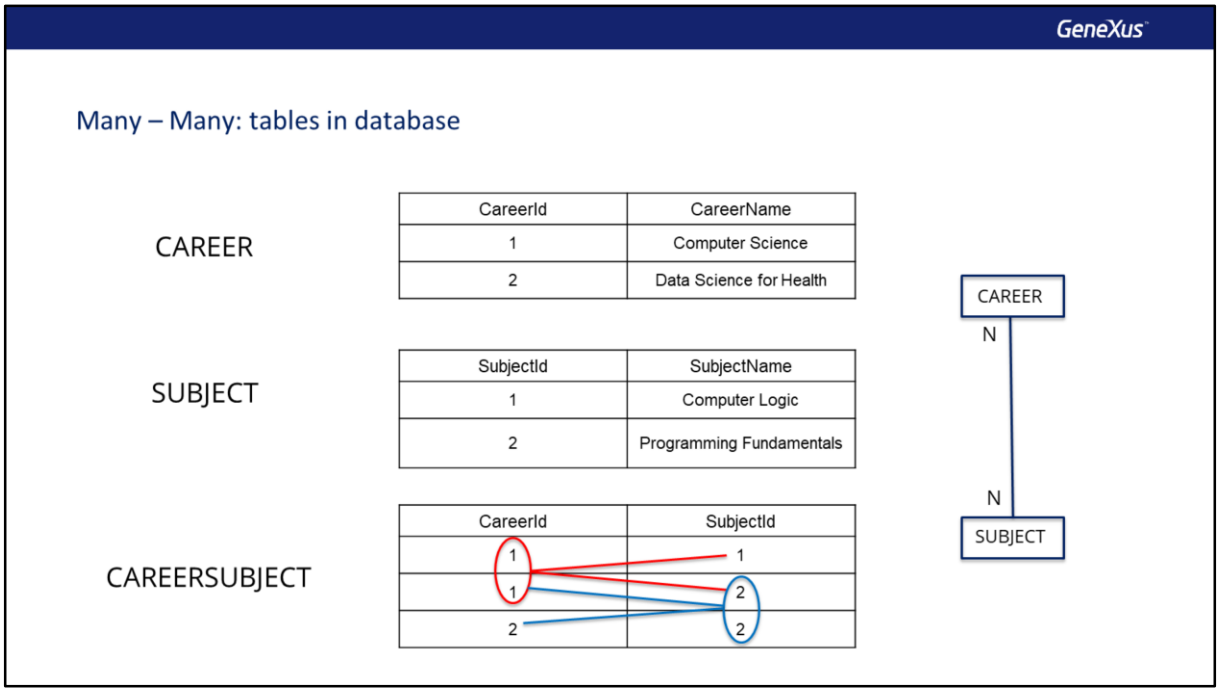
Transaction design and normalization

An encompassing look

GeneXus™

In this video, we will try to analyze different topics related to transaction design and how the decisions we make are reflected in the database structures created, or in the application's functionality.

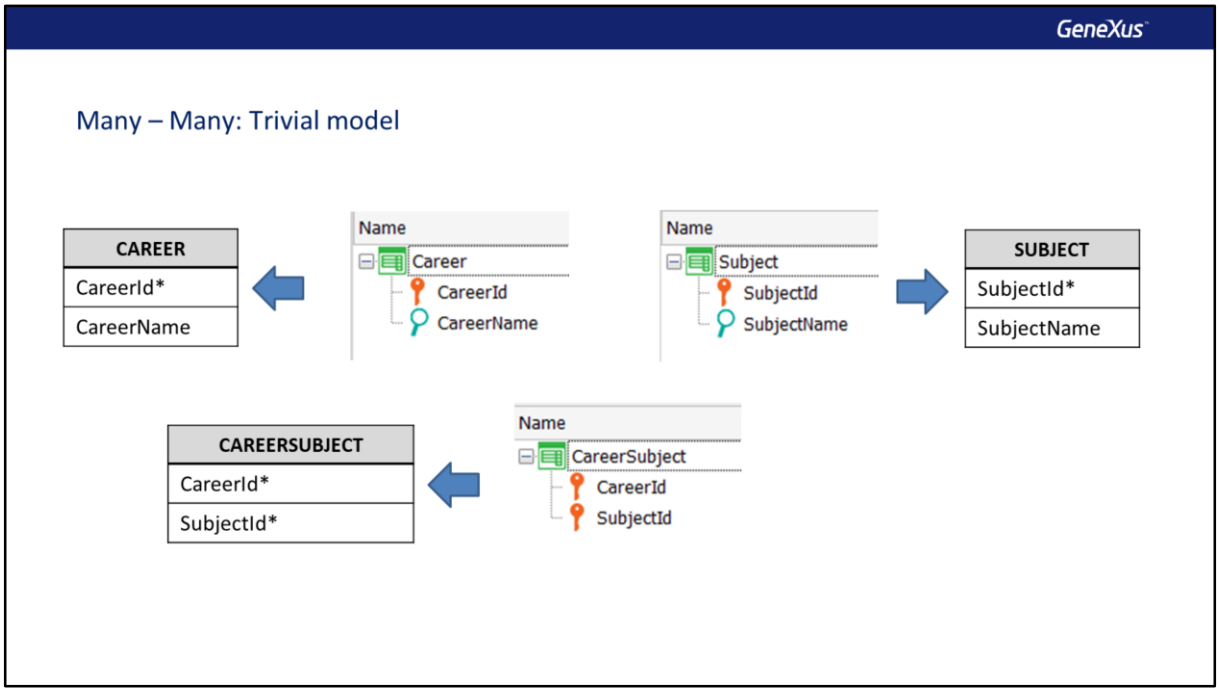
For this reason, unlike previous videos on this topic where an example was developed, in this one we will address specific cases that allow us to analyze different practical situations that can be useful in building our solution.



To represent a many-to-many relationship between two entities in a relational database, three tables are used; one for each entity and a third one (also called relationship table) that contains the identifiers of the previous tables to form a compound key.

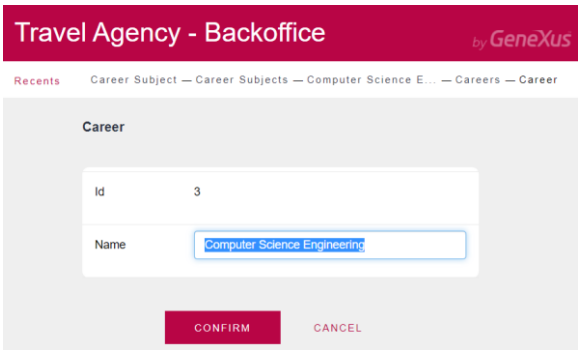
For example, let's consider the reality of a university, where each degree program has many courses and each course can be included in many degree programs. To represent this relationship, we will have a DegreeProgram table, a Course table, and a DegreeProgramCourse table whose keys are those of the previous tables forming a compound key.

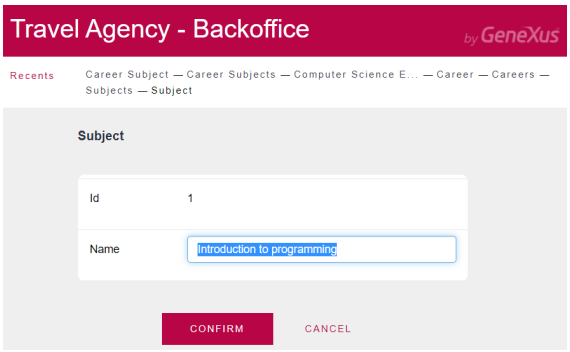
If we analyze the data, we can see that the degree program 1 contains courses 1 and 2, but that in turn course 2 is in degree program 1 and in degree program 2, so this model effectively allows us to represent a many-to-many relationship between degree programs and courses.

