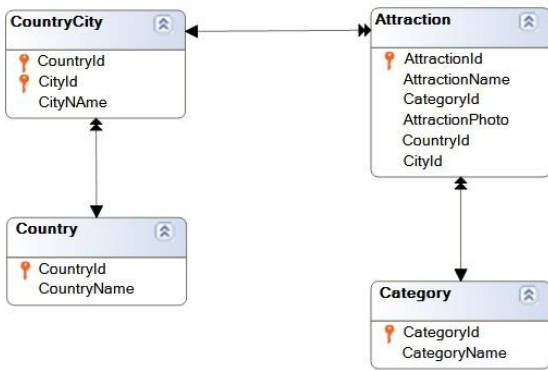


More about For each command

*GeneXus™*

## Review: Base transaction



```
print Title
```

```
for each Attraction
  print Attractions
endfor
```






Remember that GeneXus determines the base table of the For Each command, taking into account the name of the transaction that we declare next to the For Each command, which corresponds to the name of the base transaction; that is, the transaction whose associated physical table we want to run through.

In addition, the attributes declared within the For Each command, whether in printblocks, Where and Order clauses, etc., must belong to the extended table of the For Each command's base table.

In this example here, the base table of the For Each command will be ATTRACTION; that is, the table that will be run through and whose extended table will be accessed in order to retrieve the required data.

## Review: Base transaction

**Procedure AttractionsList Navigation Report**

Name	 AttractionsList	Environment	 Default (C#)
Description	Attractions List	Spec. Version	 15_0_1-106211
Output Devices	File	Form Class	Graphic
Main	Yes	Program Name	AttractionsList2
		Call Protocol	HTTP
		Parameters	



---

**Levels**

**For Each Attraction (Line: 10)**

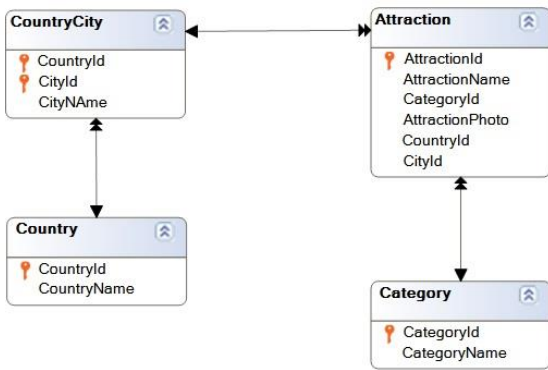
Order: [AttractionId](#)  
 Index: IATTRACTION

Navigation Start from: FirstRecord  
 filters: Loop while: NotEndOfTable  
 Join location: Server

 = [Attraction \(AttractionId\)](#)  
 = [Country \(CountryId\)](#)

The navigation list clearly says that the base table is ATTRACTION, which will be run through according to the primary key of that table—that is, ordered by AttractionId— and that the entire table will be run through, also accessing the COUNTRY table to retrieve the CountryName value, which corresponds to the country of the attraction.

## Review: Base transaction



It is not mandatory to specify a base transaction for a For Each command

```

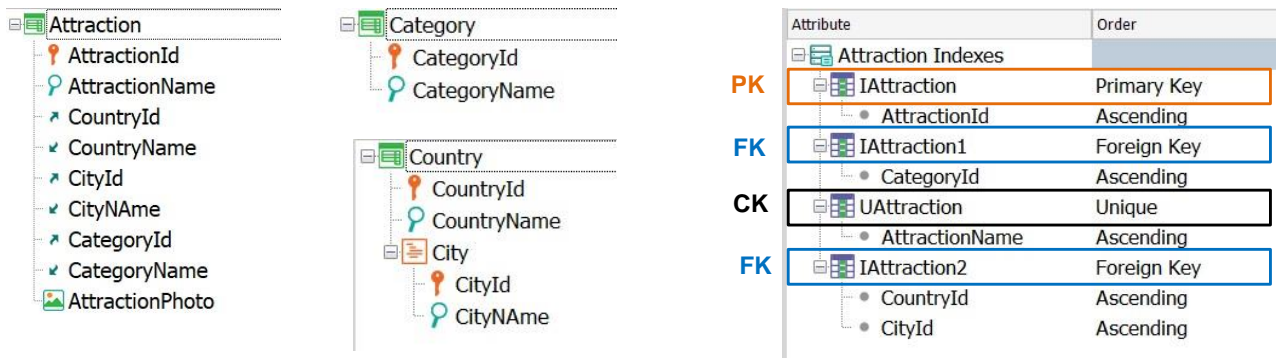
print Title
for each          ?
    print Attractions
endfor
  
```



Is it mandatory to specify a base transaction for a For Each command?

