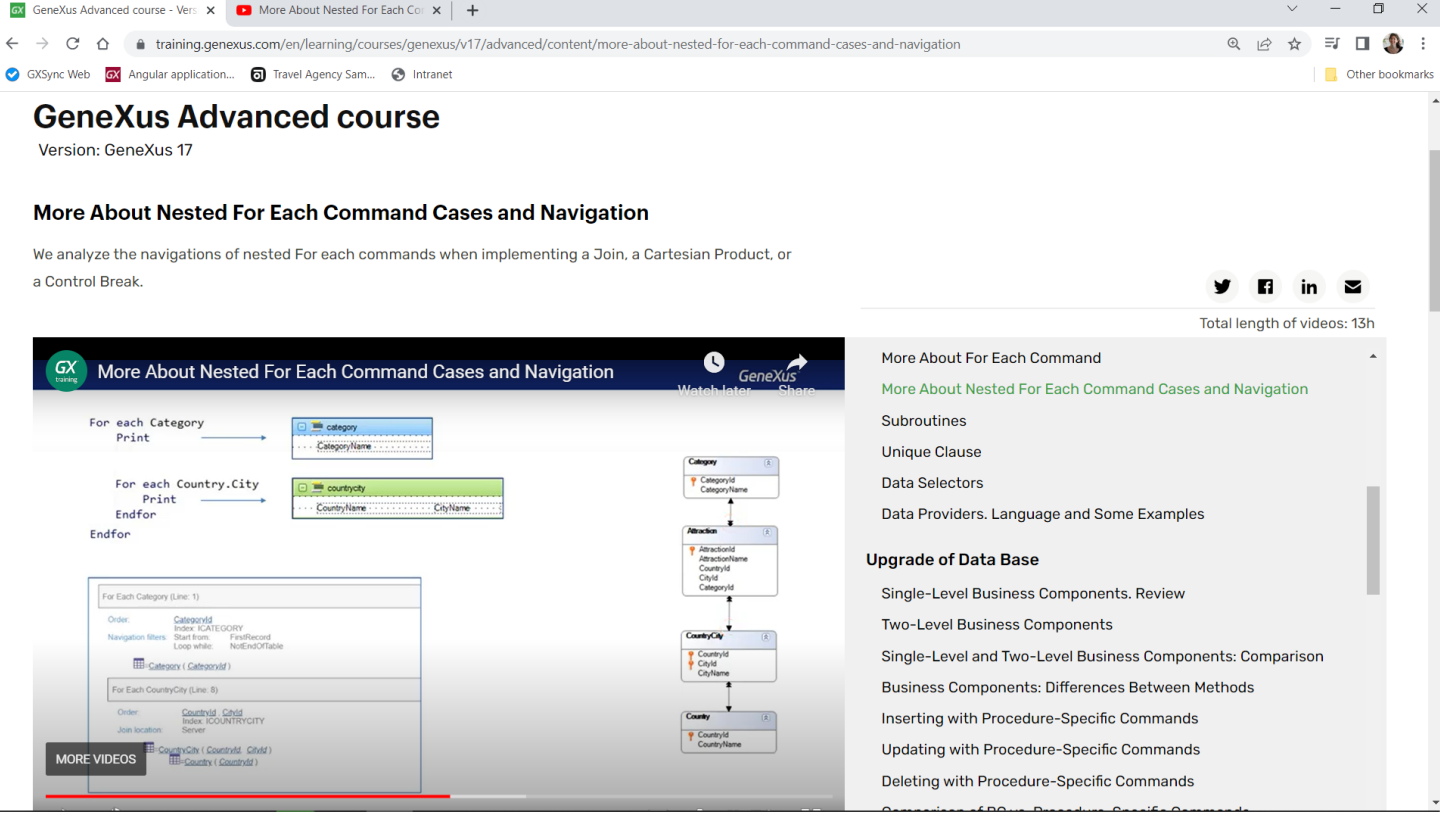
Logic for querying the database with GeneXus

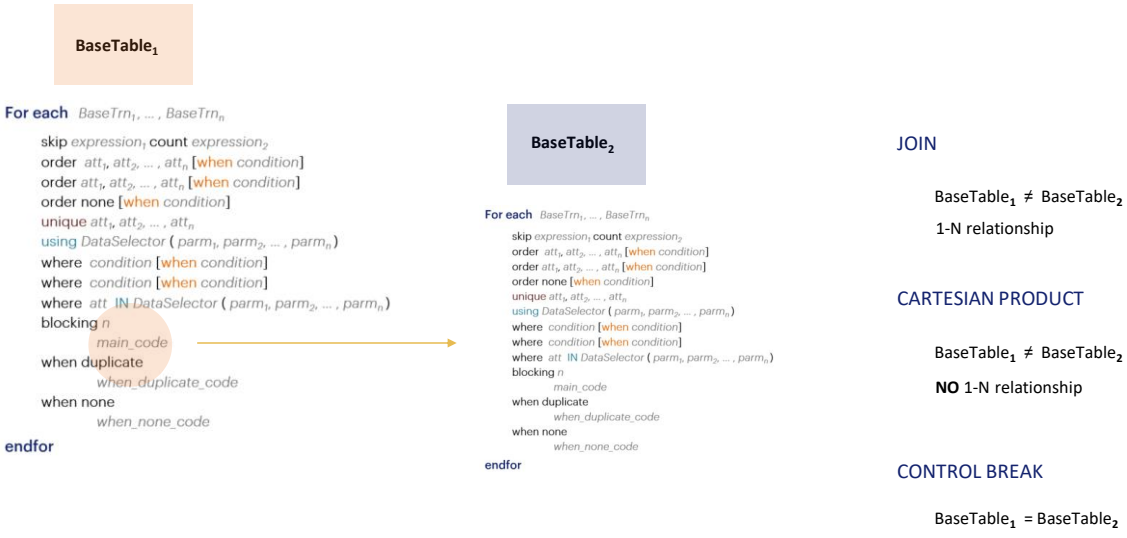
Cases of nested For each commands. Formalization

GeneXus™



In this video... we had discussed the three types of navigations that GeneXus implements when it finds nested For each commands.

We will come back to them to formalize them a bit more.

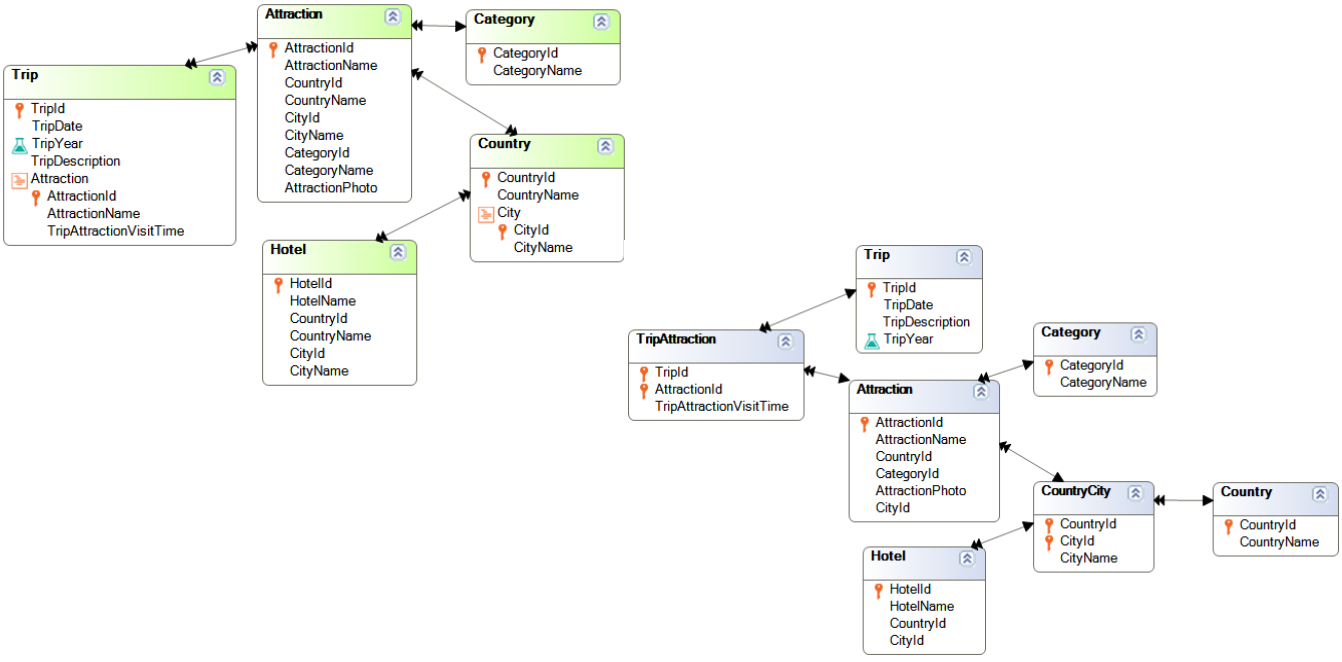


When inside the body of a For each command there is another one, it means that for each record in one navigation of a table, we want to run through many records in another navigation of the same or another table. A particular case is when a single record is retrieved.

First, the base tables are determined and then the relationship between them is explored. This will lead to one of three cases: Join, Cartesian Product, and Control Break.

We are aware of a first major difference: the first two cases will involve different base tables, while the latter is a case of the same base table.

What is the difference between a Join and a Cartesian Product? The identification of a direct or indirect 1 to N relationship. Let's formalize all cases.

