Grado en Ingeniería Informática

Bases de Datos

TEMA 4

Normalización

Docentes:

Jesús Maudes Raedo

UBUCEV
Universidad de Burgos

Índice de contenidos

I. Introducción

La normalización es una técnica propia de las bases de datos relacionales por la que intentaremos obtener esquemas de la mayor calidad posible, en cuanto a reducir a su mínima expresión las posibles redundancias que haya detrás de un diseño defectuoso.
La normalización se basa en exigir una serie de requisitos de calidad a las relaciones. El cumplimiento de algunos de estos requisitos permite catalogar a una relación como cumplidora de una determinada *forma normal*. Las formas normales van desde las más simples como la primera forma normal, hasta la más exigente, que es la forma normal de *dominio-clave*. En este capítulo, no veremos todas las formas normales posibles, sino sólo las que están basadas en la teoría de las *dependencias funcionales*, que suelen ser las que habitualmente se enseñan en los cursos introductorios de bases de datos.

II. Objetivos

OBJETIVO 1: *Conocer los problemas que dan los diseños relacionales que incluyen algún tipo de redundancia.*

OBJETIVO 2: *Conocer la teoría de dependencias funcionales y aplicarla al proceso de normalización. Mostrar de pasada el andamiaje formal y demostraciones asociado a esta teoría, para aquellos alumnos con una curiosidad más exhaustiva.*

OBJETIVO 3: *Dominar los procesos de obtención de claves de una relación y la caracterización de la forma normal de la misma hasta la forma normal de Boyce – Codd.*

OBJETIVO 4: *Conocer la necesidad de hacer descomposiciones y qué características son deseables en las mismas. Dominar de forma mecánica el proceso de descomposición mediante la síntesis de Bernstein.*

OBJETIVO 5: *Conocer algunos algoritmos útiles en el proceso de normalización, primando la sencillez de la versión del algoritmo explicado, sobre otro tipo de consideraciones (e.g., eficiencia, capacidad de resolver situaciones atípicas/complejas etc …). Aprovechar esa sencillez para intentar que el alumno razone por qué funcionan esos algoritmos,*

III. Contenidos específicos del tema

IMPORTANTE:
No es necesario aprenderse las demostraciones. Las demostraciones sólo están presentes a fin de ilustrar que todos los planteamientos son fruto de un razonamiento matemático sólido. Tampoco es necesario que el alumno se aprenda las definiciones matemáticas si no se siente cómodo con ellas. Basta con que sea capaz de definir los conceptos y

teoremas con sus propias palabras cuidando ser exhaustivo y no dejándose ningún matiz por el camino.

1. Problemas de las representaciones relacionales con redundancia de información

Supongamos que estamos representando mediante tablas la información correspondiente a este enunciado (basado en el ejemplo del capítulo 17 de [de Miguel y Piattini 1993]):

> *Se quiere registrar la información de una pequeña biblioteca que registrará los libros, identificados por un código, el título y el año de publicación. Cada libro puede ser escrito por varios autores y un autor puede escribir varios libros. De los autores registraremos su nombre, con el cual les identificaremos, y su nacionalidad. Todo libro es editado siempre por una editorial, y una editorial puede editar varios libros. De las editoriales registraremos su nombre, con el cual las identificaremos, y su dirección.*

Autores

nombreA	nacionalidad
Pepe	España
John	UK
Yang	China
Gupta	India

Libros

cod_libro	título	año	nombreE
#L1	El libro 1	2001	Editorial1
#L2	El libro 2	2001	Editorial1
#L3	El libro 3	2003	Editorial2
#L4	El libro 4	2004	Editorial3

Editoriales

nombreE	dirección
Editorial1	C/ Uno, nº 1
Editorial2	C/ Dos, nº 2
Editorial3	C/ Tres, nº 3

Escribe

nombreA	cod_libro
Pepe	#L1
Pepe	#L2
John	#L1
John	#L3
Yang	#L3
Gupta	#L4

Figura 1: Esquema descompuesto en 4 tablas.

Un diseñador experimentado desompondría la información de la base de datos en las 4 tablas de la Figura 1, lo que mantendría la primera forma normal que ya vimos en capítulos anteriores. Además, como ya sabemos, al hacer el *join* de las cuatro tablas resulta otra con toda la información:

nombreA	nacionalidad	cod_libro	título	año	nombreE	dirección
Pepe	España	#L1	El libro 1	2001	Editorial1	C/ Uno, nº 1
Pepe	España	#L2	El libro 2	2001	Editorial1	C/ Uno, nº 1
John	UK	#L1	El libro 1	2001	Editorial1	C/ Uno, nº 1
John	UK	#L3	El libro 3	2003	Editorial2	C/ Dos, nº 2
Yang	China	#L3	El libro 3	2003	Editorial2	C/ Dos, nº 2
Gupta	India	#L4	El libro 4	2004	Editorial3	C/ Tres, nº 3

Figura 2: Resultado de hacer las 3 operaciones de join entre las 4 tablas resultantes de la Figura 1

¿Por qué una de las dos representaciones de la misma información (i.e., la primera), es preferible a la otra?. La descomposición de la información en dos tablas relacionadas, nos ha venido dada, bien a través de aplicar el sentido común, bien de haber aplicado las reglas de paso del modelo entidad-relación al modelo relacional que aprenderemos en el próximo tema. Pero ¿por qué hemos descartado desde el principio el haber representado toda la información en una única tabla como la de la Figura 2?

La razón evidente de no hacerlo así es que en la representación de la Figura 2 hay una gran redundancia de información. Por ejemplo, cuando un autor escribe tiene varios libros, todos los datos del mismo (e.g., *nacionalidad)* aparecen de forma repetida en todas las filas de dicho libro, tal y como ocurre con las dos filas de *Pepe* y con las de *John;* y cuando una editorial edita varios libros, todos sus datos (e.g., *dirección*) se repiten en cada libro; y cuando en un libro hay varios coautores, los datos de *año* y *título* del libro, también se repiten.

Esta redundancia puede ser vista como un desperdicio de espacio físico si almacenáramos los datos en una única tabla. Pero ese desperdicio en espacio de almacenamiento es el menor de los problemas, pues la redundancia es la causante de una serie de anomalías y otros fenómenos indeseados que se presentan a continuación.

1.1. Anomalías en altas bajas y modificaciones

1.1.1 Anomalías en la Modificación

Si una editorial cambiase de nombre y/o dirección (campos *nombreE* y *dirección)*, ese cambio habría que trasladarlo a todos los libros editados por dicha editorial, no sólo a una fila, que sería lo que ocurriría si el esquema se hubiese descompuesto de manera que hubiese dado lugar a una tabla independiente de *Editoriales*.

De la misma forma si un autor cambiase de nacionalidad, habría que haber cambiado la nacionalidad en todas las filas donde estuviese presente ese autor. Sin embargo, si existiera una tabla separada de *Autores,* este cambio sólo se produciría en una única fila. La ausencia de una tabla específica de *Libros* también nos lleva a que si un libro cambia de título y/o año también habría que propagar el cambio a todas las filas donde aparece.

1.1.2 Anomalías en Inserción

Todo parece indicar que la clave primaria del ejemplo es la compuesta por *nombreA* y *cod_libro*, pues parece que existe una única fila por cada combinación de éstos.

Esto nos lleva a que no podemos insertar información de autores de los que aún no se han adquirido libros. Si hubiese

una tabla de *autores*, no tendríamos ese problema. De la misma manera, no podemos insertar información de libros sin autor. Si hubiese una tabla de *libros*, no tendríamos ese problema.
Nota que tampoco podemos insertar información de editoriales de las que aún no hayamos adquirido libros sin una tabla específica para las editoriales.

1.1.3 Anomalías en el Borrado

Si borrásemos un libro con un sólo autor, el cual sólo ha escrito ese libro y no lo ha escrito de forma conjunta con otro autor (e.g., el libro *#L4*), perderíamos los datos de dicho autor. De la misma forma, si borrásemos el autor de ese libro (e.g., *Gupta*), perderíamos el libro; si borrásemos el único libro que ha editado una determinada editorial, perderíamos también la editorial, (e.g., borrando el libro *#L4* perderíamos la editorial *Editorial3*).

1.2. Incapacidad para representar cierta información

Una forma de paliar las anomalías de inserción y borrado recurriésemos a los valores nulos. Por ejemplo, si queremos insertar al autor *Ana* sin libros, añadiríamos una fila como esta:

nombreA	nacionalidad	cod_libro	título	año	nombreE	dirección
Ana	Argentina	Null	Null	Null	Null	Null

Figura 3: Al forzar la inserción de un autor sin libros tenemos que recurrir a los nulos llegando a una representación poco natural.

Y si quisiésemos borrar el libro 4, tendríamos que hacer una actualización (i.e., un UPDATE) de manera que la última fila de la Figura 2, ahora quedaría así

nombreA	nacionalidad	cod_libro	título	año	nombreE	dirección
Gupta	India	Null	Null	Null	Editorial3	C/ Tres, nº 3

Figura 4: Al forzar el borrado de un autor sin libros mediante un UPDATE tenemos que recurrir a los nulos llegando a una representación poco natural.

Sin embargo, la representación inicial de la Figura 1 no requiere recurrir al artificio de los valores nulos para representar que un autor no tiene libros (i.e, basta insertar sin usar los nulos al autor en la tabla de autores), ni el borrado del libro requiere llenar de nulos la fila correspondiente en la tabla de *libros*.

1.3. Pérdida de Información

De los problemas presentados anteriormente podríamos deducir que es necesaria una descomposición de esas tablas mal diseñadas, en varias tablas más pequeñas. Sin embargo, no es válida cualquier descomposición. Basándonos en el ejemplo de la Figura 1, ahora tenemos
No hubiera sido correcto haber partido la tabla de la manera que se ha hecho en la Figura 5, en la que vemos que el campo común entre las tablas *A* y *B* es el *año.* Porque al hacer el *join* de esas dos tablas no recuperamos la tabla sin descomponer (e.g., la de la Figura 1), sino que como muestra la Figura 6, de 6 filas que teníamos inicialmente, resulta que ahora tenemos esas 6 más otras 3 mas, incurriendo así en una pérdida de información. Estas nuevas filas que no estaba en la tabla original, son conocidas como filas *espurias*
Todo esto es debido a que el campo común de ambas (i.e., *año*) no es el más adecuado. Como, en el fondo, ya sabemos del tema anterior, el campo común debería haber sido clave primaria de alguna de las tablas, y *año* no lo es.

Como ya sabemos,al hacer el *join* de las cuatro tablas resulta otra con toda la información:

Tabla A

nombreA	nacionalidad	cod_libro	año
Pepe	España	#L1	2001
Pepe	España	#L2	2001
John	UK	#L1	2001
John	UK	#L3	2003
Yang	China	#L3	2003
Gupta	India	#L4	2004

Tabla B

título	año	nombreE	dirección
El libro 1	2001	Editorial1	C/ Uno, nº 1
El libro 2	2001	Editorial1	C/ Uno, nº 1
El libro 3	2003	Editorial2	C/ Dos, nº 2
El libro 4	2004	Editorial3	C/ Tres, nº 3

(**Nota**: en esta tabla B aparecen menos filas que en A porque se han eliminado las filas repetidas, ya que en el modelo relacional puro no hay filas repetidas).

Figura 5: Descomposición incorrecta de la tabla de la Figura 2.

nombreA	nacionalidad	cod_libro	título	año	nombreE	dirección
Pepe	España	#L1	El libro 1	2001	Editorial1	C/ Uno, nº 1
Pepe	*España*	*#L1*	*El libro 2*	*2001*	*Editorial1*	*C/ Uno, nº 1*
Pepe	*España*	*#L2*	*El libro 1*	*2001*	*Editorial1*	*C/ Uno, nº 1*
Pepe	España	#L2	El libro 2	2001	Editorial1	C/ Uno, nº 1
John	*UK*	*#L1*	*El libro 1*	*2001*	*Editorial1*	*C/ Uno, nº 1*
John	*UK*	*#L1*	*El libro 2*	*2001*	*Editorial1*	*C/ Uno, nº 1*
John	UK	#L3	El libro 3	2003	Editorial2	C/ Dos, nº 2
Yang	China	#L3	El libro 3	2003	Editorial2	C/ Dos, nº 2
Gupta	India	#L4	El libro 4	2004	Editorial3	C/ Tres, nº 3

Figura 6: Resultado de hacer las 3 operaciones de *join* entre las tablas de la Figura 5. Se han resaltado las filas espurias.

2. Las Formas Normales

En el ejemplo anterior el sentido común nos podría haber guiado para encontrar las tablas que mejor representasen la información y no dieran todos los problemas presentados. Otra forma posible de actuación más sistemática, y que abordaremos en un tema próximo, consistiría en obtener un diagrama entidad-relación y derivar las tablas

correspondientes a través de las reglas de transformación ya conocidas. Sin embargo, esta solución, aun siendo muy práctica, asume que el diagrama entidad-relación obtenido es correcto.

La teoría de la normalización es un soporte formal que nos sirve para pensar qué problemas tiene un determinado diseño relacional y qué soluciones hay. Para la mayoría de los casos, por su simplicidad, nos basta y nos sobra con utilizar el modelo entidad-relación que ya explicaremos más adelante, pero es posible encontrarse con situaciones complejas que requieran pensar con mayor detenimiento cuál es la solución más adecuada, y para ese tipo de situaciones es para las que nos convendrá usar la teoría de la normalización.

La teoría de la normalización se basa en la definición de un conjunto de posibles estados de una relación o tabla. Cada uno de estos estados se conoce como ***Forma Normal***. Las formas normales están definidas a modo de niveles, de manera que para que una tabla esté en la forma normal *n* habrá de estar primero en las *n-1* formas normales anteriores.

Esto es lo que pretende mostrarse con la Figura 7, en la que se ve que el mayor de los superconjuntos es el que engloba al total de todas las posibles relaciones o tablas. Dentro del mismo sólo algunas relaciones verificarán la primera forma normal (1FN). A su vez, el conjunto de las relaciones que verifican la segunda forma normal (2FN) está dentro del conjunto de las que verifican la primera, y así sucesivamente hasta llegar a las que verifican la forma normal más elevada (FNDC o forma normal de *dominio-clave*).

En este curso sólo se abordarán las formas normales que van de la primera hasta la forma normal de *Boyce-Codd* (FNBC). Las formas normales primera, segunda y tercera aparecen por primera vez en [Codd 1972], y posteriormente en [Codd 1974] se presentó la forman normal de Boyce-Codd. La primera forma normal ya es conocida del tema 2, por ser una restricción inherente del modelo relacional, mientras que las formas normales que van de la segunda a la de *Boyce-Codd* se obtienen todas a partir de la ***teoría de dependencias funcionales*** que se describe más adelante.

