be solved in version 18, but for now as a workaround we could make the load explicit.

This is the end of the in-depth theoretical analysis on how to update through the Business Component.

In the following video, we will see all this with an example.

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