

# Bases de Datos

## Cartesianas

carles franquesa i niubo

*A mis estudiantes Jusseth, Álex, Fernando y Jairo, de IKIAM,  
por encender en mí el deseo de realizar este texto.*

# Prólogo

*Cuando el Dr. Franquesa me pidió que escribiera el prólogo de su libro "Bases de Dades Cartesianes" me sentí honrado pero también a la vez preocupado por saber si podría estar a la altura de lo que se me requería. Supongo que pensó en mi recordando los años que compartimos en la asignatura de "Introducción a las Bases de Datos". Fueron los últimos años antes de mi jubilación y guardo un gran recuerdo que creo que los alumnos de entonces compartirán con nosotros.*

*En fecha de hoy, no he tenido conocimiento de otros libros que versen sobre estos contenidos en nuestra lengua catalana, y tan solo por eso, ya deberíamos felicitarnos los que amamos las bases de datos y al mismo tiempo esta lengua. Pero no solo es eso.*

*A medida que he ido cosiendo un camino de más de cuarenta años de docencia en estudios superiores, he ido adquiriendo la convicción que lo más motivador y estimulante ha sido la vocación. Para los profesores vocacionales no hay gratificación mayor que observar la magnitud de los proyectos donde ahora trabajan alumnos que un día llegaron con la incertidumbre en los ojos, y al mismo tiempo con una ingenuidad voraz dispuestos a alimentarse de lo que se les indica, lo que es necesario prever es lo que trascenderá en sus actividades profesionales en un futuro más cercano de lo que parece. Para eso se requiere entusiasmo, ilusión, locuacidad y coherencia en los contenidos. Este libro es un exponente de todos esos factores. Más que un libro de teoría, parece un discurso. Una conversación.*

*Las Bases de Datos han evolucionado mucho desde sus inicios allá por los años 60 del siglo pasado. Entonces, el mismo hardware limitaba lo que se podía hacer y las primeras implementaciones fueron para resolver problemas tales como el escandalo de piezas en la fabricación con una estructura, ficticia por poco realista, de tipo jerárquico: una pieza, o bien un conjunto de piezas, nunca dependía de más de una pieza del nivel superior. A poco de empezar a utilizar estas bases de datos jerárquicas, el fundamento teórico de otras soluciones se estaba cocinando por parte de verdaderos pioneros como Edgar F. Codd y, más tarde Christopher J. Date. El conjunto de soluciones encontradas supusieron la aparición teórica de las bases de datos relacionales. Las limitaciones del hardware impidieron su desarrollo posponiendo la implementación*

*casi quince años.*

*Hoy en día muchas de las bases de datos ya son de tipo relacional a pesar de que ya se apunten nuevas soluciones atrevidas que aún deben demostrar su utilidad real y que tienen unos fundamentos matemáticos no tan robustos y por descontado mucho menos rigurosos que los del modelo relacional. En muchos casos se trata de permitir nuevos tipos de datos elementales como bases de datos de objetos, o bases de datos de tamaños gigantescos, normalmente de tipo multimedia, soportándose más en la cantidad de recursos materiales de los que se dispone que en la integridad de los datos.*

*En este libro, el Dr. Franquesa hace un estudio muy completo de las bases de datos relacionales y, en particular propone la solución real con la implementación de una de ellas.*

*El libro se propone asentar primeramente una base teórica inicial de tipo matemático seguida de lo que llama "Análisis del Proyecto" donde se recoge el ambiente en el que la solución informática se encontrará finalmente circunscrita.*

*Sigue un capítulo de diseño previo ya al estudio central del modelo Entidad - Relación para entrar en el capítulo siguiente en el Álgebra Relacional, donde se estudia qué mecanismos se pueden usar, y cómo se utilizan, para obtener resultados de la base de datos preparada, diseñada, como solución al problema abordado por el proyecto. En este capítulo, el quinto, el autor propone unos gráficos magníficos y muy esclarecedores para representar múltiples tuplas.*

*Entra después en la descripción, central en varios aspectos, del Lenguaje Estructurado de Consultas que nos debe servir para primero describir la propia base de datos y, después para llegar a utilizar los datos que contendrá. Naturalmente el hecho de tener los datos serviría de poco si no podemos consultarlos, borrarlos o añadir nuevos cuando nos convenga. Por fin, será necesario que todo lo que se ha visto y estudiado se ponga en contexto con una implementación en particular, escogiendo para hacerla el SGBD PostgreSQL que es, no solamente una de las mejores implementaciones, sino también una de las más fáciles de encontrar.*

*Necesito decir que, cuando impartíamos la asignatura de "Introducción a las Bases de Datos", me hubiera gustado tener este libro como base para las explicaciones a los alumnos. Al mismo tiempo, les serviría a ellos como referencia actual y futura sobre las bases de datos.*

*He dicho que me gustaría haber tenido este libro y no he sido exacto, lo que me hubiera gustado habría sido escribirlo!*

Josep Maria Bañeres,

*Profesor responsable de Bases de Datos,*

*ya retirado, de la Universitat de Barcelona.*

*Cabrils, octubre de 2014.*



# Índice

<b>1</b>	<b>Teoría de Conjuntos</b>	<b>11</b>
1.1	Introducción . . . . .	11
1.2	Definiciones y Nomenclatura . . . . .	11
1.2.1	Abstracción . . . . .	13
1.2.2	Descripción de Conjuntos . . . . .	14
	Por enumeración de sus elementos . . . . .	14
	A partir de propiedades de sus elementos . . . . .	15
1.3	Operaciones entre Conjuntos . . . . .	16
1.3.1	Unión . . . . .	17
1.3.2	Intersección . . . . .	19
1.3.3	Diferencia . . . . .	20
1.3.4	Producto Cartesiano . . . . .	21
1.4	Lógica de Predicados . . . . .	23
1.4.1	Cálculo de Predicados . . . . .	23
	Operación de negación lógica . . . . .	24
	Operación lógica disyuntiva . . . . .	25
	Operación lógica conjuntiva . . . . .	26
	Leyes de Augustus de Morgan . . . . .	27
1.4.2	Relación con la Teoría de Conjuntos . . . . .	27
<b>2</b>	<b>Análisis del Proyecto</b>	<b>31</b>
2.1	Sistemas Gestores de Bases de Datos . . . . .	31
2.1.1	Arquitectura Cliente-Servidor . . . . .	33
2.2	Principio de Independencia de los Datos . . . . .	33
2.2.1	Arquitectura a Tres Niveles . . . . .	35
2.3	Inmersión en el Dominio . . . . .	36
2.4	Definición de Requerimientos . . . . .	37
2.4.1	Información Estadística . . . . .	38
2.5	Trayectoria del Error . . . . .	38
2.6	Ejemplo para un Club Deportivo . . . . .	39
2.6.1	Requerimientos del Club Deportivo . . . . .	39
<b>3</b>	<b>Diseño de Bases de Datos</b>	<b>43</b>
3.1	Modelos de Datos . . . . .	46
3.2	Dominio del Problema . . . . .	47
3.3	Máximas de Calidad de un Diseño . . . . .	48

3.3.1	Prohibir la Redundancia . . . . .	48
3.3.2	Prohibir Nulos Estructurales . . . . .	49
3.3.3	Unificar Conceptos . . . . .	49
<b>4</b>	<b>Modelo Entidad Relación</b>	<b>51</b>
4.1	Entidades . . . . .	52
4.2	Atributos . . . . .	53
4.3	Atributos Clave . . . . .	56
4.4	Atributos Multivalorados . . . . .	58
4.5	Atributos Compuestos . . . . .	59
4.6	Atributos Calculados . . . . .	61
4.7	Relaciones . . . . .	63
4.7.1	Participaciones de las entidades un una relación . . . . .	64
4.7.2	Dependencias de existencia . . . . .	64
4.7.3	Cardinalidad de una relación binaria . . . . .	65
4.8	Relaciones 1:1 . . . . .	66
4.9	Relaciones 1:N . . . . .	67
4.9.1	Entidades de soporte a la interfaz . . . . .	70
4.9.2	Entidades débiles . . . . .	72
4.9.3	Clasificación de las Relaciones 1:N . . . . .	74
4.10	Relaciones M:N . . . . .	75
4.10.1	Atributos en relaciones M:N . . . . .	79
4.11	Autorelaciones . . . . .	80
4.11.1	Autorelaciones 1:1 . . . . .	80
4.11.2	Autorelaciones 1:N . . . . .	82
4.11.3	Autorelaciones M:N . . . . .	83
4.12	Cálculo Automático de las Cardinalidades . . . . .	85
4.13	Relaciones Ternarias . . . . .	87
4.13.1	Cardinalidades en las relaciones ternarias . . . . .	88
4.13.2	Relación ternaria con tres relaciones binarias . . . . .	89
4.14	Modelo Entidad Relación Extendido . . . . .	90
4.14.1	Especialización y Generalización de Entidades . . . . .	91
	Especializaciones o generalizaciones completas . . . . .	93
	Especializaciones o generalizaciones disjuntas . . . . .	94
4.14.2	Agregación de Entidades . . . . .	94
<b>5</b>	<b>Álgebra Relacional</b>	<b>97</b>
5.1	Álgebras . . . . .	98
5.2	Relaciones . . . . .	98
5.2.1	Relación . . . . .	100
	Tuplas . . . . .	100
	Cálculo relacional por tuplas . . . . .	102
	Dominios . . . . .	103
	Definición formal de relación . . . . .	103
	Compatibilidad entre relaciones . . . . .	106
5.2.2	Visión Cartesiana de una Relación . . . . .	106
5.3	Modelo Relacional de una Base de Datos . . . . .	109