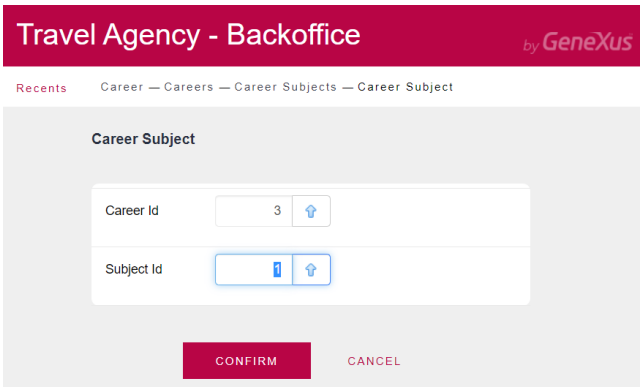


In GeneXus, we use transactions to model reality, not tables. A trivial solution to model the relationship between degree programs and courses so that GeneXus generates the three tables we need, is to create transactions with the same structure as the tables.

Since all three transactions are flat, that is, there are no sublevels, the tables will be created as expected, so we can be sure that this model does indeed represent a many-to-many relationship between degree programs and courses.

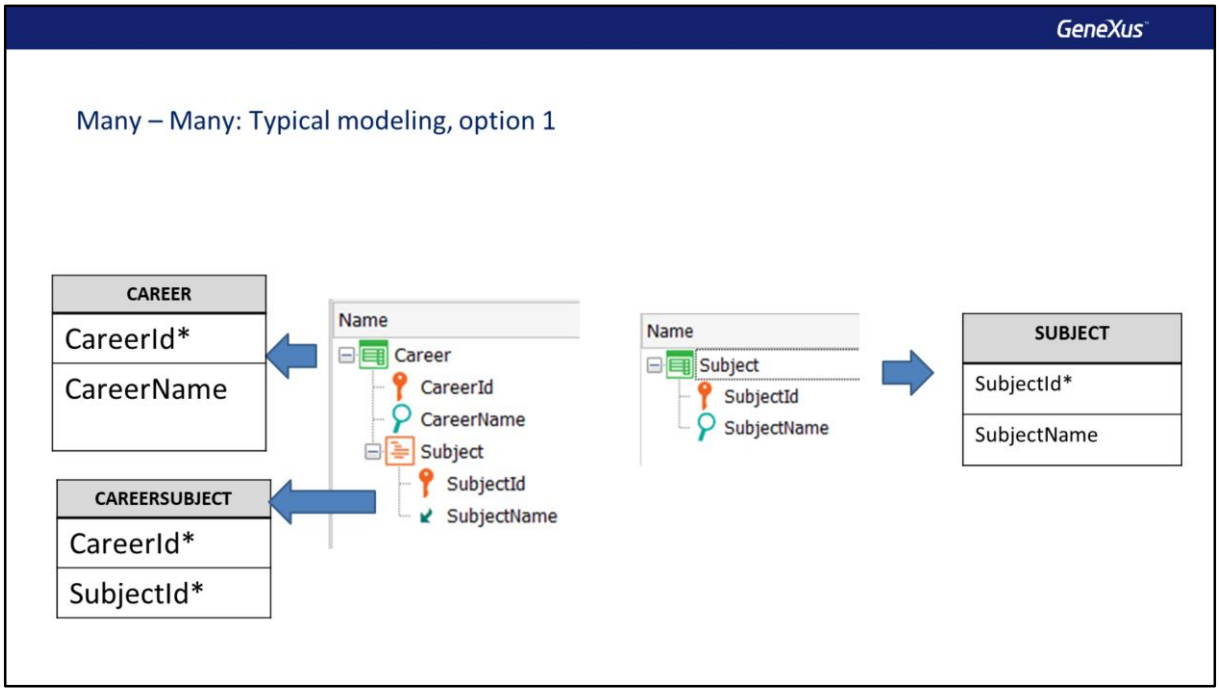






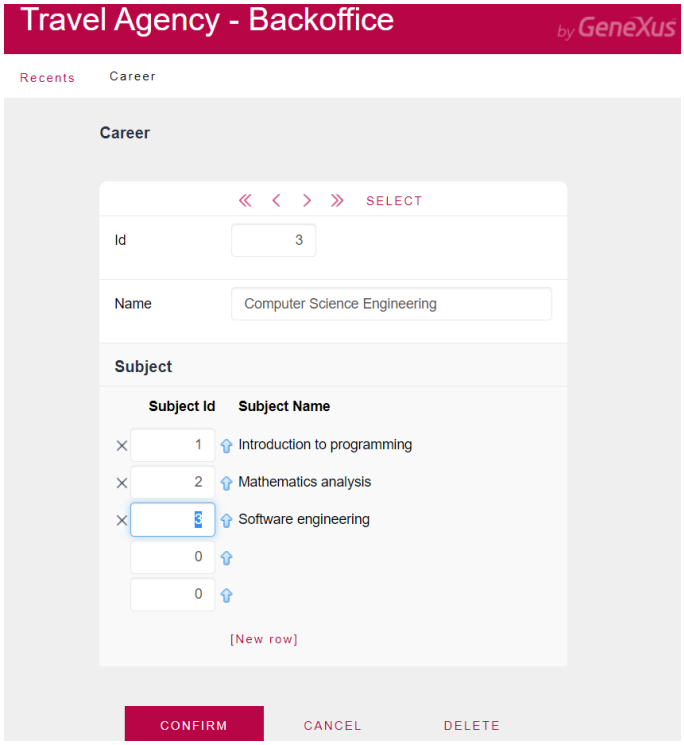
However, if we consider the user interface at the time of data entry, we would have 3 screens. The degree programs and courses would be entered smoothly, but then it is very inconvenient to have to enter the pairs of degree program and course identifiers to establish which degree programs correspond to which course and vice versa.

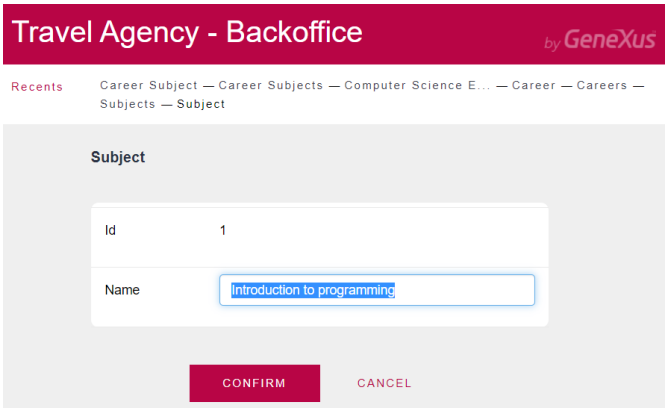
Therefore, although this design complies with modeling the many-to-many relationship, it would not be advisable. It is preferable to be able to enter the data of a degree program in one screen and then in a grid all the courses of that degree program, or vice versa...



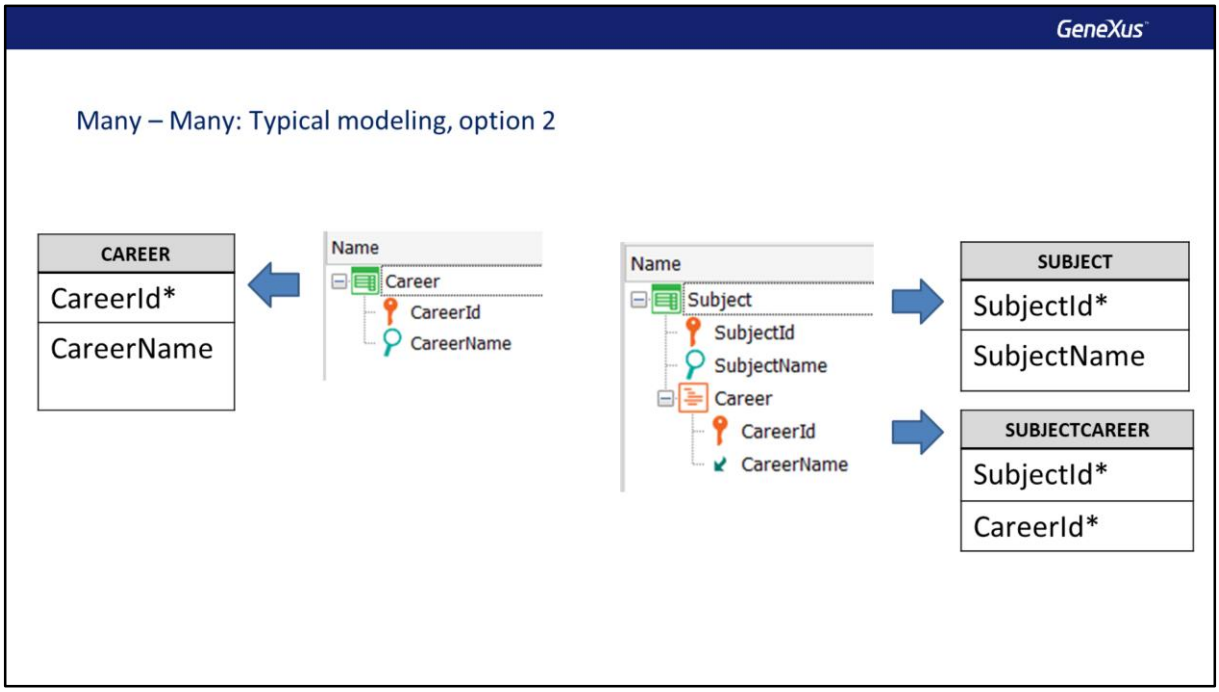
To obtain a screen where all its courses can be entered for a degree program, we can create a two-level DegreeProgram transaction, where the nested level is made up of the courses. With this transaction alone we would model a 1-to-many relationship between degree programs and courses; to make it a many-to-many relationship, we add a second Course transaction.

