Paso del E-R a tablas

Fernando Cano

Mayo 2012

1. Entidades

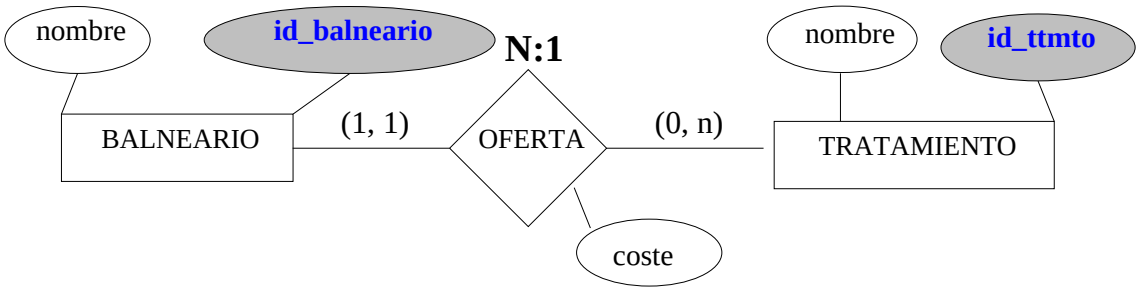
Cada entidad del modelo E-R genera una tabla. Dicha tabla contiene como columnas cada uno de los atributos de la entidad. Además puede contener otras columnas fruto de relaciones, normalmente 1:N, con otras entidades.

2. Relaciones

Dependiendo de cómo se definan las cardinalidades de las relaciones, éstas pueden generar una tabla o por el contrario traducirse en columnas dentro de las tablas asociadas a las entidades que participan en dicha relación. Vamos a verlo en diferentes casos.

1. **Relaciones 1:N**

   Empecemos por la relaciones **1:N**. Veamos un ejemplo, en el que un balneario oferta una serie de tratamientos. Cada tratamiento es ofertado por uno y solo un solo balneario.



   Las tablas resultantes serían:

```
CREATE TABLE balnearios (
    id_balneario    smallint Not Null Primary Key ,
    nombre          text
);

CREATE TABLE tratamientos (
    id_tratamiento  smallint Not Null Primary Key ,
    nombre          text,
    id_balneario    smallint not null references balnearios,
    coste           decimal (6,2)
);
```

   Como se ve el atributo cualificativo *coste* queda dentro de la tabla de *tratamientos*. Realmente dicho atributo es un atributo del tratamiento, más que de la relación que se establece entre balneario y

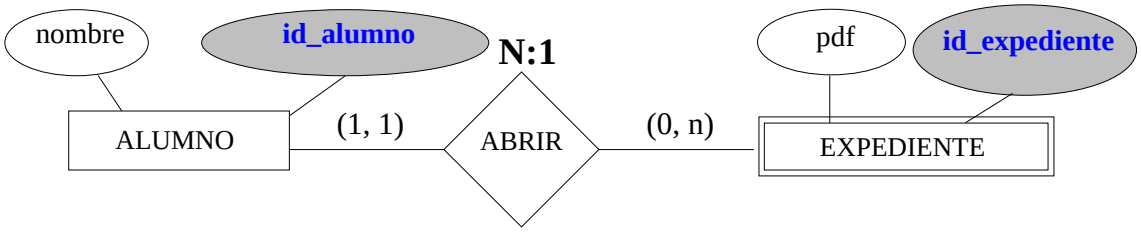
tratamiento. De hecho lo normal es que *coste* lo hubiéramos dibujado como un atributo de la entidad tratamiento, ya que no depende del balneario que oferte el tratamiento.

Otro detalle importante es que debido a la cardinalidad $(1, 1)$, que refleja que un tratamiento se relaciona con uno y solo un balneario, el identificador de balneario de la tabla tratamientos no puede contener un valor nulo. Si se tratase de una cardinalidad $(0, 1)$, entonces el identificador de balneario podría ser nulo.

NOTA: Como se puede observar se han definido las claves primarias como `smallint Not Null Primary Key`, y este el el criterio que se sigue en todos los ejemplos. Aunque la alternativa de definir las PKs como *autonumérico* o *autoincremental* o *serial* como se hace en Postgres casi siempre es mucho mejor. Pero como cada gestor de bases de datos lo hace de forma diferente no vamos a profundizar aquí en este un tema.

2. **Relaciones 1:N con entidades dependientes con clave**

   Cuando nos encontramos con una entidad dependiente existencialmente pueden presentarse varios casos. Uno de ellos es cuando la entidad dependiente tiene atributos suficientes para formar clave. Vamos a ver un ejemplo en el que a un alumno se le pueden abrir varios expedientes y cada uno tiene su identificador y el pdf con el texto del expediente.