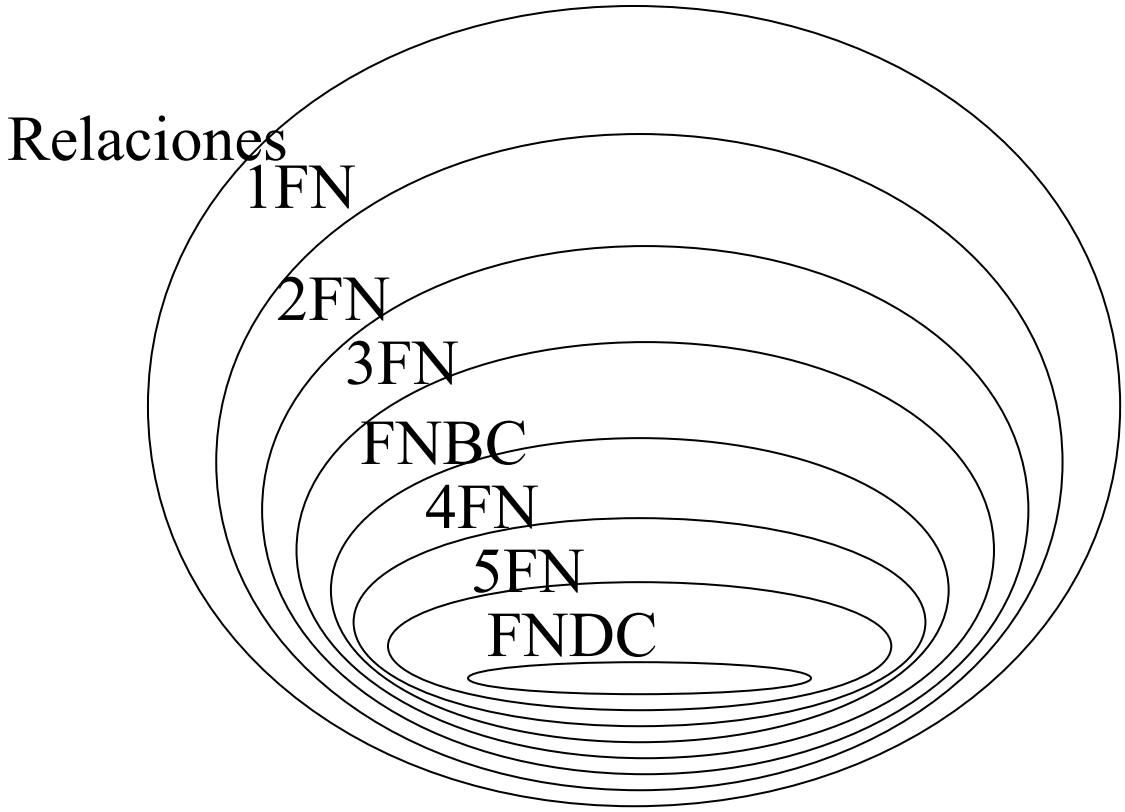

Figura 7: Las formas normales se organizan a modo de niveles.

2.1. La Primera Forma Normal (1FN)

Se dice que una relación está en primera forma normal (1FN) si dada una fila y un campo de esa tabla le corresponde un único valor. Esto significa:

1. Que no puede haber atributos multivaluados, tipo *array,* vector, lista etc...
2. Que no puede haber atributos compuestos, a modo de lo que sería un registro o *struct* en determinados lenguajes de programación.

   Nota: La primera forma normal nos viene "casi" impuesta por los propios SGBDs. El "casi" viene por dos motivos:

   1. Siempre podríamos declarar un campo, por ejemplo, teléfonos que tuviera varios valores (i.e, teléfonos) separados a través de un separador que hayamos convenido, por ejemplo una coma. Sin embargo esto dificulta luego consultar de quién es tal teléfono, quizás habría que usar búsquedas con LIKE/SIMILAR TO. Para borrar o añadir un teléfono habría que quizás´hacer un UPDATE, lo que es poco natural, y sustituir un teléfono por otro sería una operación muy compleja.
   2. El SQL ha evolucionado a partir de mediados de los 90 para permitir que haya campos que sean tengan aspecto de vectores y *structs*, en un intento de acercar el mundo relacional al orientado a objeto. Sin embargo estas extensiones del SQL no han tenido practicamente repercusión en la industria, que sigue trabajando de facto con la versión SQL-92.

2.2. La Teoría de Dependencias Funcionales

Para poder definir el resto de formas normales que van desde la segunda hasta la de Boyce-Codd, es necesario conocer previamente la teoría de dependencias funcionales, la cual se describe a continuación brevemente.

2.2.1 Dependencias Funcionales

El concepto de dependencia funcional aparece por primera vez en [Codd 1972].

> ***Definición :*** Sea ***R*** un esquema de relación (una tabla). Sean $V \subseteq R$ y $W \subseteq R$ conjuntos de atributos de ***R***. Se dice que se verifica la dependencia funcional $V \rightarrow W$, si para todas las parejas de tuplas (filas) ***t1*** y ***t2*** de la relación ***R***, en las que $t1[V]=t2[V]$ (en las que coinciden los valores en esos atributos ***V***) ocurre también que $t1[W]=t2[W]$ (coinciden los valores en los atributos ***W***).
>
> A la parte izquierda de una dependencia la llamamos ***determinante***

Por ejemplo, sea R una relación con los atributos A, B, C y D, como la de la siguiente tabla:

A	B	C	D
a1	b1	c1	d1
a1	b2	c1	d2
a2	b2	c2	d2
a2	b3	c2	d3
a3	b3	c2	d4

Figura 8: Tabla de ejemplo para discutir qué es una dependencia funcional (ejemplo tomado de [Silberschatz et. al 2006].

Veamos si se verifica la dependencia funcional A → C. En este ejemplo A representa la ***V*** de la definición, y C la ***W***. Se trata de dos conjuntos de atributos de la relación, aunque en este sencillo ejemplo, son conjuntos de un solo atributo.

A puede tomar los valores a1, a2 y a3

- Cuando toma el valor a1, C siempre vale lo mismo, esto es c1
- Cuando toma el valor a2, C siempre vale los mismo, esto es c2.
- Cuando toma el valor a3, C siempre vale lo mismo, esto es c2

Por lo tanto, si que se verifica la dependencia A → C en esa relación.

De la misma forma podríamos ver que:

- No se verifica C → A, pues basta ver que cuando la C vale c2, la A no siempre vale lo mismo, una veces vale a2 y otras a3.
- Sí que se verifica AB → C, ya que para AB=(a1, b1), la C siempre vale c1; para AB=(a1, b2), la C siempre vale c1; para AB=(a2, b2) la C siempre vale c2, para AB=(a2, b3) la C siempre vale c2, y para AB=(a3, b3) la C siempre vale c2.

Este último ejemplo es muy interesante, pues se ve que cuando hay una combinación de atributos que no toman valores repetidos, como es el caso de *AB*, no se pueden encontrar dos tuplas con el mismo valor de AB y por tanto no es posible encontrar un contraejemplo que contradiga la hipótesis de que esa dependencia existe.

Sabemos que una clave candidata es un conjunto de atributos cuyos valores combinados no se repiten a lo largo de una tabla o relación. Luego si nos dicen que un determinado conjunto de atributos ***K*** es clave candidata, entonces existirá la dependencia ***K*** → ***X***, donde ***X*** es cualquier otro conjunto de atributos de la relación. En nuestro ejemplo, podríamos encontrar cuantas dependencias si quisiéramos del tipo AB –-> ***X***.

Es más, sabemos que una clave candidata es una superclave que no contiene dentro otra superclave salvo ella misma; y que los valores combinados de una superclave, por tanto, tampoco se repiten dentro de una relación.

Por ejemplo, si el DNI es clave candidata en una relación de empleados, no encontraremos dos empleados que tengan el mismo DNI, pero tampoco encontraremos dos empleados que tengan el mismo DNI y sexo, porque al menos se diferenciarán en el DNI. Por tanto, la combinación DNI + sexo es una superclave que no es clave candidata, pues en su interior está el DNI que es clave y superclave.

Dado que, como hemos visto, los valores combinados de una superclave no se repiten, podríamos extender el razonamiento que hemos hecho para ver que siempre existen las dependencias de la forma ***K*** → ***X*** para ***K*** clave candidata, para ver que siempre existen las dependencias de la forma ***S*** → ***X***, donde ***S*** es superclave.

Es decir, si existen todas las dependencias del tipo DNI → "lo que sea", existirán todas la dependencias del tipo DNI + "más atributos" → "lo que sea".

Pero démonos cuenta que ese "lo que sea" que hay en la parte derecha de la regla, es cualquier conjunto de atributos de la relación, y esto incluye, claro, a la propia relación. Por tanto:

> ***Definición:*** Se dice que ***S*** es una superclave de una relación ***R*** si ***S*** → ***R***. O también, [Silberschatz et. al 2006], si siempre que ***t1***[***S***]=***t2***[***S***] ocurre que ***t1***[***R***]=***t2***[***R***].

Como una restricción del modelo relacional nos dice que toda relación ha de tener al menos una clave candidata (si tiene sólo una sería la primaria), podemos imaginar que en toda relación es fácil deducir un montón de dependencias sólo con poner las claves y/o superclaves de la relación en la parte izquierda de dichas dependencias.

Otras dependencias muy fáciles de deducir son las llamadas *triviales.*

> ***Definición:*** Se dice que una dependencia ***X*** → ***Y*** es trivial si ***Y*** ⊆ ***X***.

Por ejemplo, para los empleados, la dependencia < *sexo, titulación* > → <*sexo*> siempre existe, pues si encuentro dos filas de dos empleados con el mismo sexo y titulación, es claro y trivial que coincidirán en el valor del sexo.

Por tanto, las dependencias triviales van a existir siempre y no nos aportan información nueva. No tienen ningún interés.

Una dependencia ***A*** → ***B***, se puede leer como ***"A determina B"***, y tiene sentido hacerlo así. Si tengo *nombre* → *sexo*,

sabemos que dado un nombre no podré encontrarme otra fila con el mismo nombre de empleado en el que el sexo sea diferente. Por tanto, "*nombre determina sexo*", tal y como ocurre en la realidad, en donde según sea el nombre de una persona, somos capaces normalmente de predecir su sexo.

Ejemplo:

Sea la relación:

Alumnos_Asignaturas (DNI, Nombre, Color_Pelo, Cod_Asignatura, Nro_Créditos)

Donde *Cod_Asig*, representa el código de una asignatura de la que esté matriculado el alumno (podría haber varias). Vamos a deducir algunas de las dependencias funcionales de *Alumnos*:

- En primer lugar $< DNI > \rightarrow < Nombre >$, $< DNI > \rightarrow < Color\_Pelo >$ y $<Cod\_Asignatura \rightarrow Nro\_Créditos >$. porque el DNI determina todos los atributos de alumno, y el código de asignatura determina todos los atributos de asignatura..
- De las 2 primeras dependencias anteriores se deduce $< DNI > \rightarrow < Nombre, Color\_Pelo >$
- La dependencia
  $< DNI, Cod\_Asignatura > \rightarrow < DNI, Nombre, Color\_Pelo, Cod\_Asignatura, Nro\_Créditos >$
  porque lo que le falta de determinar al DNI (i.e., *Cod_Asignatura, Nro_Créditos)* lo determina el código de la asignatura. En el fondo es porque $< DNI, Cod\_Asignatura >$ es clave candidata de *Alumnos_Asignaturas.*
- De la dependencia anterior se deducen $< \boldsymbol{DNI}, Cod\_Asignatura > \rightarrow < \boldsymbol{DNI} >$,
  $< DNI, Cod\_Asignatura > \rightarrow <Nombre>$, $< DNI, Cod\_Asignatura > \rightarrow < Color\_Pelo >$,
  $< DNI, \boldsymbol{Cod\_Asignatura} > \rightarrow <\boldsymbol{Cod\_Asignatura}>$ y $< DNI, Cod\_Asignatura > \rightarrow < Nro\_Créditos >$
  descomponiendo su parte derecha en trozos con un solo atributo.
- Podríamos generar todas las triviales que imaginemos (de hecho ya hemos generado dos en el punto anterior, ver atributos en negrita). Por ejemplo: $< Nro\_Créditos, Color\_Pelo > \rightarrow < Color\_Pelo >$, es una dependencia trivial.

2.2.2 Cierre de un conjunto de dependencias funcionales

Definición: El conjunto cerrado o cierre de un conjunto de dependencias funcionales ***F***, se denota por $\boldsymbol{F}^+$ y es el conjunto de todas las dependencias funcionales que ***F*** implica lógicamente.

La implicación lógica de una dependencia a partir de otra(s) viene dada a través de los llamado ***Axiomas de Armstrong*** [Armstrong 1974] que se enumeran a continuación:

1. **Regla de Reflexividad:** Si $\boldsymbol{X}$ es un conjunto de atributos e $\boldsymbol{Y} \subseteq \boldsymbol{X}$, entonces se cumple $\boldsymbol{X} \rightarrow \boldsymbol{Y}$, (que como ya sabemos es una dependencia funcional trivial). Por tanto, la regla de reflexividad es la que permite deducir lógicamente todas las dependencias triviales.
2. **Regla de Amplificación:** Si se cumple $\boldsymbol{X} \rightarrow \boldsymbol{Y}$, y además ***W*** es otro conjunto de atributos de la misma relación, entonces $\boldsymbol{WX} \rightarrow \boldsymbol{WY}$.
3. **Regla de Transitividad:** Si se cumple $\boldsymbol{X} \rightarrow \boldsymbol{Y}$, y además se cumple $\boldsymbol{Y} \rightarrow \boldsymbol{Z}$, entonces, se verifica también $\boldsymbol{X} \rightarrow \boldsymbol{Z}$.

A partir de estos tres axiomas se definen más reglas, a saber:

1. **Regla de Unión:** Si se cumplen $\boldsymbol{X} \rightarrow \boldsymbol{Y}$, y $\boldsymbol{X} \rightarrow \boldsymbol{Z}$, entonces se cumple $\boldsymbol{X} \rightarrow \boldsymbol{YZ}$. O generalizándolo para *n* dependencias:

   $$\forall\, X \rightarrow A_i\,,\ i \in 1,..,n \Rightarrow X \rightarrow A_1 \ldots A_n$$

   Demostración:

   $\boldsymbol{X} \rightarrow \boldsymbol{Y}$, y $\boldsymbol{X} \rightarrow \boldsymbol{Z}$, entonces: $\boldsymbol{X} \rightarrow \boldsymbol{Y} \Rightarrow \boldsymbol{XZ} \rightarrow \boldsymbol{YZ}$ por la regla de Amplificación

$X \rightarrow Z \Rightarrow X \rightarrow XZ$ por la regla de Amplificación también[1]

Aplicando la regla de Transitividad a los resultados anteriores $X \rightarrow XZ$ y $XZ \rightarrow YZ$, queda finalmente que $X \rightarrow YZ$.

2. **Regla de Descomposición:** Si se cumple $X \rightarrow YZ$, entonces se cumple $X \rightarrow Y$ y $X \rightarrow Z$. O generalizándolo para *n* dependencias:

$$X \rightarrow A_1 \dots A_n \Rightarrow \forall\, X \rightarrow A_i\,,\ i \in 1,..,n$$

Demostración:

$YZ \rightarrow Y$ por reflexividad (es una trivial)

Nos dicen que $X \rightarrow YZ$

Por transitividad $X \rightarrow YZ$, $YZ \rightarrow Y \Rightarrow X \rightarrow Y$

Procederíamos de forma análoga para $X \rightarrow Z$.

3. **Regla de Psudotransitividad:** Si se cumple $X \rightarrow Y$ e $WY \rightarrow Z$, entonces se cumple $XW \rightarrow Z$.

Demostración:

De la dependencia $X \rightarrow Y$, por amplificación se deduce $XW \rightarrow WY$

Como $WY \rightarrow Z$, por transitividad se deduce $XW \rightarrow Z$.