5.3.1	Atributos estructurales y atributos descriptivos . . . . .	111
5.3.2	Claves Primarias . . . . .	111
5.3.3	Claves Foráneas . . . . .	113
5.3.4	Transformación del Modelo ER al Modelo Relacional . . . . .	115
5.3.5	Diagrama de Esquemas de una Base de Datos . . . . .	123
5.4	Operaciones Básicas con Relaciones . . . . .	126
5.4.1	Operación de Selección . . . . .	127
5.4.2	Operación de Proyección . . . . .	128
	Composición de operaciones . . . . .	131
5.4.3	Producto Cartesiano de Relaciones . . . . .	132
5.4.4	Unión de Relaciones . . . . .	135
5.4.5	Diferencia de Relaciones . . . . .	137
5.4.6	Renombramiento de Relaciones . . . . .	137
5.5	Operaciones Adicionales . . . . .	141
5.5.1	Intersección de Relaciones . . . . .	142
5.5.2	Reunión Natural . . . . .	143
	Reunión interna . . . . .	145
	Reuniones externas . . . . .	147
5.5.3	División de Relaciones . . . . .	148
5.5.4	Asignación de Relaciones . . . . .	150
5.6	Álgebra Relacional Extendida . . . . .	151
5.6.1	Proyección Generalizada . . . . .	151
5.6.2	Valores Nulos . . . . .	153
	Lógica de tres valores . . . . .	154
5.6.3	Funciones de Agregación . . . . .	154
<b>6</b>	<b>Lenguaje Estructurado de Consultas</b>	<b>161</b>
6.1	Entorno de Trabajo . . . . .	162
6.1.1	Familiarización con el SGBD . . . . .	164
	Familiarización con el <code>psql</code> . . . . .	164
	Familiarización con el <code>SQL</code> . . . . .	165
6.2	Lenguaje de Definición de Datos . . . . .	166
6.2.1	Multiconjuntos . . . . .	166
6.2.2	Tablas . . . . .	166
6.2.3	Metadatos . . . . .	167
6.2.4	Dominios de los atributos . . . . .	168
	Tipos primitivos . . . . .	168
	Tipos alfanuméricos . . . . .	169
	Tipos temporales . . . . .	170
	Tipos autoincrementales . . . . .	171
6.3	Construcción de la Base de Datos . . . . .	173
6.3.1	Espacio de Trabajo . . . . .	174
6.3.2	Esript Principal . . . . .	174
6.3.3	Creación de los Dominios . . . . .	176
6.3.4	Creación de las Tablas . . . . .	178
	Integridad referencial . . . . .	181
	Claves candidatas . . . . .	187

6.4	Lenguaje de Manipulación de Datos . . . . .	197
	Escripts de inserción . . . . .	197
6.4.1	Consultas de Lectura . . . . .	198
	Estructura fundamental . . . . .	198
	Datos . . . . .	199
	Cláusula SELECT . . . . .	202
	Cláusula FROM . . . . .	204
	Cláusula WHERE . . . . .	209
6.4.2	Funciones de Agregación . . . . .	212
	De todos los tipos de atributos . . . . .	213
	De atributos numéricos . . . . .	214
	Cláusula GROUP BY . . . . .	216
	Cláusula HAVING . . . . .	218
6.4.3	Cláusula de Ordenación . . . . .	220
	Cláusula ORDER BY . . . . .	220
	Consulta de lectura con todas las cláusulas . . . . .	221
6.4.4	Vistas . . . . .	222
6.4.5	Consultas de Actualización . . . . .	224
	Cláusula INSERT INTO . . . . .	225
	Cláusula UPDATE SET . . . . .	227
	Cláusula DELETE FROM . . . . .	229
6.4.6	Operaciones con Conjuntos . . . . .	231
	Cláusula UNION . . . . .	232
	Cláusula EXCEPT . . . . .	236
	Cláusula INTERSECT . . . . .	238
6.4.7	Operaciones de Reunión de la Cláusula FROM . . . . .	240
	Cláusula NATURAL INNER JOIN . . . . .	241
	Cláusula NATURAL LEFT OUTER JOIN . . . . .	242
	Cláusula NATURAL RIGHT OUTER JOIN . . . . .	243
	Cláusula NATURAL FULL OUTER JOIN . . . . .	244
	Cláusula INNER JOIN USING . . . . .	245
	Cláusula INNER JOIN ON . . . . .	247
6.4.8	Valores Nulos . . . . .	248
	Predicado IS NULL . . . . .	248
	Predicados IS UNKNOWN, IS DISTINCT OF . . . . .	249
6.4.9	Tablas temporales . . . . .	250
6.4.10	Consultas Anidadas . . . . .	251
	En el argumento de la cláusula SELECT . . . . .	251
	En el argumento de la cláusula FROM . . . . .	252
6.4.11	Consultas Anidadas en la Cláusula WHERE . . . . .	254
	Cláusula IN . . . . .	256
	Cláusulas SOME y ALL . . . . .	257
	Cláusula EXISTS . . . . .	260
<b>7</b>	<b>El SGBD PostgreSQL</b> . . . . .	<b>265</b>
7.1	Documentación de PostgreSQL . . . . .	266
7.2	Estructura del Proyecto . . . . .	268

7.3	Usuarios y Seguridad . . . . .	270
7.3.1	Permisos . . . . .	271
	Aportaciones al proyecto del club deportivo . . . . .	276
7.3.2	Privilegios . . . . .	277
7.3.3	Concurrencia y Transacciones . . . . .	282
7.4	Componentes Lógicos de una Base de Datos . . . . .	287
7.4.1	Componentes de Datos . . . . .	287
	Atributos indexados . . . . .	288
7.5	Componentes de Control . . . . .	291
7.5.1	Implementación de la Aplicación del Club Deportivo . . . . .	291
	Implementación de funciones genéricas . . . . .	292
	Implementación de funciones específicas . . . . .	293
7.6	Funciones . . . . .	293
7.6.1	Cabecera . . . . .	294
7.6.2	Cuerpo . . . . .	295
7.6.3	Entrada y salida de datos con PL/pgSQL . . . . .	298
7.6.4	Consultas de actualización en PL/pgSQL . . . . .	300
7.6.5	Consultas de lectura en PL/pgSQL . . . . .	304
	Consultas que retornan un solo valor . . . . .	305
	Cláusula <code>PERFORM</code> . . . . .	310
	Consultas que retornan registros . . . . .	311
7.7	Disparadores . . . . .	315
7.7.1	Funciones de Tipo <code>TRIGGER</code> . . . . .	315
	Cabecera de las funciones <code>TRIGGER</code> . . . . .	316
	Cuerpo de las funciones <code>TRIGGER</code> . . . . .	316
	Valores de retorno de los disparadores . . . . .	317
7.7.2	Sintaxis de Declaración de un Disparador . . . . .	317
7.8	Aplicaciones Cliente . . . . .	319
7.8.1	Lenguaje Anfitrión y SQL Incrustado . . . . .	319
	Conexión . . . . .	320
	Consultas de lectura ( <code>SELECT</code> ) . . . . .	320
	Consultas de actualización ( <code>INSERT</code> , <code>UPDATE</code> y <code>DELETE</code> ) . . . . .	320
7.9	Administración de la Base de Datos . . . . .	321
7.9.1	Parada e Iniciación del Sistema . . . . .	321
	MS-DOS . . . . .	321
	Linux . . . . .	322
	El programa <code>pg_ctl</code> . . . . .	324
7.9.2	Mantenimiento y Copias de Seguridad . . . . .	325
7.9.3	Configuración de un servidor PostgreSQL . . . . .	327
	Archivo de parámetros de configuración . . . . .	327
	Archivo de autenticación basada en el host . . . . .	332
	Archivo de mapeo de usuarios . . . . .	334

<b>A</b>	<b>Estilo</b>	<b>337</b>
A.1	Modelo Entidad Relación . . . . .	337
A.2	Modelo Relacional . . . . .	338
A.3	Lenguaje Procedural . . . . .	338

A.4	Versiones de los Sistemas Utilizados . . . . .	338
<b>B</b>	<b>Códigos ASCII</b>	<b>339</b>
<b>C</b>	<b>Datos para el Ejemplo del Club Deportivo</b>	<b>341</b>
C.1	Tabla provincia . . . . .	341
C.2	Tabla ciudad . . . . .	342
C.3	Tabla persona . . . . .	343
C.4	Tabla teléfonos . . . . .	344
C.5	Tabla conoce . . . . .	344
C.6	Tabla socio . . . . .	345
C.7	Tabla trabajador . . . . .	345
C.8	Tabla deporte . . . . .	346
C.9	Tabla hace . . . . .	347
C.10	Tabla pagos . . . . .	347

# Preámbulo

Cuando se formulan contenidos teóricos, los humanos mostramos predilección por soportarlos en la lógica matemática. La misma raíz etimológica del término *lógica* se usa entonces como sufijo para indicar estudio, forma de entender. Eso es así porque un contenido teórico expone un modelo de alguna faceta de la realidad, y para poderse desarrollar precisa de una coherencia que se encuentra bien fundamentada en la teoría de conjuntos.