The answer is no. GeneXus can calculate the base table of the For Each command from the attributes included in the command. The way to find the base table will not be seen in this course.

## Indexes and their relationship with database queries



Let's now move on to the indexes and their relationship with the database queries.

We already know that **indexes** are efficient ways to access data.

We have already seen that, in each table, GeneXus creates an index by the primary attribute (either a simple or compound key) and an index by each foreign key. This is done to make data consistency controls between tables more efficient.

Also, that it is possible to define indexes, indicating whether they accept duplicate values or not. If we define an index that doesn't accept duplicate values –that is, a Unique index– we are telling GeneXus that it must automatically control the uniqueness of its value; that attribute, or set of attributes over which the index is defined, becomes a candidate key.

## Indexes and their relationship with database queries

```

Attraction
├── AttractionId
├── AttractionName
├── CountryId
├── CountryName
├── CityId
├── CityName
├── CategoryId
├── CategoryName
└── AttractionPhoto
    
```

```

Category
├── CategoryId
└── CategoryName
    
```

```

Country
├── CountryId
├── CountryName
└── City
    ├── CityId
    └── CityName
    
```

```

print Title
for each Attraction order AttractionName
  print Attractions
endfor
    
```

**Warnings**

spc0038 There is no index for order AttractionName; poor performance may be noticed in group starting at line 3.

**Levels**

For Each Attraction (Line: 10)

Order: AttractionName  
! No index

Navigation Start from: FirstRecord

filters: Loop NotEndOfTable

while:

Join location: Server

Attraction (AttractionId)

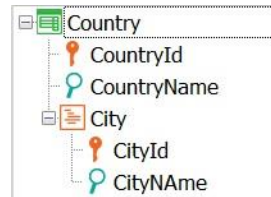
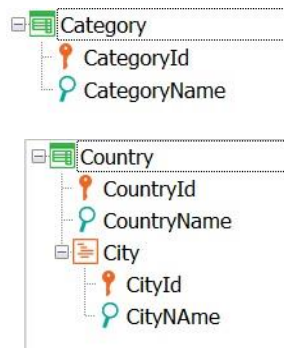
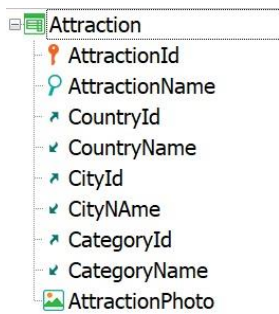
Country (CountryId)

The database has no index by AttractionName

If we add an Order clause, for example, to order by the name of the attraction, the navigation list gives us a warning, informing us that the database has no **index** by the attribute by which we need to order the information, so this query could have low performance.

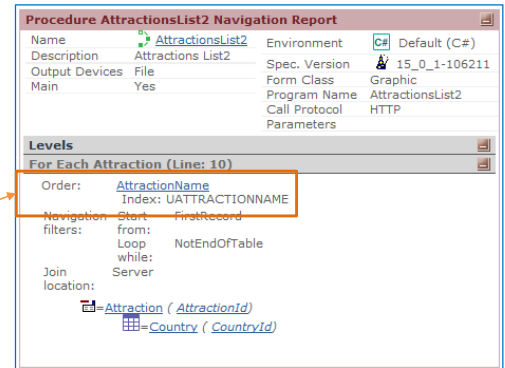
When we give GeneXus an attribute by which to order data, it tries to order it in an efficient way; therefore, it looks for an index by that attribute. But since it can't find it, it informs us about it.

## Indexes and their relationship with database queries



```
print Title
```

```
for each Attraction order AttractionName
print Attractions
endfor
```



The database has an index by  
AttractionName

If we need to obtain the records of ATTRACTION ordered by the AttractionName attribute, these records will have to be reordered because by default they are ordered by the value of the attribute that is the primary key.




When a query is defined, if there is a physical index created in the table for the attribute to order by, GeneXus will use it. In this case, the query has to be ordered by a secondary attribute: AttractionName. GeneXus warns us in the navigation list, as we have seen, that an index hasn't been created.

The existence of an index would optimize the query. However, the disadvantage of creating an index is that it must be maintained. That is, as users add, modify and delete attractions from the ATTRACTION table, this index must be rearranged.