Ejemplo:

Siguiendo con el esquema de relación

Alumnos_Asignaturas (DNI, Nombre, Color_Pelo, Cod_Asignatura, Nro_Créditos)

supongamos que nos dan un conjunto de dependencias funcionales de partida F:

$F = \{$ *DNI* $\rightarrow$ *nombre,*

DNI $\rightarrow$ *color_pelo,*

nombre $\rightarrow$ *DNI*[2]*,*

cod_asignatura $\rightarrow$ *nro_créditos}*

Vamos a ver a continuación algunas dependencias que no están en F, pero que sí están en F^+ aplicando los axiomas y reglas anteriores:

Lógicamente el axioma de reflexividad nos permite hallar un numeroso conjunto de dependencias triviales, por ejemplo:

color_pelo $\rightarrow$ *color_pelo,*

cod_asignatura, color_pelo $\rightarrow$ *color_pelo*

cod_asignatura, color_pelo $\rightarrow$ *cod_asignatura, color_pelo*

etc …

Todas ellas no estaba en el F que nos han dado, pero están en F^+. Lo mismo pasa con esta otra:

nombre, Cod_asignatura $\rightarrow$ *nro_créditos, color_pelo*

Se puede deducir así:

- De **nombre → DNI** y **DNI → color_pelo**, por transitividad **nombre → color_pelo**
- Si con el *nombre del alumno* determino el *color del pelo,* con el *nombre del alumno* **y** *código de la asignatura* determino el *color del pelo*. Haciéndolo paso a paso:
  - Por reflexividad deducimos la dependencia **nombre, cod_asig → nombre**
  - Por transitividad, de **nombre, cod_asig → nombre** y **nombre → color_pelo** deducimos **nombre,cod_asig → color_pelo**

1 Añadiendo X a ambos lados de la dependencia: X → Z => XX → XZ = X → XZ

2 Lógicamente la dependencia *nombre* ⟶ *DNI* no es real, en tanto fallaría si hay dos alumnos que se llamen igual. Asumimos que esa dependencia existe a efectos del ejemplo, nada mas.

- y como con el *código de la asignatura* determino los *créditos,* con la *combinación del nombre de la asignatura* con el *número de créditos* determino tanto el color del pelo como el número de créditos (regla de unión). Haciéndolo paso a paso:
  - Por reflexividad deducimos la dependencia **nombre, cod_asig → cod_asig**
  - Por transitividad, de **nombre, cod_asig → cod_asig** y **cod_asig → n_creditos** deducimos **nombre, cod_asig → n_creditos**
  - Aplicando la regla de unión, con **nombre, cod_asig → color_pelo** y **nombre,cod_asig → n_creditos**, se deduce finalmente **nombre, cod_asig → color_pelo, n_créditos**

2.2.3 Cierre de un conjunto de atributos X

Las formas normales de una relación *R* que vamos a estudiar están basadas en configuraciones anómalas de algunas dependencias funcionales para esta relación *R*. Esas dependencias que van a sintomatizar esas anomalías no tienen por qué estar en el conjunto de dependencias funcionales de partida (*F*), sino que puede que se deduzcan a partir de *F* y siguiendo los axiomas de Armstrong (esto es, son dependencias que están en F^+).
Un paso intermedio para calcular F^+ es calcular el cierre de los posibles conjuntos de atributos en *R*.

Definición:

- Sea ***R*** un esquema de relación donde ***F*** es el conjunto de dependencias funcionales que lleva asociado.
- Sea ***X*** un subconjunto de ***R*** (por tanto, un subconjunto de los atributos de ***R***)
- Se define el cierre de ***X*** respecto a ***F*** (y se representa como X^+), como el conjunto de atributos ***A*** de ***R***, tales que la dependencia $X \rightarrow A$ puede deducirse de ***F*** a través de los axiomas de Armstrong (o dicho de otra forma: tal que $X \rightarrow A \in F^+$).

Se dice, además, que ***X*** es un *descriptor,* y por ello también se conoce X^+ como el cierre del *descriptor* ***X***.
Intuitivamente, se trataría de ver qué dependencias se pueden deducir de manera que ***X*** fuese la parte izquierda (o determinante). Juntando todos los atributos que estuviesen a la derecha de esas dependencias obtendríamos X^+.
Es decir, X^+ no es mas que el conjunto de atributos que quedan determinados por **X** a partir del conjunto de dependencias ***F***.

Ejemplo:

Con la relación

Alumnos_Asignaturas (DNI, Nombre, Color_Pelo, Cod_Asignatura, Nro_Créditos)

y el conjunto de dependencias:

F = { *DNI → nombre,*
DNI → color_pelo,
nombre → DNI,
cod_asignatura → nro_créditos}

Hallar el cierre de *(cod_asignatura, color_pelo)* según *F*:

$$(cod\_asignatura,\ color\_pelo)_F^+$$

Solución: En primer lugar esos dos atributos se determinan así mismos por reflexividad:

$$(cod\_asignatura,\ color\_pelo)_F^+ = \{ cod\_asignatura, color\_pelo, \ldots$$

el código de asignatura, por si solo determina al número de créditos y nada más. Podemos añadir el número de créditos por la regla de unión

$$(cod\_asignatura,\ color\_pelo)_F^+ = \{ cod\_asignatura, color\_pelo, nro\_créditos \ldots$$

el color del pelo sólo se determina así mismo, y con el número de créditos pasa lo mismo. Por lo tanto, finalmente:

$$(cod\_asignatura, color\_pelo)_F^+ = \{ cod\_asignatura, color\_pelo, nro\_créditos \}$$

Que en el fondo no es más que la dependencia:

$$cod\_asignatura, color\_pelo \rightarrow cod\_asignatura, color\_pelo, nro\_créditos$$

Del ejemplo podemos intuir que el cierre de un conjunto de atributos puede obtenerse a partir de un simple algoritmo:

2.2.3.1 Algoritmo para el cálculo de X^+

(El siguiente algoritmo está basado en el que aparece en [Silberschatz et. al 2006])

```
Resultado := X   /* Por la regla de reflexividad  X → X */
cambio_en_resultado := true
WHILE ( cambio_en_resultado ) DO
        cambio_en_resultado := false
        FOR EACH dependencia funcional Y → Z IN F DO {
                IF Y ⊆ Resultado AND Z ⊄ Resultado {
                        Resultado := Resultado ∪ Z
                        /*Por la regla de reflexividad X → Resultado => X → Y
                          Por la transitiva X → Y y Y → Z => X → Z
                          Por la regla de unión X → Z y X → Resultado => X → Resultado ∪ Z */

                        cambio_en_resultado := true
                } // fin IF
        } // fin FOR
} // fin WHILE
```

Ejemplo (tomado de [Silberschatz et. al 2006] sección 7.4.1 – 7.4.2):

Sea la relación $R = \{ A, B, C, G, H, I \}$, y $F = \{ A \rightarrow B, A \rightarrow C, CG \rightarrow H, CG \rightarrow I, B \rightarrow H\}$. Se pide calcular el cierre $(AG)^+$:

Por el paso 1: *Resultado := AG*

Probamos si cambia el resultado recorriendo las dependencias

Primera pasada del bucle *FOR EACH:*

Probamos con $A \rightarrow B$: como *A* está en *Resultado* añadiríamos *B*

Resultado:= AGB

Probamos con la siguiente: $A \rightarrow C$: como A está en Resultado añadiríamos *C*

Resultado := AGBC

Probamos con la siguiente: $CG \rightarrow H$ como *C* y *G* están en *Resultado* añadiríamos *H*

Resultado := AGBCH

Probamos con la siguiente: $CG \rightarrow I$: como *C* y *G* están en *Resultado* añadiríamos *I*

Resultado := AGBCHI

Probamos con la siguiente: B --> H: como B está en *Resultado* añadiríamos *H*, pero como *H* ya estaba en *Resultado,* el resultado no cambia

Como (durante el bucle *FOR EACH*) ha cambiado el resultado damos otra pasada del bucle *FOR EACH*

Probamos con $A \rightarrow B$: como *A* está en *Resultado,* pero *B* también está en *Resultado*

Resultado no cambia, y

probando con el resto de dependencias veríamos igualmente que ya no cambia.

Por $A \rightarrow C$: (A y C en *Resultado),* por tanto no cambia

Por $CG \rightarrow H$: *(CG* y H en *Resultado)*, por tanto no cambia

Por $CG \rightarrow I$: *(CG* e I en *Resultado)*, por tanto no cambia

Por $B \rightarrow H$: (B y H en *Resultado)*, por tanto no cambia

Como en esta pasada del *FOR EACH* el resultado no ha cambiado, termina el algoritmo, y así:

(AG)+ := Resultado = AGBCHI

Por tanto: $(AG)^+$ = AGBCHI

2.2.4 Equivalencias entre conjuntos de dependencias

Cuanto mayor sea el número de dependencias más confuso y lento será intentar razonar a través de ellas (e.g., razonar cuales son las claves, en qué forma normal está la relación, cómo arreglarlo, etc...). Por tanto, debemos de buscar conjuntos de dependencias que sean equivalentes al que nos dan, pero que contengan el menor número de dependencias posibles.

La primera cuestión, por tanto, es ser capaces de determinar si dos conjuntos de dependencias distintos son en el fondo equivalentes.

Si en el ejemplo anterior en el que:

$F = \{ DNI \rightarrow nombre, DNI \rightarrow color\_pelo, nombre \rightarrow DNI, cod\_asignatura \rightarrow nro\_créditos\}$

definimos por ejemplo otro conjunto G que tenga las mismas dependencias que F más por ejemplo la dependencia

$$nombre \rightarrow color\_pelo$$

intuitivamente vemos que F y G son equivalentes, ya que cada dependencia $X \rightarrow Y$ en el fondo no es más que una restricción que impediría tener dos valores de Y diferentes cuando los valores de X se repitan. En tanto, hemos visto que la nueva dependencia se puede deducir de las de F, no estamos añadiendo ninguna restricción extra que no estuviera implícitamente en el propio F.

Esto es:

1. Si $nombre \rightarrow DNI$, dado dos nombres iguales en la tabla deberían de tener el mismo *DNI*
2. Si $DNI \rightarrow color\_pelo$, dados dos DNIs iguales deberían de tener el mismo color de pelo.
3. Luego: dado dos nombre iguales sólo les corresponde un DNI al que sólo le corresponde un color de pelo, es decir implícitamente estamos restringiendo que dados dos nombres iguales el color del pelo también ha de serlo.

Por tanto, si un conjunto de dependencias F le faltan varias dependencias que sí que tiene otro conjunto G, pero esas dependencias se pueden deducir mediante el propio F, realmente esas dependencias que le faltan no le aportan nada nuevo a F respecto a G.

El conjunto de todas las dependencias que se pueden deducir a partir de F es $\boldsymbol{F}^+$ (el cierre de F según veíamos en la sección 2.2.2). Por tanto, si dos conjuntos son capaces de deducir las misma dependencias (i.e., tienen el mismo cierre) serán equivalentes.

Definición: Dos conjuntos de dependencias funcionales $\boldsymbol{F}$ y $\boldsymbol{G}$ son equivalentes si $\boldsymbol{F}^+ = \boldsymbol{G}^+$.

Teorema: Dados dos conjuntos de dependencias funcionales $\boldsymbol{F}$ y $\boldsymbol{G}$

$$\boldsymbol{F}^+ = \boldsymbol{G}^+ <=> \boldsymbol{F} \subseteq \boldsymbol{G}^+ \text{ y } \boldsymbol{G} \subseteq \boldsymbol{F}^+$$

El resultado de este teorema es sumamente intuitivo: dos conjuntos de dependencias son equivalentes ($\boldsymbol{F}^+ = \boldsymbol{G}^+$), si todas las dependencias del uno se pueden deducir mediante el otro ($\boldsymbol{F} \subseteq \boldsymbol{G}^+$), y todas las del otro se pueden deducir mediante el uno ($\boldsymbol{G} \subseteq \boldsymbol{F}^+$).

Demostración:

Paso 1: demostrar que si $\boldsymbol{F}^+ = \boldsymbol{G}^+ => \boldsymbol{F} \subseteq \boldsymbol{G}^+$ y $\boldsymbol{G} \subseteq \boldsymbol{F}^+$

Como $F \subseteq F^+ \Rightarrow F \subseteq G^+$

y como además $G \subseteq G^+ \Rightarrow G \subseteq F^+$

Paso 2: demostrar que si $F \subseteq G^+$ y $G \subseteq F^+ \Rightarrow F^+ = G^+$

Sea f una dependencia funcional tal que $f \in F^+$, es decir f se puede deducir a partir de F con los axiomas de Armstrong

como $F \subseteq G^+$, entonces f se puede deducir de igual manera de G^+

por tanto cualquier dependencia $f \in F^+$ verifica también que $f \in G^+$, de donde $F^+ \subseteq G^+$

Razonando de idéntica forma llegaríamos a como $G^+ \subseteq F^+$.

Como $F^+ \subseteq G^+$ y $G^+ \subseteq F^+$ entonces $G^+ = F^+$

Ejemplo:

Se pide discutir la equivalencia de los siguientes conjuntos de dependencias de la relación

R = (P, H, A, M, G }

- F={ P → M, PH → G, HA → P, HG → A}
- G={ P –> M, PH → A, HA → G, HG → P}
- H={ P → M, HG → P, HA → G }

Equivalencia F-G:

Comencemos por las dependencias de F:

- P → M está en F y en G, luego está dependencia no contradice $F^+ \subseteq G^+$ y $G^+ \subseteq F^+$
- PH → G es una dependencia en F que no está en G, habría que ver si se puede deducir con G, para ello calculamos PH_G^+:

  PH_G^+= PHMAG

  Luego con G si que se puede deducir PH → G, por tanto esta dependencia tampoco contradice la posibilidad de que ambos conjuntos sean equivalentes.
- HA → P, tampoco está en G, por lo que procedemos de igual forma:

  HA_G^+= HAGPM

  Luego con G si que se puede deducir HA → P, tampoco contradice que sean equivalentes.
- HG → A, tampoco está en G

  HG_G^+= HGPMA

  Luego con G si que se puede deducir HG → A, tampoco contradice que sean equivalentes.

De todo ello se ve que todas las dependencias de F se podían haber deducido con G, o que $F^+ \subseteq G^+$

Procediendo de forma similar (se deja para el alumno) veríamos que todas las dependencias de G se podían haber deducido con F, y que $G^+ \subseteq F^+$ y que por ello ambos conjuntos son equivalentes.

Equivalencia F-H

No son equivalentes. Basta tomar la primera dependencia no común de F: PH → G, para verlo:

PH_H^+= PHM

Por tanto con H no es posible deducir la dependencia PH → G. Por ello, no se cumple $F^+ \subseteq H^+$ y por lo tanto F y H ya no pueden ser equivalentes.

Como $F^+ = G^+$ y $F^+ \neq H^+$, entonces $G^+ \neq H^+$, por lo que H tampoco es equivalente a G.

2.2.4.1 Equivalente minimal de un conjunto de dependencias

Como ya se vio en el apartado anterior, es interesante tener un conjunto de dependencias equivalente al de partida, con el menor número de dependencias posibles, porque esto nos facilita operaciones como por ejemplo la búsqueda

de la clave.