If we look at the tables created by GeneXus, we see that we are indeed modeling a many-to-many relationship between degree programs and courses, since we get the same three tables we saw before.





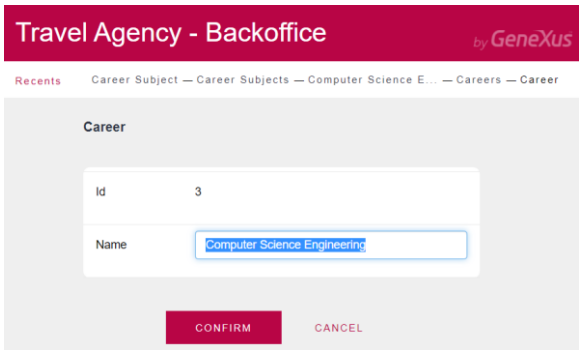
At runtime, we see that our model prioritized entering the courses, and then loading the courses of each degree program.

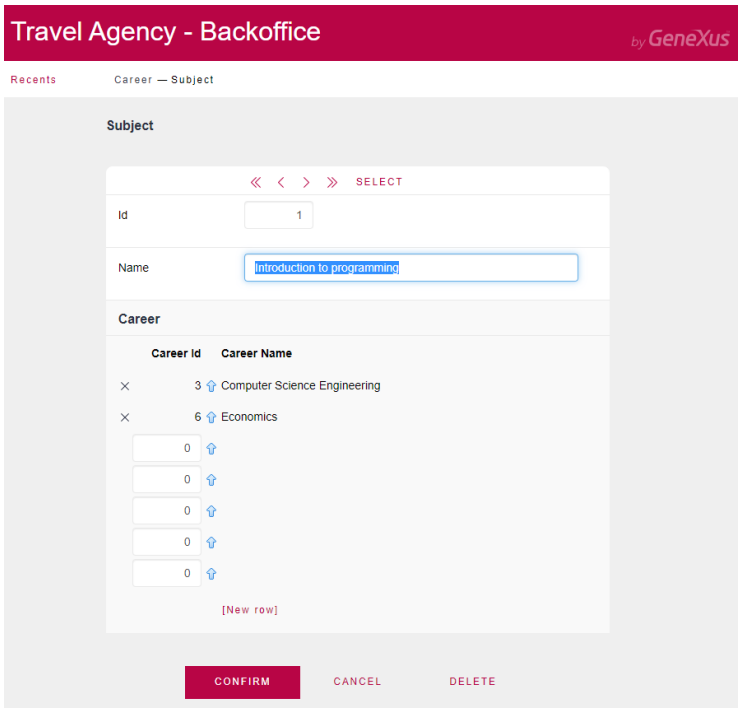


If, in the same case of the many-to-many relationship between degree programs and courses, we had been asked for a screen to enter the degree programs to which each course belongs, we would create a two-level Course transaction where the second level corresponds to the degree programs.

We would obviously also create a DegreeProgram transaction so that the many-to-many relationship between the entities is maintained.

If we look at the tables that will be created, we can see that they are exactly the same as in the previous examples, so we are sure that we are modeling a many-to-many relationship between courses and degree programs.

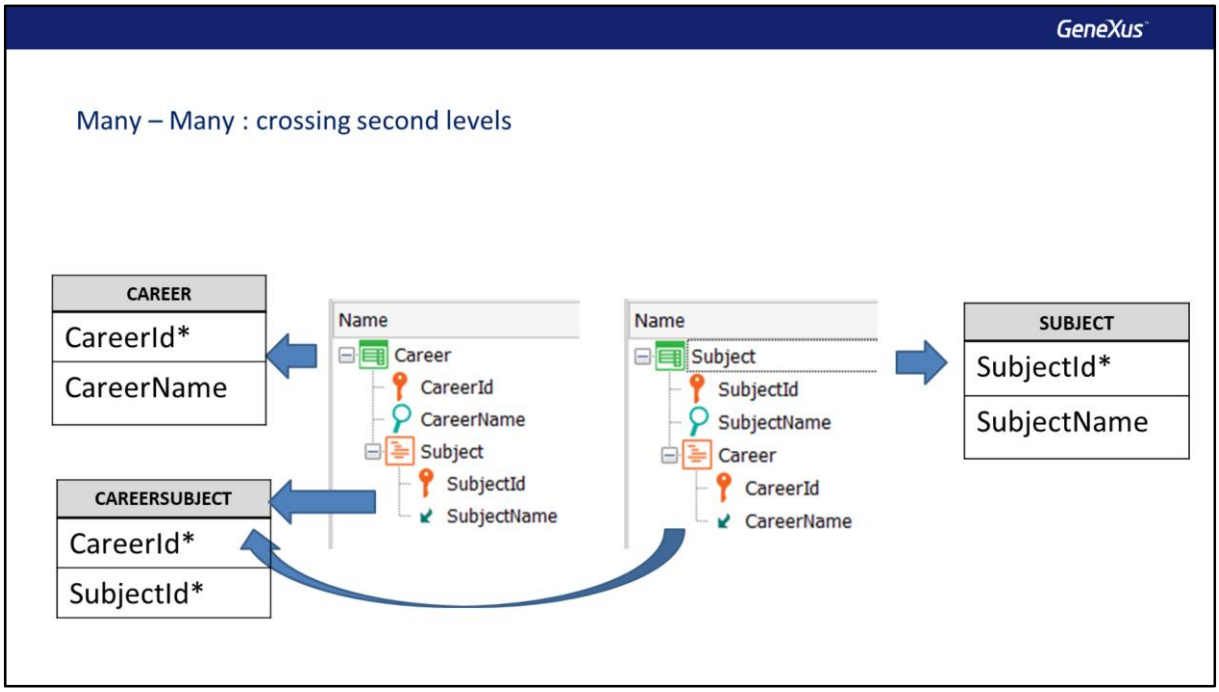




At runtime, we confirm that now the approach is that we first enter the degree programs and then, for each course, we enter the degree programs to which it belongs.

So to model a many-to-many relationship, two transactions are enough, one with two levels and for the entity we place in the second level, we also create a separate transaction.

From the point of view of the many-to-many relationship, it doesn't matter which entity we place in the second level, it is only relevant at the time of data entry.