Las bases de datos relacionales, que nos disponemos a desmenuzar, no difieren de esta aproximación, y así, estructuran unas normas que finalmente encuentran conexión con la praxis en la implementación que se hace en máquinas computadoras.

Pero bien, no avancemos historias. Observemos qué tenemos en frente, de entrada. Asumiendo la fuerza brutal que tienen dos conceptos tan importantes como el espacio y el tiempo, quede claro que lo que aquí trataremos es el espacio. El espacio son los objetos, el tiempo son las acciones. Orientar el discurso hacia las acciones o los procedimientos convertiría este texto en un libro de algoritmia. Y no se trata de eso. Aquí abordamos los objetos, o sea los espacios. De hecho es la parte sencilla, ya que parece claro que los sustantivos son más fáciles de comprender que los verbos. En la lengua catalana tenemos un buen puñado de pruebas. El diccionario, [2],[6], permite decir píxel, pero no pixelar. Una cosa hace olor, no huele. Y aunque que esté permitido besar o petonejar, en catalán acostumbramos a hacernos besos. En definitiva, está bien claro que somos proclives a soportar la semántica en los sustantivos. La prueba más clara de que el universo de los sustantivos es más simple que el de los verbos es que a los sustantivos tan solo los clasificamos en femeninos o masculinos, singulares y plurales, o sea género y número. En cambio para los verbos tenemos las conjugaciones verbales con sus pasados, presentes y futuros, aparte de las formas impersonales. Así pues, tenemos suerte, los datos viven en el mundo de los sustantivos.

De hecho, para estructurar la realidad, hay que empezar por los datos. Y más filosóficamente aún, el tiempo surge a partir de la repetición en las referencias a los objetos. El tiempo establece diferencias entre una cosa y ella misma. Los instantes de creación pueden servir para identificar casi cualquier objeto. O sea, que hay que ir con cautela al incorporar el tiempo en los datos que tienen referencias temporales.

Para comprender en profundidad donde hay que situar la tarea que aquí se propone, veamos primero una partición de las teorías tal como las entendemos. Conviengamos en que una teoría puede entenderse como un conjunto de verdades incuestionables más un conjunto de reglas que trabajan con esas verdades para construir otras nuevas. Visto desde este ángulo cualquier cosa es una teoría. Una bicicleta, y las aplicaciones que se le puedan dar, por ejemplo, constituyen una teoría. Todo eso entra en escena para dar a conocer que respecto a las teorías hay dos clasificaciones en tres grupos. Por un lado, se puede clasificar una teoría como incompleta, categórica, o contradictoria. Y por otra parte como teoría abierta, sólida, o inconsistente. Y según se mire desde el punto de vista deductivo o bien empírico, se utiliza una u otra clasificación. Pero bien, mirado desde la distancia, sin ningún rigor, todo ello nos resulta útil para disponer de una terminología que ubica los contenidos que se exponen en este texto.

Una teoría abierta, o incompleta, es aquella que sin entrar en ninguna contradicción postula un conjunto de verdades que después se pueden enriquecer con aplicaciones concretas dando paso a muchas otras teorías. Cualquier álgebra es una teoría abierta. Y de manera mucho más prosaica también podemos considerar cualquier cosa. Una bicicleta, un teléfono o un zapato. Todo eso son teorías abiertas, ya que son verdades que no se contradicen, y por tanto son teorías, y al mismo tiempo dejan margen para la interpretación, y por eso son abiertas. Podríamos hablar de bicicletas rojas o verdes, estableciendo dos teorías distintas.

Teoría categórica es la que pretende clasificar el conjunto de todas las cosas dichas. Es muy pretenciosa, ya que para cualquier predicado formulable, o sea, para cualquier pregunta que se pueda responder sí o no, el análisis matemático pretende hacerlo coherentemente. El análisis matemático es la única teoría categórica que conocemos. Por eso es una herramienta tan estimada en todas las disciplinas.

Finalmente, en las teorías inconsistentes se producen espacios de confusión en los que un mismo predicado resulta cierto y falso simultáneamente. Son teorías que mueren pronto. No parecen tener demasiada utilidad.

De todo ello es fácil desprender una analogía con la vida de una persona a lo largo de las tres edades.

Pues bien, el desarrollo de una base de datos para satisfacer un conjunto de requerimientos es una teoría abierta. O sea incompleta. Eso significa que cualquier objetivo tiene más de una aproximación correcta. Nadie nos podrá asegurar que la solución implementada sea óptima.

De hecho, a partir de su diseño, nadie puede demostrar que una base de datos dará respuesta a todas las solicitudes que se le puedan formular. Se trata de un territorio resbaladizo. Por eso, relajamos nuestros objetivos. Una pretensión más asequible es la de detectar que alguna parte de la estructura de la base chirría. Si hay errores en el diseño, es posible que los detectemos. Y cuando seamos incapaces de detectar,

entonces podremos suponer que el diseño es correcto.

Desde que empezaron a existir las bases de datos hasta ahora<sup>1</sup> ha habido un uso masivo de estas tecnologías. La misma expresión, *base de datos*, es una de las primeras expresiones que se popularizó del ámbito de la computación. Eso ha provocado que gran cantidad de desarrolladores de aplicaciones procurasen incorporar las bases de datos en sus trabajos. El uso de las bases de datos ha sido tan masivo que el conocimiento de para qué sirven se ha diluído, de forma que a día de hoy, una gran cantidad de aplicaciones que hacen gala de conectarse a bases de datos, no extraen ningún provecho que no pudieran tener con un sistema plano de ficheros.

Más grave aún que el desperdicio de recursos de espacio y tiempo que eso supone en las aplicaciones, se ha perdido la mayor parte de las ventajas que un sistema gestor de bases de datos ofrece. Es un problema grave, ya que los desarrolladores de código invierten horas de trabajo en acciones que el gestor resolvería mucho mejor de lo que pueden hacer ellos. La responsabilidad de este hecho es sin duda de todos nosotros, de la sociedad. Queda mal hablar de una aplicación que no utiliza una base de datos, y por tanto, cualquier aplicación se asegura un prestigio usándolas, incluso aunque eso las entorpezca.

Hay que tener esto en cuenta, ya que todo ello nos viene a explicar que nos encontramos en un ámbito donde hay un gran intrusismo profesional, y por tanto, hay que desconfiar mucho de las tendencias.

---

<sup>1</sup>12/02/2017

Los archivos para realizar los ejercicios de los últimos capítulos contienen tanto la implementación de la base de datos de ejemplo, como de los datos que hay en el Apéndice C. Se pueden descargar de

<https://drive.google.com/open?id=0B1-YyXS4zdgrQWhGOGg2WUExd28>

aunque esos ficheros no han sido traducidos al español.



# Capítulo 1

## Teoría de Conjuntos

### 1.1 Introducción

La teoría de conjuntos es profusamente aceptada como la teoría más importante de todas las teorías. Dicho esto con cierta precaución para no enfadar a los puristas, es seguramente la primera, la teoría más abstracta.

Pero bien, dejando aparte los superlativos, que no hacen más que competir, y aproximándonos al tema de manera más pragmática, la teoría de conjuntos está en la esencia del conocimiento analítico, y por tanto científico, y por tanto humano.

En este primer capítulo se hace un repaso de los conceptos básicos de esta teoría, ya que son cuestiones preliminares para el resto del curso. Se presenta la definición de las operaciones que tratan con conjuntos. A pesar que de alguna manera hay comentarios en que todo esto puede parecer infantil, o incluso naïve, conviene asentar sólidamente estas nociones básicas para no caer en malentendidos. En la segunda mitad se ve la lógica de predicados, prima hermana de la teoría de conjuntos aunque más emparentada con la deducción natural. Todo ello son facetas de un mismo poliedro.

### 1.2 Definiciones y Nomenclatura

Independientemente del rigor con el que se podría describir esta teoría, es necesario retener tres conceptos.