Definición: Un conjunto de dependencias funcionales ***F*** es *minimal* si:

1. Todas las dependencias funcionales están en *forma canónica*. Esto es: el lado derecho de cada dependencia contiene un único atributo.
2. No existe ningún subconjunto de dependencias ***F'***, tal que ***F'*** $\subset$ ***F*** en el que además $F'^{+} = F^{+}$ (Es decir, no se puede quitar ninguna dependencia de ***F*** sin que el cierre de ***F*** cambie; o lo que es lo mismo no sobra/es redundante ninguna dependencia).

$$\forall\ p \in F,\ F^{+} \neq (F - \{p\})^{+}$$

3. Todas las dependencias son elementales (elemental quiere decir que no se puede simplificar la parte izquierda de la dependencia (o determinante) con un descriptor más pequeño, o dicho de otra forma, no podemos quitar atributos de la parte izquierda de ninguna dependencia).

$$\forall\ X \to A \in F,\ si\ Z \subset X \Rightarrow F^{+} \neq \{F - \{X \to A\} \cup \{Z \to A\}\}^{+}$$

2.2.4.2 Algoritmo para hallar un recubrimiento minimal equivalente

Sea ***F*** el conjunto de dependencias de partida y ***G*** el recubrimiento minimal equivalente de *F* que halla el algoritmo:

G = ∅

Paso 1: /* Pasar ***F*** a forma canónica usando la regla de descomposición */

Para cada X → Y ∈ ***F***, hacer:

G:= ***G*** ∪ { X → A1, … X → An } / ***Y***={ A1, … An } y Ai es un atributo de *R*

Paso 2: /* Descomponer las dependencias que no sean elementales */

Para cada X → A ∈ ***G***, hacer{

Para cada B ∈ X, hacer{

G1 := {***G*** - { X → A }} ∪ { (X-B) → A}

Si $G^{+} = G^{+}$ entonces ***G*** := ***G1*** }

Paso 3: /* Eliminación de dependencias redundantes */

Para cada X → A ∈ ***G***, hacer{

G1 := ***G*** - { X → A }

Si $G^{+} = G^{+}$ entonces ***G*** := ***G1*** }

}

Nota: Aunque el orden en el que originalmente se ejecutan los pasos del algoritmo [Ullman 1982] (pp. 224-225) intercambia el orden de los pasos 2 y 3, existe algún caso en el que ejecutar el paso 3 antes que el 2 no es correcto [Atkins 1988]. Este problema viene descrito en [de Miguel y Piattini 1993] mediante el ejemplo F = { AB → C, C → B, A → B }.

En el ejemplo del capítulo 9 de [Date 1994] y en el algoritmo en [Elmasri y Navathe 2007] sí que se intercambia el orden original de los pasos 2 y 3, para hacerlo como aquí se sugiere.

Ejemplo:

Encontrar un recubrimiento minimal equivalente a ***F*** = { A → BC, B → AC, C → A } para R = {A,B,C}

Paso 1:

G = { A → B, A → C, B → A, B → C, C → A }

Paso 2:

En este caso no hay que hacerlo porque observamos que el todas las dependencias las partes izquierdas tienen un solo atributo, por lo que no se pueden reducir esas partes izquierdas aun más.

Paso 3:

1. Probar si sobra A → B
   sea ***G1*** = { A → C, B → A, B → C, C → A }
   Hallamos $(A)^+_{G1}$= AC
   Vemos que no deducimos A → B, luego esta dependencia no es redundante y no la podemos quitar.
2. Probar si sobra A → C
   sea ahora ***G1*** = { A → B, B → A, B → C, C → A }
   Hallamos $(A)^+_{G1}$= ABC
   Vemos que sí deducimos A → C, aun cuando la quitásemos, luego esta dependencia es redundante y la podemos quitar.
   G = ***G1***
3. Probar si sobra B → A
   G1 = { A → B, B → C, C → A }
   Hallamos $(B)^+_{G1}$= BCA
   Vemos que sí deducimos B → A, aun cuando la quitásemos, luego esta dependencia es redundante y la podemos quitar.
   G = ***G1***
4. Probar si sobra B → C
   G1 = { A → B, C → A }
   Hallamos $(B)^+_{G1}$= B
   Vemos que no deducimos B → C, luego esta dependencia no es redundante y no la podemos quitar.
5. Probar si sobra C → A
   G1 = { A → B, B → C }
   Hallamos $(C)^+_{G1}$= C
   Vemos que no deducimos C → A, luego esta dependencia no es redundante y no la podemos quitar.

Por tanto como resultado de este paso, ***G*** = { A → B, B → C, C → A }

Ejemplo:

Encontrar un recubrimiento minimal equivalente a ***F*** = { AB → C, A → B, B → A } para R = {A,B,C}

Paso 1: Las dependencias ya están en forma canónica.

Paso 2: Ver si podemos simplificar las partes izquierdas.

- A → B y B → A tienen un único atributo a su izquierda, luego ya no se pueden simplificar más.
- AB → C, ¿podría simplificarse por A → C?. Sea ***G1*** = { A → C, A → B, B → A }
  - ¿Se puede deducir AB → C con ***G1***?, $(AB)^+_{G1}$= ABC.

    <u>*Nota:*</u> La comprobación de si con el conjunto nuevo G1 se deduce el original G, nos la podemos ahorrar siempre, ya que sabemos de antemano que va a salir bien, porque como se cumple A→C, es claro que se va a cumplir AB → C.

    Demostración: A → C => AB → CB por axioma de amplificación => AB → C por regla de descomposición

    y en general siempre sería lo mismo: comprobar que la dependencia vieja sin simplificar XY → Z se puede deducir de la simplificada X → Z, donde X e Y son conjuntos de atributos; que como

vemos se va a verificar siempre.

- ¿Se puede deducir A → C con ***F***?, esta parte de la comprobación no nos la podemos saltar (i.e, comprobar si el conjunto nuevo simplificado se puede deducir con el conjunto sin simplificar.
  Al final esta comprobación se reduce a ver si la dependencia simplificada se puede deducir con el conjunto sin simplificar:

  $$(A)^+_F = ABC.$$

  Por tanto, como sí que llegamos a la C, sí que es simplificable; por lo que llegamos al resultado parcial:

  $$\boldsymbol{G} = \{ A \rightarrow C, A \rightarrow B, B \rightarrow A \}.$$

  Como ahora todas las dependencias tienen un atributo en su parte izquierda, no podemos seguir intentando simplificar más partes izquierdas con el paso 2.

Paso 3: Ver si sobra alguna:

- A → C
  - ***G1*** = {A → B, B → A }, $(AB)^+_{G1}$= AB, al no llegar a C, esta no se puede eliminar.
- A → B
  - ***G1*** = {A → C, B → A }, $(A)^+_{G1}$= AC, al no llegar a B, esta no se puede eliminar.
- B → A
  - ***G1*** = {A → C, A → B }, $(B)^+_{G1}$= B, al no llegar a A, esta no se puede eliminar.

Luego no podemos quitar ninguna

Por lo que, ***G*** = { A → C, A → B, B → A } es un recubrimiento minimal de ***F***.

Es interesante notar un par de cosas en el paso 2:

1. Como ya hemos dicho, por un lado la vieja dependencia (AB → C) siempre se va a poder deducir de la nueva (A→C). Es lógico, si con el DNI sé el color del pelo, con el DNI y cualquier otro dato también lo sé. Por ello, ese lado de la comprobación nos le podemos saltar y dar por supuesto.
   Lo contrario, sin embargo no es cierto: La nueva dependencia (A → C) no tiene por qué poder deducirse a partir de la vieja (AB → C), porque la vieja es "en principio" más exigente, significa que tengo que saber dos cosas (la A y la B) para determinar la C, mientras que la dependencia nueva sólo me exige saber una.
   Por tanto, tenemos que comprobar si ese "en principio" es cierto o no. Habrá casos como el del ejemplo en los que hay escondida algún tipo de redundancia que permita eliminar la dependencia, pero habrá otros casos en que no.
2. El recubrimiento minimal de ***F*** no tiene por qué ser único, dependerá mucho del orden en el que intente simplificar las dependencias del paso 2. Veamos que hubiera pasado si en lugar de AB → C por A → C, lo hubiéramos intentado simplificar por B → C:
   Saltándonos el paso que siempre se cumple, nos preguntamos: ¿Se puede deducir B → C con ***F***?:
   Como $(B)^+_F$= BAC. Sí que llegamos a la C. Luego en este ejemplo, también podríamos haber simplificado AB → C por B → C; por lo que el paso 2 da un resultado distinto al anterior G'={ B → C, A→B, B→A}, y si con ese resultado ejecutáramos el paso 3, veríamos que no sobra ninguna dependencia de ***G'***, por lo que al final hay dos recubrimientos minimales: { A → C, A→ B, B → A } y { B → C, A → B, B → A }.

¿ Para qué sirve el recubrimiento minimal ?. Una vez que hemos hallado el recubrimiento minimal varias tareas necesarias en el proceso de normalización se verán simplificadas, en cuanto ahora operaremos con un conjunto más reducido de dependencias. Estas tares son:

1. Calcular las claves de la relación.
2. Analizar la forma normal en la que está la relación

3. Analizar cómo descomponer una relación en otras que estén en una forma normal superior de forma más rápida.

2.2.5 *Formalización del concepto de clave*

Definición: Dado el esquema ***R*** = (A1, A2, … , An), con el conjunto de dependencias funcionales *F*. Si *X* es un subconjunto de atributos de ***R***, decimos que ***X*** es una clave de ***R*** si:

1. $X \rightarrow R \in F^+$ (esto es, ***X*** es superclave), y
2. No existe otro conjunto de atributos $Y \subset X$, tal que también $X \rightarrow R \in F^+$ (esto es, ademas de ser superclave, es minimal, o dicho de otra manera: no contiene otra superclave en su interior, salvo ella misma).

Por tanto, una clave candidata es un conjunto minimal del atributos que determina funcionalmente a todos los atributos de la relación.

2.2.5.1 Procedimiento para hallar la clave

Si el recubrimiento es el conjunto vacío (es porque todas las dependencias son triviales),
entonces la única clave está formada por todos los atributos de la relación.
Sino:
Atts := {todos los atributos que nunca están a la derecha de ninguna dependencia del recubrimiento minimal}
Por cada dependencia en el recubrimiento minimal analizar si
- cada una de las partes **izquierdas** de las dependencias U ***Atts***, ó
- cada combinación entre varias partes **izquierdas** de las dependencias U ***Atts*** con otros atributos
- determinan la relación, descartando aquellas combinaciones que tengan dentro una clave.

Observación: Es interesante percatarse de qué atributos nunca están a la derecha de ninguna dependencia del recubrimiento minimal (i.e., conjunto de atributos ***A*** en el procedimiento). Si un atributo *A* no está en la parte derecha de ninguna dependencia, es que ese atributo no le puede determinar nadie más que si mismo, esto es $A \rightarrow A$. Lo que significa que ese atributo *A* ha de ser forzosamente parte de todas las claves posibles [de Miguel y Piattini 1993][3]. Esta idea en muchos casos nos reducirá el número de combinaciones a probar si son clave, y otras veces nos permitirán hallarlas casi de forma directa. Por ello, se ha incluido ese paso en el procedimiento.

Ejemplo:

Con la relación

Alumnos_Asignaturas (DNI, Nombre, Color_Pelo, Cod_Asignatura, Nro_Créditos)

y el minimal:

$F = \{$ $DNI \rightarrow nombre$, $DNI \rightarrow color\_pelo, nombre \rightarrow DNI$, $cod\_asignatura \rightarrow nro\_créditos\}$

Hallar la(s) clave(s):

cod_asignatura (i.e., *Atts*={*cod_asignatura*}) es el único atributo que nunca está a la derecha de ninguna dependencia, por lo que según la observación que hemos hecho, es necesario que este atributo esté en todas las claves, por lo que sólo probaremos combinaciones con *cod_asignatura*

3 A su vez este libro referencia esta idea como parte del trabajo fin de carrera de la Universidad Politécnica de Madrid de 1992 de H. Pinilla *Desarrollo de una herramienta para la ayuda al diseño de Bases de Datos Relacionales.*

Probamos con la parte izquierda de cada dependencia:

- La única parte izquierda que tiene *cod_asignatura* es la de la dependencia cod_asignatura → nro_créditos, $(cod\_asignatura)^+$ = (cod_asignatura, nro_creditos) => *cod_asignatura* no es clave porque faltan atributos, como por ejemplo DNI
- Probamos con combinaciones que tengan una parte izquierda de una dependencia más el atributo *cod_asignatura*
    - Por ejemplo, con DNI → nombre
      $(DNI, cod\_asignatura)^+$ = (DNI, cod_asignatura, nombre, color_pelo, nro_creditos)
      => (DNI, cod_asignatura) es clave, y ya no probaremos ninguna combinación que tenga simultáneamente DNI y cod_asignatura, porque esa combinación sería una superclave y no una clave.
    - Por tanto, nos saltamos la dependencia DNI → color_pelo, porque nos llevaría otra vez a probar (DNI, cod_asignatura)
    - Tomamos la siguiente dependencia: nombre → DNI, y probamos con $(nombre, cod\_asignatura)^+$ = (nombre, cod_asignatura, DNI, color_pelo, nro_creditos) => (nombre, cod_asignatura) es clave, y ya no probaremos ninguna combinación que tenga simultáneamente nombre y cod_asignatura, porque esa combinación sería una superclave y no una clave.
    - Como no nos quedan más combinaciones que
        - Tengan *cod_asignatura* y al menos alguna(s) de las partes izquierdas de las dependencias, pero a la vez
        - No sean ni contengan las 2 claves halladas (DNI, cod_asignatura) y (nombre, cod_asignatura)

      No habría ninguna clave mas.

Ejemplo:

Con la relación R={ A, B, C} y su minimal G={ A → B, B → C, C → A }, hallar todas las claves:

Sol: No hay ningún atributo que no esté a la derecha de ninguna dependencia (*Atts* sería el conjunto vacío), luego no tenemos combinaciones para descartar inicialmente. Probamos con las partes izquierdas de las dependencias

- A → B, A^+=ABC, **A es clave**.
- B → C, B^+=BCA, **B es clave**.
- C → A, C^+=CAB, **C es clave**.

No ha lugar a añadir atributos en las partes izquierdas ya que por si solas son todas claves, si las añadiésemos atributos obtendríamos superclaves, pero no claves.

Ejemplo:

Con la relación R={ A, B, C, D} y su minimal G={ A → B, B → C, C → A }, hallar todas las claves:

Sol: En este caso la D es el único atributo que nunca está a la derecha, *Atts*={D}, luego estará en todas las claves. Por lo que sólo probaremos combinaciones que tengan la D. Como ninguna dependencia tiene la D en la izquierda, iremos probando cada parte izquierda más la D:

- A → B, DA^+=DABC,=> **es clave**.
- B → C, DB^+=DBCA,=> **es clave**.
- C → A, DC^+=DCAB,=> **es clave**.

Ahora habría que probar combinaciones que:

1. Tengan la D y al menos alguna parte izquierdas de alguna dependencia (esto es: A o B o C)
2. Pero no sean ni contengan AD, BD o CD, porque esas tres ya son claves

Pero no existen combinaciones que cumplan ambos requisitos, por lo que no hay más claves.

Ejemplo:

Con la relación R={ A, B, C, D} y su minimal G={ AB → C, C → D, D → A }, hallar todas las claves:

Sol: B, nunca está a la derecha, *Atts*={B} , probaremos combinaciones que tengan B

Dependencia AB → C =>$(AB)^+$=ABCD, que **sí es clave**.

Dependencia C → D, $(CB)^+$=CBDA, que **sí es clave**.

Dependencia D → A, $(DB)^+$=DBAC, que **sí es clave**.

Ahora habría que probar combinaciones que:

1. Tengan la B y al menos alguna parte izquierdas de alguna dependencia (esto es: AB o C o D)
2. Pero no sean ni contengan AB, CB o DB, porque esas tres ya son claves

Pero no existen combinaciones que cumplan ambos requisitos, por lo que no hay más claves.