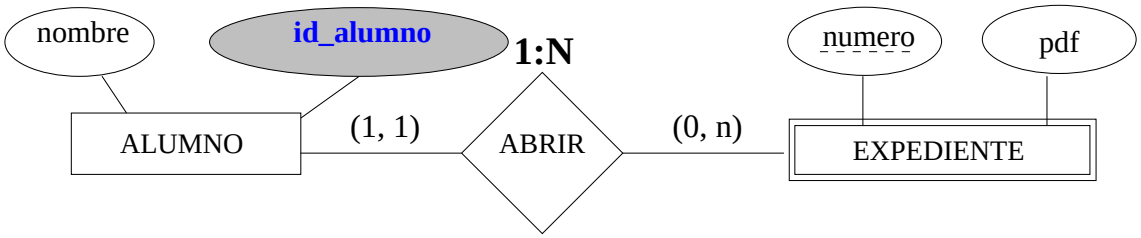


   Las tablas resultantes serían:

```
CREATE TABLE alumnos (
    id_alumno       smallint Not Null Primary Key ,
    nombre          text
);

CREATE TABLE expedientes (
    id_expediente   smallint Not Null Primary Key ,
    pdf             text,
    id_alumno       smallint not null references alumnos
);
```

3. **Relaciones 1:N con entidad con dependencia-id**

   En este caso la entidad dependiente no tiene atributos suficientes para formar clave, pero cuenta con un *discriminador*, en nuestro ejemplo el atributo *numero*, que indica el número de expediente de cada alumno (1, 2, 3, ...). Evidentemente el atributo *numero* se podría repetir para los alumnos a los que se les abra un expediente.

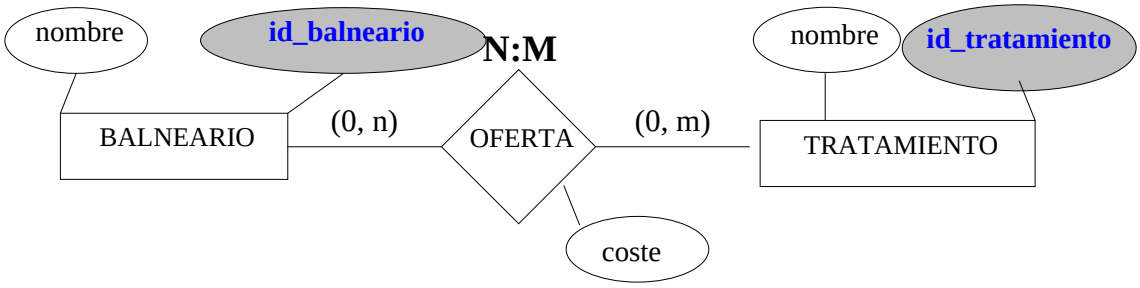


Las tablas resultantes serían:

```
CREATE TABLE alumnos (
    id_alumno       smallint not null Primary Key ,
    nombre          text
);

CREATE TABLE expedientes (
    id_alumno       smallint not null references alumnos,
    numero          smallint not null ,
    pdf             text,
    primary key (id_alumno, numero)
);
```

4. **Relaciones N:M**

   Vamos a modificar un poco el ejemplo de los balnearios y tratamientos. Ahora, un mismo tratamiento puede ser ofertado por diferentes balnearios y en cada uno puede tener diferentes costes.



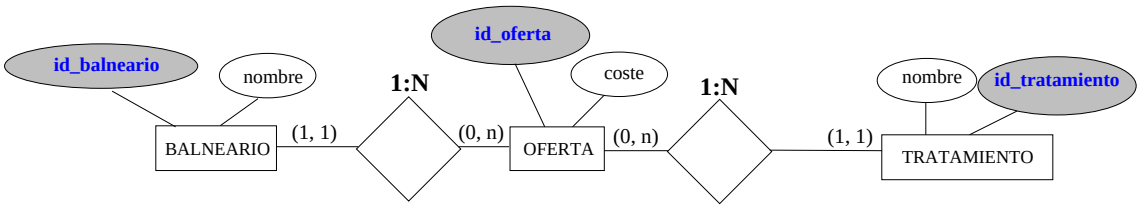
   Las tablas resultantes serían:

```
CREATE TABLE balnearios (
    id_balneario    smallint Not Null Primary Key ,
    nombre          text
);

CREATE TABLE tratamientos (
    id_tratamiento  smallint Not Null Primary Key ,
    nombre          text,
);

CREATE TABLE ofertas (
    id_balneario    smallint not null references balnearios,
    id_tratamiento  smallint not null references tratamientos,
    coste           decimal (6,2),
    primary key (id_balneario, id_tratamiento)
);
```

   Como se ve el atributo cualificativo *coste* ahora queda en la tabla de *ofertas*. Realmente dicho atributo depende de la relación que se establece entre un balneario y un tratamiento.