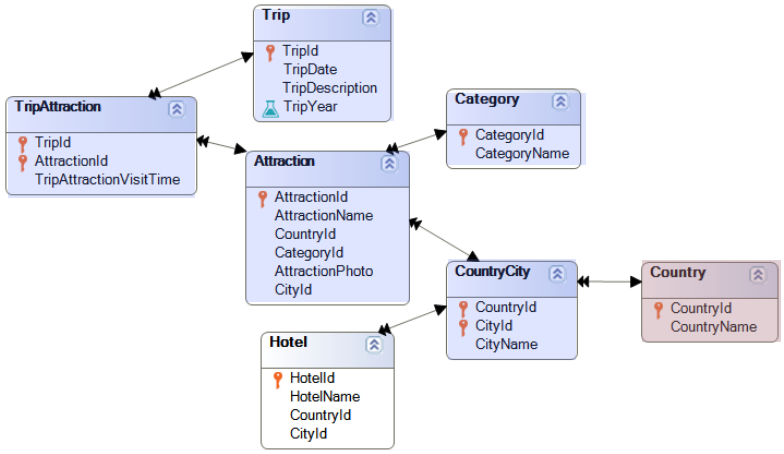


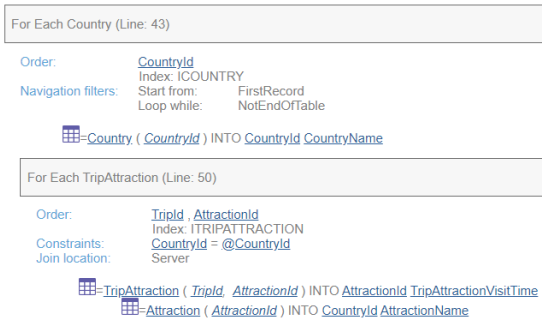
Suppose we have these 5 transactions to record the tourist attractions that can be visited on tours, where each attraction corresponds to a category (such as Museum or Monument), and is from a city in a country. On the other hand, we have hotels in each country and city.

From these transactions, we obtain these tables with their relationships.

1

```
for each Country
   print PB1 //CountryName
   for each Trip.Attraction
      print PB2 //AttractionName, TripAttractionVisitTime
   endfor
endfor
```





$BaseTable_1 \subset ext(BaseTable_2)$

Let's start with the Join. How will this code be resolved?

For each country name, you need to print the attraction names with their visit time specified in the trip. Is there a relationship between the information?

We know that a TripAttraction record belongs to a single attraction, which corresponds to a single city, which in turn corresponds to a single country. In other words, from the TripAttraction table we reach Country in a unique way. Therefore, for the second For each we can print only the tripattractions corresponding to the country of the main For each, as we can see in the navigation list as a constraint. This CountryId preceded by @ (at sign) corresponds to the CountryId of the record of the external For each where we are positioned. And this CountryId attribute is that of Attraction, which is reached through TripAttraction. Clearly this is a Join, with a 1 to N indirect relationship.

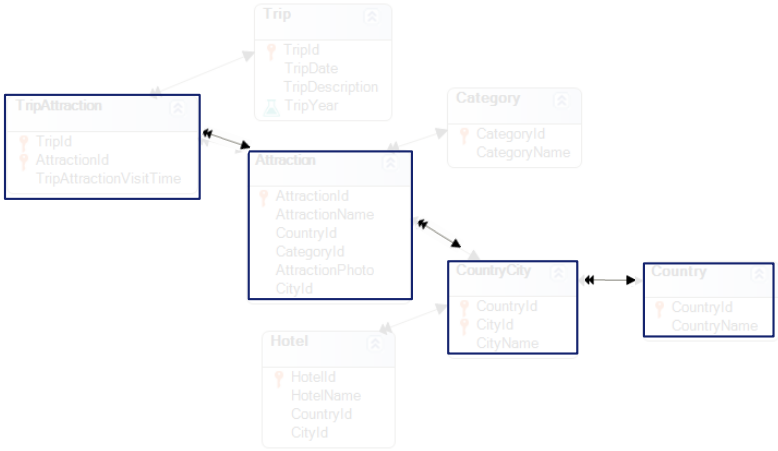
This is because for a tripattraction following the foreign keys we find a single CountryId; therefore, for a CountryId we can find N tripattractions that will find it by following this path.

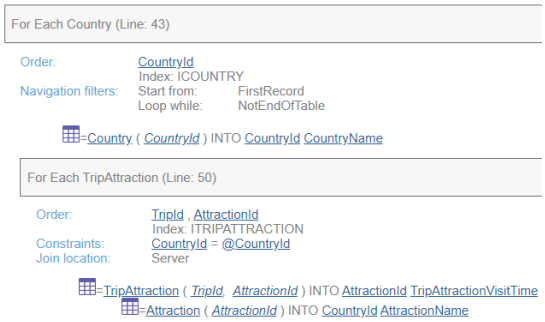
If we formalize this case of 1 to N indirect relationship, we are saying that the base table of the main For each is contained in the extended table of the nested For each.

That case includes the simplest of all. For example, consider that if the base table of the second For each were CountryCity, this formula would be satisfied.

1

```
for each Country
   print PB1 //CountryName
   for each Trip.Attraction
      print PB2 //AttractionName, TripAttractionVisitTime
   endfor
endfor
```





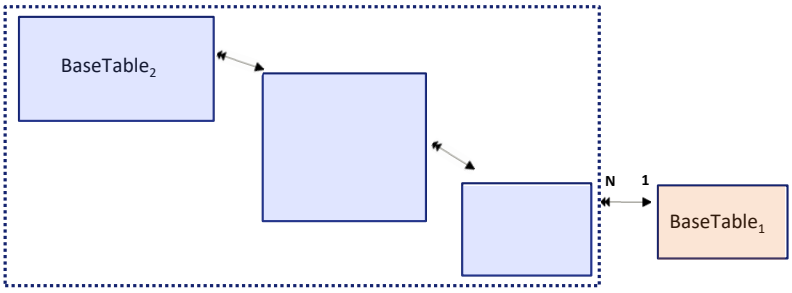
JOIN

$BaseTable_1 \subset ext(BaseTable_2)$

Here we see more clearly the relationships between the tables involved in this case.

1

```
for each Country
   print PB1 //CountryName
   for each Trip.Attraction
      print PB2 //AttractionName, TripAttractionVisitTime
   endfor
endfor
```



JOIN

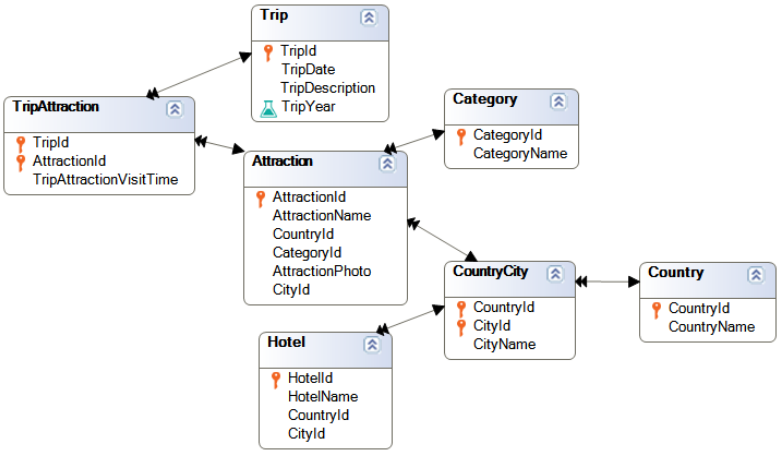
$BaseTable_1 \subset ext(BaseTable_2)$

That is, the base table of the main one is this one, and the base table of the nested one is this other one.

The indirect 1 to N relationship is clear. It is indirect through the extended table of the nested For each.

2

```
for each Trip.Attraction
   print PB1 //AttractionName, TripAttractionVisitTime
   for each Hotel
      print PB2 //HotelName
   endfor
endfor
```



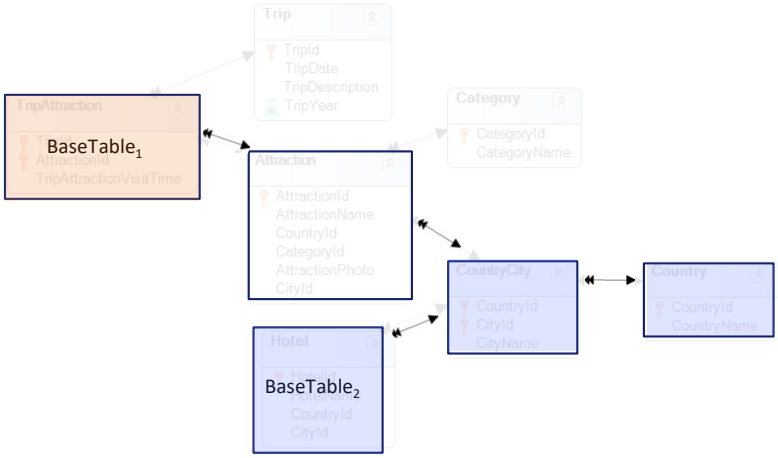
~~$BaseTable_1$ ⊂ ext($BaseTable_2$)~~

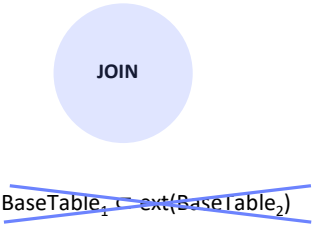
Now let's look at the other type of indirect 1 to N relationship, this time through the main one, instead of through the extended table of the nested one.

For each tripattraction, we print the attraction name and visit time on that trip, and then the hotel names.

