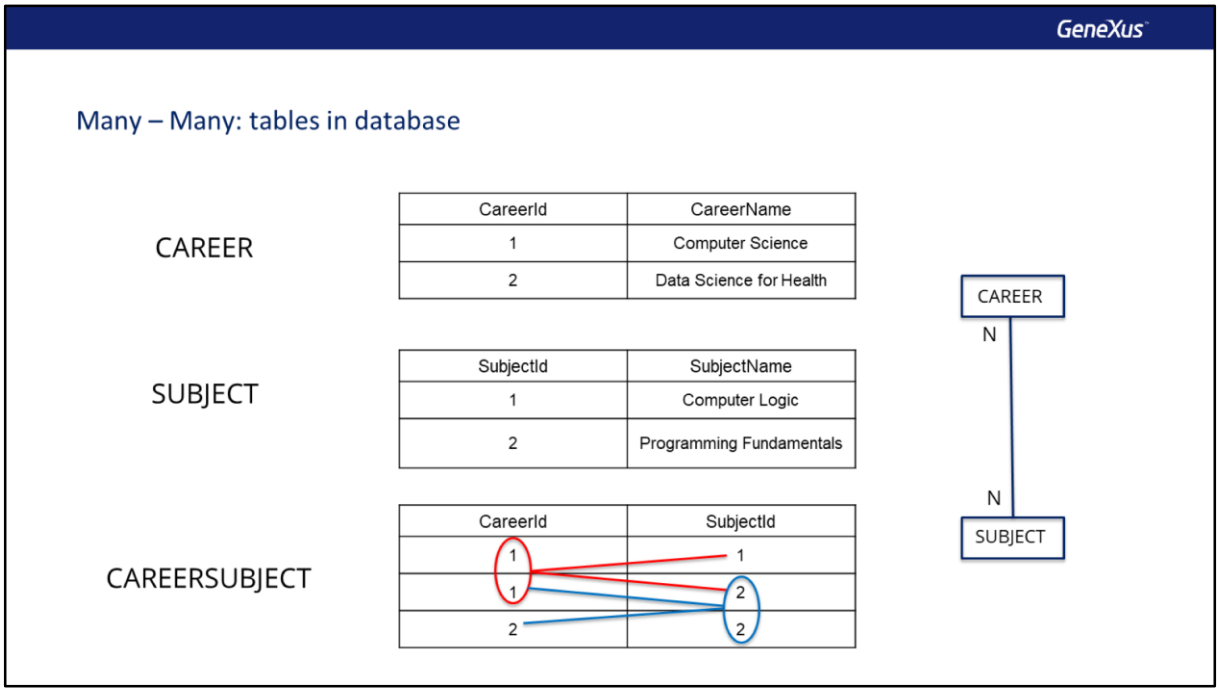
Diseño de transacciones y Normalización

Una mirada integradora

GeneXus™

En este video trataremos de analizar distintos temas relacionados con el diseño de transacciones y cómo las decisiones que tomemos se reflejan en las estructuras de la base de datos creadas, o en la funcionalidad de la aplicación.

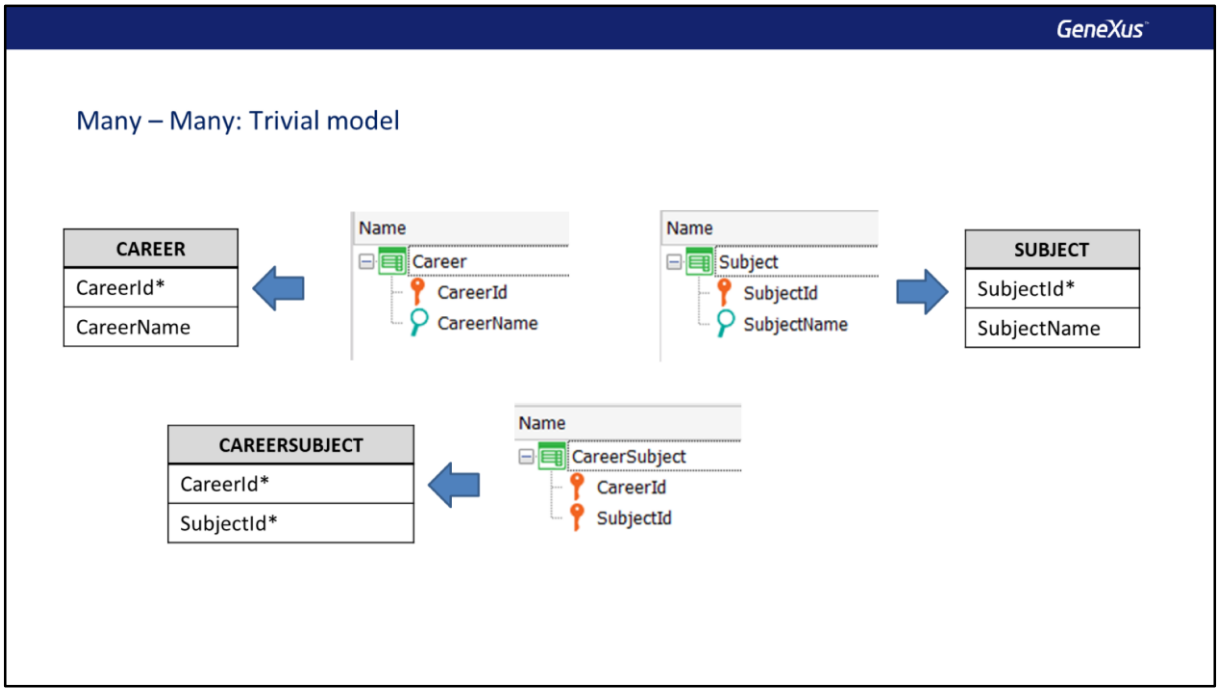
Por esa razón, a diferencia de videos anteriores del tema donde se seguía el hilo conductor del desarrollo de un ejemplo, en este caso abordaremos casos puntuales que nos permitan analizar distintas situaciones prácticas que puedan sernos de utilidad en la construcción de nuestra solución.



Para representar una relación muchos a muchos entre dos entidades en una base de datos relacional, se utilizan tres tablas; una por cada entidad y una tercera (también llamada tabla relación) que contiene los identificadores de las tablas anteriores formando una clave compuesta.

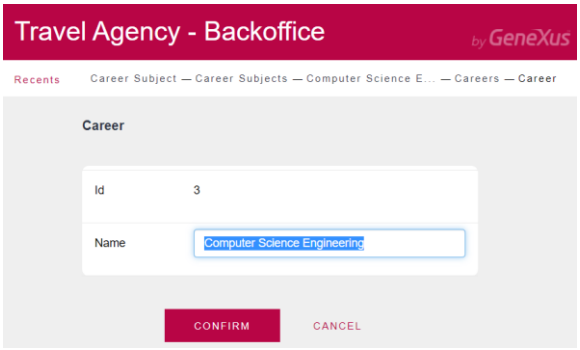
Tomemos como ejemplo el caso de la realidad de una universidad, donde cada carrera tiene muchas materias y cada materia puede estar en muchas carreras. Para representar esta relación, tendremos una tabla Career, una tabla Subject y una tabla CareerSubject que tiene como clave, a las claves de las tablas anteriores formando una clave compuesta.

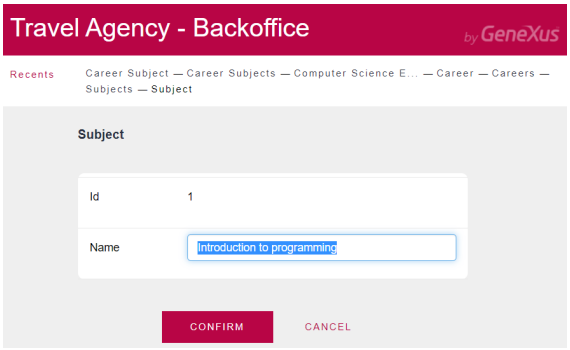
Si analizamos los datos, vemos que la carrera 1 contiene a las materias 1 y 2, pero que a su vez la materia 2 está en la carrera 1 y en la carrera 2, por lo que este modelo efectivamente nos permite representar una relación de muchos a muchos, entre carreras y materias.

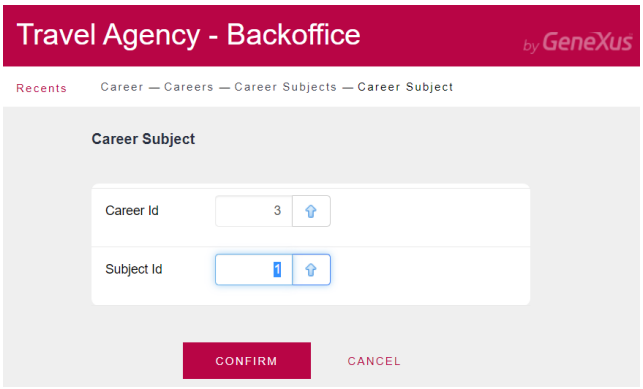


En GeneXus no usamos tablas para modelar la realidad, sino transacciones. Una solución trivial para modelar la relación entre carreras y materias de forma que GeneXus genere las tres tablas que necesitamos, es crear transacciones con la misma estructura que las tablas.