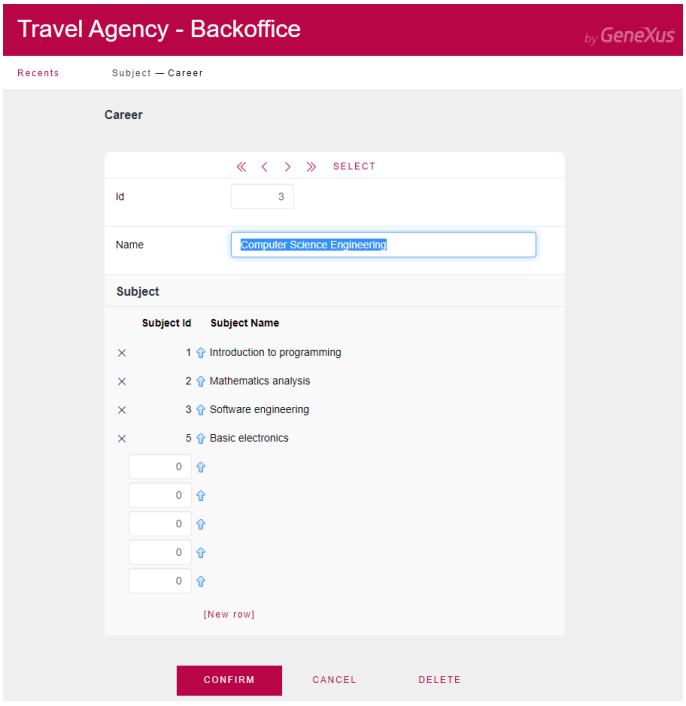
What will happen if we are asked for a screen to enter a degree program and for that degree program all the courses it contains, and at the same time another screen to enter a course and all the degree programs to which it belongs?
Following the previous reasoning, we should create two two-level transactions, one called DegreeProgram with a second level called Course, and the other called Course with a second level called DegreeProgram.

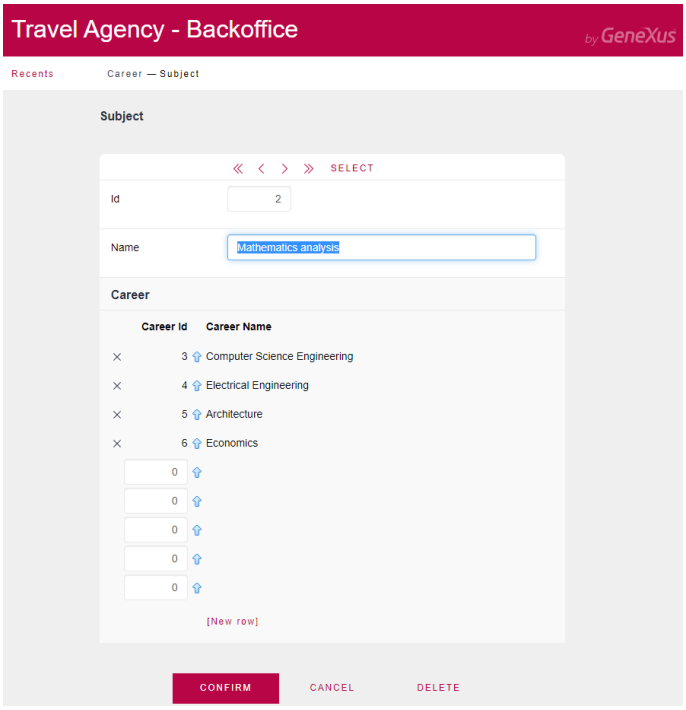
But what relationship would we be modeling in this case?
To find out, we write the tables that GeneXus will create. We know that from the DegreeProgram transaction a DEGREEPROGRAM table will be created with the structure of the first level of the transaction. Also, that from the second level a DEGREEPROGRAMCOURSE table will be created, and that because the CourseName attribute is inferred, the table will contain only the identifier attributes of the first and second level, forming a compound key.

If we analyze the tables that will be created from the two-level Course transaction, we see that from the first level a COURSE table will be created with the same structure as the header of the Course transaction; from the second level, a table will be created that we assume will be called COURSEDEGREEPROGRAM, containing only a key composed of CourseId and DegreeProgramId, since the DegreeProgramName attribute is inferred.

But GeneXus has already created a table with exactly that same structure, the DEGREEPROGRAMCOURSE table, so it doesn't create another one. The final result of this model are the same three tables that we obtained before and that we know correspond to a many-to-many relationship between degree programs and courses.

Therefore, if we create two two-level transactions and cross the entities—that is to say, in one we place as the second level what in the other is the first level and vice versa—we will be modeling a many-to-many relationship.



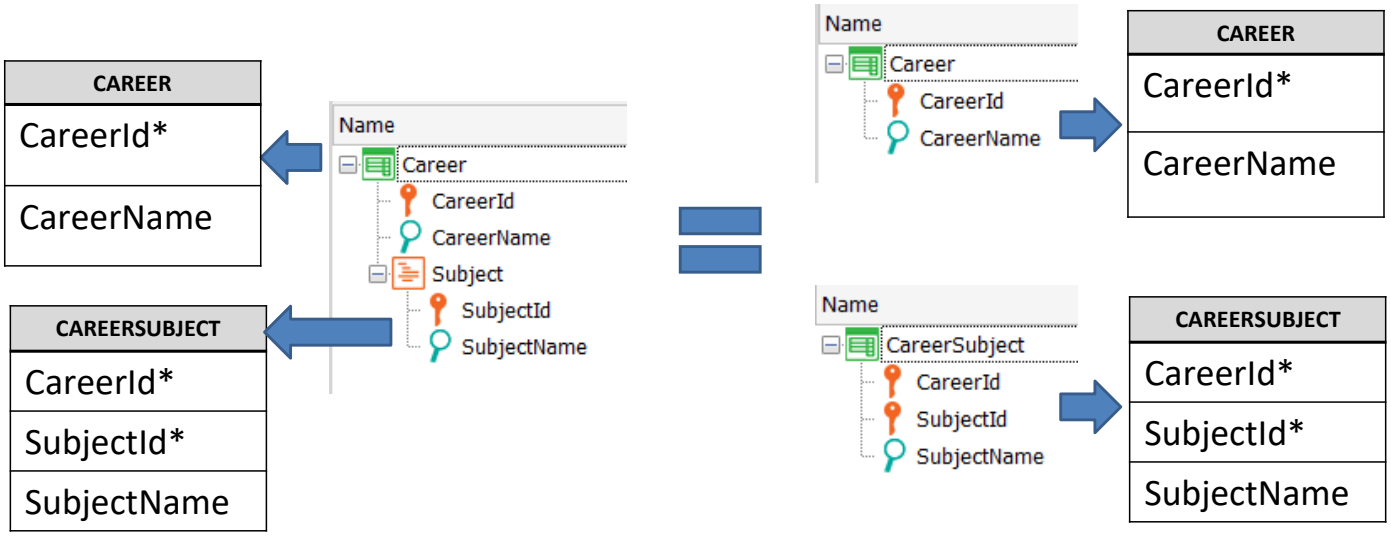


However, with this model, when entering data there are some limitations. For example, if we want to enter a degree program we can enter its header but, due to the automatic referential integrity check, we cannot enter the corresponding courses, because no course has been entered yet.

The same happens if we open the Course transaction first, as we will only be able to enter the data of each course, but not the degree programs to which they belong.

Therefore, we must first enter in one of them all the headers without lines, and then go to the other transaction and enter the header so that we can enter the corresponding lines.

Flattening the design



A multi-level transaction can be divided into single-level transactions. This operation is called flattening the model, since all transactions will become flat, that is, without sublevels.

