Diseño lógico de datos:

Ejercicio de práctica sobre el modelo relacional

por Fernando Dodino, Juan Zaffaroni

Versión 2.0
Septiembre 2013

Indice

El enunciado

Tenemos dos partes:

- el enunciado original
- y dos requerimientos para el diseño de datos

Diseño de datos

En la mayoría de los paradigmas se separan

- abstracciones de **datos** que representan información
- y los **procesos**: abstracciones de control del flujo que manipulan esos datos

Ya sabemos qué es diseñar

- es encontrar componentes
- darles responsabilidades
- y conocer las relaciones que tiene con otras partes/componentes

En el diseño lógico de datos definimos

- las entidades que representan partes de un sistema
- sus relaciones
- y ciertas responsabilidades como veremos más adelante.

Qué busco al diseñar los datos

1. Entender mejor los requerimientos que pide el negocio
2. Evitar redundancia de datos

Modelo conceptual

El modelo conceptual define las entidades y sus relaciones en base al negocio (es fácil de validar por el usuario). Es independiente de la tecnología. *Ejemplos:*

- un cliente tiene muchas sucursales,
- cada sucursal está ubicada en un domicilio.
- una materia de una facultad puede tener 0, 1 ó muchas correlativas.

Diseño lógico

Aquí definimos el esquema en el cual vamos a trabajar sin depender de un fabricante específico. Ej: trabajamos en un esquema relacional sin decidir si lo implementaremos en Oracle, MySQL o SQLServer.

Diseño físico

El diseño físico se implementa en un motor de base de datos que, además de soportar un determinado esquema, impone ciertas restricciones propias del fabricante, como el lenguaje de manipulación de datos (DML - data manipulation language) o el que nos permite crear las

entidades (DDL - data definition language). El estándar para el modelo relacional es el SQL aunque cada DBMS implementa su propio subconjunto.

Aunque en el trabajo comercial se los suele utilizar como sinónimos, en lo académico conviene hacer una pequeña distinción semántica:

- la **base de datos** es el conjunto de datos de mi sistema, que tiene un soporte físico en un medio persistente (podemos pensar en uno o n archivos en un disco rígido, o en cualquier otro medio de almacenamiento).
- del **sistema que administra esa base de datos** consiste en un conjunto de programas que ofrecen muchos servicios:
    - manejo de la concurrencia de usuarios
    - seguridad
    - una interfaz interactiva que permite ejecutar consultas para manipular datos, donde cada consulta
        - se recibe
        - se compila
        - y optimiza
    - la administración del catálogo de entidades
    - el manejo de transaccionalidad de un pedido, que permite garantizar la atomicidad de un conjunto de acciones (si alguna de las operaciones da error se cancelan todas las demás)
    - y utilitarios específicos de administración para analizar, chequear, reorganizar la información de la BD, tareas que recaen en el rol del DBA (Database administrator)

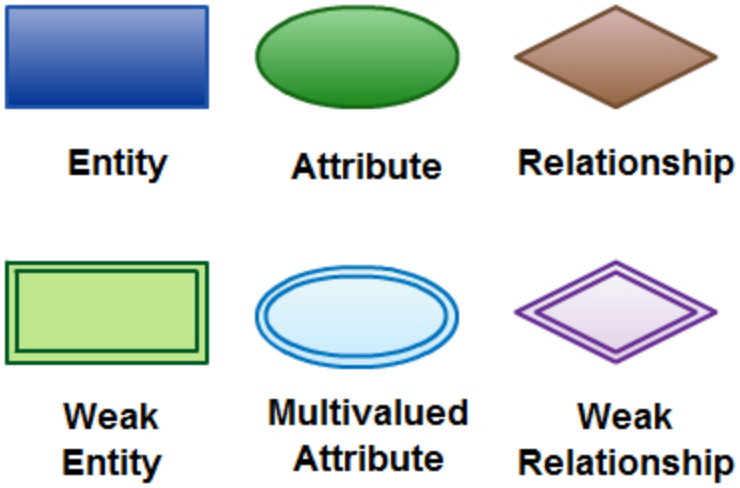
Dado que nuestro objetivo está puesto en el diseño de los datos no vamos a investigar cómo es la arquitectura que tiene un sistema de BD, en todo caso pueden investigarlo uds.