Once we've done this, the database will be reorganized by pressing F5 to create this new index. Then, in the navigation list, we will see that GeneXus will use that index that has just been created.

It is worth mentioning that just as we create it, at any time we can delete an index, and by pressing F5 and reorganizing, we will return to the previous status.

## Indexes and their relationship with database queries: Example

Attractions List		
	Colosseum	Italy
	Eiffel Tower	France
	Louvre Museum	France

```
Parm (in:&NameFrom, in:&NameTo);
```

```
print Title
```

```
for each Attraction order AttractionName
  {
  Where AttractionName >= &NameFrom
  Where AttractionName <= &NameTo
  }
  print Attractions
endfor
```

**Where AttractionName >= &NameFrom and AttractionName <= &NameTo**

Let's see this example:

Suppose we want to get a list of attractions whose names are in alphabetical order between a couple of values received by parameter. For example, between the letters "B" and "N."

That's why we specify the Where clauses we are looking at.

Having several Where clauses is the equivalent to having only one, where the conditions are combined with the "and" logical operator. In other words, only records that meet all the conditions at once will be considered.

If we're going to filter by AttractionName, and we have an index created by that attribute, we should always **order by AttractionName** to optimize the query.

Note that if we don't enter the Order clause, GeneXus will order by primary key, and the entire table will have to be run through to know if an attraction is within the specified range or not.



## When clause

```

print Title
for each Attraction order AttractionName
  Where AttractionName >= &NameFrom when not &NameFrom.IsEmpty()
  Where AttractionName <= &NameTo when not &NameTo.IsEmpty()
  print Attractions
endfor

```

What result will be obtained for the For Each command we are looking at, if the &NameFrom and &NameTo variables are empty?

If there were an attraction with an empty name, then it would be the only one returned, since it would be the only one that would meet the conditions. Otherwise, no attractions would be listed.

Is it possible, then, to consider the orderings and filters, so that they are only applied in certain circumstances? For example, to only apply the first Where clause **when** the &NameFrom variable is not empty? And to only apply the second Where clause **when** the &NameFrom variable is not empty?

The answer is yes. We achieve this by conditioning the Where clauses with **when**. Each Where clause will only be applied when the When condition is met.

In this way, at runtime, when we leave both variables empty, none of the Where clauses will be applied, so all the attractions of the table will be listed. If the &NameFrom variable is empty but &NameTo is not, the first Where clause will not be applied but the second one will be, so all attractions whose name is lower than or equal to &NameTo will be listed.

In the same way, you can set conditions for applying an order or not. In fact, a series of conditional orders can be specified, in order to choose the first one whose condition is met.

## When none clause

```

print Title

for each Attraction order AttractionName
  Where AttractionName >= &NameFrom when not &NameFrom.IsEmpty()
  Where AttractionName <= &NameTo when not &NameTo.isempty()
  print Attractions
  When none
    Print NoAttractions
endfor

```

Title		
<b>Attractions List</b>		
AttractionP	AttractionName	CountryName
NoAttractions		
<i>No attractions registered</i>		

Let's now move on to the **When none** clause.

What happens when none of the records in the base table meets the conditions indicated?

Let's suppose that in this case we want to print a warning message on the output... saying that there are no associated records.

To this end, we will program the **When none** clause.

All commands written between when none and endfor will be executed sequentially and **only when no records from the base table of the For Each command have been found that meet the conditions indicated**

In this example, we have decided to print a message, but we may also type a series of commands, such as another For Each command, for example.

Since what will be executed after the **When none** clause will imply that the search was unsuccessful, if we type a For Each command there, the When none clause will not be nested. It will be like a standalone For Each command.

## Summary

```

For each BaseTransaction
  order att1, att2, ... , attn [when condition]
  order att1, att2, ... , attn [when condition]
  where condition [when condition]
  where condition [when condition]

    main code

When none
  .....
```

**endfor**

In summary...

As we have seen, the **base table** of a For Each command is determined from the specified base transaction; the rest of the attributes mentioned, both in the body of the For Each command (main code) and in the Order and Where clauses, must belong to the extended table of that base table.

The attributes mentioned in the When none block will be not taken into account.

We gray out everything we've seen before. Here, we have added the **When** and **When none** clauses.

Later on, we will see that more clauses can be added to this essential command to access the database.