En síntesis, llamamos *conjunto*,  $C$ , a una colección de *elementos*,  $e_1, e_2$ , y  $e_3$  por ejemplo, diferentes entre ellos. Decimos que el elemento  $e_1$  *pertenece* al conjunto  $C$ , y lo escribimos  $e_1 \in C$ , o sea, diciendo "pertenece" al símbolo  $\in$ . Y también podemos decir que  $C$  *contiene* el elemento  $e_1$ , aunque en este caso no introducimos ningún símbolo.

Por definición, un conjunto no tiene elementos repetidos. Si queremos trabajar o decir alguna cosa sobre un conjunto con elementos repetidos, entonces debemos decir que nos encontramos frente a un *multiconjunto*. Pero bien, a estas alturas del discurso esta definición no tiene más trascendencia.

Conjunto, elemento, y el tercer concepto esencial es el concepto de *subconjunto*. Resulta bastante intuitivo entender que  $S$  es un subconjunto de  $C$ , cosa que anotamos  $S \subseteq C$ , si  $S$  es un conjunto de elementos que todos ellos pertenecen a  $C$ . También decimos entonces que  $S$  *está incluido* en  $C$ . O sea, un conjunto no es el conjunto de todos sus subconjuntos posibles. Y por tanto, un subconjunto no pertenece al conjunto en el cuál se ha definido: está incluido en él. Otra cosa sería hablar del conjunto de subconjuntos posibles. Es una cosa distinta.

La definición de subconjunto contempla los dos casos singulares en que  $S$  es el conjunto vacío, que denotamos  $\emptyset$  y que por definición está incluido en todos los conjuntos, o el otro extremo, en que  $S$  es igual a  $C$ . Ahora bien, si queremos indicar que  $S$  es un subconjunto de  $C$  tal como normalmente lo imaginamos, entonces decimos que  $S$  es un subconjunto *propio* de  $C$ . Formalmente, utilizando el símbolo  $\Leftrightarrow$  que significa *si y solo si*,

$$S \subset C \Leftrightarrow S \subseteq C \text{ y } S \neq \emptyset \text{ y } S \neq C.$$

Caja 1.1. *Definición de subconjunto propio.*

En este caso, podemos decir también que  $S$  está incluido en  $C$  *estrictamente*. Y lo representamos sin la rayita,  $S \subset C$ , tal como se muestra en la caja más arriba.

A veces interesa explicar que entre dos conjuntos, uno de ellos es subconjunto del otro sin la necesidad de reflejar la continencia de cuál es subconjunto de cuál. Entonces decimos que  $C$  y  $S$  son *compatibles*, y lo expresamos mediante el símbolo  $\sim$ . En la Caja 2 se formaliza la compatibilidad entre conjuntos.

$$X \sim Y \Leftrightarrow X \subseteq Y \text{ o bien } Y \subseteq X$$

Caja 1.2. *Definición de compatibilidad entre conjuntos.*

Ergo si son iguales, son compatibles. Otros ejemplos de compatibilidad entre conjuntos son los números enteros y los números reales o las cadenas de caracteres y los números de teléfono.

El *cardinal* de un conjunto es el número de elementos que contiene. Un número entero no negativo. En notación simbólica, está claro que le podemos llamar  $n$ , minúscula porque es un número y no un conjunto. Y si queremos expresar el cardinal en referencia al conjunto  $C$ , entonces utilizamos la notación de  $C$  entre barras, es decir  $|C|$ . Es fácil ver que si  $S$  es un subconjunto propio de  $C$ , y si  $n = |C|$  y  $m = |S|$ , entonces  $n > m$ .

Como se ha dicho más arriba, los elementos de un conjunto han de ser distintos entre ellos, y eso implica que tienen que ser identificables. Esta deducción resulta profundamente constructiva. Los conjuntos son colecciones de elementos identificables.

Desde un punto de vista mucho más filosófico, se podría decir que hay alguna relación entre la identificación y el número uno.

### 1.2.1 Abstracción

La capacidad de percibir características comunes en objetos o fenómenos y nombrarlo con una palabra es la abstracción. ¿Cómo es una silla? Hay sillas de una pata, o tres o cuatro, con respaldo o sin. De madera, de metal y de plástico... en fin, hay tantas variaciones de sillas que parece mentira que nos entendamos cuando utilizamos la palabra silla. Ya lo veis. Cada palabra es ya una abstracción.

Se dice que un concepto es más abstracto que otro cuando el primero designa un conjunto del cual el segundo es un subconjunto. Este proceso mental puede acarrear pérdida de información. El título de un texto es una abstracción, ya que bajo un mismo título pueden existir varios textos. La abstracción sintetiza la esencia. Y también una abstracción de gato es animal, y una abstracción de silla es mueble. A veces, pero, no es tan simple. Ya se ve que cuanto más concreto es un concepto, menos abstracto. La abstracción es la relación más vertical entre palabras. El índice de un libro, que materializa una aproximación vertical al contenido, también es una abstracción.

En la programación orientada a objetos, la idea de abstracción está implementada. Y no es por casualidad que todos los lenguajes que se desarrollan sobre esta tecnología concluyan en la idea de *objeto* como término al máximo nivel de abstracción. Por eso, todo, en el fondo, son objetos.

La abstracción es una operación mental magníficamente complicada, que produce satisfacción de resolver. No es una operación calculable, ya que precisa de una riqueza terminológica que es difícil estructurar.

El proceso que se realiza cuando se define una abstracción se puede denominar inferencia. Hay léxicos que enriquecen las semánticas y este es un caso. Comprender qué significa una inferencia ayuda a entender una abstracción. Las abstracciones de los conceptos se infieren, por definición.

## 1.2.2 Descripción de Conjuntos

Describir un conjunto significa dar los datos necesarios para que el interlocutor o lector comprenda sin sombra de incertidumbre, cuáles son los elementos que pertenecen a él. Básicamente hay dos maneras de hacerlo. Una manera es enumerando los elementos, y la otra seleccionando de entre los elementos de un conjunto más grande los que cumplan un predicado. Esto significa definir un conjunto como la colección de aquellos elementos que cumplan una expresión evaluable como cierta o falsa.

Por lo visto hasta ahora, está bien claro que en ningún momento se ha hecho mención al orden de los elementos dentro del conjunto. Estructuramos conceptos. La ordenación de los elementos de un conjunto no forma parte de la definición del conjunto. Por esa razón, en lenguaje formal denotamos con llaves,  $\{\}$ , la definición de los conjuntos. Más concretamente, la definición de un conjunto, en cualquiera de las dos maneras de describirlo, queda entre los símbolos de llave de apertura,  $\{$ , y de cierre,  $\}$ .

Filosóficamente, se percibe alguna relación entre la ordenación y el número dos, ya que para poder definir un orden entre los elementos de un conjunto es necesario poder responder cuál de los dos elementos es menor de cada pareja posible de elementos, que por cierto, son  $n(n-1)/2$ . Es fácil verlo, si tenemos  $n$  elementos y cada uno puede aparejarse con los  $n-1$  restantes, entonces tenemos  $n(n-1)$  posibles parejas, pero como no consideramos el orden entre las parejas, nos quedan la mitad. Un número entero seguro, ya que o bien  $n$  o bien  $n-1$  será par.

### Por enumeración de sus elementos

A la forma más primitiva de describir un conjunto la llamamos enumeración, y consta de una secuenciación de los elementos. Por ejemplo,  $C = \{e_1, e_2, e_3\}$ . Entonces, aunque no lo queramos, si describimos un conjunto por enumeración estamos estableciendo un orden entre sus elementos. Como consecuencia, lo que estamos describiendo es una secuencia, cosa que es más concreta que un conjunto porque además, tiene un orden. Todas las secuencias son conjuntos, pero no todos los conjuntos son secuencias.

Por eso usamos las llaves para denotar los conjuntos. Cuando se pretende indicar que el orden de la secuencia es importante, entonces en lugar de llaves se usan paréntesis. La expresión  $C = (e_1, e_2, e_3)$  indica estos elementos en ese orden.

Está claro que la ventaja de describir un conjunto por enumeración es la evidencia. Los elementos que componen el conjunto que se describe los tenemos frente a nuestros ojos. Y también es fácil ver que mediante la enumeración solo podemos describir conjuntos finitos. Es decir, que el cardinal sea un número concreto. Hay muchas disciplinas que trabajan con conjuntos de infinitos elementos. Y por tanto, esta manera de describir un conjunto no les sirve demasiado.

### A partir de propiedades de sus elementos

Otra manera de describir un conjunto es estableciendo un criterio que han de satisfacer los elementos de un conjunto más grande para pertenecer al que se está definiendo. Es fácil comprender un conjunto definido por enumeración. En cambio, definir un conjunto como el subconjunto de elementos de otro conjunto más grande que cumplan una condición requiere más inteligencia por parte del receptor de la definición, ya que le resulta necesario interpretar los criterios que definen la pertenencia.