2

```
for each Trip.Attraction
   print PB1 //AttractionName, TripAttractionVisitTime
   for each Hotel
      print PB2 //HotelName
   endfor
endfor
```

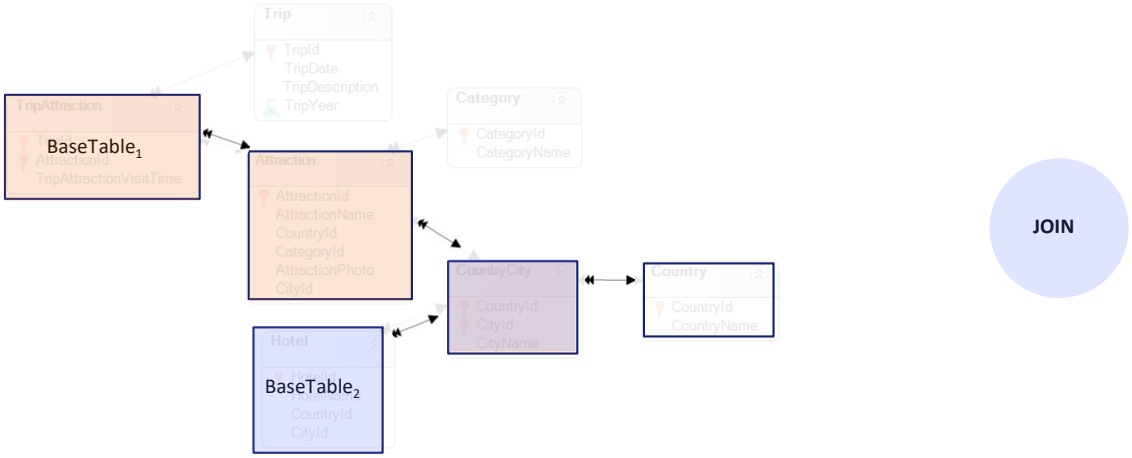




Here it is clearly not true that the base table of the main For each is included in the extended table of the nested For each. There is no indirect 1 to N relationship through this way. However...

2

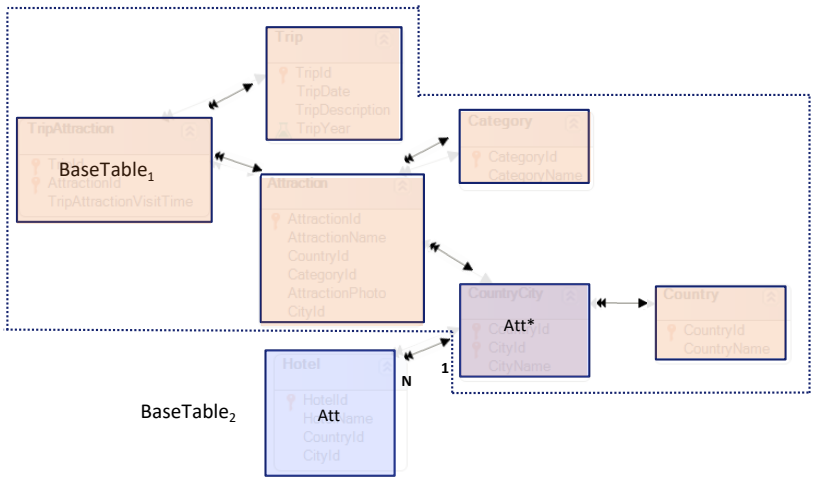
```
for each Trip.Attraction
   print PB1 //AttractionName, TripAttractionVisitTime
   for each Hotel
      print PB2 //HotelName
   endfor
endfor
```



...we know that for each record of the main For each we reach a record of this table, which is also reached directly from the base table of the nested one.

2

```
for each Trip.Attraction
   print PB1 //AttractionName, TripAttractionVisitTime
   for each Hotel
      print PB2 //HotelName
   endfor
endfor
```



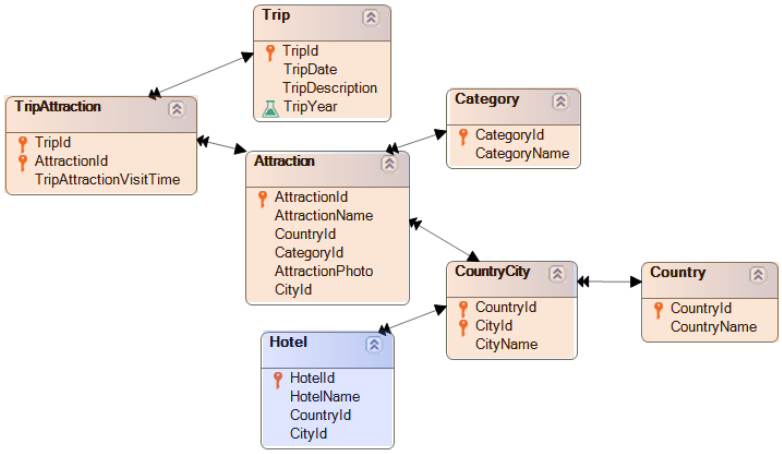
ext($BaseTable_1$) ∩ $BaseTable_2$ ≠ ϕ

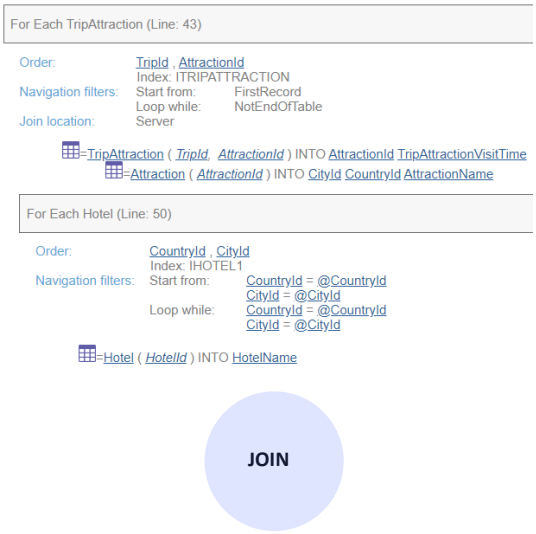
That is to say, in this one there will be a foreign key to this other one. They will share this attribute (because of it, this 1 to N relationship is established between the tables).

In short, the extended table of the main For each will have some attribute in common with the base table of the nested one, and they will establish the Join.

2

```
for each Trip.Attraction
   print PB1 //AttractionName, TripAttractionVisitTime
   for each Hotel
      print PB2 //HotelName
   endfor
endfor
```





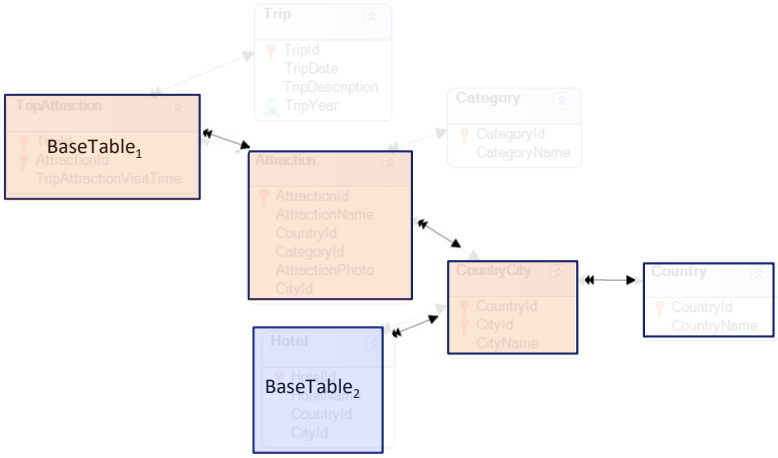
$ext(BaseTable_1) \cap BaseTable_2 \neq \phi$

Here they will be CountryId, CityId, foreign key to CountryCity.

We can see it clearly in the navigation list. Only the hotels corresponding to the same city of the trip attraction will be printed.

2

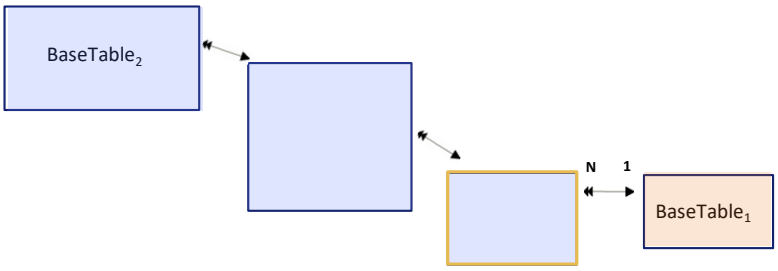
```
for each Trip.Attraction
   print PB1 //AttractionName, TripAttractionVisitTime
   for each Hotel
      print PB2 //HotelName
   endfor
endfor
```



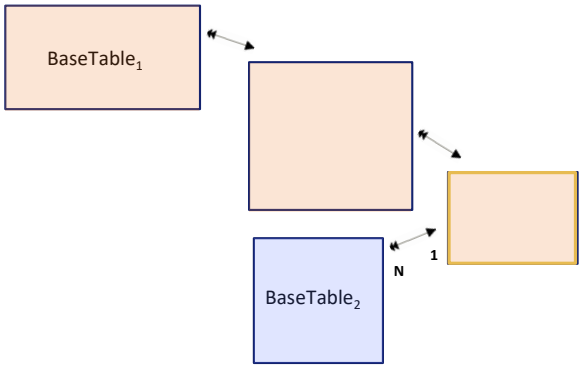
$ext(BaseTable_1) \cap BaseTable_2 \neq \phi$

Again, to see it more clearly, let's keep only the tables involved: here we see the indirect 1 to N relationship.