   Aunque esta es la forma más ortodoxa generación de tablas a partir de una relación N:M en muchos casos no es la mejor solución. La alternativa pasa por entender la relación entre balneario y tratamiento como una entidad. En nuestro caso esto se traduce en entender la oferta como una entidad en sí misma. Hay muchos casos en los que resulta más evidentes convertir una relación N:M en una entidad: matrimonio, partido, examen, etc. otros pueden ser menos intuitivos. Pero en cualquier caso siempre es posible. Una vez que creamos esta relación *intermedia* nos aparece la

necesidad de inventarnos una clave para ello. En nuestro ejemplo a la entidad *oferta* le añadimos la clave *id_oferta*. De esta forma el diagrama quedaría como sigue:



Pero ahora se nos presenta otro problema: el par *id_balneario, id_tratamiento* ya no es clave primaria por lo que se podría dar el caso de que dos ofertas diferentes ofertar el mismo tratamiento en el mismo balneario incluso a distinto coste. Para solucionar esto, es decir, imponer la restricción de unicidad de *id_balneario, id_tratamiento* basta con definirlas como *UNIQUE*. Esto en SQL es muy sencillo, en otros gestores como Access no es tan inmediato.

Con todo lo anterior nuestra tabla *ofertas* quedaría asi:

```
CREATE TABLE ofertas (
    id_oferta       smallint Not Null Primary Key ,
    id_balneario    smallint Not Null References balnearios,
    id_tratamiento  smallint Not Null References tratamientos,
    coste           decimal (6,2),
    unique (id_balneario, id_tratamiento)
);
```

Esto se traduce internamente en la creación de un índice para la clave primaria y otro para el valor único, es decir que tenemos un índice añadido. Realmente el índice que se añade está basado en un solo campo *id_oferta*, eso no quita un incremento de espacio de memoria y del coste computacional, pero por otro nos resuelve problemas más importantes como los que se mencionan a continuación:

- Foreign Keys. Siempre resulta más mucho más sencillo y más funcional establecer FKs a claves formadas por una sola columna. Cuando se crean FKs a claves formadas por más de una columna se pueden cometer errores que son difíciles de resolver. Siguiendo con nuestro si definiéramos en una nueva tabla una FK a la clave de *ofertas* formada por *id_balneario, id_tratamiento*, y lo hiciéramos de la forma:

  ```
  foreign key (id_tratamiento, id_balneario) references ofertas
  ```

  cometeríamos un error ya que el índice de la tabla *ofertas* está construido en base a *id_balneario, id_tratamiento* y no en base a *id_tratamiento, id_balneario* que generaría un índice diferente y la integridad referencial fallaría sin ser conscientes de porqué. Este problema en gestores como Access es aún más difícil de encontrar, como nos dice la experiencia.
- Updates. Imaginemos una aplicación, de escritorio o web, en la que queremos modificar una oferta en la que cometimos un error ya que asociamos el balneario 1 con el tratamiento 2 cuando realmente es el tratamiento 3. Cuando en un *grid* de nuestro interface tenemos el registro (1,2) y lo modificamos a los valores (1,3) la orden SQL que se debe enviar al servidor sería

  ```
  update ofetas
  set id_tratamiento = 3
  where id_balneario = 1 and id_tratamiento =2;
  ```

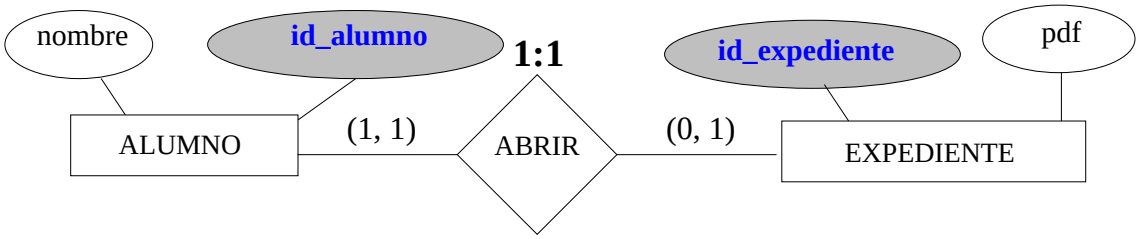
  Pero para ello es necesario almacenar en algún sitio el antiguo valor de *id_tratamiento* (2). Y esto nos plantea un problema de programación en muchos interfaces. Más aún se el *uptate* es

de múltiples filas. Si la tabla *ofertas* tuviera como clave el *id_oferta* este problema no existiría ya que nunca se modificaría el valor de la clave. el tratamiento 3.