El conjunto de todas las personas que llevan lentes es un conjunto definido a partir de propiedades de sus elementos. Está bien claro. El hecho de no tener la capacidad de enumerar cada uno de los elementos del conjunto en cuestión no impide que podamos hablar de él. Podemos decir cosas. Por ejemplo, hay menos personas que llevan lentes azules que personas que llevan lentes. O sea, un subconjunto del conjunto de las personas que llevan lentes es el de las personas que llevan lentes azules. Y esto, también significa que si lleva lentes azules, lleva lentes.

Obsérvese con atención la estructura deductiva del párrafo anterior. *Si lleva lentes azules implica que lleva lentes*. Hay alguna cosa espiritual en el hecho de asociar el concepto de subconjunto, que es un sustantivo, con la implicación que tiene como consecuencia, que es un verbo.

Pero a parte de este hecho, que por sí solo ya es desconcertante, también aquí escondido hay el secreto de por qué la teoría de conjuntos tiene el prestigio que tiene, y por qué hay tantas disciplinas que derivan de ésta. Hay muchas personas que cuando se les dice que la teoría de conjuntos es muy importante no entienden por qué. Pues ahí está. La forma de deducir que tenemos los humanos se puede explicar de una manera gráfica. La teoría de conjuntos tiene la trascendencia que tiene porque sirve para entender cómo deducimos. Y eso es muy importante.

Que un elemento pertenezca a un subconjunto implica que pertenece al conjunto, si es un gato es un animal. Y también podemos leer el silogismo de derecha a izquierda, negándolo. A la inversa, si no pertenece a un conjunto, no puede pertenecer a ninguno

de sus subconjuntos, si no es un animal, no puede ser un gato.

Para cualquier conjunto de objetos  $C$ , y subconjunto  $S \subseteq C$ , podemos formular sistemáticamente la proposición *todo*  $S$  es  $C$  pero *no todo*  $C$  es  $S$ . O sea, toda silla es mueble pero no todo mueble es silla. O todo gato es animal pero no todo animal es gato. Y de manera dual, en el mundo de las acciones, para cualquier conjunto de acciones  $C$ , y subconjunto  $S \subseteq C$ , podemos formular sistemáticamente la proposición *siempre* que se  $C$  se  $S$ , pero *no siempre* que se  $S$  se  $C$ . O sea, siempre que se canta se habla, pero no siempre que se habla se canta. O siempre que se lee se mira, pero no siempre que se mira se lee. Es interesante observar la transformación de *todo* a *siempre* en la dualidad espacio tiempo.

Describir un conjunto por medio de las propiedades de los elementos que lo componen es ciertamente una manera más complicada que la anterior. Aun así, tiene la ventaja de poder hablar de conjuntos de muchos elementos, o infinitos. Y por contra, no es una definición constructiva en el sentido que requiere de un conjunto mayor al cual se hace referencia en la definición, que en el caso del ejemplo sería el conjunto de todas las personas. De todas formas, como este requerimiento viene dado y no acostumbra a representar ningún esfuerzo partir de un universo de referencia, esta segunda manera de describir conjuntos es la más ampliamente utilizada.

### 1.3 Operaciones entre Conjuntos

Resulta muy gratificante poder establecer reglas que operen con conceptos tan abstractos como son los conjuntos. Las reglas que seamos capaces de formular en este nivel de abstracción, casi inimaginable de superar, nos resultarán útiles para todas las teorías que se deriven de ésta, que son la mayoría.

Este es el enorme valor que tienen unos postulados tan simples y tan convencionales como los que siguen.

Las operaciones entre conjuntos son operaciones como la suma o el producto, excepto que a la suma, por ejemplo, se le dan dos o más números de entrada y nos retorna un número de salida. Éstas de ahora, en cambio, son operaciones a las cuales les damos de entrada dos o más conjuntos y de salida también nos retornan un conjunto.

En adelante, diremos  $C$  y  $D$  a dos conjuntos cualesquiera. Para los ejemplos gráficos, se utilizarán los dos conjuntos que se muestran en la Figura 1.1.

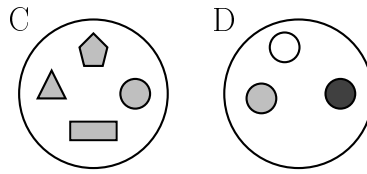


Figura 1.1: *Conjuntos C y D utilizados en los ejemplos.*

### 1.3.1 Unión

En la Figura 1.2 se muestra la imagen mental que conviene retener para la noción de unión de conjuntos.

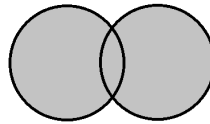


Figura 1.2: *Mnemograma para la unión de conjuntos.*

El conjunto resultante de la unión de dos o más conjuntos está compuesto por los elementos que pertenecen a cualquiera de los conjuntos dados de entrada. La unión es una operación conmutativa. Que los conjuntos de entrada se puedan conmutar significa que la unión entre dos conjuntos  $C$  y  $D$  es el mismo conjunto que la unión entre  $D$  y  $C$ .

Para las representaciones escritas de esta idea se utiliza el símbolo  $\cup$  en notación infija, o sea, entre los conjuntos que se dan de entrada a la operación. Así pues, en lenguaje formal definimos la unión de los conjuntos  $C$  y  $D$  como se indica en la siguiente caja.

$$C \cup D = \{x \mid x \in C \vee x \in D\}$$

Caja 1.3. *Definición de la unión de conjuntos.*

Unos comentarios respecto a esta definición para aclarar terminología.

- Leemos la barra vertical diciendo "tales que", o "tal que", ya que en lenguaje formal no hay singulares ni plurales. En otras nomenclaturas se utilizan los dos puntos en lugar de la barra vertical. En lenguaje formal,  $\mid$  y  $:$  son símbolos sinónimos.

- El símbolo  $\vee$  significa "o" en lenguaje formal de predicados. De hecho, está íntimamente relacionado con el símbolo mismo de la unión de conjuntos. Usamos  $\cup$  para los conjuntos, y  $\vee$  para los predicados, pero como se verá más adelante, en esencia son lo mismo.
- Toda ella, pues, se podría leer como "La unión del conjunto  $C$  y el conjunto  $D$  es el conjunto formado por los elementos  $x$  tales que  $x$  pertenece a  $C$  o  $x$  pertenece a  $D$ " (en lenguaje formal se echan de menos los pronombres).
- La definición contiene dos expresiones. La de la derecha del signo "=", y la de la izquierda. Es como el sintagma nominal y el sintagma verbal de una frase. El signo de igual aquí desempeña el papel del verbo.
- A las variables que están en expresiones donde se utiliza todo su dominio, como la  $x$  en la expresión de la parte derecha de la caja, se las denomina variables *ligadas*, o *cerradas*, en la expresión. Se las reconoce porque son las que si cambiáramos su nombre en todas sus apariciones, en lugar de  $x$  la denotáramos por  $y$  por ejemplo, el sentido de la definición global seguiría siendo correcto. Son como variables locales en los lenguajes de programación.

De manera gráfica, en la Figura 1.3 se muestra la unión de los conjuntos de la Figura 1.1.

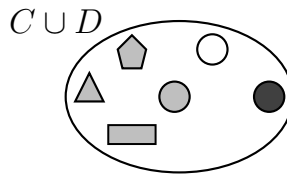


Figura 1.3: *Conjunto Unión  $C \cup D$  de los conjuntos de la Figura 1.1.*

Nótese que en este ejemplo se ilustra el hecho de que por definición un conjunto no puede tener elementos repetidos. Eso ha provocado que haya desaparecido un círculo gris.

Y también un ejemplo textual para no dejar dudas. Si  $C = \{e_1, e_2, e_3\}$  y  $D = \{e_3, e_4\}$ , entonces  $C \cup D = \{e_1, e_2, e_3, e_4\}$ .

A continuación, se expresa en lenguaje formal algunas propiedades de la unión de conjuntos. Sólo para hacer gimnasia. Podríamos pensar en otras.

- $C \cup C = C$ .
- $C \cup \emptyset = C$ .
- $S \subseteq C \Rightarrow S \cup C = C$ .
- $|C \cup D| \leq |C| + |D|$ .



### 1.3.2 Intersección

En la Figura 1.4 se puede ver una idea gráfica de la intersección de conjuntos.

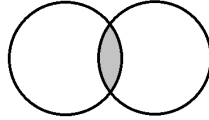


Figura 1.4: *Mnemograma para la intersección de conjuntos.*

El conjunto resultante de la intersección de dos o más conjuntos está compuesto por los elementos que pertenecen a ambos conjuntos dados de entrada. La intersección también es una operación conmutativa.