El esquema en el cual se guardan los datos puede seguir diferentes tipos de modelos:

- jerárquico o su variante en red
- de objetos
- documentales, orientada a grafos, columnares o par clave-valor, que veremos en otro momento,
- y el **relacional**, que vamos a introducir a continuación.

Repaso Diagrama Entidad-Relación (DER)

Basándonos en este tutorial



definimos entonces:

- **Entidad**: cualquier cosa que tenga relevancia para nuestro sistema. Los proyectos, las tareas, los impuestos, las complejidades...
- **Atributos**: son las propiedades o características que describen una entidad. Un cliente tiene nombre y cuit, un auto tiene modelo, marca y color, etc.
    - Atributos multivaluados: son atributos que pueden tomar más de un valor (direcciones de mail de un empleado, subordinados de un jefe, entre otros)

- **Instancia de una entidad**
    - *Ocurrencia particular de una entidad.*
- **Relaciones entre entidades**
    - *Representan asociaciones del mundo real entre entidades*
    - *Se puede representar con un verbo o preposición que conecta dos entidades.*

Características de las Relaciones

Grado

- *Unarias o Recursivas: Son relaciones que asocian a una misma entidad.*
- *Binarias: Relaciones que asocian a dos entidades.*
- *N-arias: Relaciones que asocian a N-entidades*

Cardinalidad

- *Describe la cantidad de instancias de entidad permitidas en un relación entre dos entidades*
    - *1 a 1*: un cliente tiene un domicilio y cada domicilio pertenece a un cliente.
    - *1 a n*: una empresa tiene muchas sucursales, cada sucursal pertenece a una empresa.
    - *n a n*: un profesor da clases a muchos alumnos, cada alumno tiene muchos profesores.

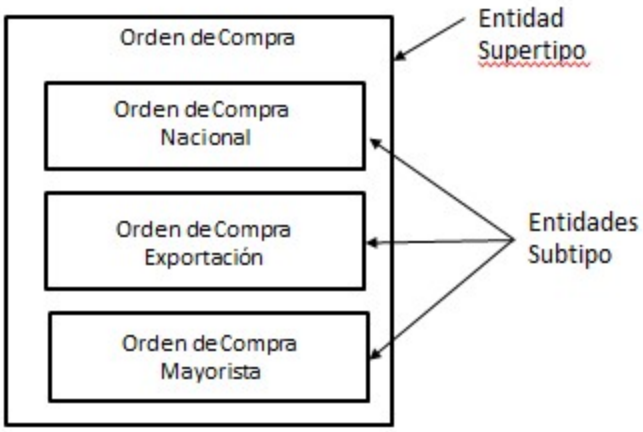
Modalidad

- *Relación Mandatoria (Obligatoria): Si la instancia de una entidad debe existir en la relación.*
  Ej.: Una Factura debe contener a lo sumo un Item.
- *Relación Opcional: Si la instancia de una entidad no necesita existir en la relación.*
  Ej.: Un cliente puede tener facturas asociadas.

Especialización o Generalización (incorporadas a partir del EED: Expanded Entity Diagram)

- *Entidad Supertipo: Entidad padre.*
- *Entidad Subtipo:*
  - *Son las entidades hijas.*
  - *Son mutuamente excluyentes*
  - *Contienen distintos atributos*

Ej.:



Muchos afirman que el DER es independiente de la tecnología, podemos estar de acuerdo si tenemos en cuenta que:

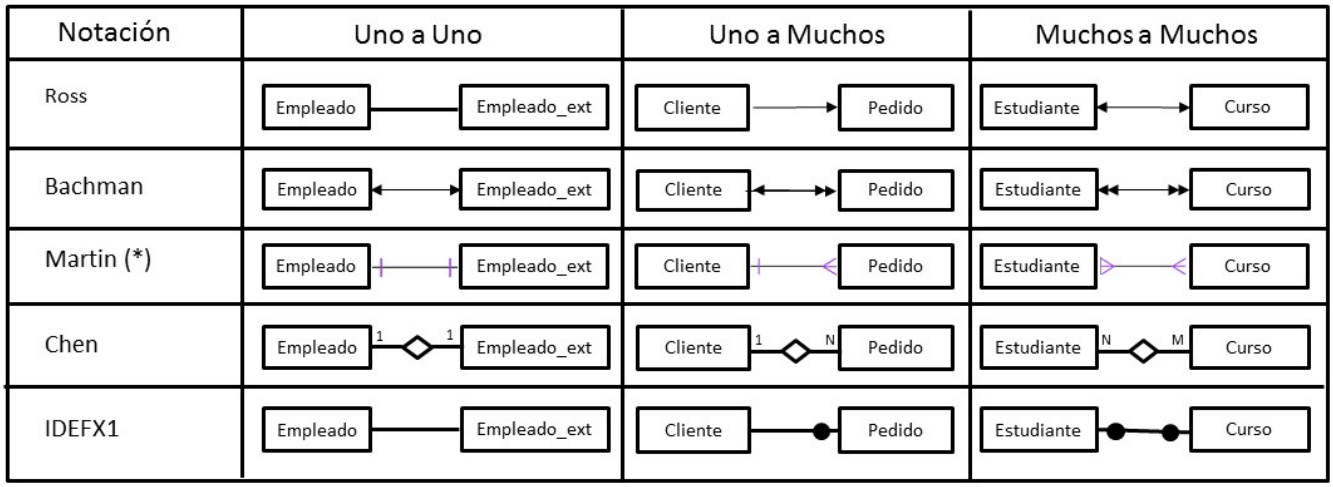
- en el modelo relacional no existen los atributos multivaluados ni podemos tener tipos/subtipos
- las relaciones entre entidades se pueden implementar con punteros/referencias (modelo jerárquico, de objetos) o con foreign keys (en el modelo relacional, como veremos más adelante)

Existen varias notaciones, elijan la que más le guste

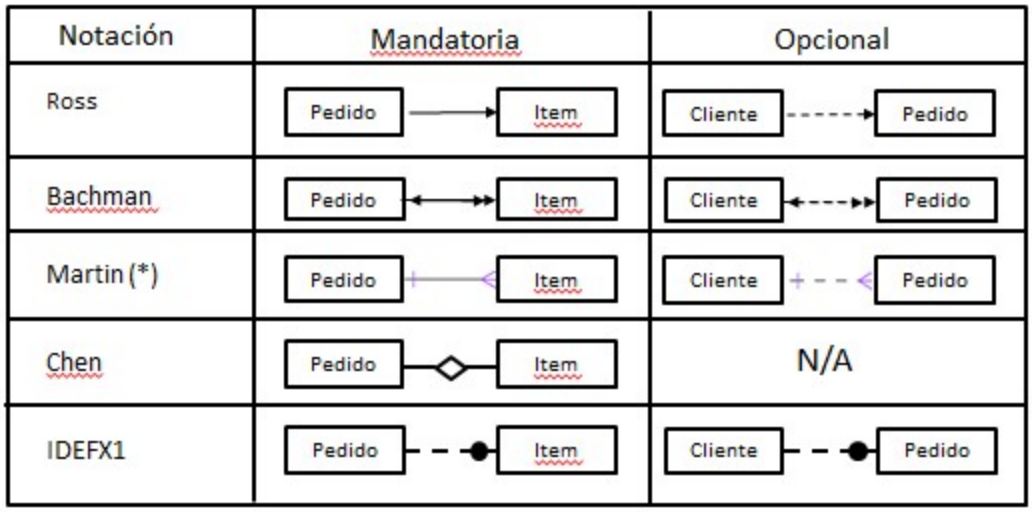
- Martin / Crow's Foot Notation
- Bachman
- Chen

- Ross
- IDEF1X (Integration Definition for Information Modeling)

Acá van algunos ejemplos:



(*) Crow's Foot Notation



(*) Crow's Foot Notation

Del modelo conceptual a un primer DER

Hacemos el DER del ejercicio del manejo de proyectos. Del modelo conceptual sabemos que