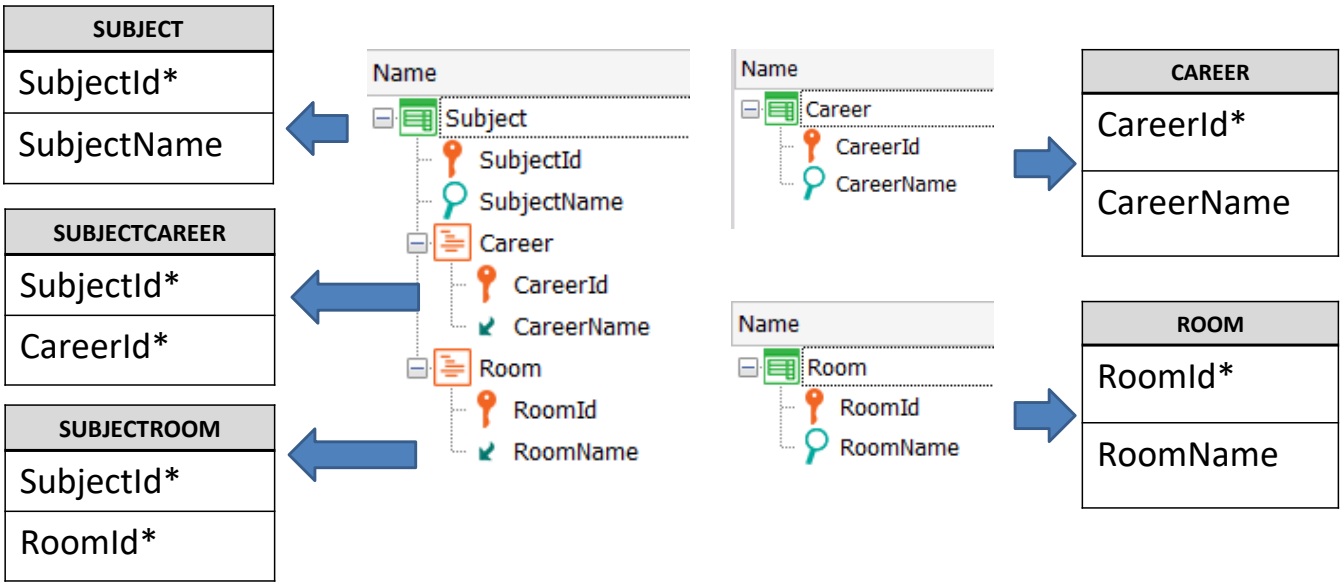
In this example, the DegreeProgram transaction, which has two levels, can be divided into two single-level transactions: DegreeProgram and DegreeProgramCourse.

The key of the DegreeProgramCourse transaction is formed as a compound key with the attributes DegreeProgramId and CourseId.

Note that there is no Course transaction, so CourseId will not be a foreign key and CourseName cannot be inferred; it will be an attribute stored in the DEGREEPROGRAMCOURSE table.

We see that with both transaction models we get exactly the same tables, so we can say that these two designs are equivalent.

Transactions with more than two levels



A transaction can contain more than one sublevel. In addition, each sublevel can in turn have sublevels.

A transaction having several sublevels is the model that will be used to represent a main entity (that of the header), which has several entities with weak 1-N relationships with it. Or to represent many-to-many relationships with several entities.

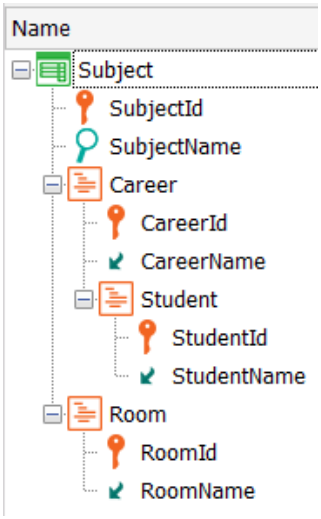
For example, let's consider the case where a course, in addition to the many-to-many relationship with degree program that we modeled before, can be taught in several classrooms and that each classroom can be used to teach many courses.

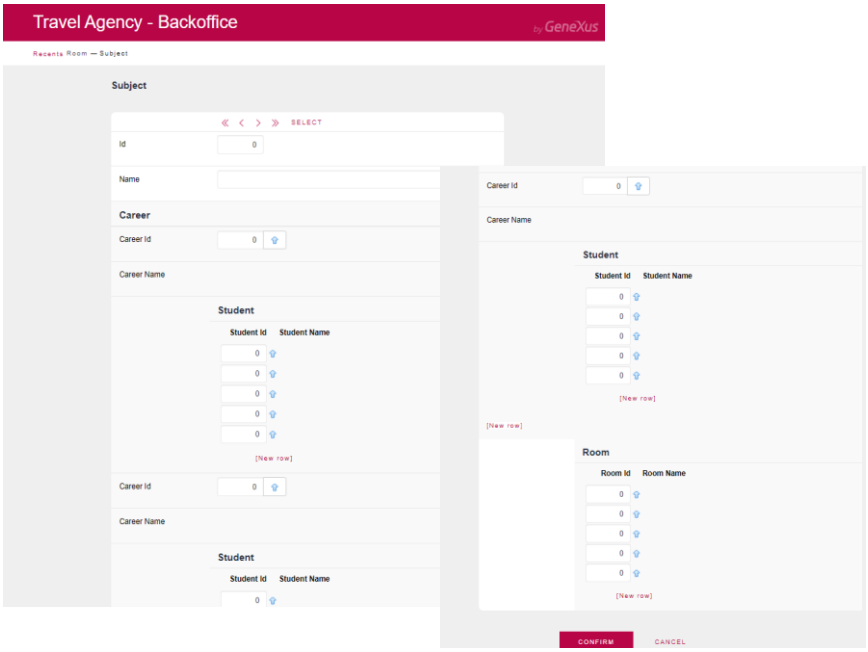
We can represent this reality by creating a Course transaction with a DegreeProgram sublevel and a Room sublevel, also adding the DegreeProgram and Room transactions to the model.

Considering the tables that GeneXus will create, we can check the many-to-many relationships between Course and each of the other entities.

The Course transaction screen will contain two grids, one for each sublevel.

Transactions with nested and sibling levels





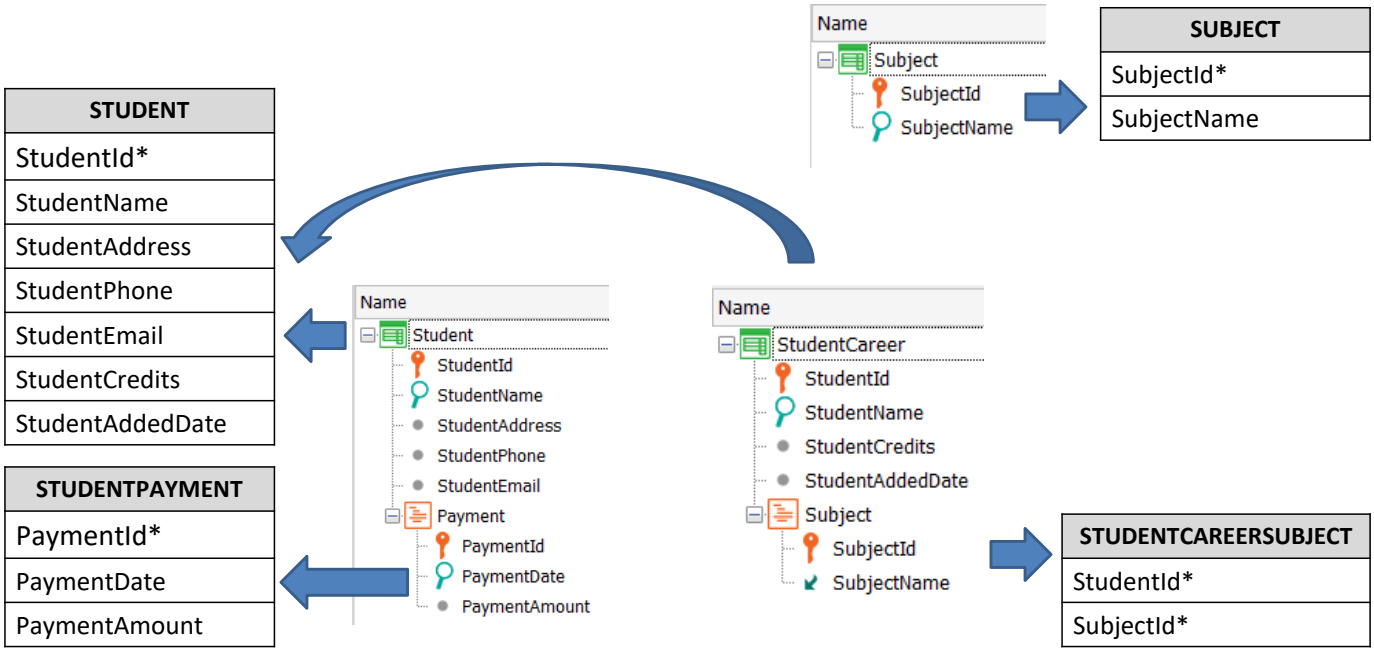
Although from a modeling point of view a sublevel may itself contain sublevels, it is necessary to consider how the user interface will look when the transaction screen is built.

In this example, a course has many degree programs and in turn each degree program has many students enrolled. In addition, the course has many classrooms where it can be taught.