## A case study

```

Customer
{
  CustomerId*
  CustomerName
}

Trip
{
  TripId*
  TripDescription
  TouristGuideId
  TouristGuideName
}

TouristGuide
{
  TouristGuideId*
  TouristGuideName
  Phone
  {
    TouristGuidePhoneId*
    TouristGuidePhoneNumber
  }
}

Reservation
{
  ReservationId*
  ReservationDate
  CustomerId
  CustomerName
  Trip
  {
    TripId*
    TripDescription
  }
}

```

Finally, let's examine a case study:

Let's consider the following transaction design:

The Customer transaction, the Trip transaction corresponding to the trips with a tour guide in charge, the Tourist Guide transaction with its set of phone numbers, and the Reservation transaction to record, for each customer, the set of trips that he or she has booked.

We need to obtain a list showing, for a given customer, and from a given date, all the trips he has booked, and for each of them the contact phone numbers of the tour guide in charge.

To solve it, we propose the following source:

## A case study

out Rules \* Conditions Variables

```
Parm(in:&ReservationDate, in: CustomerId);
```

```
For each Reservation.Trip
  Where ReservationDate >= &ReservationDate
  Print Trips
  For each TouristGuide.Phone
    Print TouristGuidesPhones
  Endfor
Endfor
```

```
Trip
{
  TripId*
  TripDescription
  TouristGuideId
  TouristGuideName
}
```

```
TouristGuide
{
  TouristGuideId*
  TouristGuideName
  Phone
  {
    TouristGuidePhoneId*
    TouristGuidePhoneNumber
  }
}
```

```
Reservation
{
  ReservationId*
  ReservationDate
  CustomerId
  CustomerName
  Trip
  {
    TripId*
    TripDescription
  }
}
```

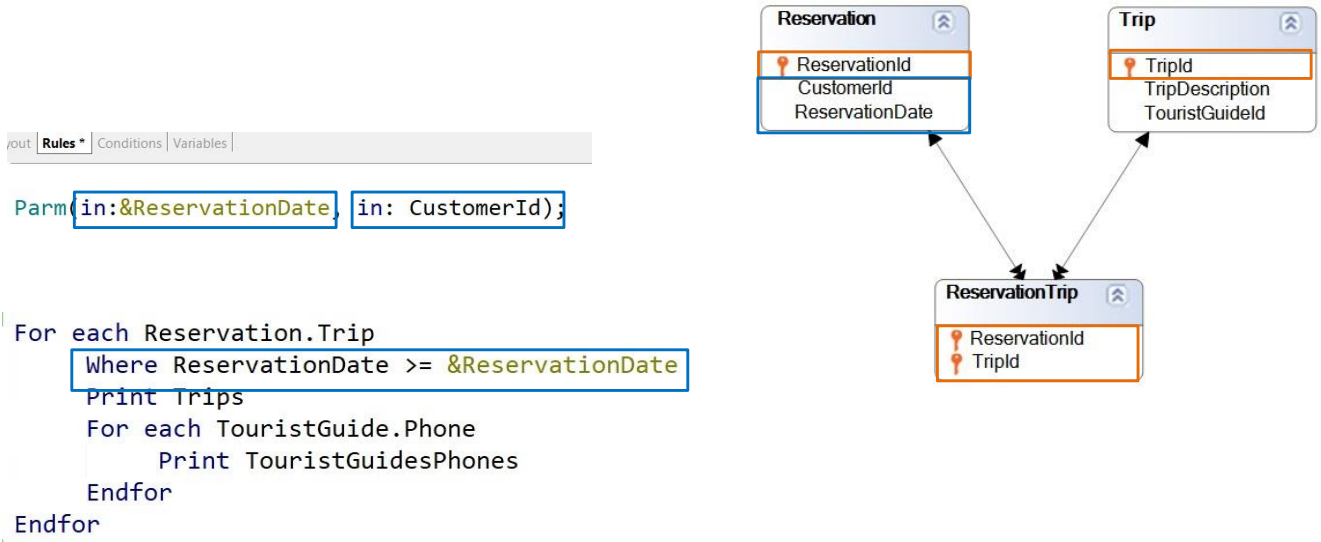
To solve it, we propose the following source:

Let's examine if the information listed is the one we are asked for. There is a couple of nested For Each commands. In the first one, we explicitly say that the base table will be the one corresponding to the Trip level of the Reservation transaction; that is to say, the one called ReservationTrip.