- Claves internas. Algunos servidores de bases de datos permiten utilizar unas claves internas, como los OIDs (identificadores de objetos) de Postgres, que permiten agilizar mucho las operaciones con los registros de las tablas. Se utiliza el propio IOD como clave, por tanto una clave múltiple nos nos valdría.
- Respeto a los padres. Teniendo en cuenta que C.J. Date es uno de los más prestigiosos padres de las bases de datos y que en su libro *Fundamento de Base de datos* recomienda que las tablas tengan la clave formada por una única columna, no parece mala idea seguir su recomendación aunque no lleguemos a entender bien sus razonamientos.

5. **Relaciones 1:1**

   En este tipo de relaciones las entidades podrían utilizar la misma clave o cada cual la suya. En el siguiente ejemplo un alumno puede tener o no un expediente, pero sólo uno. Cada entidad tiene su propia clave.



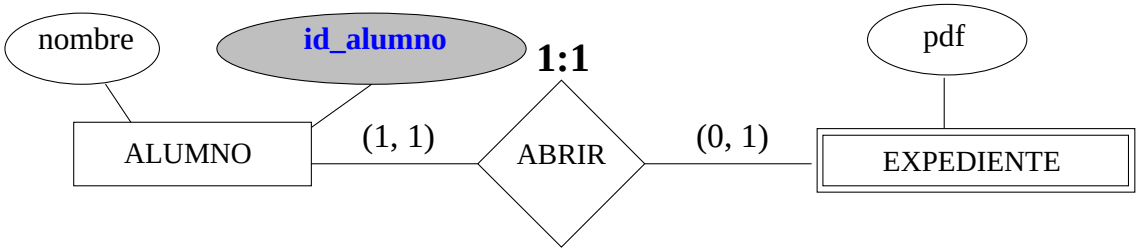
   Las tablas resultantes serían:

```
CREATE TABLE alumnos (
    id_alumno       smallint not null Primary Key ,
    nombre          text
);

CREATE TABLE expedientes (
    id_expediente   smallint not null primary key,
    id_alumno       smallint not null unique references alumnos ,
    pdf             text
);
```

   Como se puede observar hemos añadido la restricción *unique* al id_alumno para controlar que un alumno no pueda tener más de un expediente.

6. **Relaciones 1:1 con entidad con dependencia-id**

   En este caso la entidad con dependencia-id no tiene atributos suficientes para formar clave, pero podemos *chupar* la clave de la entidad de la cual depende. Realmente es una alternativa mejor a la que hemos visto antes, ya que nos ahorramos una columna y además un índice asociado al *unique*. Y por otro lado cuando hagamos el join de las tablas, podemos utilizar la clausula *using* (using(id_alumno)).

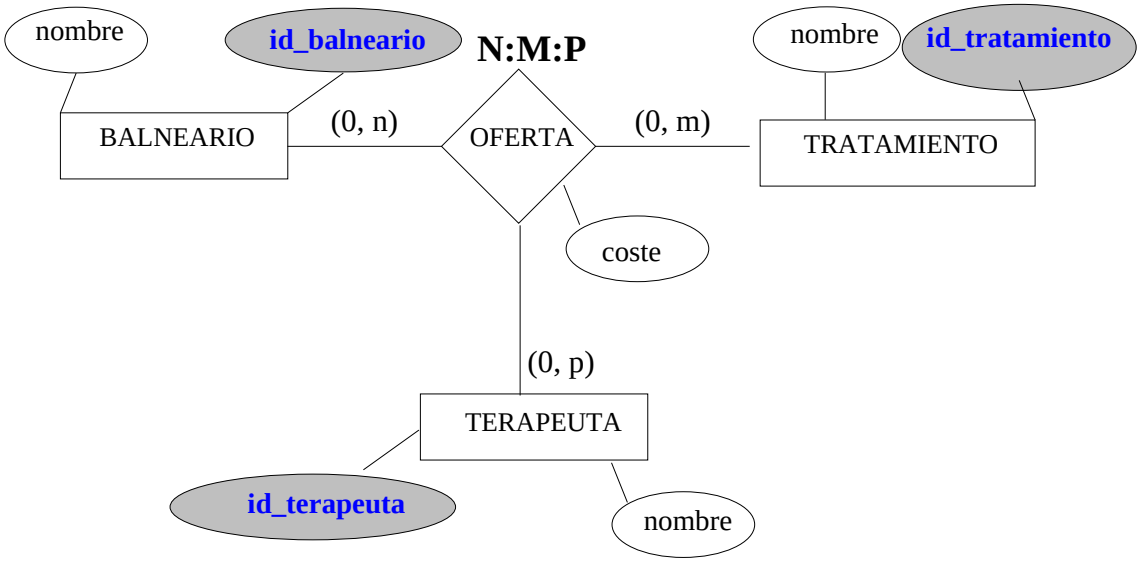


Las tablas resultantes serían:

```
CREATE TABLE alumnos (
    id_alumno       smallint not null Primary Key ,
    nombre          text
);

CREATE TABLE expedientes (
    id_alumno       smallint not null primary key
                            references alumnos,
    pdf             text
);
```