1 $BaseTable_1 \subset ext(BaseTable_2)$



2 $ext(BaseTable_1) \cap BaseTable_2 \neq \phi$



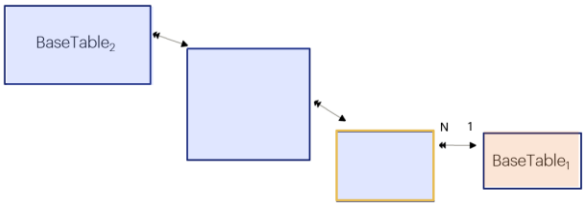
If in the first case this base table was run through and for the nested one this other one, and therefore the relationship was indirect 1 to N through the extended table of the nested one, in the second case it is through the extended table of the base table of the main one. And it is here that the 1 to N relationship is established with the base table of the nested one.

JOIN

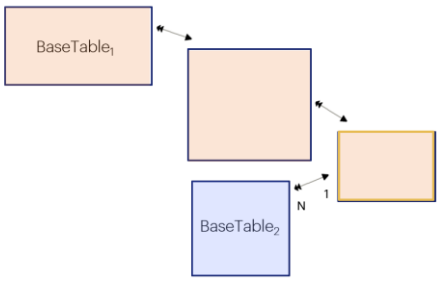
1-N relationship

$BaseTable_1 \neq BaseTable_2$

1 $BaseTable_1 \subset ext(BaseTable_2)$



2 $ext(BaseTable_1) \cap BaseTable_2 \neq \phi$



Therefore, the Join case is that of different base tables where a direct or indirect 1 to N relationship is found, according to these options: base table with extended table, or extended table with base table.

There is no extended table with extended table.

JOIN

1-N relationship

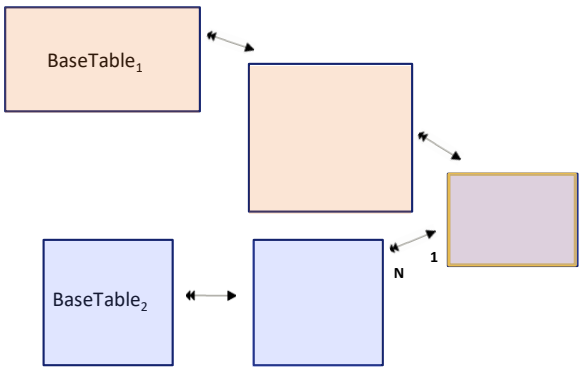
1 $BaseTable_1 \subset ext(BaseTable_2)$

2 $ext(BaseTable_1) \cap BaseTable_2 \neq \phi$

$BaseTable_1 \neq BaseTable_2$

CARTESIAN PRODUCT

NO 1-N relationship



That is to say, if the base table of the nested one is not this one but this other one, there is a relationship between the extended tables, obviously, because both arrive at the same table. However, here there will not be a Join but a Cartesian product.

Why, if for each record of the base table of the main For each we may keep only those of the base table of the nested one that correspond to the same record of this table to which both arrive in a unique way?

The more indirect the relationship, the less likely it seems that the developer is looking to take it into account, because the relationship seems more and more distant, and if the developer is looking for it, they can always make it explicit.

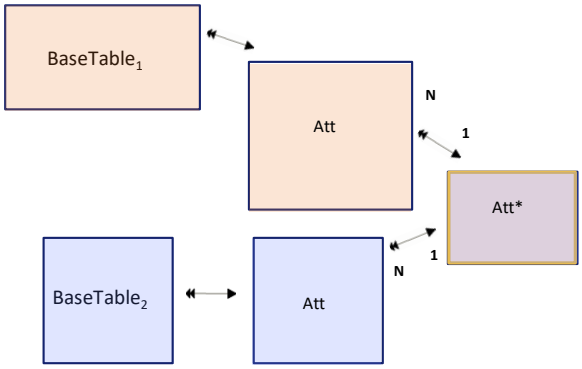
JOIN

1-N relationship

1 $BaseTable_1 \subset ext(BaseTable_2)$

2 $ext(BaseTable_1) \cap BaseTable_2 \neq \phi$

$BaseTable_1 \neq BaseTable_2$



CARTESIAN PRODUCT

```
for each
   ...
   &var = Att
   for each
      where Att = &var
         ...
   endfor
endfor
```

For example, by assigning to a variable the value of the attribute that is obtained by following the path of the extended table of the main For each... that is to say, the value of this attribute that matches this one.
And in the nested For each, **explicitly** filtering the records from which you get to this other attribute that is called the same because it is also a foreign key to the common table.

Let's take this opportunity to make a clarification: when we speak of a Join or Cartesian Product, of this difference, we are referring to the implicit determinations of GeneXus, not to whether it ends up filtering the information of the nested one. Note that in this case it is a Cartesian Product in the sense that if we had not written a Where, GeneXus would not add it implicitly either and all the records of the nested one would be returned. Actually, in this case not all the records in the nested one will be returned because we explicitly wrote a Where, so it will actually do a Join, but not the implicit Join of GeneXus.

JOIN

1-N relationship

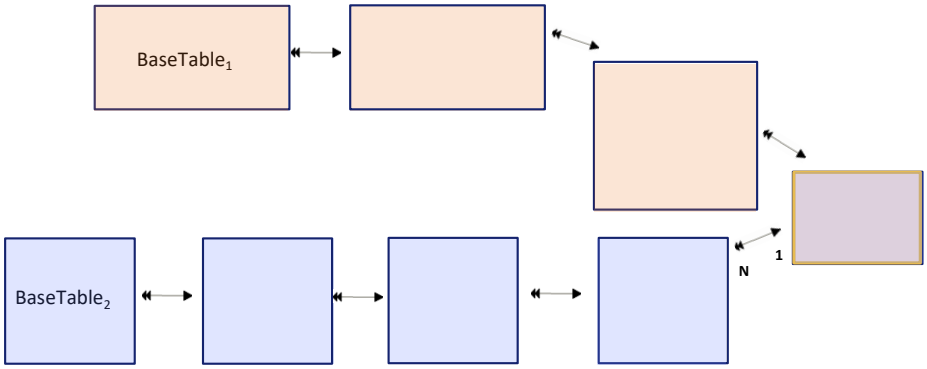
1 $BaseTable_1 \subset ext(BaseTable_2)$

2 $ext(BaseTable_1) \cap BaseTable_2 \neq \phi$

$BaseTable_1 \neq BaseTable_2$

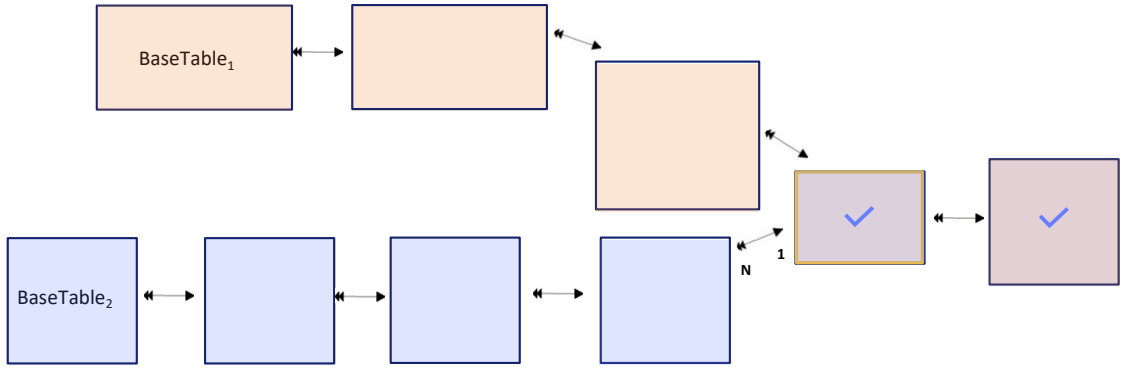
CARTESIAN PRODUCT

NO 1-N relationship



The more distant the relationship, the less likely it will be that the developer has it in mind, or is trying to enforce it implicitly.

CARTESIAN PRODUCT




```
for each
   ...
  for each
      ...
   endfor
   ...
endfor
```

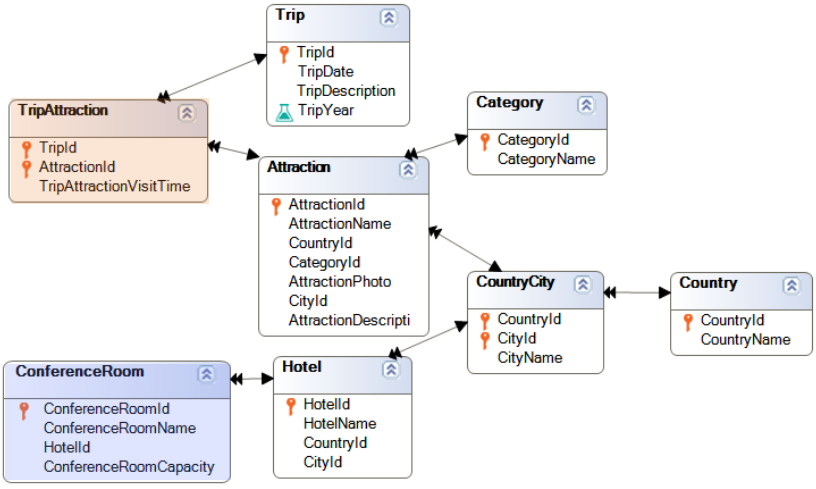
The following is an interesting aspect of this case. We add one more table to make it even clearer.

If we do the intersection of both extended tables, we are left with the tables in common. Now, if in the main For each we place attributes from these two tables, clearly the values that will be taken will be those obtained from each record of the base table that meets the filters. That is to say, they will be the ones coming from this extended table.

But what happens if these attributes are in the nested For each? Are the ones from the extended table of this For each taken?
If by one way or the other we arrived at the same record in this table, it wouldn't matter at all, because they would give the same value by either way. That's what would happen if the Join was done. But there is no Join in this case. Therefore, the values of these attributes obtained in this way will not always be the same as those obtained in this other way.