2.3. La Segunda Forma Normal (2FN)

Definición: Una dependencia funcional A → B sobre una relación *R*, se denomina ***dependencia funcional parcial***, si existe un X ⊆ *R*, tal que X ⊂ A y X → B. En tal caso diremos que B ***depende parcialmente*** de A.

Ejemplo (tomado de [de Miguel y Piattini 1993] pg 561)**:**

Sea la relación *publica (título-artículo, revista, número, página, editorial)* con las siguientes dependencias:

F = { título-artículo, revista, número → página;
revista → editorial }

Vemos que editorial depende parcialmente de (*título-artículo, revista, número*), pues depende de una parte, que es revista.

Definición de la Segunda Forma Normal (2FN): Se dice que una relación ***R*** está en segunda forma normal si:

- ***R*** está en 1FN y
- Cada atributo
  - O pertenece a una clave candidata
  - O no depende parcialmente de una clave candidata

Definición de la 2FN según Date: [Date 1994] Se dice que una relación ***R*** está en segunda forma normal si está en 1FN y todos los atributos no clave dependen por completo[4] de la clave candidata.

Ejemplo:

Sea el siguiente esquema de relación en el que *idPedido* es un número secuencial que identifica los pedidos, *fecha* es la fecha de un pedido, *cantidad* es la cantidad de cada uno de los productos que constituyen un pedido, *idProducto* es un código o referencia que permite identificar cada producto que constituye un pedido, *precio* es el precio del producto, y *cantidad_disponible* es el número de productos con ese *idProducto* que tenemos en el almacén.

Analizando el enunciado resulta:

- la relación *Tienen (idPedido, fecha, cantidad, idProducto, precio, cantidad_disponible)*
- las siguientes dependencias funcionales:

F = { idPedido → fecha;
idProducto → precio, cantidad_disponible;
idPedido, idProducto → cantidad }

Podríamos comprobar que si pasásemos ***F*** a forma canónica, sería minimal. Si halláramos las claves de *Tienen*, nos

4 Realmente en la traducción del libro de Date no dice "*por completo*" sino *"irreductiblemente",* que significa lo mismo que depender por completo, pero utilizando la terminología particular de este autor.

daríamos cuenta que tiene una única clave que es (idPedido, idProducto).

Pues bien, esta relación no estaría en 2FN, porque existen atributos que no son de la clave y dependen parcialmente de ella. Por ejemplo, *fecha* depende sólo del *idPedido,* o mismamente también tanto *precio* como *cantidad disponible* dependen sólo de *idProducto.* Esto es, hay 3 dependencias parciales de la clave candidata, bastaba con que hubiera habido tan solo una para diagnosticar que la relación *Tienen* no estaba en 2FN.

Lo ideal hubiera sido dividir la tabla en tres, como de hecho aprenderás hacer en la sección 3.3, y también en el próximo tema:

1. Pedidos (IdPedido, fecha)
2. Productos (idProducto, precio, cantidad_disponible)
3. LineaDetalle (idPedido, idProducto, Cantidad)

Puedes comprobar como ejercicio que las 3 tablas sí están en 2FN.

Date define los diagramas de dependencias [Date 1994], de forma que cada parte izquierda o derecha de una dependencia lo encierra en una cajita, y luego pinta la flecha correspondiente a cada dependencia. Para el caso del ejemplo el diagrama sería:

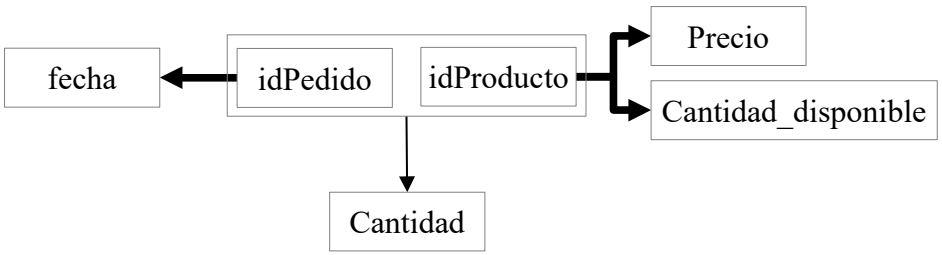

Figura 9: Diagrama de dependencias funcionales de Date para el ejemplo.

Una observación importante es darse cuenta de que para depender parcialmente de alguna clave, tiene que existir al menos una clave compuesta (que pueda dividirse en partes). **Si una relación no tiene <u>ninguna</u> clave compuesta, por tanto, podemos asegurar sin temor que está ya en 2FN.**

En el diagrama se han marcado más gruesas las dependencias parciales que provocan no estar en 2FN. Se ve muy claramente que tienen que partir de un rectángulo que esté dentro de otro, el rectángulo exterior habrá de ser una clave compuesta para causar la ausencia de 2FN.

Una aclaración más que se deriva de la propia definición: si existiese más de una clave, basta con que haya una dependencia parcial <u>de un atributo que no pertenezca a ninguna de las claves</u>, y que esa dependencia parcial sea contra <u>alguna</u> de las claves, para que la relación no esté en 2FN.

Es decir, para comprobar que no hay 2FN:

1. Si no está en 1FN => No está en 2FN
2. Si no, si no tiene alguna clave compuesta => Si está en 2FN
3. Si no
    1. Buscamos los atributos que no estén en ninguna clave
    2. Miramos si alguno de esos atributos depende parcialmente de alguna de las claves
    3. Si existiese alguno => No está en 2FN
    4. Si no => Si está en 2FN

Por ejemplo, supongamos en la tabla del ejemplo hubiésemos añadido el atributo *idLineaDePedido,* que fuese clave:

*Tienen (**idLineaDePedido,** idPedido, fecha, cantidad, idProducto, precio, cantidad_disponible)*

F = { **idLineaDePedido → idPedido, idProducto**

idPedido → fecha;

idProducto → precio, cantidad_disponible;

idPedido, idProducto → cantidad }

Con lo que hay 2 claves: la que teníamos antes *< idPedido, idProducto>*, más la nueva *idLineaDePedido*

1. Suponemos está en 1FN, así que seguimos con el siguiente paso
2. Sí que sigue habiendo una clave compuesta, así que hay que seguir con el siguiente paso
3. Los atributos que no pertenecen a ninguna clave son *fecha, precio, cantidad_disponible* y *cantidad*.
    1. *Fecha* depende parcialmente de la clave *< idPedido, idProducto>*

        => Ya no estamos en 2FN, no seguimos

En definitiva, no basta con no depender parcialmente de alguna clave, como *idLineaDePedido;* para estar en 2FN esos atributos que no pertenecen a ninguna clave, no han depender parcialmente de todas las claves.

2.4. La Tercera Forma Normal (3FN)

Definición de la Tercera Forma Normal (3FN) (tomada casi totalmente de [Silberschatz et. al 2006]): Un esquema de relaciones ***R*** está en 3FN con respecto a un conjunto de dependencias funcionales ***F***, si está en 2FN[5] y para todas las dependencias funcionales de la forma A → B, donde A ⊆ ***R*** y B ⊆ ***R***, por lo menos se cumple una de las condiciones siguientes:

1. A → B es trivial[6] o
2. A es superclave[7] o
3. Cada atributo perteneciente a A-B está contenido en alguna clave candidata de ***R***[8]

En el caso de que estuviéramos manejando un recubrimiento minimal bastaría ver si *R* está en 2FN y las partes izquierdas de las dependencias son claves o están contenidas en alguna clave.

Hay diversos autores (e.g., . [Date 1994], [de Miguel y Piattini 1993], [Connolly y Begg 2005], [Elmasri y Navathe 2007][9]) que presentan una definición en esta otra línea [Codd 1972]:

Una relación está en 3FN si está en 2FN y todos los atributos que no pertenecen a una clave dependen de manera no transitiva toda clave candidata.

Esta definición quizás parezca más fácil de recordar, pero hay situaciones en la que es poco clara. En particular se complica en el caso de que existan los que Date llama atributos "*mutuamente dependientes*". Según [Date 1994] "*Dos o más atributos son mutuamente independientes si ninguno de ellos es dependiente funcionalmente de cualquier combinación de los otros.*"

Por ejemplo, si AB → C y C → AB diremos que A y B son mutuamente dependientes de C, o también si A→ B y B→ A, entonces A y B son mutuamente dependientes entre si. Hay dependencia mutua si podemos encontrar una dependencia cíclica entre grupos de atributos, por ejemplo A → B → C → A también define unas dependencias mutuas entre A y B, entre A y C y entre B y C.

Pues bien, las dependencias transitivas a considerar en esta última definición de 3FN serían sólo aquellas que no se deduzcan usando dependencias de esos ciclos que generan los atributos mutuamente dependientes.

Es decir, si tenemos:

5 Algunos autores (e.g., [Silberschatz et. al 2006], [Ullman y Widom 1997]) en los últimos tiempos omiten la condición previa de que la relación esté 2FN. Esto es porque no explican la 2FN, y se centran únicamente en si la relación alcanza o no la 3FN sin preguntarse si es o no debido a que ni si quiera alcanza la 2FN.

6 Si trabajamos con un minimal, nunca aparecerán dependencias triviales.

7 Si trabajamos con un minimal no va haber partes izquierdas superclaves que no sean claves, por lo que en ese caso la regla podría reescribirse como A es clave.

8 Si trabajamos con un minimal A ∩ B = ∅ , por lo que la regla podría reescribirse como A está contenido en una clave de *R*.

9 En realidad [Elmasri y Navathe 2007] utiliza las dos definiciones, al igual que hacemos nosotros.

- ***F*** = { A → B, B → A y A → C} (dos claves: A y B)
- C depende transitivamente de A porque A→B→C, con lo que tomando la definición anterior al pie de la letra ya no estaríamos en 3FN
- Pero como A y B dependen entre si mutuamente no puedo usar las dependencias A → B y B → A para computar posibles dependencias transitivas de las claves.
- Por tanto, si no tengo en cuenta esa dependencia transitiva A→B→C, ya no quedan dependencias transitivas de la clave, y por onsiguiente el ***F*** del ejemplo sí que esta en 3FN, como ya sabíamos de la primera definición que hemos dado.

La conclusión es que quizás sea mejor recordar la primera definición (más larga) y no tener que pensar en estas excepciones.

Por lo tanto, para comprobar que no hay 3FN, **si el conjunto de dependencias F es minimal**:

1. Si no está en 2FN => No está en 3FN
2. Si no
    1. Buscamos una dependencia en la que la parte izquierda
        1. ni sea clave
        2. ni sea parte de alguna clave
    2. Si la encontramos => No está en 3FN
    3. Si no => Si está en 3FN

Ejemplo:

Sea el siguiente enunciado:

> *Se quiere hacer una base de datos de los socios de un club. Por cada socio guardaremos su DNI, que los identifica unívocamente, su nombre, ciudad y país de nacimiento. Por conveniencia, asumiremos que no puede haber dos ciudades con igual nombre en el mismo país.*

Nuevamente, intentemos modelarlo mediante una única tabla:

Socios (DNI, nombre, Ciudad, País)

Que da lugar al siguiente **conjunto minimal** de dependencias funcionales:

F = { DNI → nombre, DNI → ciudad, ciudad → país }

Si calculamos las claves, se obtiene una sola clave que es DNI. Ahora veamos si está en 3FN. Según la primera definición:

- Está en 2FN porque la única clave que hay es simple, y por tanto no se puede depender de una parte de la clave.
- *DNI → nombre* y *DNI → ciudad* no impiden estar en 3FN porque la parte izquierda (o determinante) es clave

    Si no estuviéramos trabajando con un minimal aplicaríamos la definición del principio de la sección y veríamos que estas dependencias tampoco impiden estar en 3FN, porque entonces exigiríamos que la parte izquierda fuese superclave, y DNI al ser clave es también superclave.
- *ciudad → país*, es una dependencia en la que la parte izquierda ni es clave ni pertenece a una clave, luego está impidiendo que la relación esté en 3FN

    Si no estuviéramos trabajando con un minimal aplicaríamos la definición del principio de la sección y veríamos que esta dependencias también impiden esta en 3FN porque (a) ni es trivial, (b) ni ciudad es una superclave, y (c) ciudad (que es un atributo que está a la izquierda pero no a la derecha) no forma

parte de una clave.

Por tanto la dependencia ciudad → país es la que está impidiendo que la relación esté en 3FN.

Quizás se vea más claro con la definición de Date:

- Está en 2NF porque la única clave que hay sólo tiene un atributo, luego no se puede depender parcialmente de ella.
- Los atributos no clave son *nombre*, *ciudad* y *país.*
  - *Nombre* depende directamente de la única clave que hay (*DNI*), *ciudad* también,
  - Pero *país* no; porque *país* de quien depende directamente es de *ciudad,* luego *país* depende de forma transitiva de la clave *DNI*:

DNI → Ciudad → País

Quizás se ve más claro utilizando un diagrama de dependencias de Date: Se aprecia como a *país* no llega una flecha directa desde *DNI*, sino que necesita un eslabón intermedio que es *ciudad.*

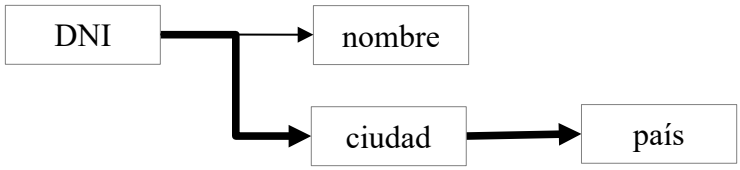

Figura 10: Diagrama de dependencias funcionales de Date para el ejemplo.

2.5. Forma Normal de Boyce – Codd (FNBC)

Definición de la FNBC (tomada de [Silberschatz et. al 2006]): Un esquema de relaciones R está en FNBC si para todas las dependencias funcionales de la forma X → A, donde X ⊆ R y A ⊆ R (por tanto, toda dependencia en forma canónica), se cumple al menos una de estas condiciones:

- X → A es una dependencia funcional trivial (i.e. A ⊆ X).
- X es superclave del esquema R.

Nota que no es necesario comprobar previamente si la relación *R* está o no en 2FN y en 3FN, pues si se verifican estas dos condiciones podemos asegurar que *R está en FNBC* y que por lo tanto estaría en 2FN y 3FN.

Además, **si el conjunto de dependencias fuese un minimal, bastaría ver si todo determinante (parte izquierda de una dependencia) es clave candidata,** ya que en un minimal no hay dependencias triviales, y si un determinante es clave, como en un minimal los determinantes están simplificados al máximo, sería clave.

Por lo tanto, para comprobar que no hay FNBC, **si el conjunto de dependencias F es minimal**:

1. Buscamos una dependencia en la que la parte izquierda no sea clave
   1. Si la encontramos => No está en FNBC
   2. Si no => Si está en FNBC

/*nota que si F= ∅ => R está en FNBC, ya que si no hay dependencias, tampoco las hay que incumplan que la parte izquierda sea clave*/

Normalmente cuando una relación está en 3FN, suele estar en FNBC. En 3FN había tres opciones: (a) que la dependencia fuese trivial, (b) que el determinante fuese superclave, y (c) una tercera opción (que es la que la FNBC no tiene) consistente en que los elementos del determinante que no estuviesen a la derecha de la dependencia fuesen

parte de alguna clave. En el ejemplo posterior veremos algún caso de 3FN que no está en FNBC.