Como las tres transacciones son planas, es decir no hay subniveles, se crearán las tablas de acuerdo a lo esperado, de modo que podemos asegurar que este modelo efectivamente representa una relación muchos a muchos entre carreras y materias.

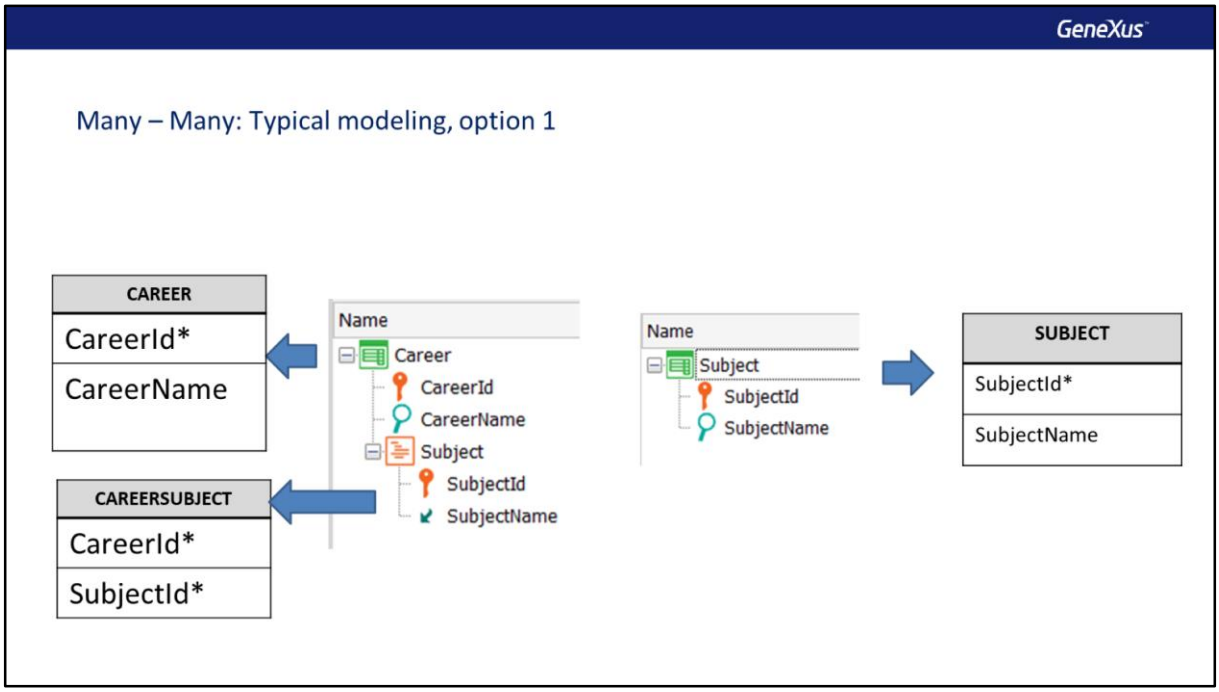






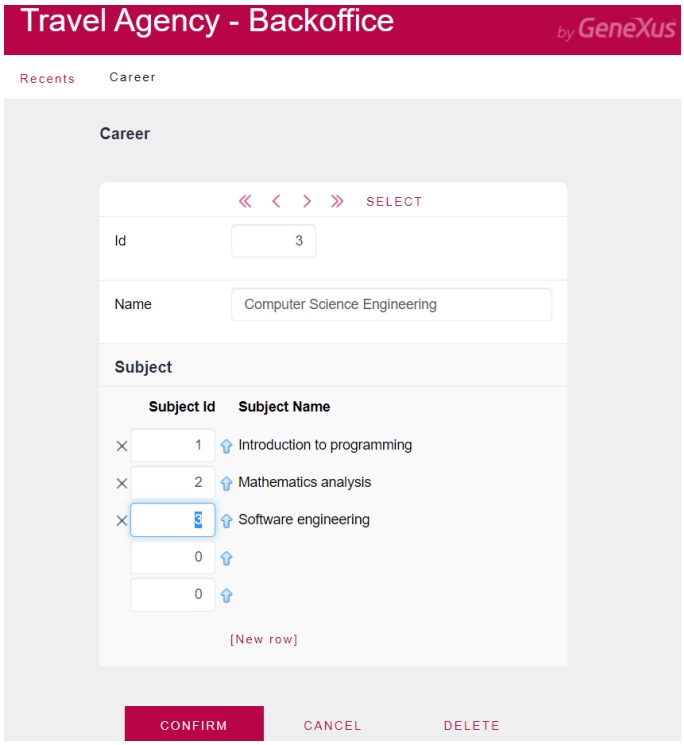
Sin embargo, si consideramos la interfase de usuario al momento de ingresar los datos, tendríamos 3 pantallas. Las carreras y las materias las ingresaríamos sin inconvenientes, pero después es muy poco amigable tener que ingresar los pares de identificadores de carrera y materia para establecer qué carreras corresponden a qué materia y viceversa.

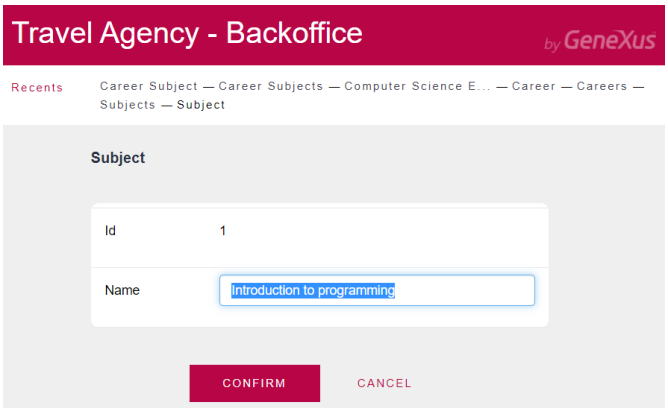
Por lo tanto, si bien este diseño cumple con modelar la relación muchos a muchos, no sería recomendable. Es preferible que en una pantalla podamos ingresar los datos de una carrera y luego en una grilla todas las materias de esa carrera, o viceversa....



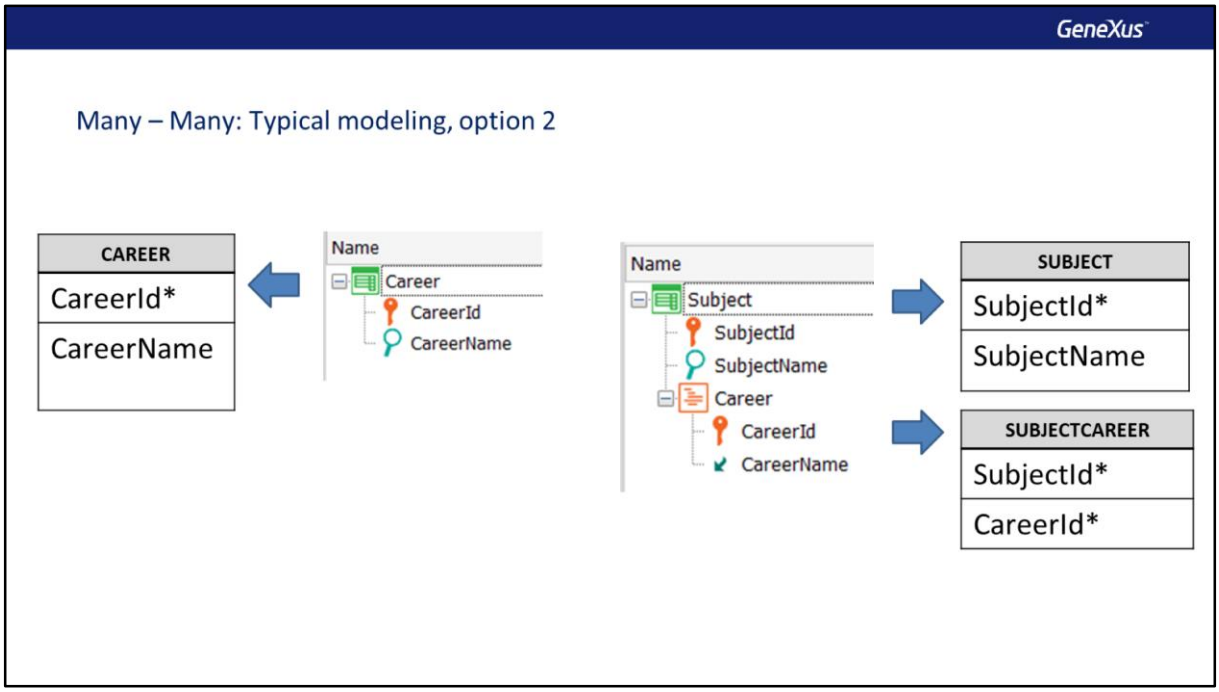
Para obtener una pantalla donde para una carrera puedan ingresarse todas sus materias, podemos crear una transacción Career de dos niveles, donde el nivel anidado son las materias. Con esa transacción sola modelaríamos una relación 1 a muchos entre carreras y materias, para que sea una relación muchos a muchos agregamos una segunda transacción Subject.