7. **Relaciones N:M:P**

   Vamos a incluir información sobre los terapeutas que imparten los tratamientos. Si no imponemos ninguna restricción el diagrama quedaría como sigue.



Las tablas resultantes serían:

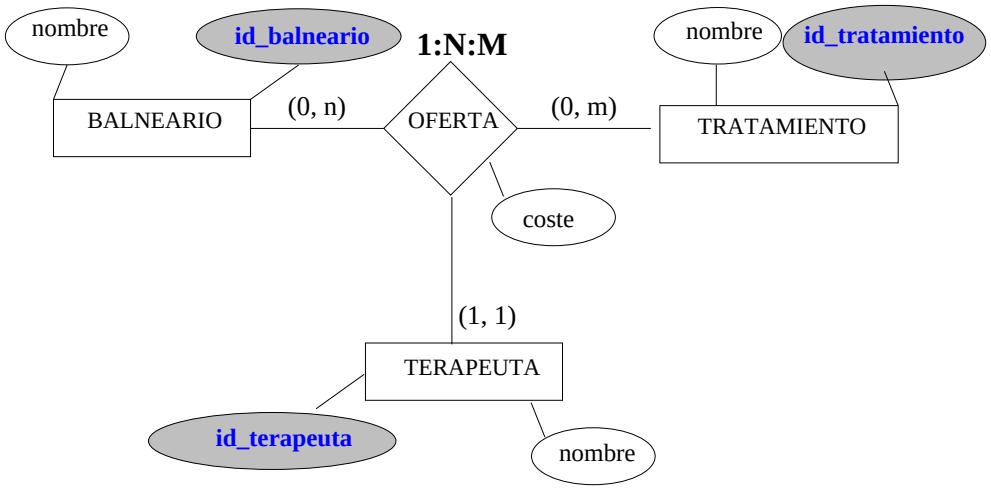
```
CREATE TABLE balnearios (
    id_balneario    smallint Not Null Primary Key ,
    nombre          text
);

CREATE TABLE tratamientos (
    id_tratamiento  smallint Not Null Primary Key ,
    nombre          text,
);

CREATE TABLE terapeutas (
    id_terapeuta    smallint Not Null Primary Key ,
    nombre          text,
);

CREATE TABLE ofertas (
    id_balneario    smallint not null references balnearios,
    id_tratamiento  smallint not null references tratamientos,
    id_terapeuta    smallint not null references terapeutas,
    coste           decimal (6,2),
    primary key (id_balneario, id_tratamiento, id_terapeuta)
);
```

En este caso también podríamos aplicar el criterio utilizado en las relaciones N:M y convertir la oferta en una entidad con lo que la tabla oferta quedaría de la forma:

```
 CREATE TABLE ofertas (
    id_oferta       smallint Not Null Primary Key ,
    id_balneario    smallint not null references balnearios,
    id_tratamiento  smallint not null references tratamientos,
    id_terapeuta    smallint not null references terapeutas,
    coste           decimal (6,2),
    unique (id_balneario, id_tratamiento, id_terapeuta)
);
```

8. **Relaciones 1:N:M**

Vamos a imponer la restricción de que un tratamiento en un balneario lo aplica un solo terapeuta.



Las tablas resultantes serían:

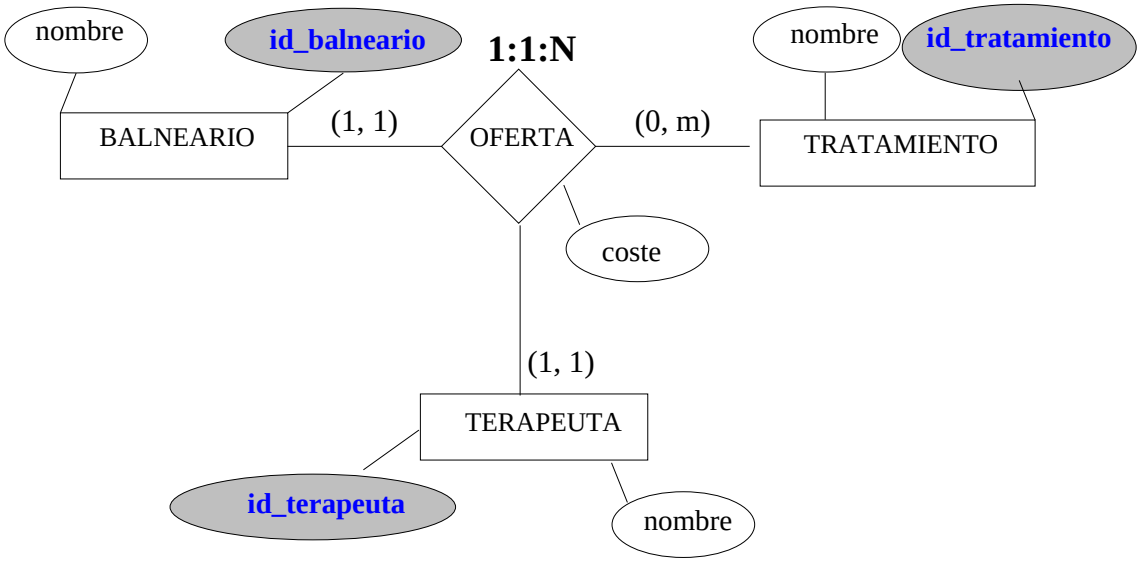
```
CREATE TABLE balnearios (
    id_balneario    smallint Not Null Primary Key ,
    nombre          text
);

CREATE TABLE tratamientos (
    id_tratamiento  smallint Not Null Primary Key ,
    nombre          text,
);

CREATE TABLE terapeutas (
    id_terapeuta    smallint Not Null Primary Key ,
    nombre          text,
);

CREATE TABLE ofertas (
    id_balneario    smallint not null references balnearios,
    id_tratamiento  smallint not null references tratamientos,
    id_terapeuta    smallint not null references terapeutas,
    coste           decimal (6,2),
    primary key (id_balneario, id_tratamiento)
);
```

En este caso convertir la relación oferta en entidad nos generaría la tabla ofertas así:

```
 CREATE TABLE ofertas (
    id_oferta       smallint Not Null Primary Key ,
    id_balneario    smallint not null references balnearios,
    id_tratamiento  smallint not null references tratamientos,
    id_terapeuta    smallint not null references terapeutas,
    coste           decimal (6,2),
    unique (id_balneario, id_tratamiento)
);
```

9. **Relaciones 1:1:N**

Vamos a imponer la restricción de que un tratamiento solo se oferta en un un balneario y solo lo aplica un solo terapeuta.



Las tablas resultantes serían:

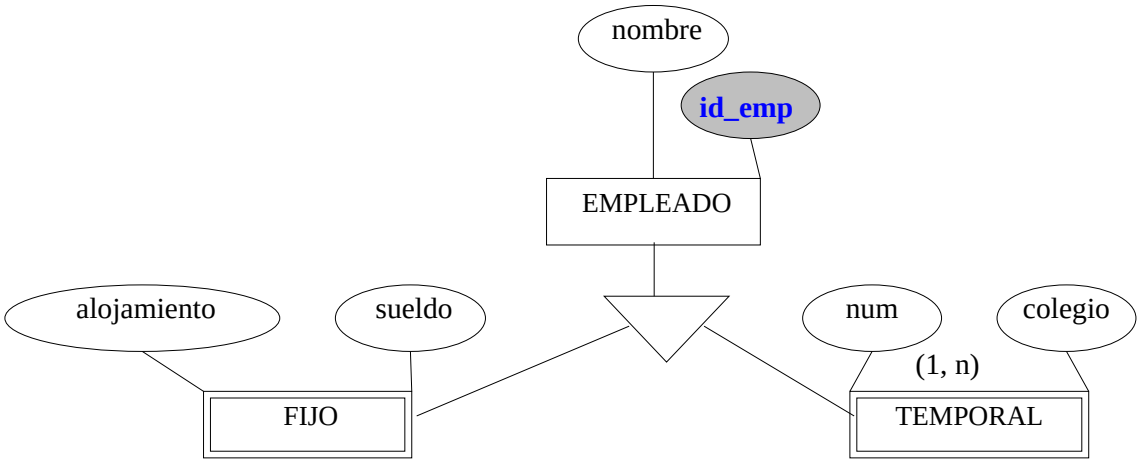
```
CREATE TABLE balnearios (
    id_balneario    smallint Not Null Primary Key ,
    nombre          text
);


CREATE TABLE terapeutas (
    id_terapeuta    smallint Not Null Primary Key ,
    nombre          text,
);


CREATE TABLE tratamientos (
    id_tratamiento  smallint Not Null Primary Key ,
    nombre          text,
    id_balneario    smallint not null references balnearios,
    id_terapeuta    smallint not null references terapeutas,
    coste           decimal (6,2),
);
```

Como se puede observar en este caso la relación no genera tabla, al igual que en el caso de las 1:N. De hecho esto se puede generalizar a las relaciones del tipo 1:1:...:1:N.