With the design we made, the data fields of the course headers are displayed, but then for each degree program its header and the student lines are added. This block of data for each degree program is repeated several times and at the end of it all is the grid of classrooms. This format results in a lot of vertical scrolling, which is not very comfortable for the application users.

The developer will have to decide whether a transaction with these sublevels should be flattened (as we saw before), or whether to leave the design as it is so that the corresponding tables are generated and the data entry screens are implemented with web panels or panels, depending on the platform to be used.

Parallel transactions with more than one level



When we want to have segmented information of the same entity, we can use parallel transactions.

These transactions have the same identifier but contain those attributes with specific information of the entity, according to how we want to segment it.

For example, if we are talking about students, there could be a transaction with their names and another with their schooling data, both with the same student identifier.

In particular, it can happen that both parallel transactions have more than one level. For example, the student's data transaction may have a second level with their payments and the parallel transaction of schooling data may contain the courses they have taken.

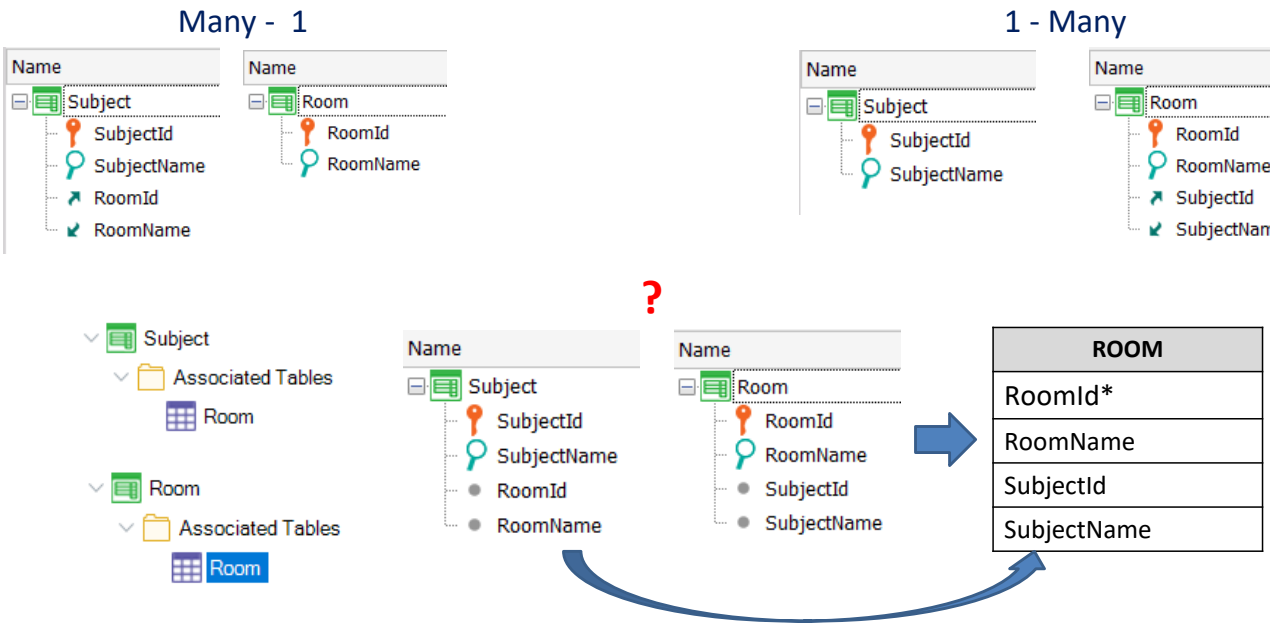
If we analyze the tables that will be created, we see that the COURSE table will be created from the Course transaction, and two tables will be created from the Student transaction: STUDENT and STUDENTPAYMENT. From the StudentDegreeProgram transaction, the same STUDENT table and the STUDENTDEGREEPROGRAMCOURSE table will be created.

If we pay attention to the STUDENT table, we see that not only the first-level attributes of the Student transaction are present, but also the attributes of the first level of the StudentDegreeProgram transaction. Those attributes that are common are added only once.
This is because the identifier of both transactions is StudentId, so GeneXus creates

a single table to contain all the attributes that functionally depend on StudentId, which in this case are the attributes of both transactions.

Therefore, from these two two-level transactions only three tables are created, not four, since they are parallel transactions and with the header attributes a single table containing all of them will be created.

Study case: crossed foreign keys



We know that if in a transaction we include an attribute that is a primary key in another transaction, the attribute will become a foreign key and a 1-to-many relationship will be established between both entities, where the "many" side of the relationship is that of the transaction where the foreign key is located.

So in the example on the left we are modeling that a classroom can be used to teach many courses and each course is taught in a single classroom (for example, if there are several class shifts). In the model on the right, one classroom is used to teach a single course, but the same course is taught in several classrooms–for example, if a course requires a classroom with special equipment (such as a laboratory) and there are several parallel classes of the same course (in several Laboratories).

What would happen if we cross the primary keys of both transactions, so that one has the primary key of the other as a foreign key and vice versa?

If we implement it in GeneXus, the first thing we notice is that when saving, the up and down arrows indicating the foreign keys and inferred attributes, respectively, are not shown.
Let's see now the tables that will be created with this design.

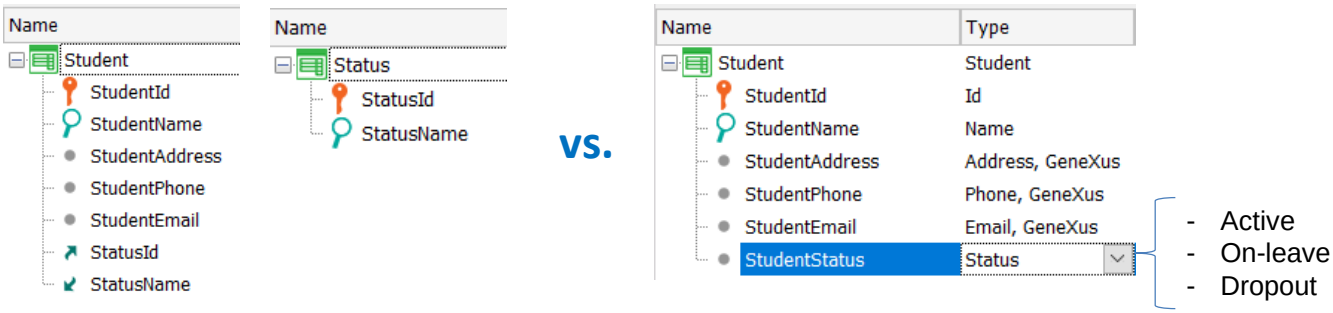
We can see that a single table is created! In this case, GeneXus created the Room table, with RoomId as primary key and the rest of the attributes as secondary attributes. What happened here?
The answer lies in the functional dependencies of the attributes.

When the identifiers are crossed, in one table an attribute is a primary key and therefore all the attributes of that transaction functionally depend on it, including the foreign key. But in the other transaction the same happens, now the primary key is another one and the one that was primary key (which is now foreign) functionally depends on the primary key of that transaction. Therefore, GeneXus must make a decision since the two identifiers cannot be primary keys at the same time given the established model where there is a double functional dependency.

For this reason, it chooses only one of them as primary key and creates a single table with that identifier, adding the rest of the attributes as secondary; that is, functionally dependent on the primary key it chose.

This result is usually not the one expected by the creator of this model, which often arises when trying to implement a 1-to-1 relationship between entities, or simply due to a modeling error.

Lookup table vs. Enumerated domain



Many times we want to assign a value to an attribute by selecting it from a list of values.

In general, to solve this we create a transaction where we store the entity we will choose, and the different records of the table make up the list of values. To do so, we make the attribute to which we want to assign one of several possible values a foreign key to the transaction that contains the values; eventually, we can retrieve other data from that entity through the extended table.

A typical case, continuing with the examples of the university, could be to select the country of a student, or the degree program of a course.

However, if the list of values is fixed, or changes very little and, in particular, doesn't have a lot of data, we could create an enumerated domain with those values.
For example, suppose a student could have “active” status if he/she is pursuing a degree; “on-leave” status if he/she has taken a year off; or “dropout” status if he/she dropped the courses.