Para las representaciones escritas de esta operación se utiliza el símbolo  $\cap$ . Entonces, en lenguaje formal definimos la intersección de los conjuntos  $C$  y  $D$  como se presenta en la caja a continuación.

$$C \cap D = \{x \mid x \in C \wedge x \in D\}$$

Caja 1.4. *Definición de la intersección de conjuntos.*

Esta definición se debería leer como "la intersección del conjunto  $C$  y el conjunto  $D$  es el conjunto formado por todos los elementos  $x$  tales que  $x$  pertenece a  $C$  y  $x$  pertenece a  $D$ ". El símbolo  $\wedge$  significa "y" en lenguaje formal de predicados. Y como en el caso de la unión, está íntimamente relacionado con el símbolo mismo de la intersección de conjuntos. Usamos  $\cap$  para los conjuntos, y  $\wedge$  para los predicados.

Otra vez, de manera gráfica, en la Figura 1.5 se muestra la intersección de los conjuntos de la Figura 1.1.

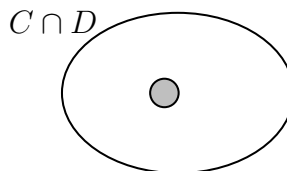


Figura 1.5: *Conjunto Intersección  $C \cap D$  de los conjuntos de la Figura 1.1.*

Un ejemplo textual. Si  $C = \{e_1, e_2, e_3\}$  y  $D = \{e_3, e_4\}$ , entonces  $C \cap D = \{e_3\}$ .

Y también como antes, expresamos en lenguaje formal algunas propiedades de esta operación. Para divertirnos. Piénsese otras...

- $C \cap C = C$ .
- $C \cap \emptyset = \emptyset$ .
- $S \subseteq C \Rightarrow S \cap C = S$ .
- $|C \cap D| \leq \min(|C|, |D|)$ .

### 1.3.3 Diferencia

La imagen asociada a la diferencia de conjuntos se encuentra en la Figura 1.6.

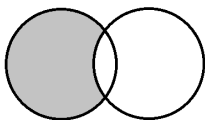


Figura 1.6: *Mnemograma para la diferencia de conjuntos.*

A veces, tan solo con estos mnemogramas hay suficiente para inspirar razonamientos válidos.

El conjunto resultante de la diferencia de dos conjuntos está compuesto por los elementos que pertenecen al primero y no pertenecen al segundo de los conjuntos dados de entrada. La diferencia de conjuntos es una operación no conmutativa.

Para las representaciones escritas de esta idea se utiliza el símbolo  $\setminus$ . En lenguaje formal definimos la diferencia del conjunto  $C$  menos el conjunto  $D$  como se introduce en la siguiente caja.

$$C \setminus D = \{x \mid x \in C \wedge x \notin D\}$$

Caja 1.5. *Definición de la diferencia de conjuntos.*

Esta definición se lee "la diferencia del conjunto  $C$  menos el conjunto  $D$  es el conjunto formado por todos los elementos  $x$  tales que  $x$  pertenece a  $C$  y  $x$  no pertenece a  $D$ ".

En la Figura 1.7 se puede observar la diferencia entre los conjuntos  $C$  y  $D$  del ejemplo.

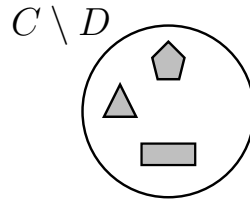


Figura 1.7: *Conjunto Diferencia  $C \setminus D$  de los conjuntos de la Figura 1.1.*

Y para el ejemplo textual, si  $C = \{e_1, e_2, e_3\}$  y  $D = \{e_3, e_4\}$ , entonces  $C \setminus D = \{e_1, e_2\}$ .

La diferencia entre conjuntos nos permite introducir la noción de *complementario*. Si  $S$  es un subconjunto de  $C$ , entonces el complementario de  $S$ , que denotamos por  $\bar{S}$ , es igual a  $C \setminus S$ . O sea, los otros de  $C$ . Obsérvese que la noción de complementario es más fácilmente comprensible cuando el conjunto ha sido definido con la segunda de las dos formas de definir conjuntos, es decir, cuando el conjunto ha sido definido a partir de las propiedades que cumplen sus elementos.

Con todo, podemos decir cosas como

- $C \setminus D = C \setminus \{C \cap D\}$ .
- $S \cup \bar{S} = C$ .
- $S \cap \bar{S} = \emptyset$ .

### 1.3.4 Producto Cartesiano

El producto cartesiano de dos conjuntos es una operación esencialmente distinta de las anteriores. Eso es así porque en las anteriores, los elementos del conjunto resultante eran de los mismos elementos que los conjuntos participantes en la operación.

Los elementos que forman el conjunto producto cartesiano de dos conjuntos son parejas.

Cada elemento del producto cartesiano es una pareja formada por un elemento del primer conjunto, y un elemento del segundo conjunto de entrada. O sea, que por cada combinación formada por un elemento del primer conjunto y un elemento del segundo

tenemos un elemento del producto cartesiano. Que sea commutativa depende de que lo sea el aparejamiento de los elementos.

Obsérvese la imagen que debería servir para recordar un producto cartesiano entre conjuntos, en la Figura 1.8. Considérese que cada una de las dos líneas de trazo grueso representa uno de los dos conjuntos. Cada marca que hay en esas líneas significa un elemento. Entonces, en el plano tenemos un punto asociado a cada pareja de elementos de los conjuntos iniciales.

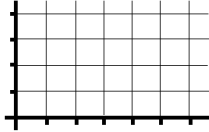


Figura 1.8: *Mnemograma para el producto cartesiano de conjuntos.*

Es notable que para definir el producto cartesiano se necesita secuencializar los elementos de los conjuntos de entrada mediante algún orden. Y es interesante observar que el hecho de poner los elementos de un conjunto en una disposición secuencial se soporta sobre la base de que sean identificables, ya que una secuencia requiere de un orden, y por tanto, también requiere una operación de comparación entre los elementos del conjunto. En este extremo hay alguna cosa que chirría. Un conjunto no tiene orden por definición, pero en cambio, a la hora de calcular un producto cartesiano, se supone que los conjuntos de entrada a la operación se pueden ordenar... Bien, en cualquier caso, como siempre trabajaremos con conjuntos ordenables, la cosa no tiene más trascendencia.

Para las representaciones escritas de esta operación se utiliza el símbolo  $\times$ . El producto cartesiano de los conjuntos  $C$  y  $D$  se define formalmente tal como se muestra a continuación.

$$C \times D = \{xy \mid x \in C \wedge y \in D\}$$

Caja 1.6. *Definición del producto cartesiano de conjuntos.*

En la Figura 1.9 se presenta el producto cartesiano de los conjuntos del ejemplo.

Y siguiendo con el ejemplo textual, dados los conjuntos  $C = \{e_1, e_2, e_3\}$  y  $D = \{e_3, e_4\}$ , entonces  $C \times D = \{\{e_1, e_3\}, \{e_1, e_4\}, \{e_2, e_3\}, \{e_2, e_4\}, \{e_3, e_3\}, \{e_3, e_4\}\}$ . Obsérvese con atención, porque es necesario saber describir el resultado de esta operación.

Un ejemplo más. El conjunto de los números enteros del 0 al 9 producto cartesiano

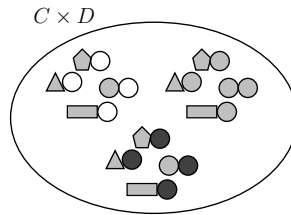


Figura 1.9: *Producto Cartesiano  $C \times D$  de los conjuntos de la Figura 1.1.*

consigo mismo es el conjunto de los números del 00 al 99. O sea, todas las parejas posibles formadas por un elemento del 0 al 9 y otro elemento del 0 al 9. Esta reflexión es interesante. Si tenemos un conjunto, el producto cartesiano de él multiplicado por él mismo es como si tratáramos los elementos del conjunto como si fueran dígitos de un sistema de numeración que en lugar de tener 10 dígitos básicos tuviera tantos como elementos haya en el conjunto. Y entonces el producto cartesiano es como contar hasta cien. Para cualquier conjunto  $C$ , el conjunto de todas las parejas posibles de elementos de  $C$  es  $C \times C$ .

Algunas propiedades del producto cartesiano de conjuntos son

- $C \times \emptyset = \emptyset$ .
- $|C \times D| = |C| |D|$ .

## 1.4 Lógica de Predicados

La lógica de predicados está íntimamente relacionada con el Álgebra de Boole por un lado, cosa que la aproxima a la computación. Además, en capítulos posteriores será necesario tener claras algunas cuestiones supuestamente conocidas por los alumnos, y por esa razón aquí hacemos un repaso utilizando un lenguaje común a los dos ámbitos como es la lógica de predicados.

### 1.4.1 Cálculo de Predicados

Un predicado es una expresión que puede ser evaluada como cierta o falsa.