Si vemos las tablas que crea GeneXus, vemos que efectivamente modelamos una relación muchos a muchos entre carreras y materias, ya que obtenemos las mismas tres tablas que vimos antes.





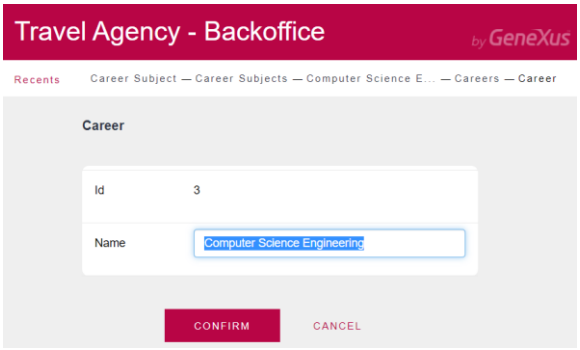
En ejecución vemos que nuestro modelo priorizó el enfoque de ingresar las materias, para luego cargar las materias de cada carrera.

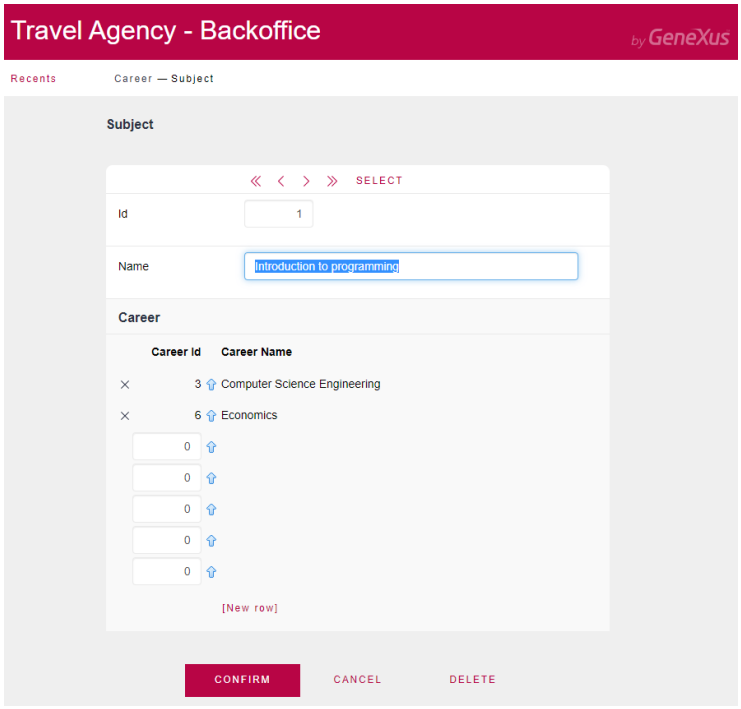


Ahora, si para el mismo caso de la relación muchos a muchos entre materias y carreras, nos hubieran pedido una pantalla que para cada materia le ingresáramos las carreras a las que pertenece, entonces crearíamos una transacción Subject de dos niveles, donde el segundo nivel corresponde a las carreras.

Obviamente también creamos una transacción Career para que se mantenga la relación muchos a muchos entre las entidades.

Si vemos las tablas que se crearán, comprobamos que son exactamente las mismas que en los ejemplos anteriores, por lo que estamos seguros que modelamos una relación muchos a muchos entre materias y carreras.

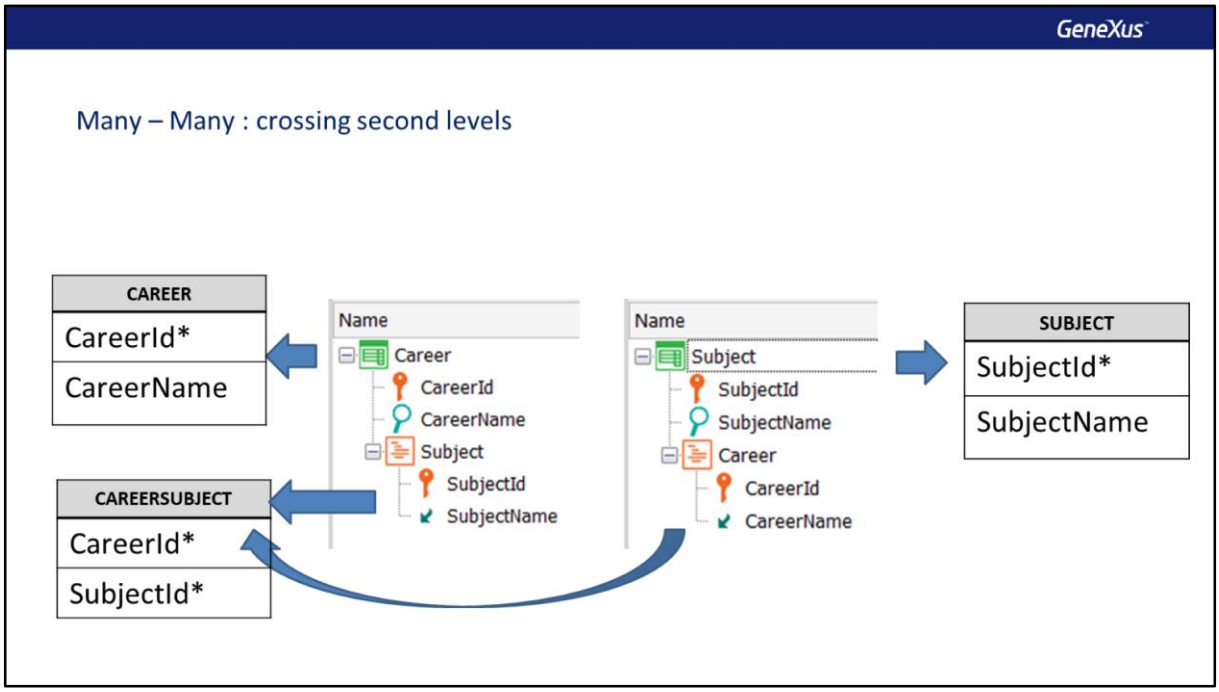




En ejecución comprobamos que ahora el enfoque es que primero ingresamos las carreras y luego para cada materia, ingresamos las carreras a las que pertenece.

De modo que para modelar una relación muchos a muchos, alcanza con dos transacciones, una con dos niveles y para la entidad que ponemos en el segundo nivel, creamos también una transacción independiente.

Desde el punto de vista de la relación muchos a muchos, no importa qué entidad ponemos en el segundo nivel, solamente tiene relevancia a la hora de ingresar los datos.



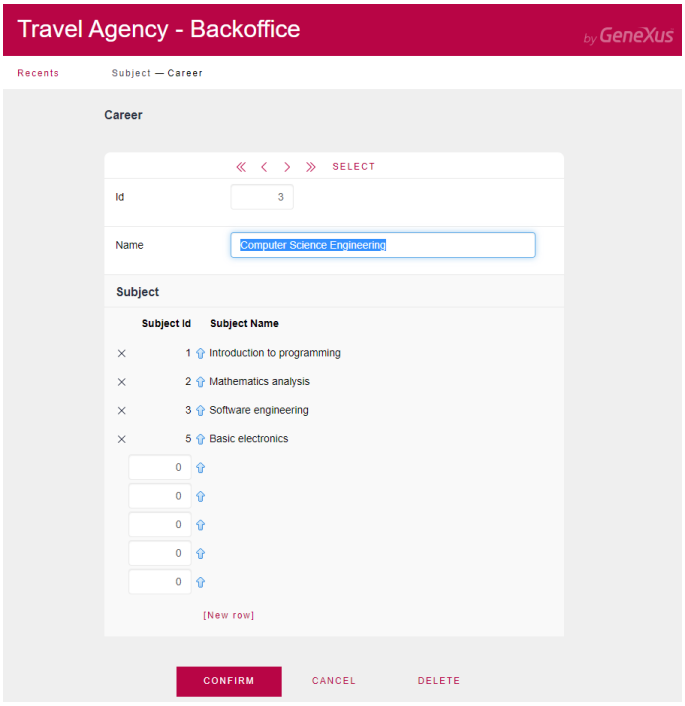
Ahora ¿qué pasaría si nos piden que en una pantalla se pueda ingresar una carrera y para esa carrera todas las materias que contiene y a la vez otra pantalla donde se pueda ingresar una materia y todas las carreras a la que pertenece?
Siguiendo el razonamiento anterior, deberíamos crear dos transacciones de dos niveles, una llamada Career con el segundo nivel Subject y otra llamada Subject con el segundo nivel Career.