In that case, we could create a Status domain with the 3 values: A for active, L for on-leave, and D for dropout.
However, it may happen that later on it is decided to add a new status; for example, when a student does an internship at another university and you want to register this status that is new and doesn’t match any of the previous ones.

If we use an enumerated domain and the university needs to add that new status,

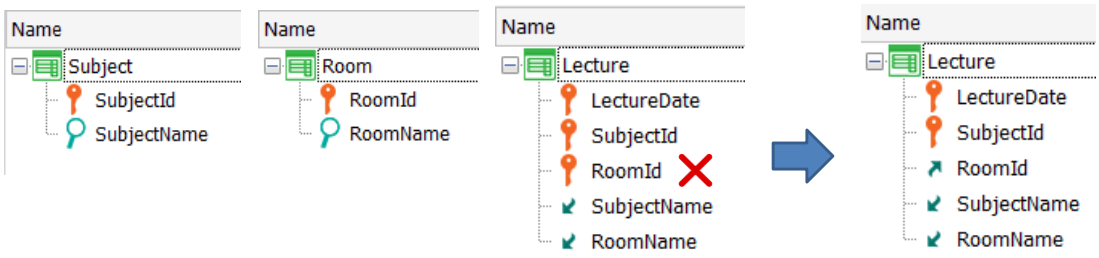
we will have to add that value to the domain and then generate the application again; this means that the university system users cannot easily add new information about their reality, but they depend on the developer to generate the programs again, test them, and deploy the new version.

In addition, if for some reason an enumerated domain value had been used as part of a compound primary key in a transaction, it would not only be necessary to generate the application again, but also the data in the database, making sure that the existing data is not lost when creating a table with a new primary key.

From that point of view, we would never use enumerated domains; however, there may be some exceptions for values that we know cannot change, such as the names of the days of the week or the months of the year.

In summary, except for well-known cases where we are sure that the values are not going to change, if we know that there is even a remote possibility that this data may change (such as the states of a country, or the names of existing countries), it is always best practice to create a table containing these values, which ensures that the application user has the flexibility to update the data, even if this means increasing the structures to be maintained in the database.

Compound primary key vs. simple primary key



SUBJECT

SubjectId	SubjectName
1	Software engineering
2	Basic electronics

ROOM

RoomId	RoomName
1	A101
2	A102
3	A103

LECTURE

LectureDate	SubjectId	RoomId
24/10/2022	1	1
24/10/2022	1	2
24/10/2022	1	3

When we choose the identifier of an entity, sometimes several options are available, since there may be several candidate keys that can be chosen as the primary key, following the principles of uniqueness (that is, there can't be two records with the same key) and irreducibility (that the key must be the minimum set of attributes that uniquely identify the records).
It is quite common to comply with uniqueness, but irreducibility is not always taken into account and we end up defining primary keys with unnecessary attributes.

Let's see this with an example.

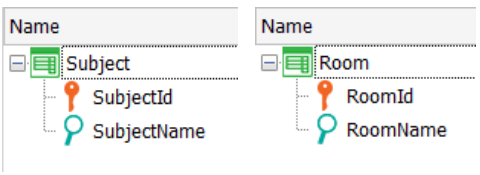
Suppose we want to model the case of a course taught in a classroom on a given date. We assume that this course is taught only once on that date. The transactions shown have been modeled: the Lecture transaction has a key composed of LectureDate, CourseId, and RoomId, to "make sure" that the combination of the course with the date and with the classroom is unique.

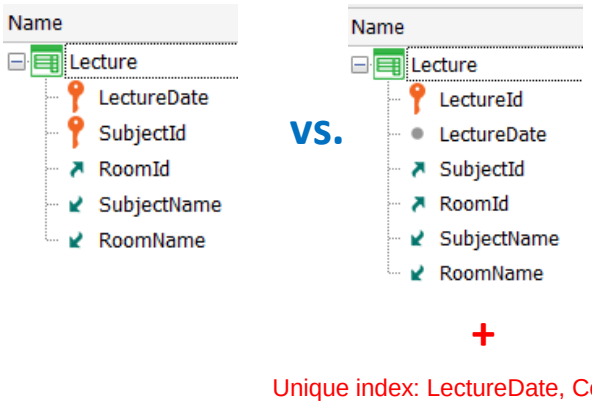
However, if we analyze the data, we see that it was possible to enter a lecture for a course on a date and that it can be taught in several classrooms at the same time, since it is enough that one of the components of the key changes its value for the other components of the key to be able to repeat the value in different records.

It is clearly not necessary for RoomId to be part of the key, since only with LectureDate and CourseId we can properly identify the class to ensure that the same subject is not repeated on the same date. The RoomId attribute can be a

foreign key.

Compound primary key vs. simple primary key





A doubt that may arise is that if, instead of defining a compound key in Lecture formed by the date and course attributes whose combination we want to be unique, we define a LectureId artificial key that for example is an autonumbered numeric identifier.

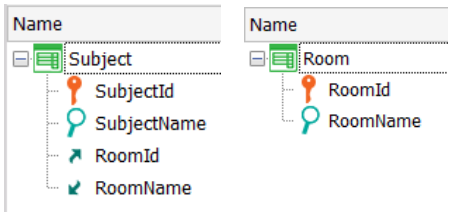
In both cases, uniqueness is being controlled and the key is irreducible; that is, it is the minimum key to correctly identify the records in the table.

However, in the case of the compound key, we cannot modify the date or course values because they are part of the key; to do so, we must delete the record and define another one.

On the other hand, in the case of the key with the class identifier, we can change these data. However, when changing them we must control that the combination of date and course values is not repeated, since it would be possible to define two records with different LectureId, but the same date-course pair.

To avoid this, we could define a unique index by date and subject in the LECTURE table.

Referential integrity: under the hood



SUBJECT

SubjectId	SubjectName	RoomId
1	Software engineering	3
2	Basic electronics	2
3	Mathematics analysis	1
4	English	4

ROOM

RoomId	RoomName
1	A101
2	A102
3	A103
?	

GeneXus pays special attention to maintaining adequate referential integrity in order to ensure data consistency. This means that it is not possible to have a foreign key value that doesn't exist as a primary key in the source table, nor is it possible to delete a primary key value that has related records where that key is foreign.

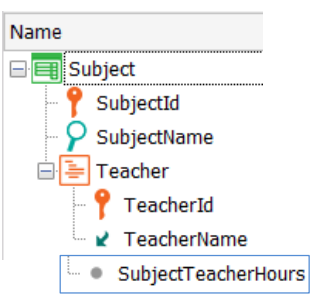
When data is updated by means of transactions, either by executing their form or a business component of that transaction, the referential integrity check is automatic.

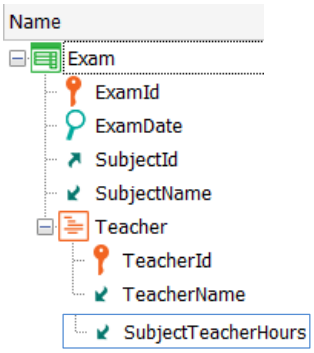
However, there are some special cases in which it may seem that the check is made, but actually it is not, or that unwanted checks are made and we want to avoid them, or that a check is automatically replaced with another one under certain conditions.

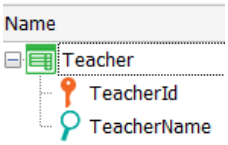
Let’s analyze some examples that show these situations.

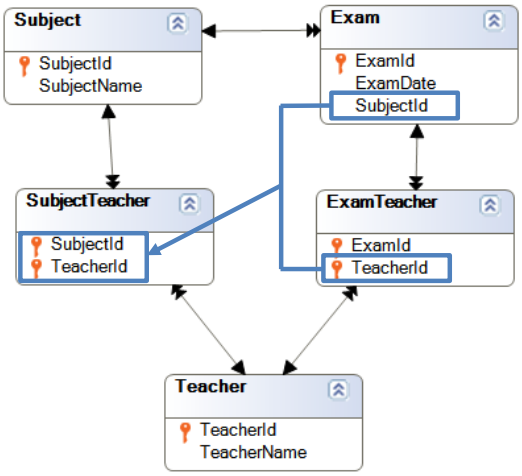
Referential integrity: under the hood

Case 1: Logic subordination









Suppose we want to record the exams of a certain course and that an exam will be administered by one or more professors. For each course, the professors who teach it are recorded.
When we add an exam, it must be checked that the professor assigned to the exam teaches the course of the exam. Is this ensured by the design?