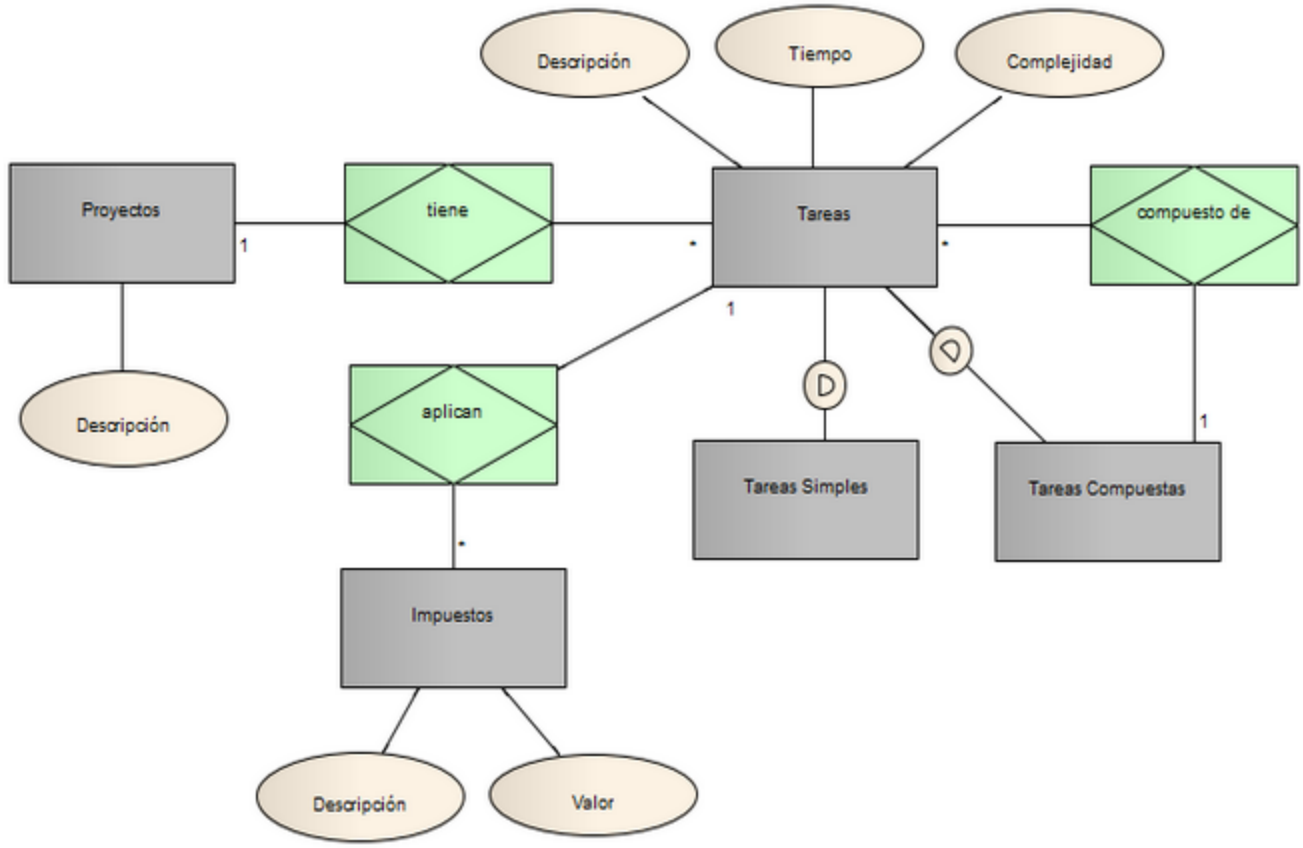
- Un proyecto tiene muchas tareas, debe tener al menos una. Cada tarea está en sólo un proyecto.
- Una tarea puede tener 0, 1 o más subtareas.
- Una tarea tiene muchos impuestos. Los impuestos están asociados a muchas tareas.

- Una tarea tiene una complejidad
- Subtipo: si modelamos TareaSimple y Compuesta y tenemos a la Tarea a secas como supertipo.

¿Qué entidades surgieron?

- Proyectos,
- Tareas,
- Impuestos,
- fíjense que no tiene mucho sentido modelar la complejidad como una entidad, no tiene información. Es inevitable volver sobre la solución en objetos,
    - si los strategies son stateless son candidatos a no ser entidades,
    - por otra parte esto va en contra de la reificación: la Complejidad Media, Mínima y Máxima no son entidades, sino códigos escondidos en la tabla de tareas (encima difíciles de mapear: todos arrancan con "M"). Esto tiene un costo.