Ejemplo:

Supongamos el siguiente enunciado:

> *Una base de datos quiere registrar la farmacias de un país con la localización en las mismas de los medicamentos. Para ello, se registrarán:*
>
> - *Las farmacias que se identificarán mediante un campo idFarmacia. Además guardaremos la dirección y ciudad de cada farmacia. Alternativamente una farmacia podría identificarse por la combinación de valores de la dirección con la ciudad.*
> - *Los medicamentos, que se identifican por su número de serie, pero también por su nombre.*
> - *Además por cada medicamento registraremos la estantería que está en cada farmacia. La estantería es un campo de texto que puede tomar valores como por ejemplo, "segunda estantería a mano derecha".*

Al juntar todo en una sola relación, queda:

Medicamentos (Nro_serie, nombre, estantería, idFarmacia, dirección, ciudad)

Entonces el conjunto de dependencias que se obtiene es:

F = { nro_serie → nombre; nombre → nro_serie;
idFarmacia → dirección, ciudad, dirección, ciudad → idFarmacia
nro_serie, idFarmacia → estantería }

y además es minimal. Hay varias claves: *<nro_serie, idFarmacia>, <nombre, idFarmacia>, <nro_serie, dirección, ciudad>* y *<nombre, dirección, ciudad>*

Veamos si estamos en 2FN: El único atributo que no pertenece a una clave es *estantería*, y *estantería* depende por completo de la clave en las 4 claves halladas. Por tanto sí que estamos en 2FN.

> **Ojo**, a lo largo de mi vida como profesor he visto como muchos alumnos han puesto en un examen que una relación como la del ejemplo no estaba en 2FN. Ese error lo cometen al identificar dependencias parciales de una clave candidata, como por ejemplo *nro_serie→nombre*, sin pararse a pensar que esta dependencia no habría que tenerla en cuenta debido a que nombre pertenece a una clave candidata.

Veamos si estamos en 3FN:

1. En la dependencia *nro_serie, idFarmacia → estantería*, el determinante es clave
2. En las otras 4 dependencias, los determinantes no son claves, pero pertenecen siempre a alguna clave.

Por tanto, estamos en 3FN; pero no estamos en FNBC precisamente por esas últimas 4 dependencias. Pues BCFN exige que en todas las dependencias no triviales el determinante sea clave, y eso sólo ocurre con *nro_serie,idFarmacia → estantería*.

Con diagramas de Date veríamos lo siguiente:

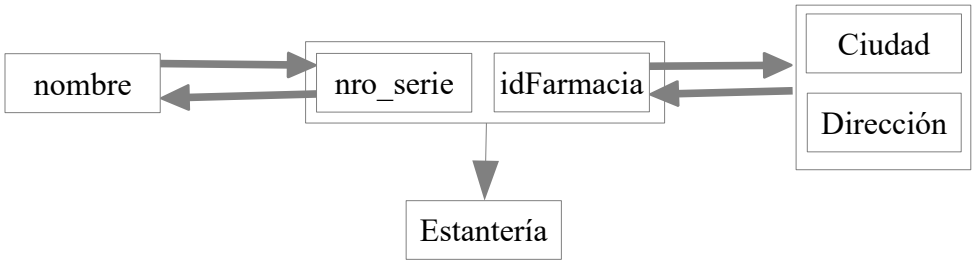

Figura 11: Diagrama de Date correspondiente al ejemplo.

Vemos que existen 4 determinantes que no son claves (ver flechas más gruesas), por lo que no estamos en FNBC. Siguiendo la definición de Date para 3FN, el único atributo que no es de alguna clave (*estantería*) no depende transitivamente de ninguna clave. Observa que hay 4 claves, siendo uno de esos casos en los que no podemos tener en

cuenta las dependencias mutuas para ver si *estantería* depende transitivamente de alguna clave. Esto es: podríamos pensar, por ejemplo que:

< nombre, ciudad, dirección > → < nro_serie, farmacia > → < estantería >

dando así lugar a la transitiva por la que la clave < nombre, ciudad, dirección > determina a < estantería >.
Pero recordemos que Date nos dice que no consideremos las transitividades que se deriven de dependencias mutuas, como en este caso < nombre, ciudad, dirección > y < nro_serie, farmacia > que dependen entre si mutuamente.

3. Descomposiciones de un esquema de relación

De los ejemplos que hemos visto podemos deducir que no hemos llegado a la forma normal más elevada debido a que hemos intentado poner todos los atributos en una sola relación, cuando realmente habría que haber ***descompuesto*** esa única relación en varias.
De alguna forma, las reglas de paso del modelo entidad-relación a relacional, que veremos en el próximo tema, ya nos proveen de unos criterios de descomposición muy potentes, y muy probablemente serán las reglas que casi siempre utilicemos en la práctica.
En esta sección prescindiremos por el momento de esos criterios para razonar unos criterios de descomposición basados únicamente en la teoría de normalización y las dependencias funcionales.

Si tenemos una relación no está en la forma normal más elevada posible, habrá que descomponerla en varias de manera que cada una de estas:

1. Esté **en la forma normal más elevada posible** (i.e., al menos 3FN).
2. La unión de los atributos de las descomposiciones sea igual a al conjunto de atributos que teníamos en la relación de partida **(**i.e., **no haya pérdidas de atributos).**
3. **No haya pérdidas de dependencias**.
4. **No haya pérdidas de producto**.

Es decir sea **una descomposición "sin pérdidas"**. Dada una relación *R* que no está en 3FN **siempre es posible descomponer** *R* **en varias tablas más pequeñas que sí estén, al amenos, en 3FN obteniendo una descomposición sin pérdidas**. Sin embargo, hay ocasiones en las que no se puede encontrar una descomposición de *R* en tablas más pequeñas que estén todas en FNBC.

3.1. Conservación de las Dependencias

Veamos en qué consiste con un ejemplo. Sea la relación ***R*** con un conjunto de dependencias ***F***:

R = (Nro_Pedido, Cod_Cli, nombre_Cli, Nro_Producto)
F = { Nro_pedido → Cod_Cli, nombre_Cli, Nro_Producto,
Cod_Cli → nombre_Cli }

La clave lógicamente es el número de pedido. Como todas las claves que tiene son de un solo atributo no puede haber dependencias parciales de la clave, luego está en 2FN. Sin embargo en la dependencia *CodCli → nombre_cli*, la parte izquierda ni es clave ni pertenece a una clave, por lo que no está en 3FN. También podríamos haberlo visto al darnos cuenta de que el nombre del cliente depende transitivamente de la clave candidata. Por tanto hay que descomponer ***R*** en relaciones más pequeñas para intentar llegar a varias relaciones en una forma normal superior.
Hagamos por ejemplo la descomposición:

R1 = { Nro_Pedido, Cod_Cli, Nro_Producto }

R2 = { Nro_Pedido, nombre_Cli }

En principio vamos bien, porque no hemos perdido ningún atributo. La unión de los atributos de ***R1*** y ***R2*** es ***R***.

Sean F_i los subconjuntos de ***F*** que se pueden verificar utilizando únicamente atributos de R_i. Entonces:

F1 = { Nro_pedido → Cod_Cli, Nro_Producto }

F2 = { Nro_pedido → nombre_Cli }

En esta descomposición se conservarían las dependencias, si las dependencias originales pudieran deducirse a partir de las que quedan. Es decir si:

$$(F_1 \cup F_2)^+ = F^+$$

En el ejemplo vemos que en la unión de *F1* y *F2* falta la dependencia *Cod_Cli → nombre_Cli* y que esa dependencia no puede deducirse a través de las que quedan. Por tanto es una descomposición con pérdidas de dependencias funcionales.

¿Por qué es importante mantener las dependencias?.

Las dependencias en el fondo no son más que restricciones. Si declaramos *Cod_Cli → nombre_Cli* estamos prohibiendo que haya dos filas con el mismo número de cliente y distinto ~~mismo~~ nombre.

Nota que esa dependencia la llevamos incluso al SQL cuando en la tabla de *clientes* (que es una de las que teníamos que haber obtenido al descomponer) hubiéramos declarado *Cod_Cli* como clave primaria. En ese caso 2 clientes con el mismo *Cod_Cli* no pueden tener el ~~mismo~~distinto *nombre*, sencillamente porque al ser el *Cod_Cli* clave primaria, no podría haber 2 clientes con el mismo *Cod_Cli*.

Por ello, en la descomposición obtenida *(R1* y *R2),* es fácil controlar que cada vez que por ejemplo insertemos una fila en *R1* no se viole la dependencia *Nro_pedido → Cod_Cli, Nro_Producto*, basta con hacer que *Nro_pedido* sea clave primaria, no se repetirán filas con igual *Nro_pedido,* y por tanto, cuando recuperemos *R* haciendo el *join* entre *R1* y *R2* mediante el único campo común (i.e., *Nro_pedido),* no encontraremos dos números de pedidos iguales con dos códigos de cliente distintos, o dos números de pedidos iguales con dos números de producto distintos.

El mismo razonamiento cabría para cuando insertásemos una fila en *R2*. La dependencia en *F2* se mantendría con sólo declarar *Nro_Pedido* como clave primaria en *R2.*

En las bases de datos modernas estas comprobaciones que van asociadas a la clave primaria se realizan muy rápidamente, pues las tablas crean siempre internamente un índice por dicha clave primaria. Un índice permite hacer búsquedas haciendo pocas lecturas, es decir rápidamente; de la misma manera que los índices de los libro permite buscar contenidos sin que tengamos que leernos el libro “de cabo a rabo”. Una simple búsqueda en el índice, del número de pedido que estamos insertando, nos dirá si ya hay una fila con ese valor, rechazando dicha inserción en tal caso.

Sin embargo, al perder *Cod_Cli → nombre_Cli*, si inserto una fila en *R1* con un código de cliente, y otra en *R2* con un nombre, ¿cómo comprobamos que no se viola esa dependencia?, ¿cómo evitamos que el mismo cliente tenga dos nombres?, ¿cuál de los dos nombres es el bueno?.

Ya no hay clave primaria que haga un índice por *cod_cli*. Tendríamos que hacer el *join* de ambas tablas y buscar en el resultado si hay alguna otra fila con el mismo código de cliente que tenga un nombre distinto (ver Figura 12), quizás lanzando esa consulta desde un aserto, *triggers* o el propio programa de aplicación desde el que se hagan las inserciones. Hacerlo así sería muy costoso, y además habría que ejecutar esa comprobación en multitud de ocasiones (i.e., cada vez que insertemos una fila en *R1* o *R2,* pero también cuando cambiemos los valores en uno de los dos campos *nombre* o cod_*cli*).

```
SELECT Cod_cli FROM R1 natural join R2
GROUP BY Cod_cli
HAVING COUNT(DISTINCT nombre_cli) > 1;
```

Figura 12: Consulta que habría que ejecutar para comprobar que se mantiene la dependencia *CodCli → nombre_cli* si hiciéramos la descomposición propuesta. Si la consulta no devolviese filas no se violaría la dependencia. En caso contrario sí se violaría.

3.2. El Problema de las Pérdidas de Producto

Este problema en realidad ya le habíamos comentado al principio del tema, en la sección *1.3.Pérdida de Información*. Volvamos al mismo problema con más conocimientos para analizarlo.

Supongamos ahora el siguiente enuciado:

> *Se quiere hacer una base de datos en la que se registren los profesores con las asignaturas que imparten. Un profesor puede impartir varias asignaturas y una asignatura puede ser impartida por un grupo de profesores. Los profesores se identifican por su nombre (nompreP), y de ellos guardaremos su titulación, las asignaturas de identifican por su nombre (nombreA) y registraremos su número de créditos.*

supongamos que lo juntamos todo en una única tabla:

$$R = (\ nombreP,\ título,\ nombreA,\ nCréditos\ )$$

con el conjunto de dependencias:

$$F = \{\ nombreP \rightarrow título,\quad nombreA \rightarrow nCréditos\ \}$$

La clave es obviamente es la compuesta por *nombreP* y *nombreA*. Como *título* y *nCréditos* no pertenecen a ninguna clave y dependen parcialmente de ésta, podemos decir que ***R*** no está en 2FN. Por tanto hay que descomponer ***R*** en relaciones más pequeñas para intentar llegar a varias relaciones en una forma normal superior.

Hagamos por ejemplo la descomposición:

R1 = (nombreP , título)

R2 = (nombreA, nCréditos)

En principio vamos bien, porque no hemos perdido ningún atributo. La unión de los atributos de ***R1*** y ***R2*** es ***R***. Los conjuntos de dependencias que se obtienen ahora son:

F1 = { nombreP → título }

F2 = { nombreA → nCréditos }

y por lo tanto, tampoco se pierden dependencias porque ***F1*** ∪ ***F2*** = ***F***, luego obviamente $(F_1 \cup F_2)^+ = F^+$.

En cada relación el *nombreP/A* respectivamente es la única clave. Son dos relaciones, por tanto en FNBC, porque en todas las dependencias la parte izquierda es clave.

Sin embargo, no podemos recuperar la relación inicial a partir de *R1* y *R2*. Si nos pidieran un listado de profesores con sus asignaturas, no podríamos hacerlo, porque no tenemos ninguna forma de relacionar las filas de ***R1*** con las de ***R2,*** al no tener campos comunes.

Si ambas tablas tuviesen algún o algunos campos comunes podríamos utilizar la operación de *join* para intentar recuperar la relación de partida ***R***.

El problema de las pérdidas de producto lo que hace es señalar que **la elección de estos atributos comunes no puede ser aleatoria** si pretendemos recuperar la relación de partida a partir de operaciones de *join* entre las descomposiciones obtenidas.

Lógicamente, cualquiera que sepa un poco no elegiría como atributo común, por ejemplo, el número de créditos. Supongamos que lo hacemos mal adrede mediante esta nueva descomposición:

R1 = (nombreP , título, nCréditos), F1 = { nombreP → título }

clave compuesta: nombreP+nCréditos, ni siquiera en 2FN

R2 = (nombreA, nCréditos), F2 = { nombreA → nCréditos }

clave: nombreA, en FNBC

demos algunos valores a las tablas, sea ***R***:

NombreP	Título	NombreA	Ncréditos
Pepe	Ingeniero Informático	Bases de Datos	6
Ana	Lda. Física	Bases de Datos	6
Pepe	Ingeniero Informático	Fundamentos de Programación	8
Ana	Lda. Física	Ofimática	6

Figura 13: Valores de ejemplo para *R*.

Con lo que ***R1*** y ***R2*** quedan así:

R1

NombreP	Título	Ncréditos
Pepe	Ingeniero Informático	6
Ana	Lda. Física	6
Pepe	Ingeniero Informático	8

R2

NombreA	Ncréditos
Bases de Datos	6
Fundamentos de Programación	8
Ofimática	6

Figura 14: Extensiones de las relaciones R1 y R2 resultantes de la descomposición.

En ambas relaciones el número de filas es menor que en *R* porque desaparecerían las filas repetidas, ya que es una restricción inherente del modelo relacional que toda relación tenga clave primaria, y por consiguiente, que no haya filas repetidas.

Si intentamos recuperar ***R*** haciendo join por el campo común se obtiene:

NombreP	Título	NombreA	Ncréditos
Pepe	Ingeniero Informático	Bases de Datos	6
Ana	Lda. Física	Bases de Datos	6
Pepe	Ingeniero Informático	Fundamentos de Programación	8
Ana	Lda. Física	Ofimática	6
Pepe	Ingeniero Informático	Ofimática	6

Figura 15: Resultado de R1 join R2. La última fila es *espuria.*

La última fila no estaba en la relación de partida *R*, es lo que se llama una ***fila espuria***. La razón de que aparezca es que hay dos asignaturas de 6 créditos y al relacionar la fila de Pepe con 6 créditos, ésta se combinará con las dos asignaturas.