The answer is no because with this design the check will not be performed automatically. When a record is going to be entered in the EXAMPROFESSOR table, we want to check if there is a record in the COURSEPROFESSOR table with the SubjectId value corresponding to that of the exam and with the ProfessorId value corresponding to the professor that is being inserted.

Although both values are saved in memory, {CourseId, ProfessorId} do not form a foreign key since they are not in the same table.

However, we can say that they form a logical foreign key even though it does not exist at the relational database level. Since there is a logical subordination relationship between the tables (and not physical subordination), GeneXus will not perform the referential integrity check we need.

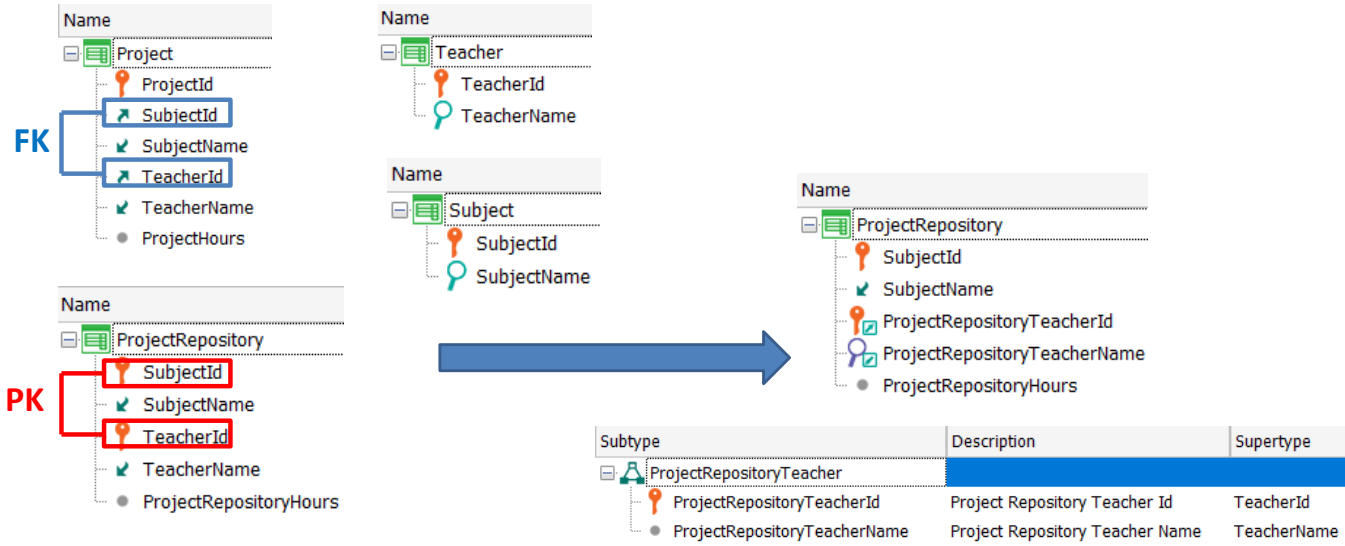
One way to solve this is to define a rule in Exam that invokes a procedure that performs the check and then with an Error rule we evaluate the result of the invocation.

Another way to solve this is to add a secondary attribute to the second level of the Course transaction; for example, CourseProfessorHours that records the hours

that the professor has assigned to that course, in order to infer it in the Exam transaction and thus force GeneXus to become aware of the relationship.

Referential integrity: under the hood

Case 2: Unintended referential integrity checks



Now let's suppose that in the reality of the university we have been developing we want to record the final projects developed to pass certain courses. A project has an identifier, an assigned professor, a course, and the number of hours assigned to that project.

In addition, there should be a repository with all the projects developed at the university, so that at the end of the semester the projects that were developed in that semester are entered into the repository.
To this end, a ProjectRepository transaction was created and a compound primary key was defined, consisting of the professor and the course of the project.

The problem with this design is that CourseId and ProfessorId form in the Project transaction a foreign key to ProjectRepository. Therefore, when trying to enter a project in the Project transaction, it is required that the project be previously entered in the PROJECTREPOSITORY table, which is impossible, because first the project is registered and only later—at the end of the semester—it is added to the repository.

To avoid this, we can create a ProjectRepositoryProfessor subtype group, with the subtypes ProjectRepositoryProfessorId, subtype of ProfessorId, and ProjectRepositoryProfessorName, subtype of ProfessorName.
Then, we replace the ProfessorId and ProfessorName attributes in the ProjectRepository transaction with the corresponding subtypes.

In this way, with the subtype, what we do is change the name of ProfessorId in the table in which this attribute is part of the primary key.

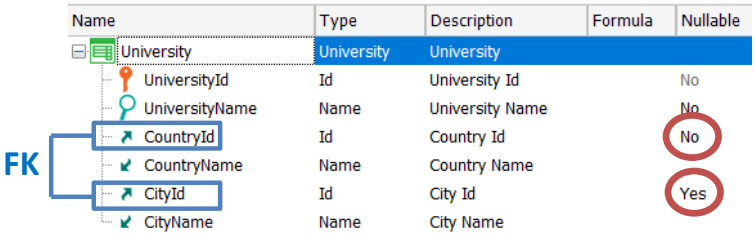
This doesn’t prevent GeneXus from checking if the value exists in the PROFESSOR table when we enter the professor identifier in the ProjectRepository transaction.

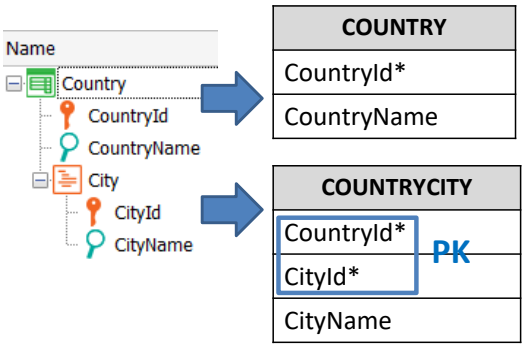
But this name change prevents the {CourseId, ProfessorId} attribute pair from being identified as a foreign key in Project, since now the names of the attributes that form the primary key in ProjectRepository are different.

In this way, by using subtypes, we were able to avoid performing a referential integrity check that was not intended in our reality.

Referential integrity: under the hood

Case 3: Compound foreign key partially nullified





If we set an attribute that is a simple foreign key as nullable, when entering a record, if we do not enter the value, the referential integrity check will not be performed. If we enter a value instead, it will be checked that the value entered in the foreign key exists as a primary key in the corresponding table.

However, when the foreign key is a compound key, we could set as nullable only some of the attributes of the key.

In the example, a university belongs to a city. Cities are recorded in the COUNTRYCITY table, which has CountryId and CityId as a compound primary key. In the University transaction, the attributes CountryId and CityId form a compound foreign key and, in particular, we can set both attributes, none, or only one of them as nullable.

Let's suppose that when entering a university we know the country but not the city it belongs to, so in the CityId attribute we set Nullable to Yes.

What will happen when we enter a university? Will there be any kind of referential integrity check?
The answer is Yes. If, when entering a university, we don't specify the value of CityId, the integrity check will not be performed to verify if the city exists in the cities table. However, since the CountryId attribute still has the Nullable property set to No, an integrity check will be performed on the COUNTRY table to verify that the entered country exists as a country in the countries table.

This integrity check on the COUNTRY table was not performed if both attributes of

the compound foreign key had the Nullable value set to No, but the check was performed only on the COUNTRYCITY table.

That is, GeneXus added an integrity check that was not originally done, since the CountryId attribute is still a foreign key regarding the COUNTRY table.

Again, this shows the goal of always keeping data consistent, even in cases where you try to disable certain controls.

In this summary, we have tried to integrate several topics related to transaction design and its impact on the database and the application's functionality. We encourage you to learn more about some of these topics in other videos published specifically for each topic.

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