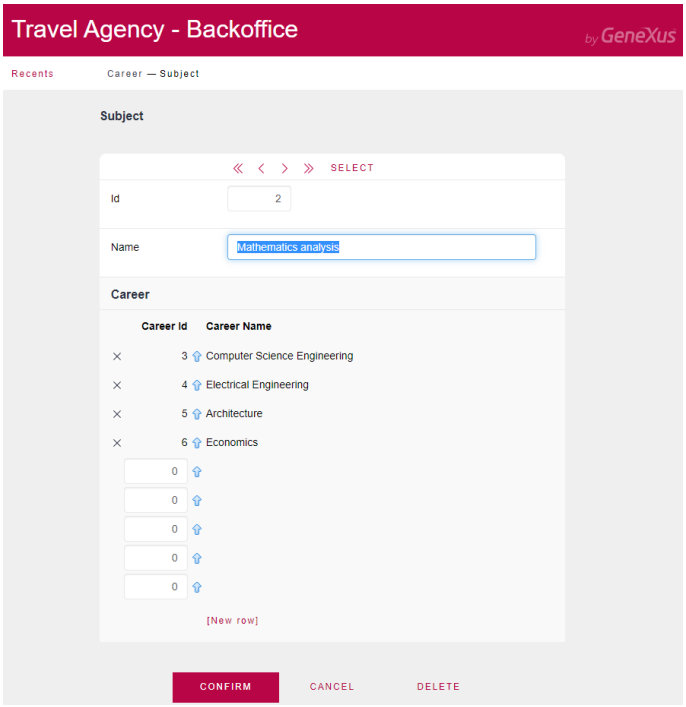
¿Pero qué relación estaríamos modelando en este caso?
Para saberlo, escribimos las tablas que GeneXus creará. Sabemos que de la transacción Career se creará una tabla CAREER con la estructura del primer nivel de la transacción y que del segundo nivel se creará una tabla CAREERSUBJECT que, debido a que el atributo SubjectName es inferido, la tabla contendrá únicamente a los atributos identificadores del primer y del segundo nivel, formando una clave compuesta.

Si analizamos las tablas que se crearán a partir de la transacción de dos niveles Subject, comprobamos que del primer nivel se creará una tabla SUBJECT con la misma estructura que el cabezal de la transacción Subject y del segundo nivel una tabla asumimos que se llamaría SUBJECTCAREER conteniendo únicamente a una clave compuesta por SubjectId y CarrerId, ya que el atributo CareerName es inferido.

Pero GeneXus ya creó una tabla con exactamente esa estructura, la tabla CAREERSUBJECT, por lo tanto no crea otra y el resultado final de este modelo son las mismas tres tablas que obtuvimos antes y que sabemos que corresponden a una relación muchos a muchos entre carreras y materias.

Por lo tanto, si creamos dos transacciones de dos niveles y cruzamos las entidades, es decir que en una ponemos como segundo nivel lo que en la otra es el primer nivel y viceversa, estaremos modelando una relación muchos a muchos.



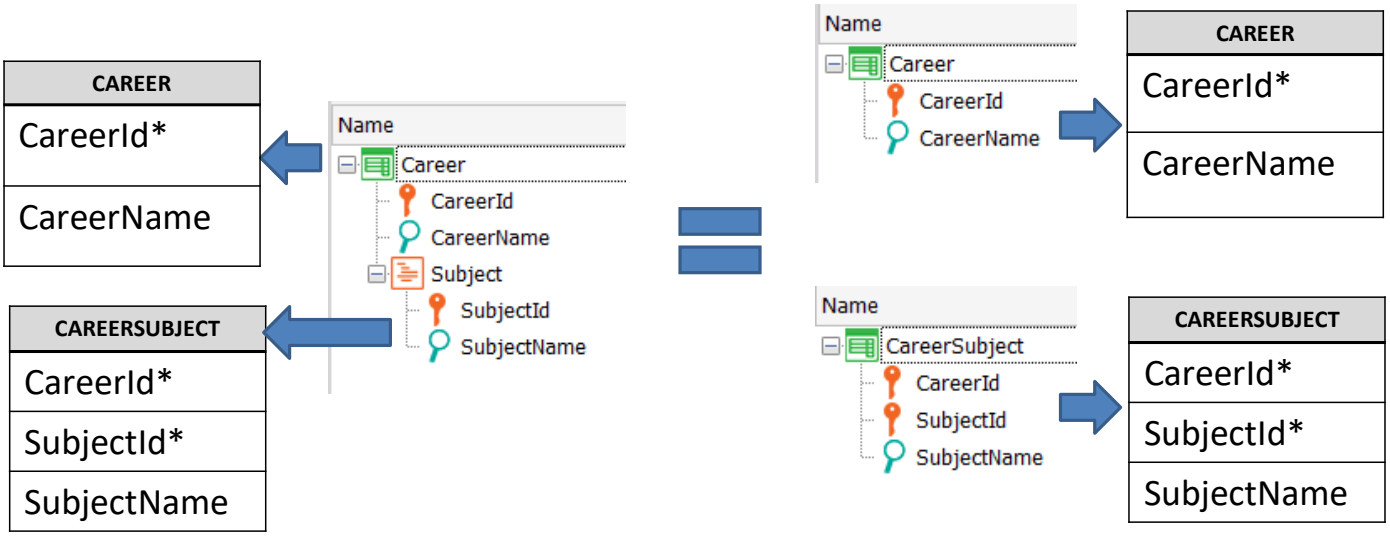


Sin embargo, con este modelo, a la hora del ingreso de datos tenemos algunas limitaciones. Por ejemplo, si queremos ingresar una carrera podemos ingresar su cabezal pero, debido al control automático de integridad referencial, no podemos ingresar las materias que le corresponden, porque aún no fue ingresada ninguna materia.

Lo mismo pasa si abrimos primero a la transacción Subject, solamente podremos ingresar los datos de cada materia, pero no las carreras a las que pertenecen.

De modo que debemos primero ingresar en una de ellas todos los cabezales sin líneas, y luego ir a la otra transacción e ingresar cabezal y luego sí podremos ingresar las líneas correspondientes.

Flattening the design



Una transacción de varios niveles, puede ser descompuesta en transacciones de un solo nivel. A esta operación le decimos aplanar el modelo, ya que todas las transacciones pasarán a ser planas, es decir sin subniveles.

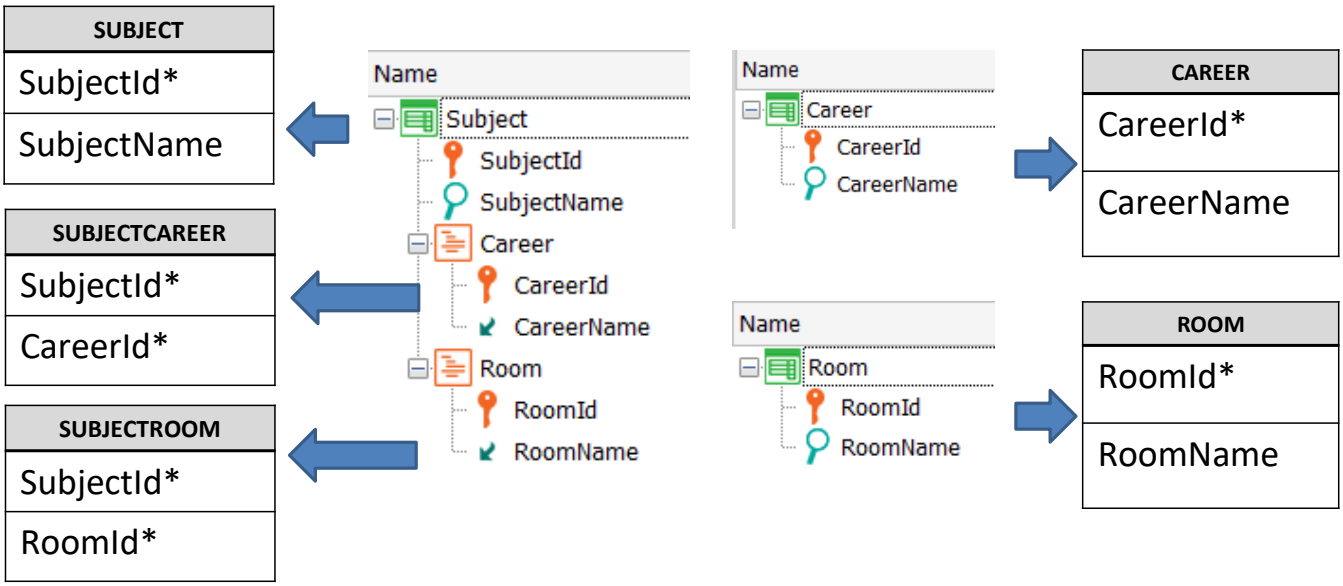
En este ejemplo, la transacción Career que tiene dos niveles, podemos descomponerla en dos transacciones de un solo nivel: Career y CareerSubject.