Ahora al no generar tabla la relación el considerar la relación como una entidad nos generaría las mismas tablas.

10. **Relaciones ISA**

Dada una relación ISA, normalmente se generan las siguientes tablas: una para la entidad de orden superior, con los atributos comunes, y otra tabla para casa una de las entidades subordinadas, con la misma clave que la entidad superior y los atributos propios de esa sub-entidad. Veámoslo con el siguiente ejemplo.



Las tablas resultantes serían:

```
CREATE TABLE empleado (
    id_empleado     smallint not null Primary Key ,
    nombre          text
);

CREATE TABLE fijo (
    id_empleado     smallint not null primary key
                            references empleados,
    sueldo          decimal (6,2),
    alojamiento     text
);

CREATE TABLE temporal (
    id_empleado     smallint not null primary key
                            references empleados,
    colegio         text,
    num             int
);
```

En algún caso, poco frecuente, se podrían generar únicamente tablas por cada sub-entidad, añadiendo en cada una los atributos comunes de la super-entidad. Como ejemplo:

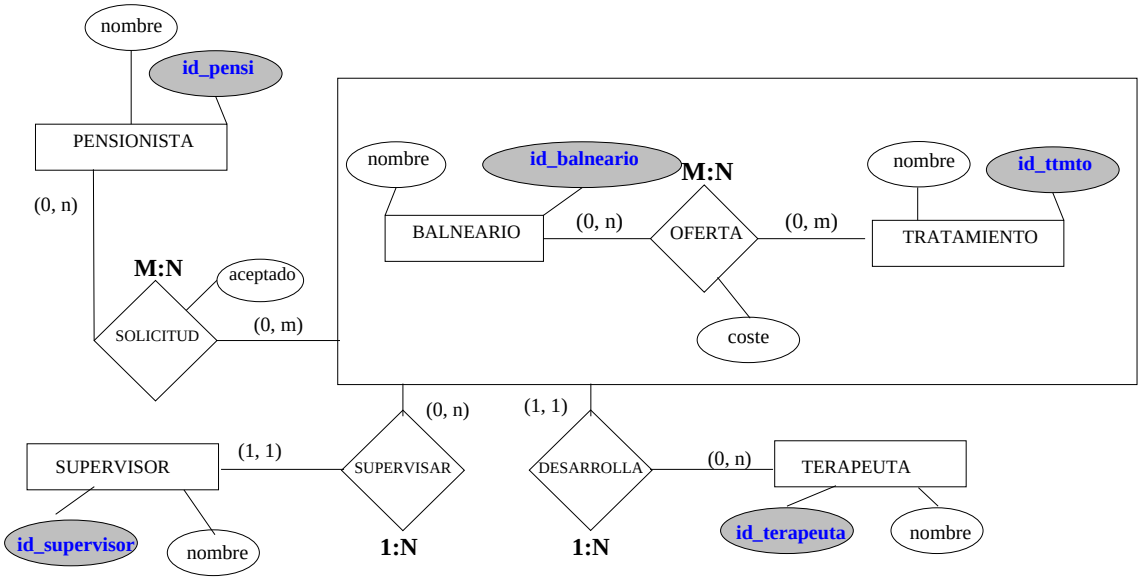
```
CREATE TABLE fijo (
    id_empleado     smallint not null primary key,
    nombre          text
    sueldo          decimal (6,2),
    alojamiento     text
);

CREATE TABLE temporal (
    id_empleado     smallint not null primary key,
    nombre          text
    colegio         decimal (6,2),
    num             int
);
```

El problema de esta opción es que no se puede establecer una integridad referencial a la entidad empleado. Dado que no podemos especificar mediante una foreign key que un id_empleado debe hacer referencia a un identificador de la tabla de fijos o de la de temporales. Además cada vez que quisiéramos hacer una consulta sobre los empleados en general deberíamos hacer consultas de *unión*.

Aunque en el diagrama se especifique *totalidad* o *solapamiento*, a la hora de crear tablas no podemos implementarlo. Para hacerlo sería necesario añadir código, ya sea mediante triggers o funciones.