¿Y entonces? Entonces como diseñadores tenemos que tomar esa decisión sabiendo los pros y contras. Eso sí no es nuevo para nosotros. Vamos a decidir no tener una entidad aparte, para simplificar nuestra solución (de paso evitamos que piensen que nos gusta el sobrediseño).



DER lógico del manejo de proyectos, en notación UML 2.1

DER físico

Ahora vamos a bajar el DER lógico del Manejo de proyectos a una implementación siguiendo las reglas del modelo relacional[1]:

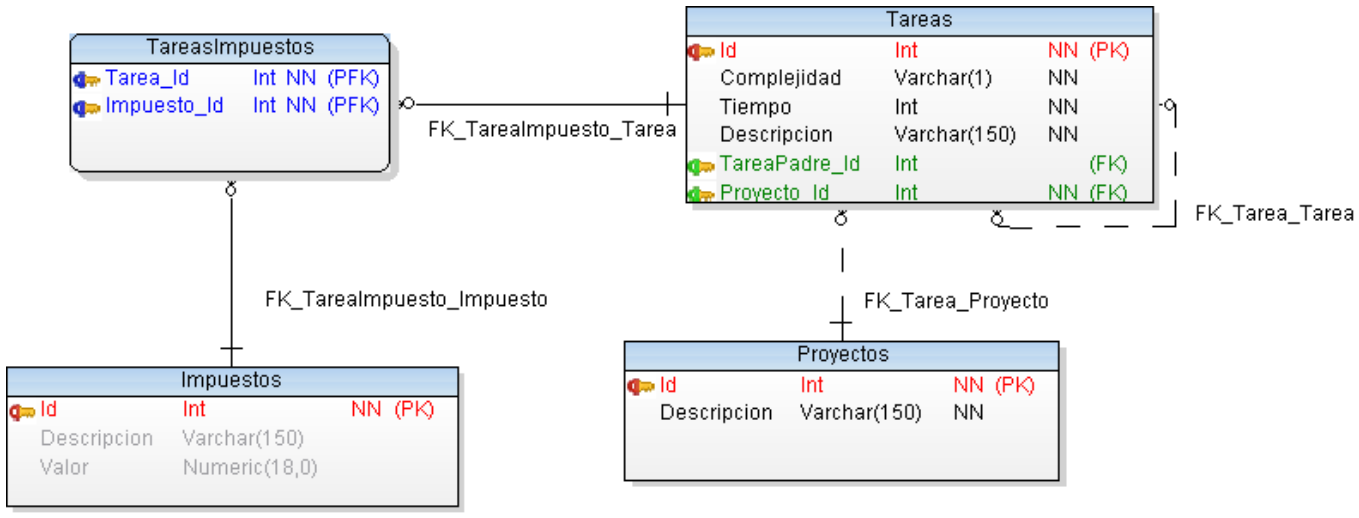
- podemos definir tipos de datos para cada atributo
- y también identificar qué atributo identifica unívocamente a cada fila de esa entidad

Relaciones:

- un proyecto tiene muchas tareas, entonces la entidad tarea necesita una clave foránea (foreign key) que apunte al id de un proyecto. ¿Por qué no puede ser al revés? Porque no tengo atributos multivaluados (colecciones) en el modelo relacional.
- cada tarea puede tener subtareas, entonces la entidad tarea hija necesita una foreign key que apunta al id de la tarea padre. Como no hay generalización entre entidades en el modelo relacional y tampoco hay una diferencia estructural importante, vamos a definir una entidad Tareas que englobe a las tareas simples y compuestas.
- ¿Cómo modelo una relación muchos a muchos entre Tareas e Impuestos?, mientras que en Objetos podíamos tener dos colecciones (una en Tarea y otra en Impuestos si lo necesitábamos), aquí necesitamos 3 tablas
    - la entidad Tareas
    - la entidad Impuestos
    - y una tabla que modela la relación entre Tareas e Impuestos, cuya PK puede ser
        - un ID autoincremental
        - Tarea_ID + Impuesto_ID (clave compuesta, fíjense que ambas son foreign keys que apuntan a las claves primarias de Tarea e Impuesto respectivamente)
    - mientras que la relación 1 a n se modela con una foreign key en la entidad n hacia la primary key de la entidad 1 (como en el caso del 1 proyecto-n tareas), esa estrategia no me sirve en la relación muchos a muchos y debo trabajarlo con una tabla específica.