So which path will be chosen if in the nested For each attributes are placed from here or here? It will be that of the main For each. In fact, if we don't place a base transaction for the nested For each and we let GeneXus calculate it, it will first remove all the attributes of the nested For each that belong to the extended table of the main one. And with those that are left, with those alone it will determine the base table. That is to say, it will remove them, because it assumes that they are reached through the main For each.

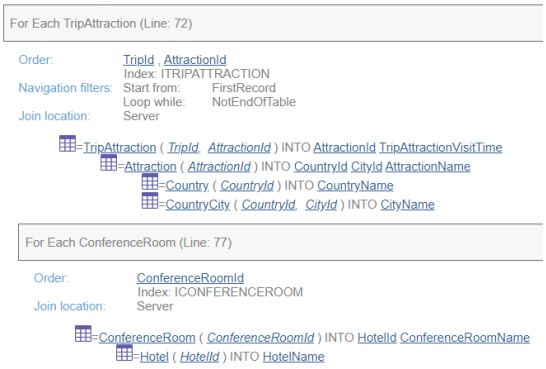
CARTESIAN PRODUCT

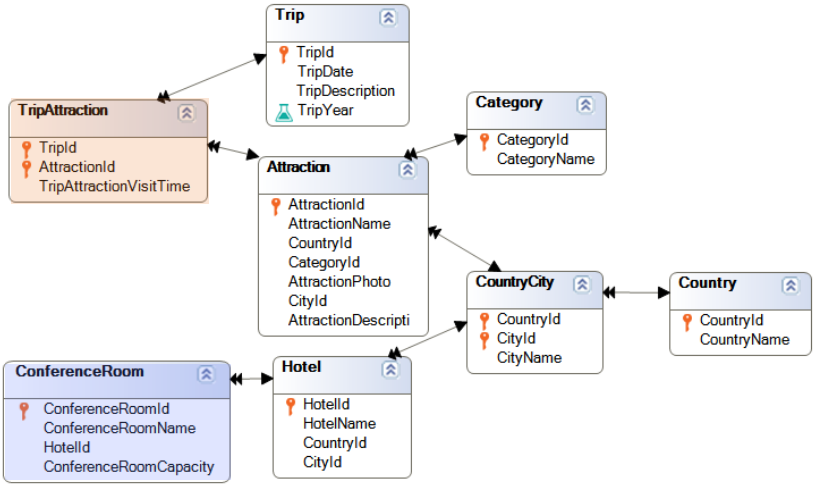



```
for each Trip.Attraction
   print PB1 //AttractionName, TripAttractionVisitTime
   for each ConferenceRoom
      print PB2 //ConferenceRoomName, HotelName, CountryName, CityName
   endfor
endfor
```

For example, if we add a ConferenceRoom table with an N to 1 relationship with Hotel, and specify these nested For each commands, where these are clearly the base tables, we see that in the first For each nothing is requested from the tables in common. In addition, from the base table it would only be necessary to access Attraction to obtain AttractionName.

But if we look at the nested For each, besides an attribute of Hotel, CountryName of Country and CityName of CountryCity are displayed. We might think that it is then going to use the associated ones through ConferenceRoom.
However, if we look at the navigation list....






```
for each Trip.Attraction
   print PB1 //AttractionName, TripAttractionVisitTime
   for each ConferenceRoom
      print PB2 //ConferenceRoomName, HotelName, CountryName, CityName
   endfor
endfor
```

...we see that it doesn't. In the nested For each it only accesses ConferenceRoom to obtain the ConferenceRoomName and the HotelId, through which it accesses this record in Hotel to retrieve HotelName. And there it stays. (Note that there is no Join.)
So where does it retrieve the values of CityName and CountryName from?

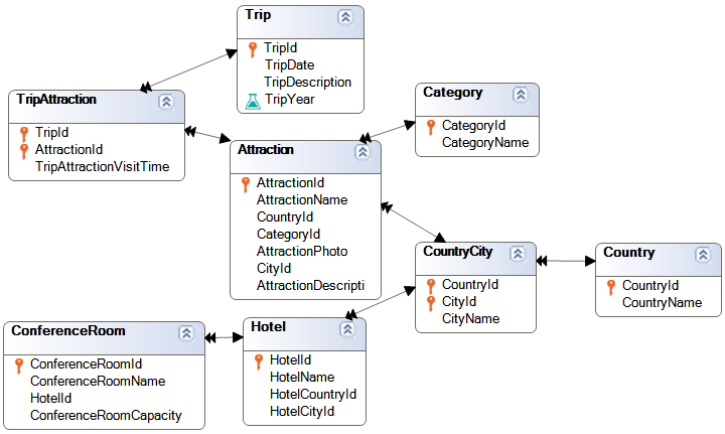
From the main For each. Let's see that it accesses Attraction to retrieve AttractionName, but also CountryId and CityId to be able to access these two tables and retrieve CountryName and CityName, respectively.

And what would we do if we wanted the values of the country and city of the Hotel of the ConferenceRoom?

An initial idea could be through a subtype.

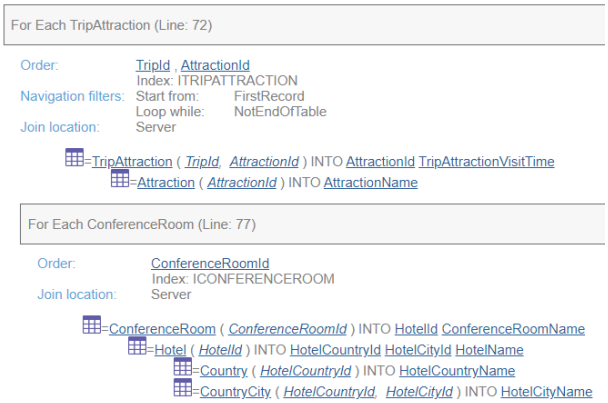
Subtype	Description	Supertype
HotelCountryCity		
HotelCountryId	Hotel Country Id	CountryId
HotelCityId	Hotel City Id	CityId
HotelCountryName	Hotel Country Name	CountryName
HotelCityName	Hotel City Name	CityName

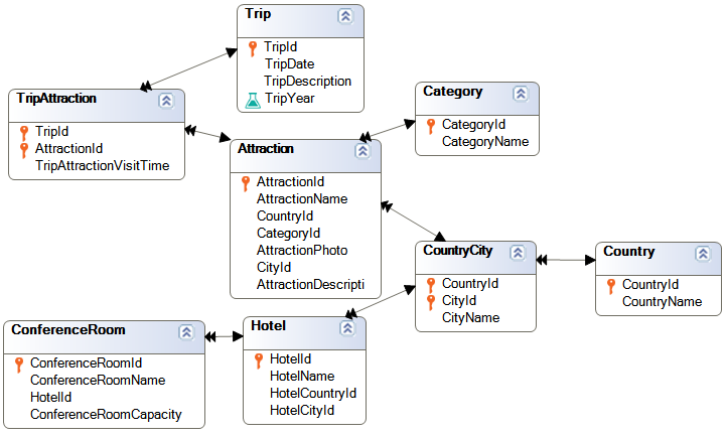
Name
Hotel
HotelId
HotelName
HotelCountryId
HotelCountryName
HotelCityId
HotelCityName




```
for each Trip.Attraction
   print PB1 //AttractionName, TripAttractionVisitTime
   for each ConferenceRoom
      print PB2 //ConferenceRoomName, HotelName, CountryName, CityName  HotelCountryName, HotelCityName
   endfor
endfor
```

For example, if we define this group of subtypes and use it in Hotel instead of the supertypes... We see that now the table has the subtypes, and it will be enough to replace the supertypes with the corresponding subtypes in the nested For Each.

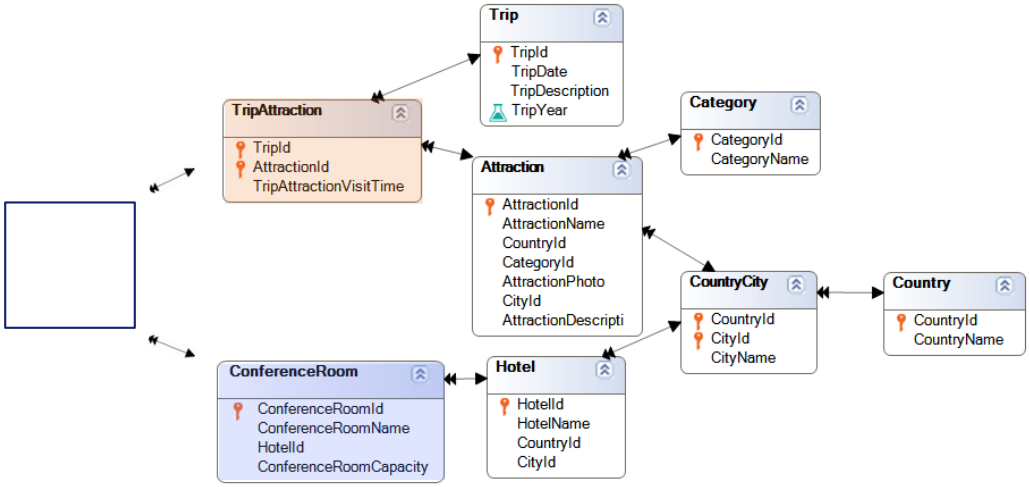




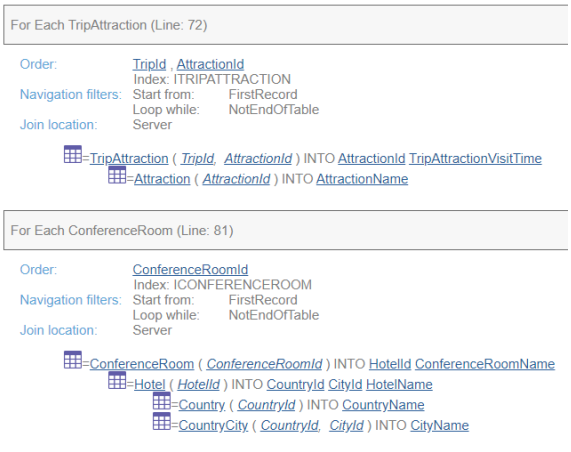

```
for each Trip.Attraction
   print PB1 //AttractionName, TripAttractionVisitTime
   for each ConferenceRoom
      print PB2 //ConferenceRoomName, HotelName, CountryName, CityName  HotelCountryName, HotelCityName
   endfor
endfor
```