La clave de la transacción CareerSubject la formamos como clave compuesta con los atributos CareerId y SubjectId.

Observemos que no existe una transacción Subject, por lo que SubjectId no será clave foránea y SubjectName no podrá inferirse, será un atributo almacenado en la tabla CAREERSUBJECT.

Vemos que con ambos modelos de transacciones obtenemos exactamente las mismas tablas, por lo que podemos afirmar que estos dos diseños son equivalentes.

Transactions with more than two levels



Una transacción puede contener más de un subnivel. Además cada subnivel puede a su vez tener subniveles.

Que una transacción tenga varios subniveles, es el modelo que servirá para representar una entidad principal (la del cabezal) que tiene varias entidades con relaciones 1-N débil con ella. O bien para representar relaciones muchos a muchos con varias entidades.

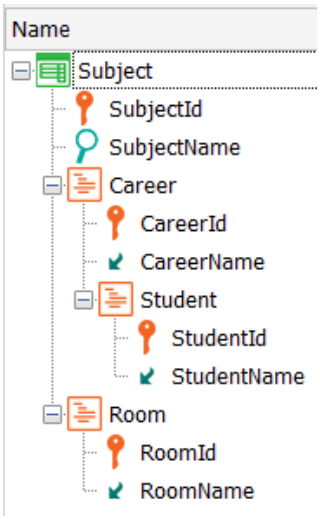
Por ejemplo consideremos el caso de que una materia, además de la relación muchos a muchos con carrera que modelamos antes, pueda dictarse en varios salones y que cada salón puede ser utilizado para dictar muchas materias.

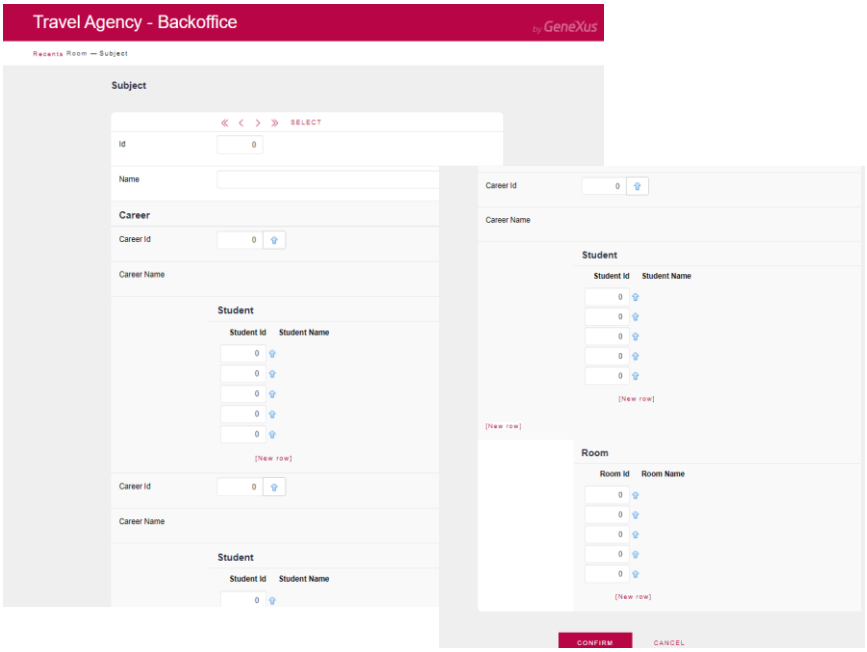
Podemos representar esta realidad creando una transacción Subject con un subnivel Career y otro subnivel Room, agregando también al modelo las transacciones Career y Room.

Viendo las tablas que GeneXus construirá, podemos corroborar las relaciones muchos a muchos entre Subject y cada una de las otras entidades.

La pantalla de la transacción Subject contendrá dos grids, uno por cada subnivel.

Transactions with nested and sibling levels





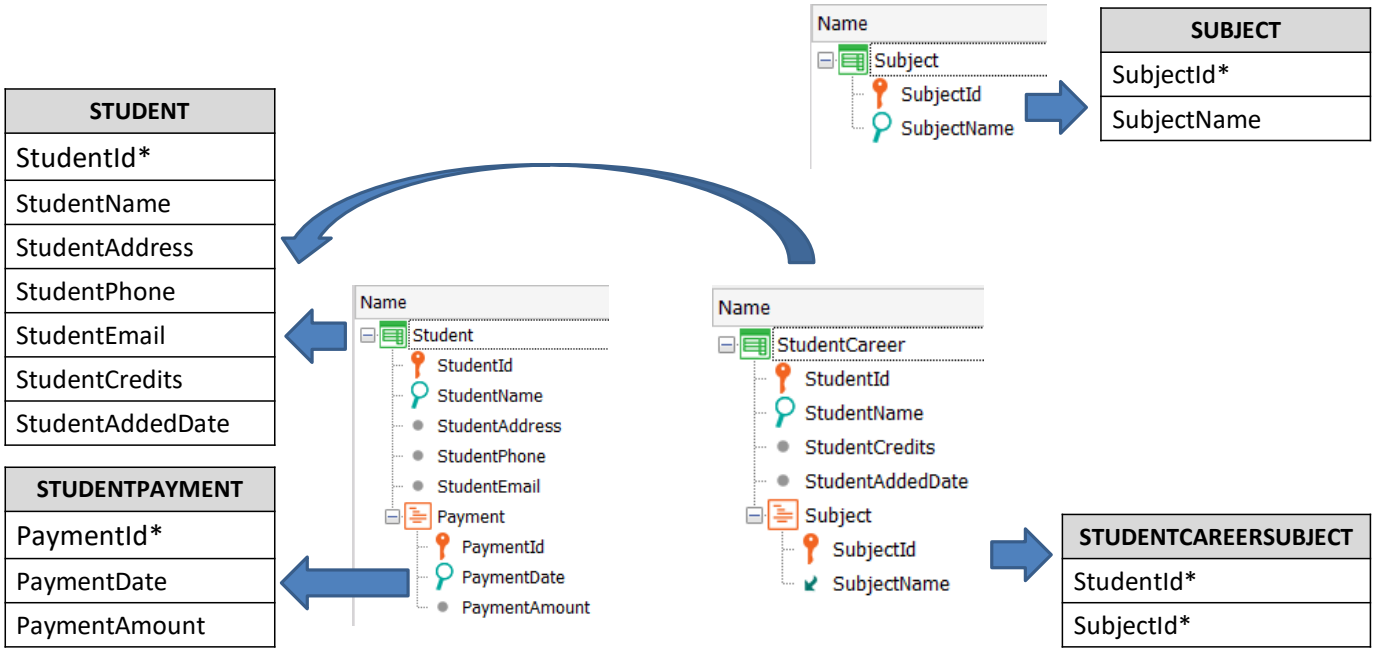
Si bien desde el punto de vista del modelado un subnivel puede a su vez contener subniveles, hay que considerar cómo quedará la interfase de usuario cuando se construya la pantalla de la transacción.

En este ejemplo, una materia tiene muchas carreras y a su vez cada carrera tiene inscriptos muchos estudiantes. Además la materia tiene muchos salones donde puede dictarse.

Con el diseño que hicimos, aparecen los campos de datos del cabezal de las materias, pero luego para cada carrera se agrega su cabezal y las líneas de los estudiantes. Ese bloque de datos de cada carrera se repite varias veces y al final de todo está la grilla de salones. Este formato provoca bastante scroll vertical que no es muy cómodo para los usuarios de la aplicación.

El desarrollador tendrá que decidir si en lugar de una transacción con estos subniveles no le conviene aplanar algún nivel (como vimos antes) o si deja el diseño como está para que se generen las tablas correspondientes y las pantallas de ingreso de datos las implementa con web panels o panels, dependiendo de la plataforma que vaya a utilizar.

Parallel transactions with more than one level



Cuando queremos tener la información segmentada de una misma entidad, podemos optar por usar transacciones paralelas.

Estas son transacciones que tienen el mismo identificador, pero contienen aquellos atributos con información específica de la entidad, de acuerdo a cómo queremos segmentarla.

Por ejemplo si hablamos de alumnos podría ser una transacción con los datos patronímicos y otra con los datos de su escolaridad, ambas con el mismo identificador de alumno.

En particular puede darse el caso de que ambas transacciones paralelas tengan más de un nivel. Por ejemplo, puede pasar que la transacción de datos del alumno tenga un segundo nivel con los pagos y la transacción paralela de datos de escolaridad tenga las materias cursadas.