Lo graficamos con la notación IE:

[1] Para introducir al lector en el modelo relacional recomendamos leer el apunte de *Modelo relacional* de Juan Zaffaroni



¿Con qué hicimos esto? Vale con lápiz y papel, ¿eh?
Bueno, si quieren poner que saben herramientas en el currículum, hay un ERWIN gratuito dando vueltas por ahí, nosotros usamos el TOAD Data Modeler, Visio también puede andar.

Les ponemos nombres a las entidades:

- **Independientes** son las que surgen como entidades naturales: Proyecto
- **Entidades débiles o de Dependencia simple**: Tarea (no existe sin un proyecto)
- **Asociativas**: Tarea_Impuesto

Sobre las relaciones:

- En objetos tenemos que indicar una dirección (qué objeto conoce a cuál). En el modelo relacional no tenemos un "origen" y un "destino", por default lo elige el motor
- Además de tener cardinalidad (número de entidades que participan, una o muchas) podemos indicar la modalidad (si admite 0 o necesita al menos una entidad participando de la relación)

Ejemplos en Álgebra Relacional y en SQL

El ejemplo nos sirve como excusa para conocer algunas operaciones del álgebra relacional y una analogía en el lenguaje de consultas estándar SQL

Restricción

Filtramos las tareas que lleven más de 2 días. Si el tiempo se mide como un int (cantidad de días), hacemos

Algebra Relacional

```
TAREAS WHERE TIEMPO > 2
```

SQL

```
SELECT *
  FROM TAREAS
 WHERE TIEMPO > 2
```

Ustedes ya conocen la misma idea, por supuesto

en Objetos	en Funcional

`tareas select: [ :tarea | tarea tiempo > 2 ]`
 	
inclusive como tarea puede tener responsabilidades podríamos pensar en escribir:
 	
`tareas select: [ :tarea | tarea llevaMasDe: 2 ]`
 	
Claro, en el modelo relacional la tabla Tareas actúa como un container de las tareas que existen para mí, mientras que en objetos necesito un objeto que conozca al conjunto de tareas.
	`filter`
	`  (\tarea -> tiempo tarea > 2)`
	`  tareas`

Proyección

Hacemos una transformación a partir de una relación base. Por ejemplo, partiendo del conjunto de tareas podemos armar un conjunto con los nombres de las tareas:

Algebra Relacional

```
TAREAS {CODIGO, DESCRIPCION}
```

SQL

```
SELECT CODIGO, DESCRIPCION
  FROM TAREAS
```

¿Qué concepto similar conocen?
Claro, el map de funcional y el collect: de objetos.

Join (Junta)

Como no existen los punteros (como en el modelo de objetos o el jerárquico) la relación se hace a través de la foreign key. Entonces si queremos saber las tareas de complejidad mínima tenemos que:

- Hay que preguntar cuáles son las tareas de complejidad mínima. Supongamos que la complejidad mínima es "I": COMPLEJIDAD = "I" (donde "A" => máxima, "E" => media, "I"

=> mínima)

- ¿Cómo relacionamos proyecto con tarea? La tarea tiene un PROYECTO_ID que es una foreign key que apunta a la primary key de Proyectos, hacemos un JOIN entre esos campos.

Algebra Relacional

```
(TAREAS WHERE COMPLEJIDAD = 'I') JOIN PROYECTOS {PROYECTO_ID}
```

SQL

```
SELECT *
  FROM PROYECTOS
       INNER JOIN TAREAS
       ON TAREAS.PROYECTO_ID = PROYECTOS.ID
 WHERE TAREAS.COMPLEJIDAD = 'I'
```

Otra forma:

```
SELECT *
  FROM PROYECTOS,
       TAREAS
 WHERE TAREAS.PROYECTO_ID = PROYECTOS.ID
   AND TAREAS.COMPLEJIDAD = 'I'
```