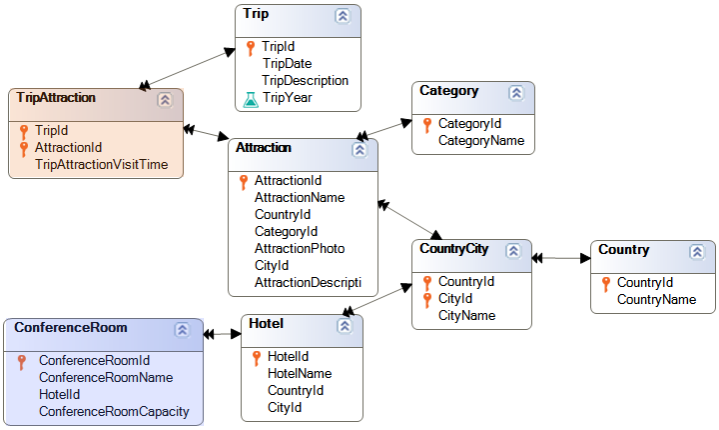
Here is the navigation list with the information we were looking for.

However, it doesn't seem a good idea to place a subtype just because we want to remove the ambiguity in a case of nested For Each commands. Note that here there is no ambiguity in the model.



It would be different if this table existed to introduce two paths to get to these other ones.






```
for each Trip.Attraction
   print PB1 //AttractionName, TripAttractionVisitTime
   Do 'PrintRooms'
endfor

Sub 'PrintRooms'
   for each ConferenceRoom
      print PB2 //ConferenceRoomName, HotelName, CountryName, CityName
   endfor
endsub
```

The smartest way to solve this problem, then, is to leave the model unchanged and write the second For each in a subroutine. Now the navigation list shows exactly what we want.

Here we are accessing CountryName and CityName. From ConferenceRoom, and not from TripAttraction.

JOIN

1-N relationship

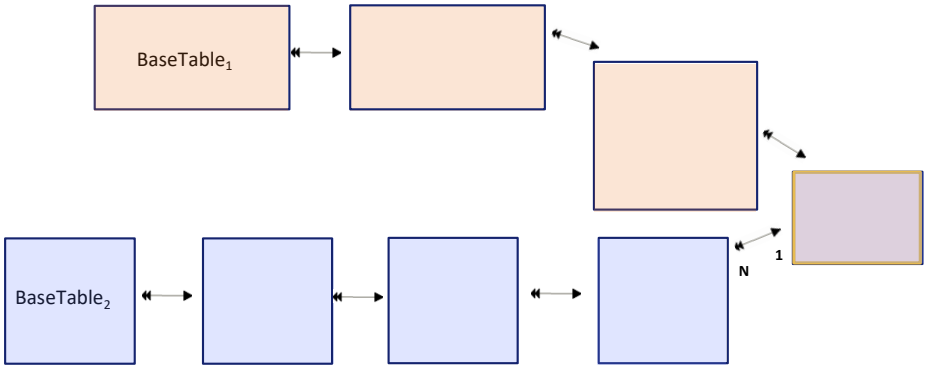
1 $BaseTable_1 \subset ext(BaseTable_2)$

2 $ext(BaseTable_1) \cap BaseTable_2 \neq \phi$

$BaseTable_1 \neq BaseTable_2$

CARTESIAN PRODUCT

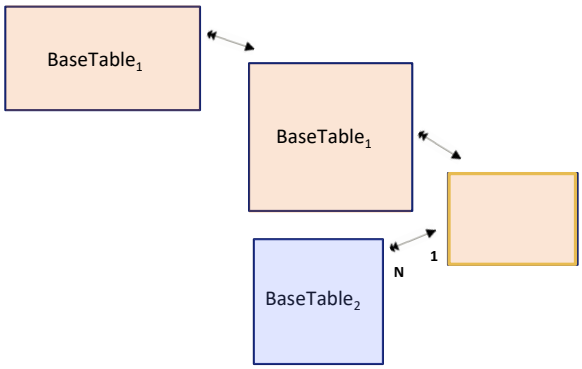
NO 1-N relationship



Before moving on, let's focus again on the second Join case....

1 $BaseTable_1 \subset ext(BaseTable_2)$

2 $ext(BaseTable_1) \cap BaseTable_2 \neq \phi$

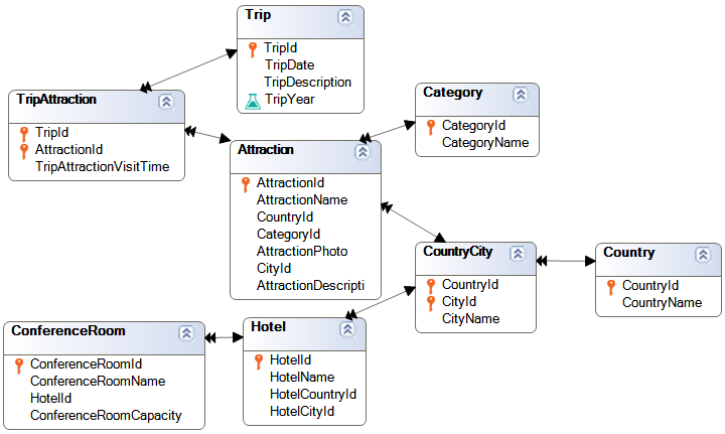


... that we had seen, to examine for a moment what happens if the relationship exists through subtypes. Actually, let's look at the simplest example of all.

What will happen if this relation is not created using supertypes, but subtypes?

Subtype	Description	Supertype
HotelCountryCity		
HotelCountryId	Hotel Country Id	CountryId
HotelCityId	Hotel City Id	CityId
HotelCountryName	Hotel Country Name	CountryName
HotelCityName	Hotel City Name	CityName

Name
Hotel
HotelId
HotelName
HotelCountryId
HotelCountryName
HotelCityId
HotelCityName




```
for each Attraction
   print PB1 //AttractionName
   for each Hotel
      print PB2 //HotelName
   endfor
endfor
```

In the example we saw, if instead of CountryId and CityId, in Hotel we place these subtypes...

If our nested For each commands are written like this: that is, first the Attraction table is run through and for each one the Hotel table is run through, then GeneXus finds the relationship...

For Each Attraction (Line: 72)

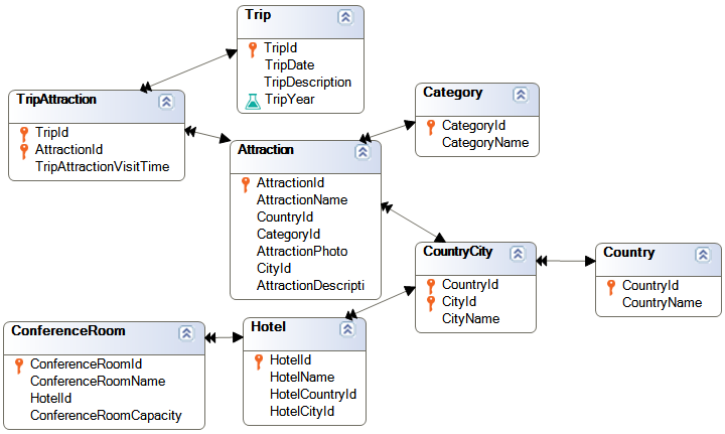
Order: AttractionId
Index: IATTRACTION
Navigation filters: Start from: FirstRecord
Loop while: NotEndOfTable

=Attraction (*AttractionId*) INTO CityId CountryId AttractionName

For Each Hotel (Line: 77)

Order: HotelCountryId , HotelCityId
Index: IHOTEL1
Navigation filters: Start from: HotelCountryId = @CountryId
HotelCityId = @CityId
Loop while: HotelCountryId = @CountryId
HotelCityId = @CityId

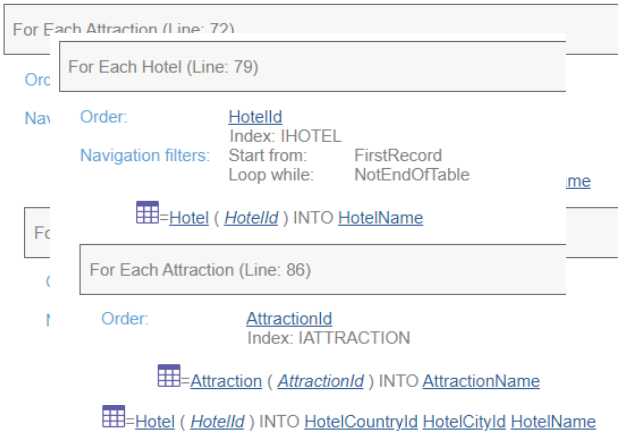
=Hotel (*HotelId*) INTO HotelCountryId HotelCityId HotelName

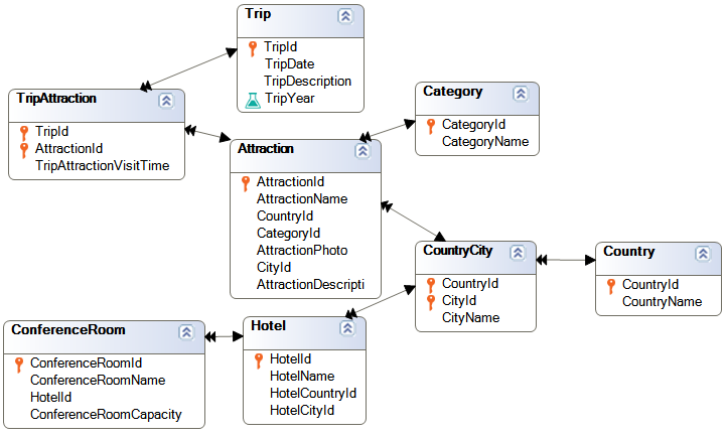



```
for each Attraction
   print PB1 //AttractionName
   for each Hotel
      print PB2 //HotelName
   endfor
endfor
```

... and performs the Join. We see that in the first For each it retrieves the values of CountryId and CityId, and then in the nested For each it filters the Hotel records for which the subtypes match these.