We confirm that within that external For Each command no attribute is being used that doesn't belong to the extended table of RESERVATIONTRIP. If so, the navigation list will show a warning that this attribute is not accessible.

The attributes that we must check are those found in the Where clause and within the printblock named Trips, which in this case are the TripDescription attributes, included in TRIP, and ReservationDate, included in RESERVATION.

## A case study



In the table diagram:

We can clearly see that from RESERVATIONTRIP we access a single record of the TRIP table and a single record of RESERVATION. Therefore, GeneXus must access those two tables every time it iterates in the For Each command.

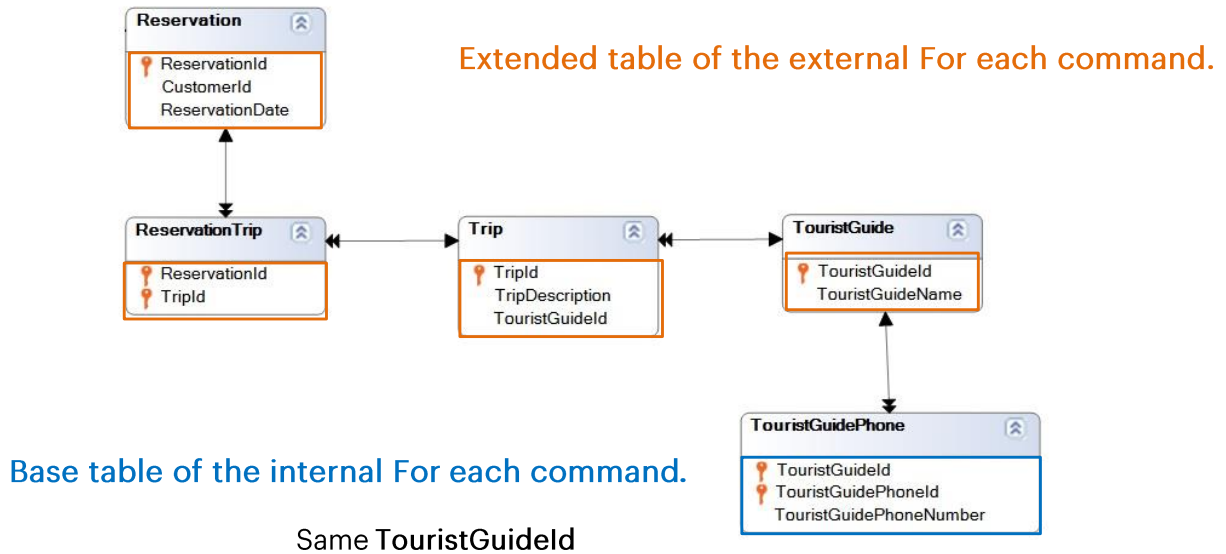
So, which records of the base table will the For Each command work with? With those that comply with the following: when going to the RESERVATION table to evaluate the ReservationDate value, it is greater than or equal to the value of the &ReservationDate variable received in a parameter. Also, the CustomerId value must also match the value directly received through a parameter in that attribute. Remember that to “receive in an attribute” is to determine that this attribute will be instantiated. In other words, whenever that attribute is used anywhere, it will be used with the value received when the object was invoked.

Since the For Each command being examined already accesses the RESERVATION table containing it, an automatic filter will be applied for that value of CustomerId.

It is important to clarify the following: The fact that the attribute received in the Parm rule belongs to the extended table of the For Each command **DOES NOT** make it automatically apply as a filter. For this to happen, the For Each command must access that particular table of the extended table to perform an action.

## A case study

## Indirect 1-N relationship



Think about what happens with the nested For Each command. Its base table will clearly be the one associated with the Phone level of the TouristGuide transaction. The following question is: Does it establish implicit filters for the information it will use? Yes, it will show the phones of each tour guide.

Why? GeneXus looks for a relationship between the extended table of the external For Each command and the base table of the nested For Each command:

It's another way of looking for a 1 to N relationship, although in this case it is an indirect one. If each RESERVATIONTRIP has a TouristGuideld, and in the table to be navigated there is also a TouristGuideld, then GeneXus understands that they will be the same due to the relationship between them. That's why it will make a Join.

In the next few videos, we will continue to learn more about the For Each command.

# GeneXus™

[training.genexus.com](http://training.genexus.com)  
[wiki.genexus.com](http://wiki.genexus.com)  
[training.genexus.com/certifications](http://training.genexus.com/certifications)