Por ejemplo, "hoy hace sol". O sea, que la noción de predicado es análoga a la de expresión booleana en los lenguajes de programación. Esta expresión está formada de proposiciones que juegan el papel de variables booleanas, y de operadores que

se corresponden con las operaciones lógicas que se describen a continuación. La definición de estas operaciones proviene del cálculo proposicional de orden cero, que es una teoría que consta de nueve principios que constituyen los principios básicos de la deducción natural.

Para definir el funcionamiento de una operación se utiliza una tabla, que es la definición más detallada posible para cualquier relación. Se denominan tablas de verdad por los valores de su contenido. En las columnas de la izquierda se ponen todas las posibles combinaciones de los valores de las proposiciones, y en la columna de la derecha se pone la respuesta de la operación para la combinación de valores correspondiente a la fila.

Las operaciones de la lógica de predicados y las del Álgebra de Boole son iguales. La diferencia más notoria es que los posibles valores para una proposición (y por tanto para un predicado) son *cierto*, C, y *falso*, F. En cambio los valores posibles de una variable booleana son  $\{0,1\}$ . Otra diferencia es que el uso aritmético de variables booleanas como la suma o el producto, o la misma propiedad asociativa de la suma respecto al producto es legítimo. En cambio, con proposiciones no hacemos estas cosas, porque el cálculo proposicional está antes que la teoría de números. Y por tanto, no tiene sentido utilizar una suma de proposiciones. De todas formas, la línea que separa ambas teorías es muy fina, y hay contenidos, como por ejemplo las leyes de De Morgan, que bailan entre los dos mundos.

Que dos proposiciones sean independientes significa que ninguna de ellas condiciona el valor de la otra. Eso significa que se puede producir cualquier combinación de valores entre ellas. Las proposiciones "esto pesa más de un quintal" y "esto pesa más de dos quintales" no son independientes. Las proposiciones "soy carpintero" y "me llamo Juan" son independientes.

Seguidamente se introducen las tres operaciones del cálculo de predicados.

### Operación de negación lógica

La operación de negación tiene como operando de entrada una sola proposición. El operador correspondiente se escribe  $\neg$  como prefijo del predicado que se niega, y se pronuncia "no". Es decir,  $\neg p$  se lee "no p".

Sea  $p$  una proposición. Entonces, la tabla de verdad que caracteriza la operación de negación lógica se muestra en la Tabla 1.1.

La propiedad más relevante que se desprende del comportamiento de esta operación es el hecho de que  $\neg\neg p = p$ . Esto, en muchos lenguajes naturales, especialmente los descendientes del latín, acarrea graves problemas por el fenómeno de la doble

$p$	$\neg p$
F	C
C	F

Tabla 1.1: *Tabla de verdad de la negación lógica.*

negación... no he dicho nada. De hecho, cualquier pregunta formulada utilizando la palabra *no* acostumbra a conducir a confusiones. Cualquier analista, o científico, debería evitar las preguntas negadas. ¿No se entiende? Pues eso.

Y por la misma razón es de muy mal programador denominar con identificadores que empiecen con *no* a las variables booleanas. Difícilmente justificable, ya que siempre se puede ajustar el código para dejar la variable en forma directa, no negada. Y exactamente lo mismo para las sentencias alternativas completas. Las que tienen *if then else*. Es buena práctica poner las expresiones del *if* en positivo.

### Operación lógica disyuntiva

Esta operación también es conocida como "o lógica". Está emparentada con la unión de conjuntos, vista en la Sección 1.3.1.

La operación disyuntiva tiene como operandos de entrada dos predicados. Para la unión de predicados se utiliza el operador  $\vee$  que se pronuncia "o". Esto es,  $p \vee q$  se lee "p o q".

Si  $p$  y  $q$  son dos proposiciones independientes entre ellas, entonces la tabla de verdad que define la disyunción lógica es la que se puede ver en la Tabla 1.2.

$p$	$q$	$p \vee q$
F	F	F
F	C	C
C	F	C
C	C	C

Tabla 1.2: *Tabla de verdad de la disyunción lógica.*

Es decir, la o lógica entre dos proposiciones es cierta si cualquiera de las dos lo es.

Es una operación conmutativa, como se puede deducir a partir de las filas segunda y tercera de la tabla. Además es asociativa, de forma que si  $p$ ,  $q$ , y  $r$  son tres proposiciones, entonces  $p \vee q \vee r = (p \vee q) \vee r = p \vee (q \vee r)$ . Esto provoca que la definición, más que para dos operandos, valga para dos o más.

Algunas propiedades de esta operación nos ayudarán a ver el parentesco que tiene

con la unión de conjuntos.

- $p \vee \mathbf{F} = p$ .
- $p \vee \mathbf{C} = \mathbf{C}$ .
- $p \vee p = p$ .
- $p \vee \neg p = \mathbf{C}$ .

### Operación lógica conjuntiva

Esta operación también es conocida como "y lógica". Está emparentada con la intersección de conjuntos, vista en la Sección 1.3.2. La operación conjuntiva también tiene como operandos de entrada dos proposiciones. Utiliza el símbolo  $\wedge$ , que se lee "y". O sea,  $p \wedge q$  se pronuncia "p y q".

La tabla de verdad que define la conjunción lógica entre dos proposiciones independientes  $p$  y  $q$  se muestra en la Tabla 1.3.

$p$	$q$	$p \wedge q$
$\mathbf{F}$	$\mathbf{F}$	$\mathbf{F}$
$\mathbf{F}$	$\mathbf{C}$	$\mathbf{F}$
$\mathbf{C}$	$\mathbf{F}$	$\mathbf{F}$
$\mathbf{C}$	$\mathbf{C}$	$\mathbf{C}$

Tabla 1.3: *Tabla de verdad de la conjunción lógica.*

Es decir la y lógica entre dos proposiciones es cierta si lo son las dos. También es conmutativa y asociativa.

Es importante notar que los términos disyunción y conjunción son peligrosos porque es extremadamente fácil confundirlos. Es una confusión muy normal. Esto es debido a que los símbolos de la unión  $\cup$  de conjuntos, la operación  $\vee$  entre predicados, incluso las mismas palabras "o", y "unión", se parecen. Por otro lado, con la intersección ocurre lo mismo. O sea, "intersección" se parece a "y", a  $\wedge$ , y a  $\cap$ . En cambio, con los nombres de las operaciones, va al contrario, la disyunción, que suena a "y", es la operación "o", y la conjunción que recuerda la "o", es la operación "y", o sea, que cuidado con los nombres de las operaciones lógicas. Esta confusión es tan habitual que hay quien denomina esas operaciones con los nombres de las operaciones de la teoría de conjuntos.



### Leyes de Augustus de Morgan

Concluimos el tema del cálculo de predicados enunciando las leyes de De Morgan. Aquí se muestra en toda su plenitud la dualidad entre la unión y la intersección de conjuntos, o entre las operaciones lógicas disyuntiva y conjuntiva.

Las leyes de De Morgan se postulan sobre ámbitos diversos, cosa que refleja la analogía entre ellos. Su interés estriba en el hecho de relacionar formalmente las operaciones de la lógica de predicados, o de la teoría de conjuntos. Dicen que la negación de una conjunción es una disyunción, y a la inversa.

- |  |
|--|
| <ol style="list-style-type: none"> <li>1. <math>\neg(p \vee q) = \neg p \wedge \neg q.</math></li> <li>2. <math>\neg(p \wedge q) = \neg p \vee \neg q.</math></li> </ol> |
|--|

Caja 1.7. *Leyes de De Morgan.*

Con todo rigor, la negación de la disyunción es la conjunción de las negaciones. Y a la inversa, la negación de la conjunción es la disyunción de las negaciones.

Supongamos que  $p$  significa "llueve". Y  $q$ , "voy a la playa". La disyuntiva sería "llueve o voy a la playa". Y para negarla deberíamos decir "no llueve y no voy a la playa". Tiene una gran lógica. Si  $a$  o  $b$  es mentira, entonces  $a$  es mentira y  $b$  también es mentira, porque si una de las dos cosas fuera verdad, entonces una o la otra también lo sería.

Las leyes de De Morgan son profusamente utilizadas en multitud de demostraciones en la lógica matemática. Y también en la electrónica digital. Utilizando estas leyes, somos capaces de fabricar todos los tipos de puertas lógicas de los circuitos integrados a partir de tan solo un tipo, la conjunción negada, o puerta *nand*.