However...






```
for each Attraction
   print PB1 //AttractionName
   for each Hotel
      print PB2 //HotelName
   endfor
endfor
```

```
for each Hotel
   PB2 //HotelName
   for each Attraction
      print print PB1 //AttractionName
   endfor
endfor
```

... if the For each commands are written in reverse order—that is, in the external one the hotels are run through and in the nested one the attractions are run through—a join will not be made, but a Cartesian product.

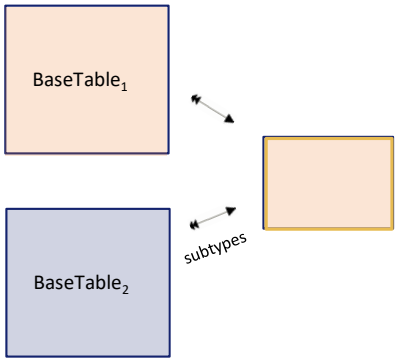
The rule is: a Join is made between supertype and subtype, but not the other way around. That is to say, since in hotel we don't have CountryId and CityId but subtypes of these—particular cases—it is not clear for GeneXus that the developer wants to go from the particular to the general and then only list the attractions with the same values for the supertypes.

JOIN

```
for each
   //SUPERTYPE
   for each
      //SUBTYPE
   endfor
endfor
```

CARTESIAN PRODUCT

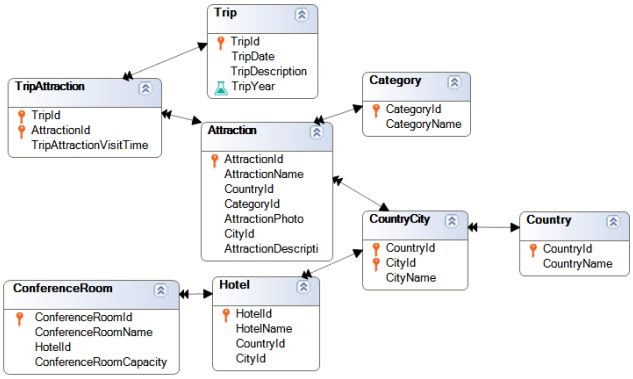
```
for each
   //SUBTYPE
   for each
      //SUPERTYPE
   endfor
endfor
```



Here it is summarized.

JOIN

```
for each TripAttraction
   print PB1 //AttractionName, TripAttractionVisitTime
   for each Hotel
      print PB2 //HotelName, CategoryName
   endfor
endfor
```



$BaseTable_1 \neq BaseTable_2$

CARTESIAN PRODUCT

```
for each TripAttraction
   print PB1 //AttractionName, TripAttractionVisitTime
   for each ConferenceRoom
      print PB2 //ConferenceRoomName, HotelName, CategoryName
   endfor
endfor
```

Finally, (going back to the case without subtypes) before moving on to the case of the same base table, let's add that both in the case of a Join and of a Cartesian Product, it is possible to use in the nested For each attributes of the extended table of the main one that are not in the extended table of the nested one. In the two cases shown in the nested For each, a request is made to print CategoryName which is not in the extended table of its For each. But it is in the extended one of the parent For each.

Therefore, for each tripattraction the values of these two attributes are printed, and the value of CategoryName is obtained, which will be used in the nested For each as the given value. In the case of the Join, the values of CountryId and CityId are also obtained precisely to make the Join.

Then, in the first case, all the Hotels of the same country and city are run through (there the retrieved values of CountryId and CityId are used), and for each one the name of the hotel and the name of the category obtained in the first For each are printed.

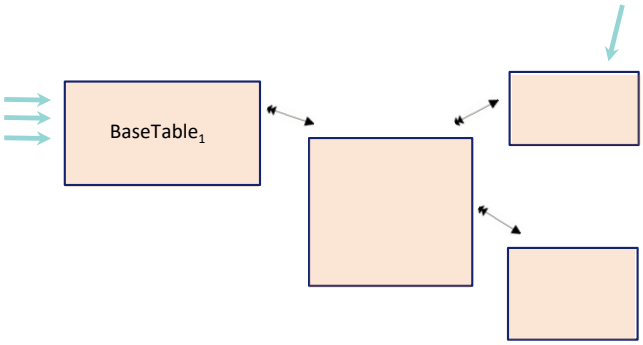
In the second case, all the conferencerooms are run through (without filters because there is no Join) and their name, the name of their hotel and the value of CategoryName from the parent For each attraction are printed.

Remember that to determine the base table of the nested For each (as well as to solve the navigation) the attributes that are in the nested one and are already part of the extended one of the main one are removed first. It is for this reason.

```
for each Trip.Attraction order CategoryName
   print PB1 //CategoryName
   for each Trip.Attraction
      print PB2 //AttractionName, TripAttracionVisitTime, CityName
   endfor
endfor
```

CONTROL BREAK

$BaseTable_1 = BaseTable_2$



Now, let's examine the control break. We know that it takes place when the base table of every For each command is the same and it only makes sense when we want to process grouped information. It can be grouped by any attribute or set of attributes of the extended table.

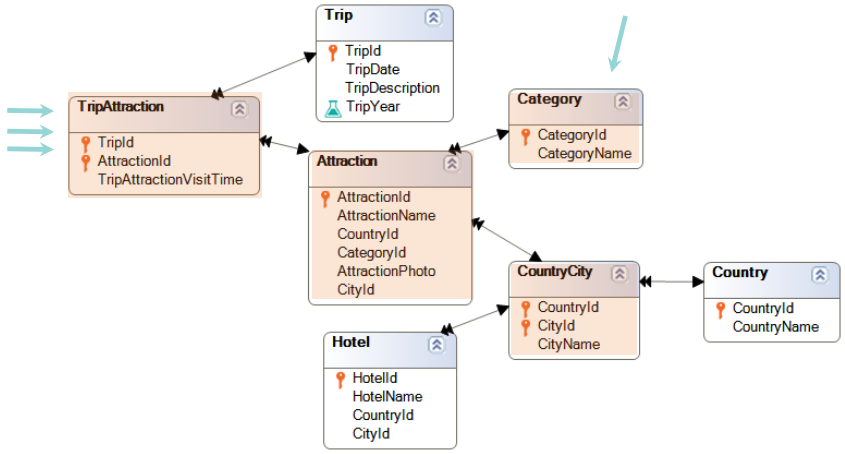
For example, we group according to the value of an attribute of this table, which should appear in the order clause, and we process all the associated records of this table (and the extended table) that have the same value for that attribute.

So, for example...

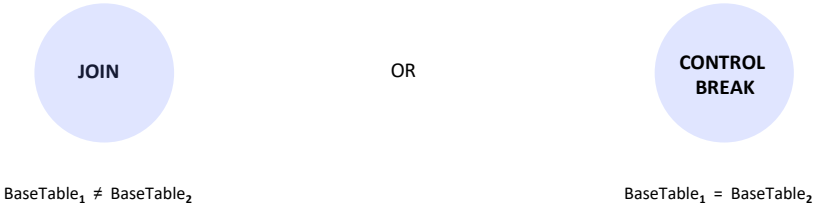
```
for each Trip.Attraction order CategoryName
   print PB1 //CategoryName
   for each Trip.Attraction
      print PB2 //AttractionName, TripAttracionVisitTime, CityName
   endfor
endfor
```

CONTROL BREAK

$BaseTable_1 = BaseTable_2$



We group the tripattractions by category, list for each group the category name, and scroll through the tripattractions of that category, printing the attraction name, visit time and city name of the attraction, for each of the tripattractions with the same category.

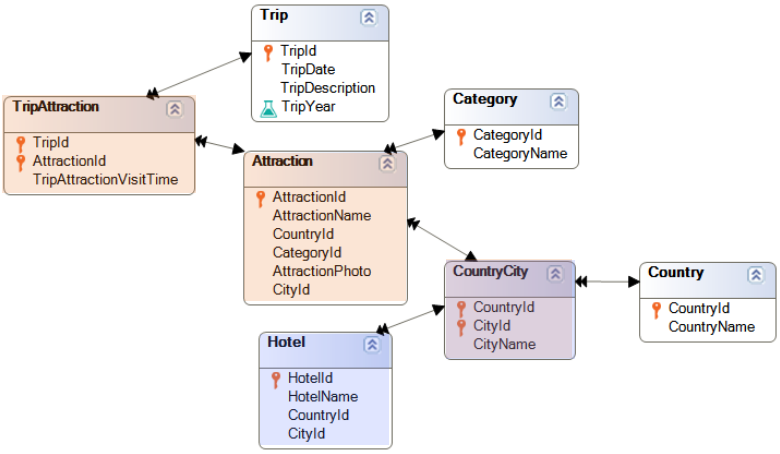


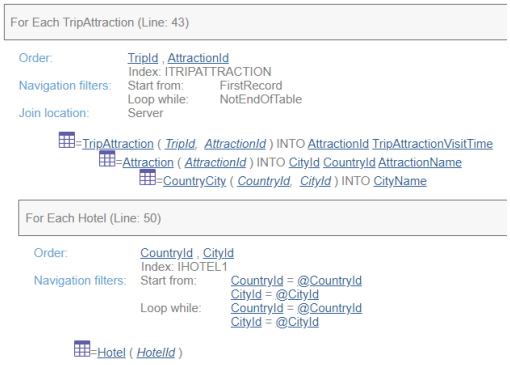
So far we didn't worry about the determination of the base table of every For each command because we took it for granted when using the base transaction. But when we leave this task to GeneXus, what was a Join can become a Control Break.