11. **Agregaciones**

Para explicar la generación de tablas asociadas a una agregación vamos a mostrar el siguiente ejemplo. Los balnearios ofertan tratamientos, los cuales son solicitados por los pensionistas. Los pensionistas pueden solicitar distintos pares de balneario-tratamiento, pero para ello, dicho tratamiento debe ser ofertado por ese balneario, de ahí la agregación. Por otro lado cada tratamiento de cada balneario es supervisado por un único supervisor. Y cada terapeuta se la asocia un único tratamiento en un único balneario.



Las tablas resultantes serían:

```
CREATE TABLE balnearios (
    id_balneario    smallint Not Null Primary Key ,
    nombre          text
);

CREATE TABLE tratamientos (
    id_tratamiento  smallint Not Null Primary Key ,
    nombre          text,
);

CREATE TABLE supervisores (
    id_supervisor   smallint Not Null Primary Key ,
    nombre          text,
);

CREATE TABLE ofertas (
    id_tratamiento  smallint Not Null references tratamientos ,
    id_balneario    smallint not null references balnearios,
    id_supervisor   smallint not null references supervisores,
    coste           decimal (6,2),
    primary key ( id_tratamiento, id_balneario)
);

CREATE TABLE pensionistas (
    id_pensionista  smallint Not Null Primary Key ,
    nombre          text,
);

CREATE TABLE solicitudes (
    id_tratamiento  smallint not null,
    id_balneario    smallint not null,
    id_pensionista  smallint not null references pensionistas,
    aceptada        boolean,
    foreign key (id_tratamiento, id_balneario)
    references ofertas (id_tratamiento, id_balneario),
    primary key (id_tratamiento, id_balneario,id_pensionista)
);

CREATE TABLE terapeutas (
    id_terapeuta    smallint Not Null Primary Key ,
    nombre          text,
    id_tratamiento  smallint not null,
    id_balneario    smallint not null,
    foreign key (id_tratamiento, id_balneario)
    references ofertas (id_tratamiento, id_balneario)
);
```

Como se ve, la relación solicitud genera tabla por ser de tipo N:M, mientras que la relación supervisar no la genera por ser del tipo 1:N. La clave la la entidad *balneario-tratamiento* que surge fruto de la agregación, tiene como clave la primary key de la tabla de *ofertas* y se trata como si fuera una entidad más.

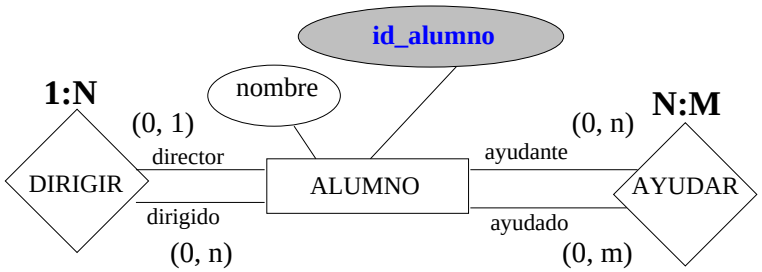
Hay un pequeño detalle que nos puede generar algunos quebraderos de cabeza. Si al crear la tabla de los terapeutas hubiéramos optado por esta orden:

```
CREATE TABLE terapeutas (
    id_terapeuta    smallint Not Null Primary Key ,
    nombre          text,
    id_tratamiento  smallint not null,
    id_balneario    smallint not null,
    foreign key (id_balneario, id_tratamiento) references ofertas
);
```

en la que observamos dos diferencias. La primera es que hemos cambiado el orden de los campos en la foreign key, ahora es *id_balneario, id_tratamiento*. Y no fue así como se crearon en la tabla de ofertas. Además es difícil darse cuenta ya que hemos omitido los nombres de los campos de la tabla ofertas. Esto generaría problemas al estar cruzados los campos que además son del mismo tipo. Todo esto se solucionaría, como ya se ha comentado antes convirtiendo alguna de las relaciones en entidades y además de ser más funcional quedaría todo muchísimo más claro y en consecuencia con menor riesgo de introducir errores oscuros.

12. **Relaciones con roles**

El caso en el que una entidad se relaciona consigo misma, sigue las mismas pautas que cuando son dos entidades diferentes. Lo que cambia es la forma en la que se generan las tablas en SQL.



Las tablas resultantes serían:

```
CREATE TABLE alumnos (
    id_alumno       smallint not null Primary Key ,
    nombre          text
);

ALTER TABLE alumnos ADD id_director smallint references alumnos;

CREATE TABLE ayudar (
    id_ayudado      smallint not null references alumnos ,
    id_ayudante     smallint not null references alumnos,
    primary key (id_ayudado, id_ayudante)
);
```

En este caso la *foreign key* del alumno que hace la función de director se establece después de crear la tabla mediante un *alter table*.