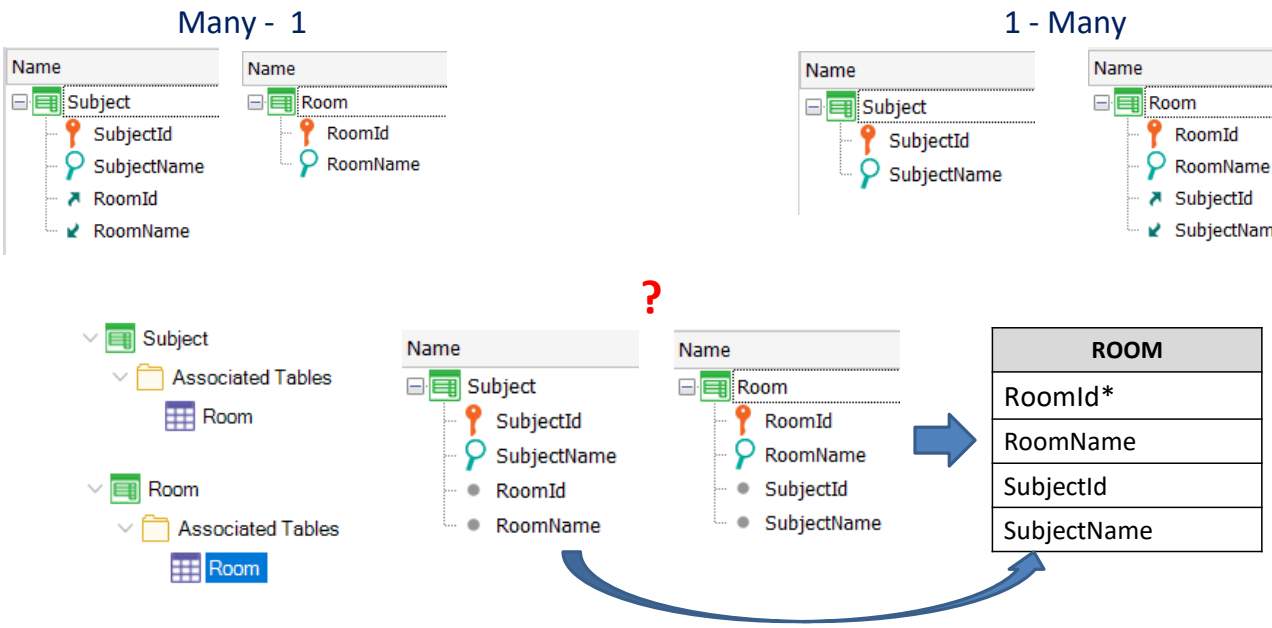
Si analizamos las tablas que se crearán, vemos que a partir de la transacción Subject se creará la tabla SUBJECT, a partir de la transacción Student se crearán dos tablas: STUDENT y STUDENTPAYMENT y a partir de la transacción StudentCareer se creará la misma tabla STUDENT y la tabla STUDENTCAREERSUBJECT .

Si prestamos atención a la tabla STUDENT vemos que no solamente están presentes los atributos del primer nivel de la transacción Student, sino también los atributos del primer nivel de la transacción StudentCareer. Aquellos atributos que son comunes se agregan una sola vez.

Esto es porque el identificador de ambas transacciones es StudentId, por lo cual GeneXus crea una única tabla para contener todos los atributos que dependen funcionalmente de StudentId, que en este caso son los atributos de ambas transacciones.

Por lo tanto, a partir de estas dos transacciones de dos niveles no se crean cuatro tablas, sino solamente tres, ya que son transacciones paralelas y con los atributos de los cabezales se creará una única tabla que los contiene a todos.

Study case: crossed foreign keys



Sabemos que si incluimos en una transacción un atributo que es clave primaria en otra transacción, el atributo pasará a ser clave foránea y se establecerá una relación 1 a muchos entre ambas entidades, donde el lado muchos de la relación es el de la transacción donde está la clave foránea.

De modo que en el ejemplo de la izquierda estamos modelando que un salón puede usarse para dictarse muchas materias y cada materia se dicta en un único salón (por ejemplo si hay varios turnos de clases). En el modelo de la derecha, un salón se usa para dictar una única materia, pero la misma materia se dicta en varios salones (por ejemplo si una materia precisa de un salón con equipamiento especial (ej. Laboratorio) y hay varias clases paralelas de esa misma materia (en varios Laboratorios).

Ahora ¿qué pasaría si cruzamos las claves primarias de ambas transacciones, de modo que una tenga como clave foránea a la primaria de la otra y viceversa?

Si lo implementamos en GeneXus lo primero que apreciamos es que al salvar, no se muestran las flechas hacia arriba y hacia abajo indicando a las claves foráneas y atributos inferidos respectivamente.
Veamos ahora las tablas que se crearán con este diseño.

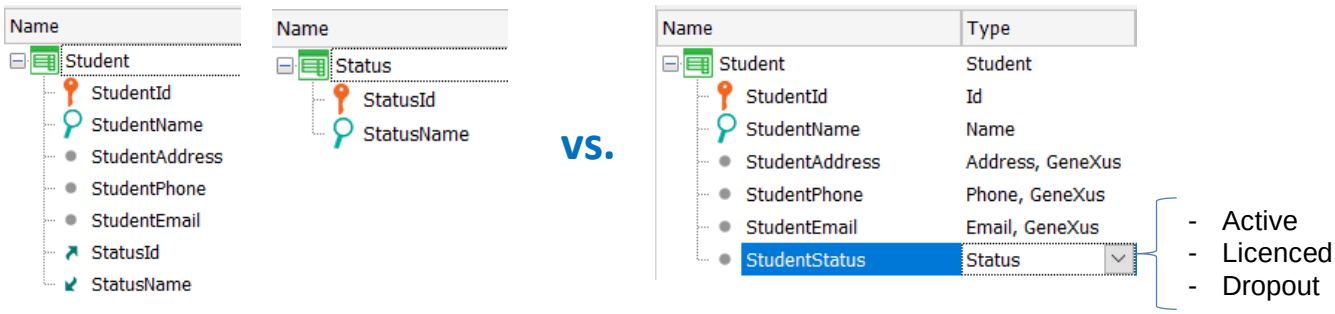
Comprobamos que se crea una única tabla! En este caso GeneXus creó la tabla Room, con RoomId como clave primaria y al resto de los atributos como atributos secundarios. ¿Qué pasó aquí?
La respuesta está en las dependencias funcionales de los atributos.

Al cruzarse los identificadores, en una tabla un atributo es clave primaria y por lo tanto todos los atributos de esa transacción dependen funcionalmente de él, inclusive el que es clave foránea. Pero en la otra transacción pasa lo mismo, ahora la clave primaria es otra y el que era clave primaria (que ahora es foránea) depende funcionalmente de la clave primaria de esa transacción. Por lo tanto, GeneXus debe tomar una decisión ya que los dos identificadores no pueden ser claves primarias a la vez dado el modelo establecido donde hay una doble dependencia funcional.

Debido a esto, elige a una sola de ellas como clave primaria y crea una única tabla con ese identificador, agregando al resto de los atributos como secundarios, es decir dependiendo funcionalmente de esa clave primaria que eligió.

Este resultado generalmente no es el esperado por el que realiza este modelo, que muchas veces surge al querer implementar una relación 1 a 1 entre las entidades, o simplemente por un error de modelado.

Lookup table vs. Enumerated domain



En muchas ocasiones, queremos asignar un valor a un atributo, seleccionándolo de una lista de valores.

En general para solucionar esto creamos una transacción donde almacenamos la entidad que vamos a elegir y los distintos registros de la tabla conforman la lista de valores. Para esto hacemos que el atributo al que le queremos asignar un valor de varios posibles sea una clave foránea a la transacción que contiene los valores y eventualmente podemos recuperar otros datos de esa entidad a través de la tabla extendida.

Un caso típico, siguiendo con nuestros ejemplos de la universidad, podría ser seleccionar el país de un alumno, o la carrera de una materia.

Sin embargo, si la lista de valores es fija, o cambia muy poco y en particular no tiene muchos datos, podríamos crear un dominio enumerado con esos valores. Por ejemplo, supongamos que un estudiante puede estar en estado “activo” si está cursando una carrera, en estado “licencia” si pidió un año sabático, o en estado “baja” si abandonó los cursos.

En ese caso podríamos crear un dominio Status con los 3 valores: A para active, L para licensed y D para dropout.
Sin embargo, puede pasar que más adelante se decida agregar un estado nuevo, por ejemplo cuando un alumno hace una pasantía de estudios en otra universidad y se quiere registrar ese estado que es nuevo y no coincide con ninguno de los anteriores.