Asignación

Este concepto de álgebra relacional tiene como analogía en SQL a las instrucciones insert, update y delete.

```
INSERT INTO PROYECTOS
(DESCRIPCION)
VALUES ('MIGRACION SQL');
```

También existe el INSERT masivo, a partir de un SELECT (en este caso mostramos cómo hacer un select de un dato fijo):

```
INSERT INTO PROYECTOS
(DESCRIPCION)
SELECT 'RRHH 5.5';
```

Si necesitamos modificar la descripción de un proyecto:

```
UPDATE PROYECTOS
   SET DESCRIPCION = 'MIGRACION SQL 1.2'
 WHERE ID = 1;
```

También existe el UPDATE masivo, el lector curioso puede investigar al respecto.

Cuando queremos eliminar un proyecto, el DBMS chequea las constraints que aplican sobre la entidad (en nuestro caso, no puedo borrar un proyecto que tenga tareas, porque hay una referencia desde tarea hacia proyecto):

```
DELETE
  FROM PROYECTOS
 WHERE ID = 1;
```

¿Por qué hablamos de asignación?
Si lo vemos desde el punto de vista de lógica de conjuntos, lo que estamos haciendo al borrar un proyecto es generar un nuevo conjunto que no contenga al elemento borrado.

Algebra Relacional

```
PROYECTOS := PROYECTOS - ( PROYECTOS WHERE ID = 1 ) ;
```
[2]

donde "-" es la diferencia de conjuntos. Aquí vemos como tanto el insert, como el update y el delete tienen efecto colateral.

Ejercitación modelo físico

Definición de entidades mediante un DDL

Para implementar lo que hemos definido en el DER necesitamos un lenguaje que nos permita agregar entidades en el catálogo del DBMS. Lo vamos a implementar en MySQL que es un DBMS Open Source. ¿En qué lenguaje? En SQL (Structured Query Language), que provee un DDL estándar; recordamos que el DDL es el lenguaje que permite definir estructuras de datos según el modelo relacional.

Nos conectamos a la base con un perfil administrador, creamos la base de datos ManejoProyectos si no existe y abrimos una ventana de Script SQL interactivo para tipear:

```
CREATE TABLE Proyectos (
	Id int AUTO_INCREMENT NOT NULL PRIMARY KEY,
	Descripcion varchar (150) NOT NULL
) ;
```

Algunas cosas:

- el modelo relacional define dominios para cada elemento de una tupla, entonces no extraña ver definición de tipos para cada columna de la tabla (int para id, varchar 150 para descripción)
- ID es un campo que no va a ser actualizado por nosotros, sino que se va a generar una secuencia autoincremental. El DBMS me garantiza que no habrá dos filas con el mismo

[2] *Extractado de la explicación de C.J.Date, pág.64 "An Introduction to Database Systems", Addison Wesley*

ID, en MySQL se define como AUTO_INCREMENT, en Oracle se asocia a un objeto SEQUENCE (definido aparte) y en SQL Server como IDENTITY. Esta clave se denomina **clave subrogada**.

- La definición de la primary key **es una decisión de diseño**: yo podría utilizar otro campo como identificador unívoco, un campo que el usuario ingrese (como la descripción, o definir un campo específico como un código de proyecto alfanumérico de 5 posiciones).
- es importante definir si cada campo acepta nulos, en nuestro caso tanto el ID como la descripción son campos obligatorios/mandatorios

Hacemos lo mismo para los impuestos.

Veamos la definición de la tarea:

```
CREATE TABLE Tareas (
	Id int AUTO_INCREMENT NOT NULL PRIMARY KEY,
	Complejidad varchar (1)  NOT NULL ,
	Tiempo int NOT NULL ,
	Descripcion varchar (150)  NOT NULL ,
	TareaPadre_Id int NULL ,
	Proyecto_Id int NOT NULL
) ;
```

Tenemos dos campos que son foreign keys:

- Proyecto_Id que referencia al ID del proyecto
- TareaPadre_Id tiene una referencia a la propia entidad (una relación recursiva, donde apunta al primary key de la tarea padre)