We will see this by returning to the last Join case that we had analyzed.

2

```
for each Trip.Attraction
   print PB1 //AttractionName, TripAttractionVisitTime
   for each ~~Hotel~~ Country.City
      print PB2 //CityName
   endfor
endfor
```





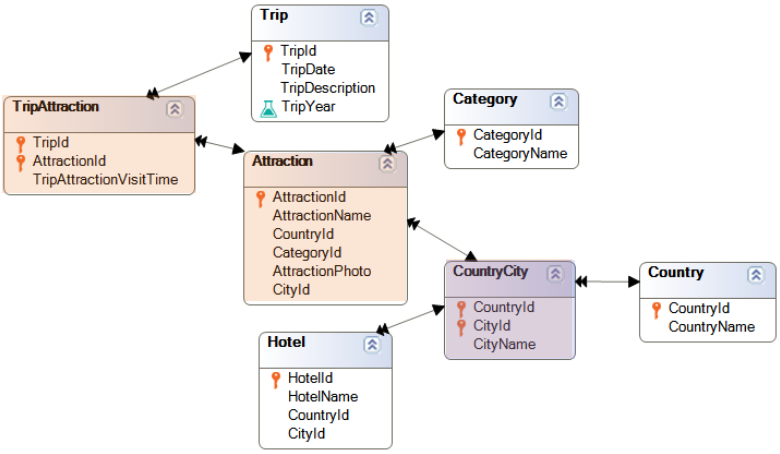
$ext(BaseTable_1) \cap BaseTable_2 \neq \phi$

There is a slight difference: in the nested For each we are printing, instead of the names of the hotels with the same city of the trip attraction, the names of the city of each one of those hotels.

Note that if we had specified Country.City as the base transaction instead of Hotel, then this would be the base table of the nested one, and it would be a particular case where the second For each would only return one record.

2

```
for each Trip.Attraction
   print PB1 //AttractionName, TripAttractionVisitTime
   print PB2 //CityName
endfor
```



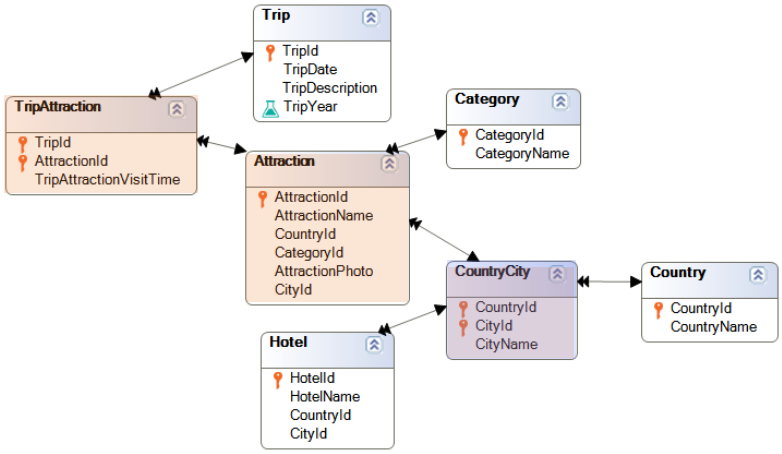
$ext(BaseTable_1) \cap BaseTable_2 \neq \phi$

We see it clearly in the navigation list that indicates **For First** instead of For Each. Because, of course, CountryId, CityId are the primary key of the table.

It's as if the For each had not been specified and the printblock had been printed directly, because CityName is in the extended table of TripAttraction.

2

```
for each Trip.Attraction
   print PB1 //AttractionName, TripAttractionVisitTime
   for each Country.City
      print PB2 //CityName
   endfor
endfor
```



BREAK

$BaseTable_1 = BaseTable_2$

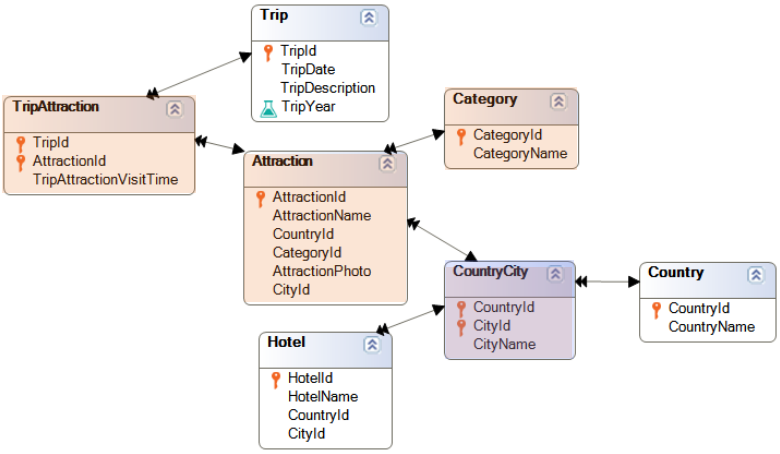
That's why if we don't set this base transaction and we let GeneXus determine the base table by itself, it will choose TripAttraction; that is, it will understand that we want to implement a control break, because assuming that the developer doesn't write unnecessary For each commands, nothing else will make sense.

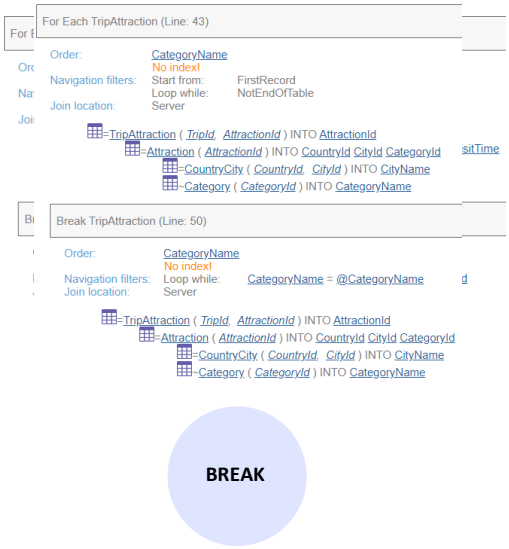
The navigation list will look like this.

However, this will not make any sense either, because it will be a control break by the primary key; that is, it will work in the nested For each with the same record of the main For each every time.

2

```
for each Trip.Attraction order CategoryName
   print PB1 //CategoryName
   for each
      print PB2 //CityName
   endfor
endfor
```

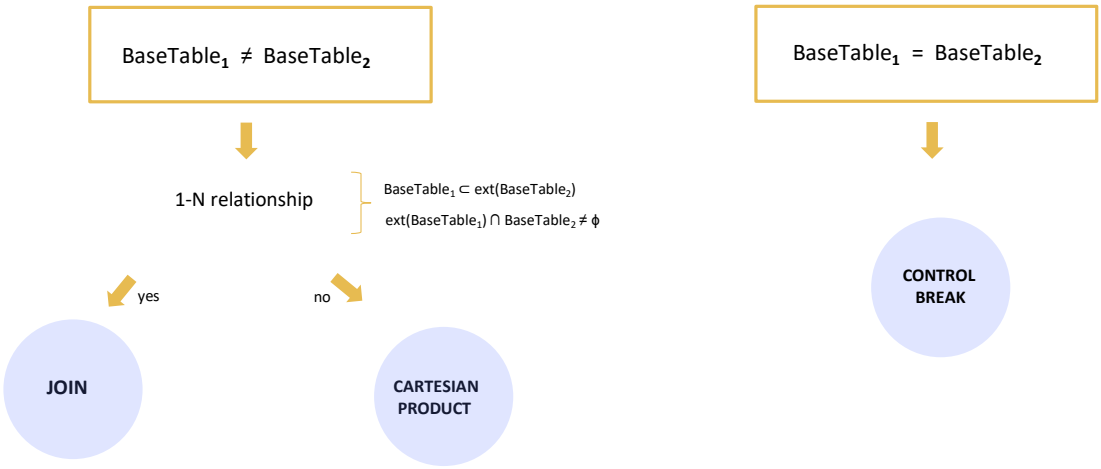




BREAK

$BaseTable_1 = BaseTable_2$

It would still be necessary to specify an order clause to break by some attribute or set of attributes whose values can be repeated. For example, CategoryName. And so we have the same example of control break that we saw before.



In summary, if the base tables are different, GeneXus will look for a 1 to N relationship of one of these two types. If it finds it, it will implement an implicit Join. Otherwise, it will not implement any Join. We call this case a Cartesian product. However, note that calling it a Cartesian product doesn't mean that it is actually one. If the developer explicitly adds a filter, it will clearly bring the filtered records, so there will be a sort of Join, but it will not be the automatic Join. We say the case is a Cartesian product only from the point of view of the automatic filters that GeneXus determines: that is, in this case, none.

The control break is clear.

This is the end of the formal analysis of the three types of possible navigations when there are nested For each commands.

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