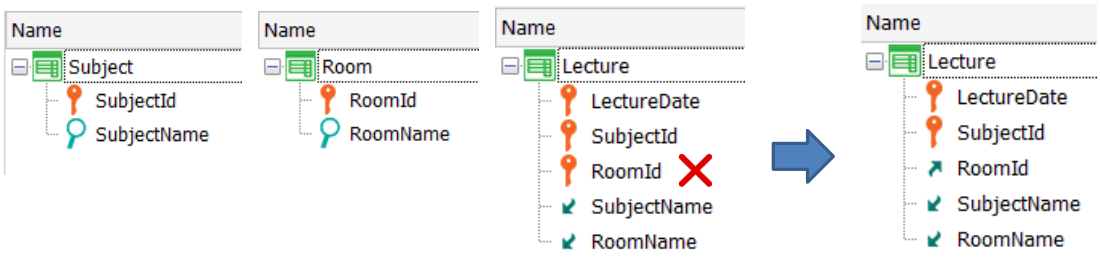
Si usamos un dominio enumerado y la universidad necesita agregar ese estado nuevo, deberemos agregar ese valor al dominio y luego regenerar la aplicación, es decir que los usuarios del sistema en la universidad no tienen la libertad de agregar un dato nuevo de su realidad fácilmente, sino que dependen de que el desarollador vuelva a generar los programas, testearlos y poner en producción la nueva versión.

Si además por algún motivo se hubiera utilizado un valor de un dominio enumerado como parte de una clave primaria compuesta en una transacción, no solo habría que regenerar la aplicación, sino también los datos de la base de datos, asegurando que no se pierdan los datos existentes al crear una tabla con una nueva clave primaria.

Mirado desde ese punto de vista no usaríamos nunca dominios enumerados, sin embargo puede haber algunas excepciones para valores que sabemos que no pueden cambiar, como los nombres de los días de la semana o los meses del año.

En resumen, salvo casos muy conocidos en los que estemos seguros de que los valores no van a cambiar, si tenemos la presunción de que aunque sea remota, existe la posibilidad de que estos datos puedan cambiar (como podría ser los estados de un país, o los nombres de países existentes), siempre es mejor práctica crear una tabla que contenga esos valores, lo que nos asegura que el usuario de la aplicación disponga de flexibilidad a la hora de actualizar los datos, aunque esto implique aumentar las estructuras a mantener en la base de datos.

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
10/24/2022	1	1
10/24/2022	1	2
10/24/2022	1	3

Cuando elegimos el identificador de una entidad, en ocasiones tenemos varias opciones, ya que puede haber varias clave candidatas que puedan ser elegidas como clave primaria, siguiendo los principios de unicidad (es decir que no puede haber dos registros con la misma clave) y de irreductibilidad (que la clave debe ser el conjunto mínimo de atributos que identifican a los registros en forma única). Es bastante común que se cumpla con la unicidad, pero no siempre se presta atención a la irreductibilidad y se terminan definiendo claves primarias con atributos de más.

Veamos esto con un ejemplo.

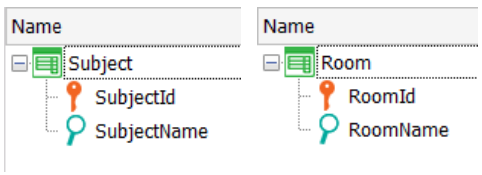
Supongamos que queremos modelar el caso de una materia que se dicta en un salón en una fecha determinada. Asumimos que esa materia se dicta únicamente una vez en esa fecha. Se modelaron las transacciones que se muestran, la transacción Lecture tiene una clave compuesta por LectureDate, SubjectId y RoomId, para “asegurar” que la combinación de la materia con la fecha y con el salón sea única.

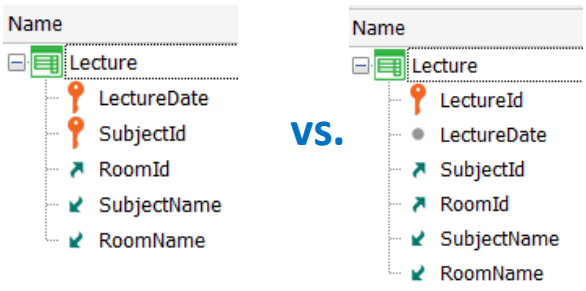
Sin embargo, si analizamos los datos, vemos que fue posible ingresar una clase para una materia en una fecha y que se dicte en varios salones a la vez, ya que basta con que uno de los componentes de la clave cambie de valor, para que los demás componentes de la clave puedan repetir el valor en distintos registros.

Claramente es innecesario que el RoomId forme parte de la clave, ya que sólo con el LectureDate y el SubjectId podemos identificar adecuadamente la clase para asegurar que no se repita la misma materia en la misma fecha. El atributo RoomId

puede estar como clave foránea.

Compound primary key vs. simple primary key





\+

Unique index: LectureDate, SubjectId

Una duda que nos puede surgir es que si en lugar de definir una clave compuesta en Lecture, formada por los atributos de la fecha y la materia cuya combinación queremos que sea única, definimos una clave artificial LectureId que por ejemplo sea un identificador numérico autonumerado.

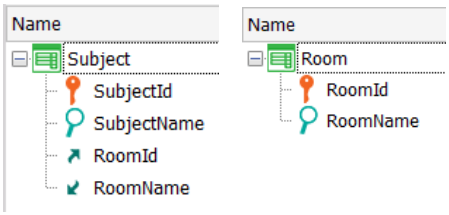
En ambos casos se está controlando la unicidad y la clave es irreductible, es decir es la mínima para identificar correctamente a los registros de la tabla.

Sin embargo, en el caso de la clave compuesta, no podremos modificar los valores de la fecha o de la materia porque son parte de la clave, y si queremos hacerlo, deberemos borrar el registro y definir otro.

En cambio en el caso de la clave con el identificador de la clase, sí podremos cambiar estos datos, aunque como contrapartida, al cambiarlos deberemos controlar que no ser repita la combinación de los valores de fecha y materia, ya que sería posible definir dos registros con distinto LectureId, pero el mismo par fecha-materia.

Para evitar esto, podríamos definir un índice único por fecha y materia en la tabla LECTURE.

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
?	

Un tema en el que GeneXus pone especial atención es en el de mantener una adecuada integridad referencial, de forma de asegurar la consistencia de los datos. Esto implica que no se posible la existencia de un valor de clave foránea que no exista como clave primaria en la tabla de origen, ni tampoco que sea posible eliminar un valor de una clave primaria que tenga registros relacionados donde esa clave es foránea.

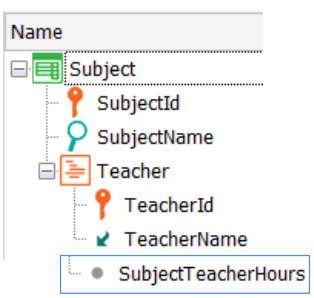
Cuando se actualizan datos mediante las transacciones ya sea ejecutando su form o un business component de esa transacción, el control de integridad referencial es automático.

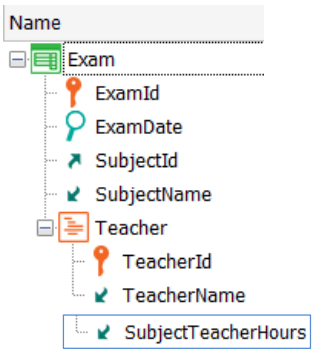
Sin embargo, hay algunos casos especiales en los que nos pueden parecer que el control se realiza, cuando en realidad no se cumple, o que se realizan controles indeseados y queremos evitarlos, o que automáticamente se sustituye un control por otro bajo ciertas condiciones.

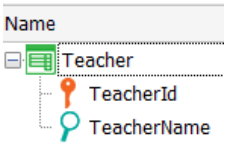
Analicemos algunos ejemplos que muestran estas situaciones.

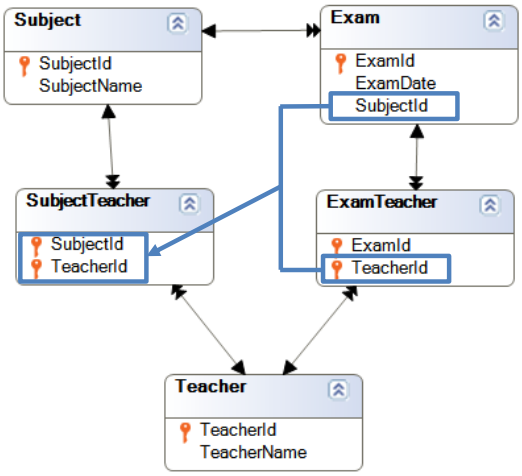
Referential integrity: under the hood

Case 1: Logic subordination









Supongamos que queremos registrar los exámenes de una cierta materia y el examen será tomado por uno o más profesores. Cada materia tiene registrados los profesores que la dictan.
Cuando agregamos un examen, debe controlarse que el profesor que se asigna al examen, dicte la materia del examen. ¿Esto está asegurado por el diseño?

La respuesta es no, ya que con este diseño no se realizará automáticamente el chequeo. Estamos pretendiendo que cuando se vaya a ingresar un registro en la tabla EXAMTEACHER, se controle que exista un registro en la tabla SUBJECTTEACHER con el valor de SubjectId correspondiente al del examen y con el valor del TeacherId correspondiente al profesor que se está insertando.