En otro query definimos estas foreign keys:

```
ALTER TABLE Tareas
    ADD CONSTRAINT FK_Tarea_Proyecto
FOREIGN KEY
(
Proyecto_Id
) REFERENCES Proyectos (
	Id
) ON DELETE CASCADE
  ON UPDATE CASCADE ,
```

```
ADD CONSTRAINT FK_Tarea_Tarea FOREIGN KEY
(
	TareaPadre_Id
) REFERENCES Tareas (
	Id
);
```

Se agregaron dos restricciones (constraints) en la entidad tareas:

- cuando se elimina un proyecto todas las tareas se eliminan (es lo que dice ON DELETE CASCADE ON UPDATE CASCADE)
- no se puede eliminar una tarea que tenga hijos (definido por la FK_Tarea_Tarea)

No importa tanto la implementación, sino que **las restricciones que vimos son propias del negocio y un buen diseñador tiene que ser capaz de definirlo**.

Acá tenés la definición del modelo de datos para MySQL.

Definiendo un fixture

Es importante poder contar con un juego de datos o fixture para poder resolver los requerimientos que vienen más abajo, entonces vamos a desarrollar un script que inserta datos suponiendo que la base de proyectos esté vacía. Les dejamos aquí el fixture en MySQL para que corran en sus casas: no es tan importante que recuerden cómo son las sentencias INSERT sino **que es parte del trabajo del diseñador construir ejemplos concretos para validar un modelo de datos**.

El fixture permite trabajar con dos proyectos:

1. WebMail 2.1 (el de la guía 6.1.1 sobre la que pueden practicar)
2. el Proyecto A del enunciado, definiendo que la tarea 1 es de complejidad máxima, la tarea 2 y subtarea 2 son de complejidad media y la tarea 3 y subtarea 1 son de complejidad mínima.

Vemos cómo queda implementado en el modelo relacional la jerarquía de tareas:

Id	***Complejidad***	***Tiempo***	***Descripción***	***TareaPadre_Id***	***Proyecto_Id***
9	A	30	Tarea 1	NULL	8
10	E	15	Tarea 2	NULL	8
11	I	25	Tarea 3	NULL	8
12	I	8	Subtarea 1	10	8
13	E	5	Subtarea 2	10	8

Acá tenés la definición del fixture para MySQL

A continuación les pasamos la sintaxis para las sentencias SQL que hemos utilizado en los ejemplos de este apunte.

Sintaxis Sentencias SQL - DDL- Data Definition Language

CREATE TABLE *table-name*

(*column-name datatype*

[PRIMARY KEY]

[REFERENCES *foreing-table* [(*col_name*)]]

[NOT NULL]

[DEFAULT *value*]

[UNIQUE] [CONSTRAINT *constr-name*]

[CHECK (*condition*)]

] [, ...]

[PRIMARY KEY (*primary-col-list*)]

[FOREIGN KEY (*foreign-col-list*) REFERENCES *foreing-table* [(foreing-*col-list*)]]

[UNIQUE (*unique-col-list*)]

[CHECK (*condition-more-than-one-cols*)]

)

ALTER TABLE *table-name*

{ ADD (*newcol-name newcol-type*

[NOT NULL] [...]]

[, ...])

[BEFORE *oldcol-name*]

|

DROP (*oldcol-name* [, ...])

|

MODIFY (*oldcol-name newcol-type*

[NOT NULL] [, ...])

|

ADD CONSTRAINT UNIQUE

(*oldcol-name* [, ...])

[CONSTRAINT *constr-name*]

|

DROP CONSTRAINT (constr-*name* [,...])

} [, ...]

Sintaxis Sentencias SQL – DML - Data Manipulation Language

SELECT [DISTINCT]

select-list

FROM

table-name [*table-alias*]

{INNER | LEFT OUTER | RIGTH OUTER | FULL OUTER} JOIN second-table-name

ON column-first-table = column-second-table [AND ...]

[WHERE *condition*]

[GROUP BY *column-list*]

[HAVING *group-condition*]

[ORDER BY *column-name* [ASC|DESC][, ...]]

INSERT INTO *table-name* [(*column-list*)]

{VALUES (*value-list*) | SELECT-*statement}*

DELETE FROM *table-name*

[WHERE *condition*]

UPDATE *table-name* SET

{

column-name = expr [, ...]

|

(column-list) = (expr-list)

}

[WHERE *condition*]