3.2.1 La regla de Rissanen

> ***Teorema:*** Sea ***R*** una relación y ***F*** un conjunto de dependencias funcionales en ***R***. Entonces ***R1*** y ***R2*** forman una descomposición sin pérdida de producto de ***R*** si ***R1*** ∩ ***R2*** → ***R1*** o bien ***R1*** ∩ ***R2*** → ***R2***
> [Rissanen 1977].

Es decir; ¿por qué hay pérdidas de producto según la regla de Rissanen en la descomposición anterior?, porque el atributo de la intersección, esto es: el atributo en común, que es *nCréditos*, no es clave ni de R1, ni de R2.
En capítulos anteriores hemos procedido de manera intuitiva, identificando relaciones padre-hijo entre tablas y llevando cada clave primaria de cada tabla padre como cave ajena de cada tabla hija. Si lo hiciésemos así obtendríamos el siguiente esquema descompuesto en 3 relaciones, R1, R2 y R3, que representan respectivamente a los profesores, las asignaturas, y los pares profesor – asignatura que indican qué asignaturas imparte cada profesor:

R1 = (nombreP , título), F1 = { nombreP → título }
clave nombre~~A~~P, en FNBC
R2 = (nombreA, nCréditos), F2 = { nombreA → nCréditos }
clave: nombreA, en FNBC
R3 = (nombreA, nombreP), F3 =∅
clave compuesta < nombreA, nombreP >, en FNBC porque no existen dependencias no triviales en la que la parte izquierda no sea clave.

R1 (profesores) es padre de R3, y R2 (~~alumnos~~asignaturas) también es padre de R3. Las claves ajenas en R3 son precisamente esos atributos comunes con R1 y R2 que harían que se cumpliese la regla de Rissanen, pues por ejemplo:

- *R1* hace *join* con *R3* sin pérdidas de producto, porque su atributo en común, *nombreP* es clave de *R1*.
- La relación resultante del join *R1joinR3 = (nombreP , título, nombreA)* también hace *join* con *R2* sin pérdidas de producto, porque el atributo común entre las dos (i.e., *nombreA*) es clave de una de las dos (de *R2*).

La regla de Rissanen nos dice algo que en el fondo ya habíamos asimilado hace mucho tiempo, que es que los campos que utilizamos para hacer *join* entre dos tablas, son en una de las dos tablas clave ajena, y en la otra clave primaria. Cuando hacemos un *join* siempre igualamos la clave primaria de una tabla con la clave ajena de la otra. Pues bien, Rissanen dice precisamente que los campos comunes de las dos tablas han de ser clave primaria de una de ellas[10].

Ejemplo (tomado de la sección 7.5.1.2 de [Silberschatz et. al 2006])**:**
Sea la relación:

Prestar (sucursal, activo, ciudad_suc, nro_préstamo, nom_cliente, importe)

con las dependencias:

F = { sucursal → activo, ciudad_suc, nro_préstamo → nom_cliente, importe, sucursal }

El alumno puede comprobar que pasando *F* a forma canónica, *F* es minimal, y que la única clave es ~~la compuesta por sucursal y~~ número de préstamo, y que ~~ni si quiera~~ está en 2FN pero no en 3FN.

10 Realmente la regla de Rissanen también validaría (i) que esos campos comunes fuesen una superclave que no fuese clave, pero en ese caso habría al menos un campo innecesariamente duplicado en las 2 tablas, creando alguna redundancia que impediría que ambas tablas alcanzaran la FN más elevada posible, y (ii) también valdría una clave candidata UNIQUE (por eso SQL permite poner REFERENCES tabla_padre (campo_UNIQUE_ en la tabla padre), en lugar de simplemente REFERENCES tabla_padre cuando se declara una clave ajena, si bien la utilización de esta posibilidad rara vez está justificada).

Si elegimos la descomposición:

R1 = (sucursal, activo, ciudad_suc) *R2 = (sucursal , nro_préstamo, nom_cliente, importe)*

- No hay pérdida de atributos
- Las dependencias que quedan en cada relación son:

  F1 = { sucursal → activo, ciudad_suc }

  F2 = { nro_préstamo → nom_cliente, importe, sucursal }

  Por lo que no se pierden dependencias ($F_1 \cup F_2 = F \Rightarrow (F_1 \cup F_2)^+ = F^+$).
- El alumno puede comprobar que *R1* esta en FNBC con clave *sucursal*, y *R2* con también, con clave *nro_préstamo* .
- Vamos a comprobar además, si hay pérdidas de producto aplicando la regla de Rissanen. Para ello buscaremos los atributos comunes a *R1* y *R2* (la intersección de ambas relaciones), esto es: *sucursal.* Como *sucursal* es clave de *R1*, la sucursal determina *R1*. Por tanto si que se verifica la regla y no hay pérdidas de producto.

3.2.2 Algortimo para comprobar que una descomposición es sin pérdida de producto

Hay algunas ocasiones en que no es sencillo ver si hay o no pérdidas de producto aplicando la regla de Rissanen. El siguiente algoritmo se presenta en el libro [Ullman 1982] (pp. 227 – 228), también disponible en [Elmasri y Navathe 2007] pp. 321-322, y está basado en la propia regla de Rissanen, pero facilita la comprobación de si hay pérdidas de producto aún cuando el caso sea complejo. Consiste en:

1. Crear una tabla en la que cada fila es una de las descomposiciones y cada columna uno de los atributos de la relación de partida
2. Marcar en cada fila las celdas correspondientes a los atributos que tiene esa descomposición
3. Es con perdida = falso
4. Repetir hasta que una fila tenga todas casillas marcadas o no se puedan aplicar más dependencias

   Por cada dependencia funcional de la relación de partida (la relación a descomponer)

   1. Buscar 2 filas de la tabla en las que:
      1. Estén marcadas todas las casillas correspondientes a los atributos de la parte izquierda
      2. En una de esas dos filas no están marcados todos los de la derecha, pero en la otra sí
   2. Si la encuentra

      Marcar en la fila donde faltan atributos de la derecha los que faltan
5. Si llegado a este punto no hay una fila con todas las casillas marcadas:

   Es con perdida = verdadero

Ejemplo 1 (tomado de la sección 7.5.1.2 de [Silberschatz et. al 2006])**:**

Sea *R = { sucursal, activo, ciudad_sucursal, número_préstamo, importe }* y

F = { sucursal → activo, sucursal → ciudad_sucursal,

número_préstamo → sucursal, número_préstamo → importe }

que descomponemos en :

R1 = { sucursal, activo ciudad_sucursal}

R2 = { número_préstamo, sucursal, importe }

Vamos a comporbar si dicha descomposición tiene pérdidas de producto.

Tras el paso 2 del algoritmo, la tabla resultante sería:

	sucursal	activo	ciudad_sucursal	número_préstamo	importe
R1	X	X	X		
R2	X			X	X

Aplicamos *sucursal → activo, ciudad_sucursal* pues en ambas filas *sucursal* está marcada, y queda

	sucursal	activo	ciudad_sucursal	número_préstamo	importe
R1	X	X	X		
R2	X	X	X	X	X

Y como la fila *R2* ya tiene todas las columnas marcadas, es una descomposición sin pérdida.

En este caso el algoritmo lo que nos ha dicho es que al hacer *join* por sucursal entre *R1* y *R2,* obtenemos *R*, ya que sucursal es un campo común que es clave de *R1,* y que determina a los campos que nos faltaban en *R2.*

Ejemplo 2:

R = ABCDE, F={ ABC → D, E → BDA } claves = compuesta EC

Que descomponemos en:

R1 = ABCD, F1 = { ABC → D }

R2 = EBDA F2 = { E → BDA } y

R3 = EC, F3=∅

Comprobamos si hay pérdidas de producto:

Tabla inicial:

	A	B	C	D	E
R1	X	X	X	X	
R2	X	X		X	X
R3			X		X

ABC → D no se puede aplicar porque no hay dos filas con ABC

E → BDA se puede aplicar con *R2* y *R3:*

	A	B	C	D	E
R1	X	X	X	X	
R2	X	X		X	X
R3	X	X	X	X	X

R3 ya tiene todas las casillas marcadas, luego es sin pérdida.

En este caso el algoritmo lo que nos está diciendo es que hagamos el *join* a través del campo *E* entre las dos filas de la tabla que tienen ese campo en común, y que por otro lado es clave de *R2.* El resultado de ese *join* entre *R2* y *R3* sería el propio *R.*

Ejemplo 3:

R = ABCDE, F = { A → B, A → C, DE → C, DE → B, C → D }

que descomponemos en:

R1 = ABC F1 = { A → B, A → C}

R2 = DECB F2 = { DE → C, DE → B, C → D }

R4 = AE F4 = ∅

Comprobamos si hay pérdidas de producto

Tabla inicial:

	A	B	C	D	E
R1	X	X	X		
R2		X	X	X	X
R4	X				X

$A \rightarrow BC$ se puede aplicar en *R1* con *R4*

	A	B	C	D	E
R1	X	X	X		
R2		X	X	X	X
R4	X	X	X		X

Todavía no tenemos una fila completa en la tabla. Seguimos con otra dependencia:

- $DE \rightarrow BC$ no se puede aplicar porque no hay 2 filas con DE marcada
- $C \rightarrow D$ se puede aplicar tanto en *R2* con *R1* como en *R2* con R4

	A	B	C	D	E
R1	X	X	X	X	
R2		X	X	X	X
R4	X	X	X	X	X

Y como la fila de *R4* esta completa tampoco tiene pérdidas.

En este caso, el algoritmo nos está diciendo que:

- primero hagamos el *join* entre *R1* y *R4* utilizando el campo *A* que es común a ambas, y clave de *R1*.
  Con ese *join* llegamos a una tabla intermedia a la que todavía la falta el campo *D,* como se puede ver en la fila de *R4* en la segunda tabla.
- con el resultado intermedio anterior, representado en la fila *R4,* el algoritmo parece que nos dice que hagamos *join* con *R2* a través del campo *C* que es común tanto a R2 como a ese resultado intermedio. Sin embargo, en este caso *C* no es clave de *R2*, pero *CE* si lo es, y es común entre *R2* y *R4*. Por lo tanto, el algoritmo lo que nos dice es que el *join* entre el resultado intermedio anterior y *R1*, lo hagamos usando *CE*, que es clave de *R1*.

Observa que el algoritmo se ha dado cuenta de que los campos por los que hay que hacer el segundo *join* son conjuntamente *C* y *E,* mientras que si hubiéramos aplicado directamente la regla de Rissanen hubiéramos que haber estado muy atentos para darnos cuenta de ese hecho, e incluso podíamos haber concluido equivocadamente que sí que había pérdida de producto. He ahí la utilidad del algoritmo, para cuando surgen casos con este grado de complejidad.

3.3. Descomposición hasta FNBC y 3FN

Existen múltiples algoritmos para descomponer una relación de partida que no está en la forma normal más elevada posible, en un esquema compuesto de varias relaciones que sí lo están y que no dan lugar ningún tipo de pérdida. En esta sección nos centraremos en la *Síntesis de Bernstein* como algoritmo de descomposición.

3.3.1 El algoritmo de síntesis de Bernstein

Nota: Este algoritmo aparece originalmente en [Bernstein 1976], pero la versión inicial no resuelve bien la pérdida de producto, y por otro lado se complica mucho por intentar contemplar el caso de que haya ciclos entre dependencias. La versión que se presenta aquí es más sencilla y resuelve las posibles pérdidas de

producto:

1. Parte de obtener el recubrimiento minimal. También calcularemos las claves, pues las necesitamos para el último paso.
2. A partir del recubrimiento minimal junta en una misma dependencia aquellas que tengan el mismo determinante (i.e., misma parte izquierda).
3. Por cada dependencia que le queda define una relación *Ri* con todos los atributos de la dependencia. Este paso garantiza que no habrá perdidas de atributos. Además las relaciones así creadas tienen como clave el determinante, y por tanto están en FNBC.
4. Si alguna relación *Ri* estuviese incluida en otra *Rj* quitamos la más pequeña (i.e., *Ri*). En este paso la relación grande *Rj* se queda con su dependencia y la de *Ri*, y es por ello, que ya no lleguemos a FNBC y nos quedemos en 3FN.
5. Finalmente, para garantizar que no haya pérdidas de producto, si ninguna de las relaciones resultantes de la descomposición contiene alguna de las claves de partida calculadas en el paso 1, se añade una última relación con los atributos de una cualquiera de esa claves.
   Este paso cumple la misma función que añadir la típica tabla que representa el rombo de la interrelación en las interrelaciones binarias N:M o en las ternarias N:M:P, son tablas que tan sólo contienen una clave formada por todos sus atributos, y que sirve para conectar a las otras tablas, que no tienen entre sí atributos comunes.

Formalmente, el algoritmo sería como sigue[11]:

Sea F_c un recubrimiento canónico de F
Crear un conjunto F' juntando las dependencias funcionales que tienen igual parte izquierda vía regla de unión
i := 0
for each dependencia funcional en F' $\alpha \rightarrow \beta$ **do**
    if ninguno de los esquemas R_j, j= 1, 2, … , i contiene α y β {
        i := i+1
        $R_i := \alpha\beta$
    }//Fin if
if ninguno de los esquemas R_j j = 1, 2, … ,i contiene la clave candidata de R
    i := i +1
    R_i := cualquier clave candidata de R
}//Fin if
return (R_1, R_2, … , R_i)

Ejemplo 1 (tomado de la sección 7.5.1.2 de [Silberschatz et. al 2006]):
Sea *R = { sucursal, activo, ciudad_sucursal, número_préstamo, importe }*, y F minimal

F = { sucursal → activo, sucursal → ciudad_sucursal,
número_préstamo → sucursal, número_préstamo → importe }

Se deja para el alumno comprobar que **la clave es número_préstamo**. Como la clave tiene un solo atributo ***R*** está en 2FN. Como hay dependencias que tienen en su parte izquierda la sucursal, que no es clave, ***R*** no está en FNBC. Es más, como sucursal no forma parte de una clave, ni siquiera esté en 3FN.
Descompondremos R con la síntesis de Bernstein:

11 Ilustración con el algoritmo tomada de [Silberschatz et. al 2006].

Como F ya es minimal, creamos F':

F' = { sucursal → activo, ciudad_sucursal; número_préstamo → sucursal, importe }

Para la primera dependencia nos sale:

R1 = { sucursal, activo ciudad_sucursal}

Para la segunda

R2 = { número_préstamo, sucursal, importe }

como:

1. R2 no contiene a R1 y viceversa y
2. R2 ya contiene la clave

ya habríamos acabado, y la descomposición obtenida es la formada por R1 y R2.

Queda para el alumno ver qué forma normal se ha alcanzado y si hay algún tipo de pérdida. Está garantizado por el algoritmo que como poco se alcance la 3FN en todas la Ri en las que hemos descompuesto, y está garantizado que no hay ningún tipo de pérdidas; pero es un buen ejercicio comprobarlo.

Ejemplo 2:

R = ABCDE, F={ ABC → D, E → BDA } claves = compuesta EC

F' ya nos le dan hecho

Con cada dependencia hago una interrelación

R1 = ABCD, F1 = { ABC → D },

R2 = EBDA F2 = { E → BDA }

R2 no contiene a R1 y viceversa, luego se conservan ambas relaciones.

Como la clave no está por completo en ninguna de las relaciones incluimos la clave en una relación aparte.

R3 = EC, F3=∅

La descomposición a la que llegamos incluyes entonces esas 3 relaciones. Como en el ejemplo anterior es un ejercicio interesante que compruebes a que forma normal se ha llegado y que no hay pérdidas.