Si bien ambos valores se tienen en memoria, {SubjectId, TeacherId} no forman una clave foránea dado que no están en la misma tabla.

Sin embargo podemos decir que forman una clave foránea lógica aunque eso no exista a nivel de la base de datos relacional. Dado que existe una relación de subordinación lógica entre las tablas (y no subordinación física), GeneXus no realizará el control de integridad referencial que necesitamos.

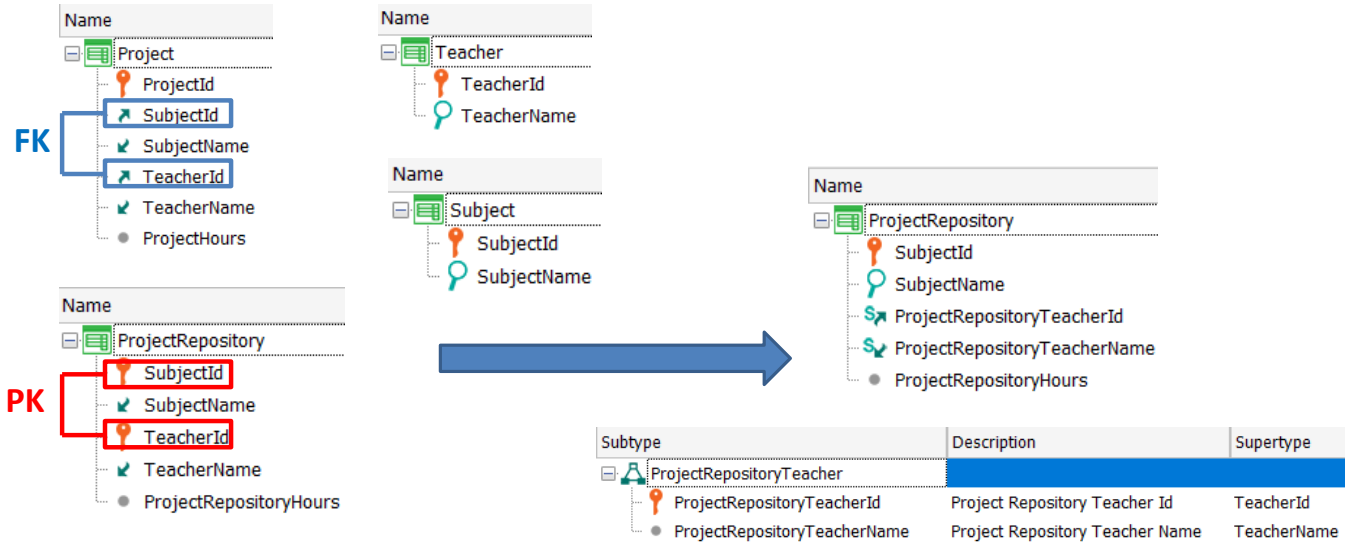
Una forma de resolver esto es definir una regla en Exam que invoque un procedimiento que realice el chequeo y luego con una regla Error evaluamos el resultado de la invocación.

Otra forma de solucionarlo es agregar al segundo nivel de la transacción Subject un atributo secundario, por ejemplo SubjectTeacherHours que registra las horas

que el docente tiene asignadas a esa materia, de manera de inferirlo en la transacción Exam y así forzar a GeneXus a darse cuenta de la relación.

Referential integrity: under the hood

Case 2: Unintended referential integrity checks



Supongamos ahora que en la realidad de la universidad que vinimos desarrollando, se desean registran los proyectos finales que se llevan a cabo en ciertas materias para aprobarla. Un proyecto tiene un identificador, un profesor asignado, una materia y la cantidad de horas asignadas a ese proyecto.

Además se desea tener un repositorios con todos los proyectos que se realizan en la universidad, de modo que al terminar el semestre los proyectos que se llevaron a cabo en dicho semestre, se ingresan al repositorio.
Para eso se creó una transacción ProjectRepository y se definió una clave primaria compuesta formada por el profesor y la materia del proyecto.

El problema con este diseño es que SubjectId y TeacherId forman en la transacción Project una clave foránea a ProjectRepository, de modo que al intentarse ingresar un proyecto en la transacción Project, se exige que el proyecto esté previamente ingresado en la tabla PROJECTREPOSITORY, lo cual es imposible, porque primero se registra el proyecto y recién después a fin de semestre se agrega al repositorio.

Para evitar esto, podemos crear un grupo de subtipos ProjectRepositoryTeacher, con los subtipos ProjectRepositoryTeacherId subtipo de TeacherId y ProjectRepositoryTeacherName subtipo de TeacherName.
Luego, sustituimos los atributos TeacherId y TeacherName en la transaction ProjectRepository por los subtipos correspondientes.

De esta manera, lo que hacermos es con el subtipo, cambiar el nombre de TeacherId en la tabla en la que este atributo es parte de la clave primaria.

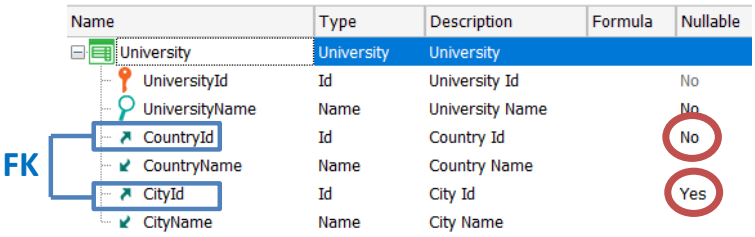
Esto no evita que cuando ingresamos el identificador de profesor en la transacción ProjectRepository, GeneXus verifique si existe el valor en la tabla TEACHER.

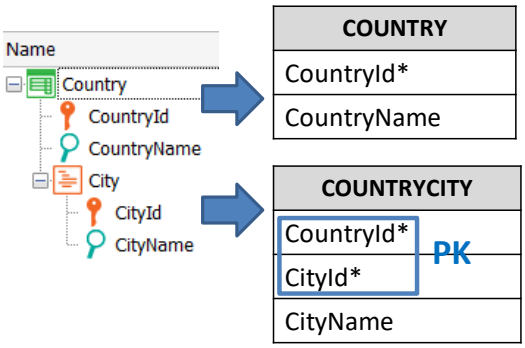
Pero este cambio de nombre evita que el par de atributos {SubjectId, TeacherId} sea identificado como clave foránea en Project, ya que ahora los nombres de los atributos que forman la clave primaria en ProjectRepository es diferente.

De esta manera, utlizando subtipos, pudimos evitar que se realice un control de integridad referencial que no era deseado en nuestra realidad.

Referential integrity: under the hood

Case 3: Compound foreign key partially nullated





Cuando ponemos como nullable un atributo que es clave foránea simple, al ingresar un registro si no ingresamos el valor, no se realizará el control de integridad referencial. Si en cambio ingresamos un valor, sí se controlará que el valor ingresado en la clave foránea exista como clave primaria en la tabla correspondiente.

Sin embargo, cuando la clave foránea es compuesta, podríamos setear como nullable solamente a algunos los atributos de la clave.

En el ejemplo, una universidad pertenece a una ciudad. Las ciudades se registran en la tabla COUNTRYCITY, que tiene como clave primaria compuesta a CountryId y CityId. En la transacción University, los atributos CountryId y CityId forman una clave foránea compuesta y en particular podemos setear como nullables a ambos atributos, a ninguno o solamente a uno de ellos.

Supongamos que al momento de ingresar una universidad conocemos el país pero no la ciudad a la que pertenece, por lo que seteamos al atributo CityId con Nullable en Yes.

¿Qué sucederá cuando ingresemos una universidad? ¿Se hará algún tipo de control de integridad referencial?
La respuesta es sí. Si al ingresar una universidad no especificamos el valor de CityId, no se realizará el control de integridad para verificar si la ciudad existe en la tabla de ciudades, pero como el atributo CountryId aún tiene la propiedad Nullable en No, se realizará un control de integridad sobre la tabla COUNTRY para verificar que el país ingresado, exista como país en la tabla de países.

Este control de integridad sobre la tabla COUNTRY no se realizaba si ambos atributos de la clave foránea compuesta tenían el valor Nullable en No, sino que el control se realizaba únicamente sobre la tabla COUNTRYCITY.

Es decir que GeneXus agregó un control de integridad que originalmente no se hacía, dado que el atributo CountryId sigue siendo clave foránea respecto a la tabla COUNTRY.

Esto nos demuestra nuevamente el objetivo de mantener siempre los datos consistentes, aún en casos en que se intenten deshabilitar ciertos controles.

En este resumen hemos intentado integrar varios temas relacionados con el diseño de transacciones y su impacto en la base de datos y en la funcionalidad de la aplicación. Lo invitamos a profundizar en algunos de estos temas en otros videos publicados específicos para cada tema.

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