Ejemplo 3:

R = ABCDE, F = { A → B, A → C, DE → C, DE → B, C → D }

La clave es AE, como hay atributos que no pertenecen a la clave y dependen de parte de ella, como por ejemplo B, no está en 2FN.

Hallamos F' y las relaciones resultantes en un mismo paso para ganar tiempo:

R1 = ABC	F1 = { A → B, A → C}
R2 = DECB	F2 = { DE → C, DE → B}
R3 = CD	F3 = { C → D }

Como R2 tiene dentro CD, es decir R3, F2 sería realmente *F2 = { DE → C, DE → B, C → D }*

y R3 desaparecería por estar dentro de R2

Como la clave no está en en ninguna relación se añade:

R4 = AE F4 = ∅

Solución: R1, R2 y R4

Se deja para el alumno ver que hemos llegado a 3FN (que es lo que garantiza Bernstein) pero no a 3FNBC.

Comprueba también que no hay ningún tipo de pérdida, y que si **no** hubiésemos incluido R4 hubiera habido pérdidas de producto.

IV. Resumen

Las técnicas de normalización permiten dar un enfoque más formal y sistemático al proceso de diseño de esquemas relacionales. Su punto de mira está en conseguir esquemas en los que haya la menor redundancia de información posible, debido a los problemas que la redundancia genera. Para ello, primero miden la bondad de un esquema relacional a partir de la forma normal que alcanza.

Aunque hay algunas formas normales más, nosotros sólo hemos visto hasta la forma normal de Boyce-Codd. Las formas normales vistas se definen utilizando el concepto de dependencia funcional. Las dependencias funcionales son restricciones que imponen que cuando unos atributos toman un valor esté determinado el valor que van a tomar otros atributos. Estas restricciones al final se implementan de manera eficiente en las tablas a través de claves primarias o candidatas.

Para utilizar las dependencias funcionales a fin de definir las formas normales, hemos tenido previamente que ver la teoría de las dependencias funcionales, lo que nos ha permitido encontrar conjuntos de dependencias equivalentes a uno de partida, pero lo más reducido posible (i.e, equivalente minimal o recubrimiento minimal). Este resultado nos ha permitido acelerar y simplificar la búsqueda de claves de una relación *R*, determinar su forma normal, y encontrar descomposiciones de *R* que arreglen los problemas de redundancia que se han expresado mediante la forma normal que presenta *R*.

Para ello, se ha presentado (i) un algoritmo sencillo que permite encontrar las claves, (ii) la determinación de las formas normales también se ha presentado en forma algorítmica a fin de sistematizarla, y finalmente, (iii) se ha presentado la síntesis de Bernstein como algoritmo de descomposición que nos permite descomponer *R* en nuevas relaciones que normalmente estarán en la forma normal de Boyce-Codd, pero que al menos estarán en tercera forma normal; no presentando dicha descomposición ningún tipo de pérdida (i.e, ni de atributos, ni de dependencias, ni de producto).

La no - pérdida de dependencias, permitirá el mantenimiento de las dependencias de partida de forma eficiente a través de claves, y la no – pérdida de producto permitirá volver a obtener *R* aplicando la operación de *join* sobre las relaciones fruto de la descomposición obtenida. Debido a que hay casos en los que es más complicado analizar la pérdida de dependencias, también se ha presentado un algoritmo que sistematiza ese proceso.

V. Glosario

Clave. Superclave mínima o que en su interior no tiene otra superclave salvo ella misma.

Cierre

de un descriptor o de un conjunto de atributos. Es el conjunto de atributos que es determinado por ese descriptor o conjunto de atributos.

de un conjunto de dependencias funcionales. Es el conjunto de todas las dependencias funcionales que se pueden deducir partiendo de dicho conjunto de dependencias y utilizando los *Axiomas de Armstrong*.

Dependencia funcional. Es una restricción representada en forma de regla o implicación, de manera que en la parte izquierda y derecha de la regla hay una seria de atributos significando que los atributos de la derecha no pueden tomar una combinación de valores diferentes en dos filas en las que los valores de los atributos a la izquierda de la

regla sean los mismos.

Mutua. Entre 2 descriptores entre los que existe una dependencia funcional, es una dependencia en la que además la parte izquierda también depende de la derecha. Una dependencia mutua puede estar escondida por dependencias transitivas que generan ciclos en el diagrama de dependencias en los que intervienen más de dos descriptores.

Parcial. Una dependencia es parcial respecto a un conjunto de atributos, sí su parte izquierda está incluida estrictamente en ese conjunto de atributos.

Transitiva. Una dependencia es transitiva, si existe otra dependencia con la misma parte izquierda, otra con la misma parte derecha, y la parte derecha de la primera es la misma que la parte izquierda de la segunda.

Trivial. Si la parte derecha de la dependencia coincide o está incluida en la parte izquierda.

Descomposición. De una relación es un conjunto de relaciones en las que la unión de sus atributos coincide con los atributos de la relación de partida.

Descriptor. Conjunto de atributos de una relación.

Determinante. Parte izquierda de una dependencia funcional.

Espuria (fila o tupla). Una fila es espuria si no estaba en una relación *R* pero aparece al hacer *join* entre las relaciones de una descomposición de *R* evidenciando así que hay un problema de pérdida de producto en esa descomposición.

Forma canónica (de una dependencia funcional). Una dependencia funcional está en forma canónica si sólo tiene un atributo en la parte derecha.

Forma normal. Categorización del nivel de redundancia de una relación mediante un criterio objetivo.

Pérdida.

De dependencias. Característica indeseable de una descomposición por la que es imposible volver a deducir todas las dependencias de la relación original a partir de las dependencias que han quedado en las relaciones fruto de la descomposición. La perdida de dependencias ocasiona que en la descomposición no sea eficiente el comprobar si se siguen verificando las restricciones que imponían esas dependencias funcionales que ya no se pueden deducir.

De producto. Característica indeseable de una descomposición de una relación *R,* por la que al hacer *join* entre las relaciones de la descomposición, no se recupera integramente *R*, sino que aparecen algunas fila más que no estaban inicialmente en *R* y que se conocen como filas espurias.

Recubrimiento minimal (de un conjunto de dependencias). Es otro conjunto de dependencias equivalente que por conveniencia viene dado en forma canónica, y que es mínimo en el sentido de que no le sobra ninguna dependencia, y los determinantes de todas sus dependencias no pueden simplificarse.

Superclave. Descriptor que determina toda la relación.

VI. Bibliografía

Referencias:

[Armstrong 1974] Armstrong, William W. ***Dependency Structures of Data Base Relationships***. Proc. IFIP Congress, Suecia. 1974. pp. 580 – 583.

[Atkins 1988] Atkins John ***A note on minimal covers.*** ACM SIGMOD Vol. 17(4). 1988. pp. 16 – 21

[Bernstein 1976] Bernstein, Philip A. ***Synthesizing Third Normal Form Relations from Functional Dependencies.*** ACM TODS Vol 1(4). 1976, pp 277 – 298
(Disponible en acceso abierto en https://personal.comp.nus.edu.sg/~lingtw/papers/bernstein.pdf.).

[Codd 1972] Codd, Edgar F. ***Further Normalization of the Database Relational Model***. En Courant Computer Science Courant Computer Science Symposia Series 6, "Data Base Systems", Data Base Systems: Courant Computer Science Symposia Series 6, Prentice-Hall, Englewood Cliffs, N.J., 1972.

[Codd 1974] Codd, Edgar F. ***Recent Investigations into Relational Data Base Systems.*** IBM Research Report RJ1385 (April 23, 1974). Republished in Proc. 1974 Congress (Stockholm, Sweden, 1974). New York, N.Y.: North-Holland (1974).

[Connolly y Begg 2005]. Connolly, Thomas M. y Begg, Carolyn E. ***Sistemas de Bases de Datos 4ª ed***. Pearson, 2005.
(Disponible en UBUvirtual: http://0-www.ingebook.com.ubucat.ubu.es/ib/NPcd/IB_BooksVis?cod_primaria=1000187&codigo_libro=2155)

[Date 1994] Date, Chris J. ***An Introduction to Database Systems 6th ed.*** Addison Wesley 1994.

[de Miguel y Piattini 1993] de Miguel, Adoración y Piattini, Mario. ***Concepción y Diseño de Bases de Datos. Del Modelo E/R al Modelo Relacional.*** RA-MA, 1993.

[Elmasri y Navathe 2007] Elmasri, Ramez y Navathe, Shamkant B. ***Fundamentos de Sistemas de Bases de Datos 5ª ed***. Pearson, 2007
(Disponible en UBUvirtual: http://0-www.ingebook.com.ubucat.ubu.es/ib/NPcd/IB_BooksVis?cod_primaria=1000187&codigo_libro=2886)

[Rissanen 1977] Rissanen, Jorma. ***Independent Components of Relations.*** ACM TODS Vol 2(4). 1977. pp 317 – 325.
(Disponible en acceso abierto en http://www.iai.uni-bonn.de/III/lehre/vorlesungen/Informationssysteme/WS05/materialien/p317-rissanen.pdf).

[Silberschatz et. al 2006] Silberschatz, Abraham, Korth, Henry F. y Sudarshan, S. ***Fundamentos de Bases de Datos 5ªEd.*** McGraw-Hill, 2006.
(Disponible en UBUvirtual: http://0-www.ingebook.com.ubucat.ubu.es/ib/NPcd/IB_BooksVis?cod_primaria=1000187&codigo_libro=4356).

[Ullman 1982] Ullman Jeffrey D. ***Principles of Database Systems 2nd Ed***. Computer Science Press, 1982.

[Ullman y Widom 1997] Ullman, Jeffrey D. y Widom, Jennifer. ***A First Course in Database Systems***. Prentice Hall, 1997.

Bibliografía comentada:

Los textos que mejor se adaptan a el orden en el que se ha expuesto cada concepto, y que más o menos son igula de exhaustivos son: los capítulos 17 y 19 de [de Miguel y Piattini 1993] y los capítulos 10 y 11 de [Elmasri y Navathe 2007]:

- En [de Miguel y Piattini 1993] el Capítulo 17 hace una revisión paralela a la de los apuntes pero sin entrar en algoritmos ni demostraciones, aunque explicando el trasfondo práctico de todos los conceptos. Utiliza siempre un par de ejemplos reales muy claros.

  El Capítulo 19 está dedicado íntegramente a ver algoritmos y es mucho más exhaustivo que lo que se ha presentado aquí. Se ven entre los presentados, los algoritmo para hallar el cierre de un descriptor, el recubrimiento minimal; junto con sendas versiones mejoradas que reducen los tiempos al realizar estas tareas computacionalmente. También presenta versión original de la síntesis de Bernstein, así como algoritmos más precisos que el sugerido en los apuntes para hallar las claves, entre otros.
- En [Elmasri y Navathe 2007] el capítulo 10 tiene inicialmente una estructura similar a los apuntes en cuanto a como secuencia los contenidos hasta la definición de las formas normales, y entra incluso también en algunas de las demostraciones. El capítulo 11 se encarga de presentar los algoritmos en la sección 11.2.

Además [Date 1994] es muy recomendable. Su Capítulo 9 hace una revisión de las definiciones (dependencia funcional, axiomas de Armstrong, cierres de un conjunto de atributos y de un conjunto de dependencias) utilizando el ejemplo de siempre de proveedores y productos, y el Capítulo 10 que presenta las formas normales desde la 1FN hasta la FNBC, las pérdidas sin descomposición. La parte distinta e interesante por la que se recomienda es, en ese capítulo 10, el planteamiento de posponer la discusión de las anomalías que generan las redundancias hasta el momento en el que explica la 3FN, ligándolas a la ausencia de la 3FN, y justificando así la necesidad de verificarla. Date no sólo explica la 3FN y la FNBC, sino que deduce excelentemente su necesidad a partir de la observación de las anomalías. El alumno debe de tener en cuenta que en sus explicaciones utiliza con frecuencia el término *varrel* (i.e., *var*-iable *rel*-acional) que al final significa tabla o relación.

En [Ullman y Widom 1997] el planteamiento es distinto y mucho más simple del expuesto en estos apuntes (secciones 3.5, 3.6 y 3.7). Se centra en la FNBC, y las otras formas normales las explica al final de la última sección: la 3FN como forma normal a la que se debe de llegar en caso de llegar a una descomposición con pérdidas (estos autores no descomponen utilizando la síntesis de Bernstein, como se comente en *Material complementario*), y la 2FN no la explican porque se centran en sí se cumple o no la 3FN sin entrar a valorar si a lo mejor no se cumple la 3FN porque no si quiera se cumple la segunda.

Del capítulo 7 de [Silberschatz et. al 2006] se han extraído muchas de las definiciones y bastantes ejemplos de estos apuntes pueden encontrarse en este libro., tampoco se explica la 2FN, por los mismos motivo vistos en [Ullman y Widom 1997]. En la segunda mitad del capítulo [Silberschatz et. al 2006] explica casi todos algoritmos revisados en los apuntes.

Finalmente, quizás el menos profundo de todos es [Connolly y Begg 2005], que tiene un capítulo 13 que es muy básico, donde presenta las dependencias funcionales y la 2FN y 3FN. El capítulo 14 ya desarrolla la teoría de dependencias funcionales, la FNBC, y la descomposición la realiza de forma intuitiva pero sin algoritmos.

VII. Material complementario

Como ya hemos comentado en la bibliografía, la lectura del capítulo 10 de [Date 1994] es muy interesante si se quiere comprender el porqué de la 3FN y la FNBC.

Para profundizar en los algoritmos vistos y conocer variantes aún más sofisticadas de los mismos, se recomienda el Capítulo 19 de [de Miguel y Piattini 1993]. En este particular, y en lo referente a algoritmos de descomposición, existen oras alternativas a la síntesis de Bernstein. En ese sentido, por su claridad conviene leerse la estrategia de descomposición que viene en la sección 3.7.4 de [Ullman y Widom 1997], la cual garantiza llegar a la FNBC pero al precio de poner en riesgo la no existencia de pérdidas de dependencias. En [Silberschatz et. al 2006] aparece ese mismo algoritmo de descomposición en la sección 7.5.1.2. y en [Elmasri y Navathe 2007] sección 11.2.2.

Las formas normales avanzadas pueden encontrarse en:

- Capítulo 11 de [Date 1994], Capítulo 18 de [de Miguel y Piattini 1993] y secciones 11.3, 11.4.11,5 y 11.6 de [Elmasri y Navathe 2007]: revisan 4FN, 5FN, FNDC y añaden la *forma normal de inclusión*, estas últimas sirven para dar soporte a las interrelaciones ISA del modelo E/R.
- Las secciones 14.4 y 14.5 de [Connolly y Begg 2005] tratan brevemente la 4FN y 5FN.
- La sección 3.8 de [Ullman y Widom 1997] y la 7.6 de [Silberschatz et. al 2006] sólo trata la 4FN.

En cuanto a material en vídeo y cuestionarios on – line:

- ***DB8 Relational Design Theory***. *Prof. Jennifer Widom, Standford University.* https://lagunita.stanford.edu/courses/DB/RD/SelfPaced/courseware/18c87a89721d4284840f5c8537c6afb3/

  Los vídeos 1, 2 y 3 para ver hasta FNBC; el vídeo 4 es sobre la 4FN, y el vídeo 5 es una discusión sobre un esquema de ejemplo se hasta dónde normalizar, teniendo en cuenta que ellos en su curso sí que han explicado la 4FN.