### 1.4.2 Relación con la Teoría de Conjuntos

Como se ha introducido en la Sección 1.2.2, hay una correspondencia entre dos hechos que parecen pertenecer a universos distintos. Que una proposición implique otra,  $p$  implica  $q$ , significa que el conjunto de verdades que llamamos  $P$  es un subconjunto del conjunto de situaciones en las que  $Q$  es verdad. Esta asociación entre la teoría de conjuntos y la lógica de predicados es un pilar fundamental sobre el que se sustenta gran parte del conocimiento analítico. Y como sabemos que si un elemento pertenece a un subconjunto debe pertenecer al conjunto, podemos afirmar formalmente lo que

sigue.

$$p \rightarrow q \Leftrightarrow \bar{P} \cup Q$$

Caja 1.8. *Relación entre la lógica de predicados y la teoría de conjuntos.*

La equivalencia de esta caja se leería como,  $p$  implica  $q$  significa lo mismo que no  $p$  o  $q$ . Para entendernos, supongamos que  $p$  es "es bombero", y  $q$  es "es persona". Está claro que la parte izquierda de la expresión se cumple ya que efectivamente si es bombero, es persona. La parte de la derecha, que significa lo mismo dicho de otra forma, dice que una de dos, o no es bombero, o es persona. Obsérvese la utilización del complementario de  $P$  en la parte derecha de la expresión. Complementario se asocia a negación.

*En este primer capítulo se han mostrado las nociones más elementales de la teoría de conjuntos, las operaciones embleáticas de esta teoría, en las cuales se ha incluido el producto cartesiano, así como de la lógica de predicados. Estos conceptos serán intensamente utilizados en los temas siguientes, y por eso, aunque sean conceptos sencillos, deberían quedar sólidamente fundamentados.*



# Capítulo 2

## Análisis del Proyecto

El análisis es un proceso de absorción de conocimiento. En ningún caso se trata de una actitud creativa. Es una postura observadora. El analista debe tener una mentalidad esponjosa, muy preparada para comprender todo aquello que se le explica con la finalidad de sumergirse en el universo de la aplicación. En este capítulo se tocan varios aspectos que hay que considerar.

### 2.1 Sistemas Gestores de Bases de Datos

Un Sistema Gestor de Bases de Datos, SGBD, o *Database Management System*, DBMS, es un programa informático que gestiona bases de datos.

En las bases de datos relacionales, una información elemental se representa en una cadena de caracteres, por ejemplo, que se guarda como valor de un atributo de un elemento concreto de una relación concreta que forma parte de una base de datos concreta. Esa es la estructura jerárquica. Normalmente un gestor de bases de datos gestiona varias bases de datos a la vez, y al conjunto de todas ellas las llamamos *clúster*. Un clúster es una instalación de un sistema gestor de bases de datos.

A grandes rasgos, los programas informáticos se pueden agrupar en tres categorías. Los programas *batch*, los servicios, y los programas de interfaz.

Los programas batch son muy anteriores a los otros dos tipos. Originalmente, a parte del sistema operativo, eran el único tipo de programa que había. Se caracterizan porque empiezan y acaban. No tienen casi interfaz. A menudo corren de fondo, sin intervención del usuario, a partir de una programación de tareas, o formando parte de

procesos más grandes. Programas de backup o de formateo de archivos son programas batch. Su naturaleza estriba en la máquina de Turing, aquella que tenía una cinta de entrada y un estado interno y según el valor que entrase por la cinta y el estado actual producía una salida y un cambio de estado.

Los servicios son un tipo de programas activados por otros programas, no por usuarios finales. Antiguamente se les llamaba programas residentes en memoria. Es habitual que se inicien cada vez que se arranca la máquina, o cada vez que accede un nuevo usuario al sistema. Entonces, esos programas se quedan en memoria principal y esperan peticiones mediante algún mecanismo. En este tipo de programas es en el que se enmarcan los SGBDs. Su manera de trabajar adquirió protagonismo con la emergencia contundente de las redes. Y a partir del momento que se dedicaron a escuchar peticiones de otras computadoras se llamaron servicios. Es incuestionable que la hegemonía que hay hoy día con la preponderancia de los ordenadores pequeños en red ha eliminado las arquitecturas basadas en supercomputadores. En otras palabras, la unión hace la fuerza. Hoy día las redes mandan. En definitiva, la auténtica inteligencia reside en los actores pequeños, pero bien comunicados. Las altas frecuencias.

Los de interfaz, GUI de *Graphic User Interface*, o sea de diálogo, son los que han evolucionado más desde el principio de la informática. En ese ámbito hay un salto histórico importante marcado por la aparición de pantallas gráficas, o sea que podían pintar un solo punto del monitor. Eso, introducido por Steve Jobs a mediados de los ochenta, marca un cambio de proporciones inconmensurables. El diálogo hombre máquina pierde la sincronización, ya que hasta entonces las cosas se decían de una en una. La interacción pues, pasa de una a dos dimensiones. De línea de comandos a interfaz gráfica. El término *línea* debe hacer pensar en una dimensión. El término *gráfica* en dos. Aparecen las ventanas, y el ratón. Ya se pueden decir dos cosas a la vez. A partir de ahí, los programas empiezan a ser conducidos por casos, de manera que tienen un bucle infinito en su núcleo, y mientras no se da el caso que el usuario indique que se cierre, el programa no se para.

Los programas que actualmente no tienen interfaz gráfica, o sea ni ratón ni ventanas, funcionan mediante línea de comandos. Se trata de software no dirigido a usuarios finales. O sea que sus usuarios son o bien técnicos informáticos, o bien otros programas que hacen las tareas de interfaz para los usuarios finales.

De todo ello, se debe sacar la certeza fundamental de que un gestor de bases de datos no contempla ninguna interacción con el exterior. No espera ningún diálogo con usuarios finales. Eso es importante a la hora de estructurar el trabajo. En el desarrollo de una base de datos no se debe tener en cuenta el aspecto de la interfaz.

### 2.1.1 Arquitectura Cliente-Servidor

La idea de arquitectura *cliente-servidor* significa que el SGBD como servicio se ejecutará en el ordenador, o *host*, servidor. Por eso también se le llama *back-end*. Porque es el extremo lejano de la conexión, respecto a los usuarios.

En cambio, la aplicación de interfaz que se dedicará a traducir las peticiones de estos usuarios, se llama *cliente*, o *front-end* (en inglés la palabra *end* también se puede entender como extremo).

En la Figura 2.1 se pueden ver los dos componentes, y el medio que los une, que normalmente será internet, intranet, o cualquier otro tipo de conexión.



Figura 2.1: *Arquitectura Cliente-Servidor*.

Microsoft Acces introdujo la primera interfaz gráfica para una base de datos. En ciertos aspectos fue un éxito, prueba es la gran cantidad de aplicaciones GUI que después le han seguido la línea. Sin embargo, también ha tenido su parte corrosiva. Las personas que aprenden bases de datos con sistemas GUI pierden las nociones de cliente y servidor. Y no distinguen qué opciones de los menús de la aplicación forman parte del núcleo del SGBD, y qué otras son utilidades adicionales añadidas, que no tienen nada que ver, o bien poco, con la base de datos real.

## 2.2 Principio de Independencia de los Datos

Una diferencia importante entre las aplicaciones que gestionan bases de datos y las que realizan otras tareas es la capacidad de reconstruirse desde la nada.

Los programas de cálculo pueden recompilarse y reconstruirse las veces que haga falta porque no tienen ningún otro compromiso con el exterior que los datos que gestionan en sus interfaces. Un inconveniente de las aplicaciones de gestión de bases de datos pues, es que el compromiso con el exterior no solo es con los datos que introduce el usuario, sino que también con la estructura de la base.

Cuando una aplicación de gestión de bases de datos se pone en producción, ya nunca más puede volver a empezar de cero, porque una vez haya datos ya no podremos

rectificar las estructuras con facilidad. Por eso hay lo que se llama principio de independencia de los datos, y de hecho es una de las razones que justifican la existencia de los SGBDs. Los cambios en la estructura de la base no deben provocar cambios en los programas clientes que la usan.

El estado deseable al que se debería alcanzar es el de poder efectuar desde la línea de comandos de la consola del SGBD la totalidad de las acciones que finalmente harán las aplicaciones cliente, sin ventanas ni colores, ni fuentes de texto. En definitiva, ofrecer unas determinadas funciones como interfaz de la base de datos. Y deben ser estas mismas funciones las que utilicen las aplicaciones de interfaz. El concepto es análogo al de una clase en java. El estado desable es la implementación de un conjunto de métodos públicos.

En la Figura 2.2 se muestra la clásica estructura de cebolla que ilustra el funcionamiento de las aplicaciones de bases de datos. En el núcleo hay la circuitería, en la que se soporta la microprogramación, el lenguaje ensamblador, el sistema de archivos,... y todo lo que compone el sistema operativo. Y sobre esa capa, o dentro, los SGBDs.

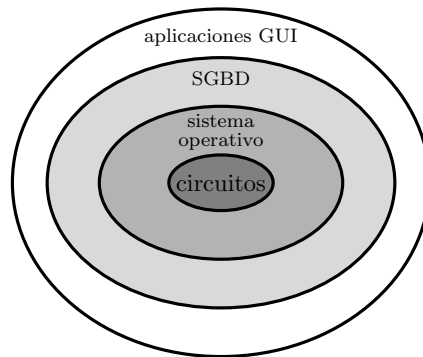


Figura 2.2: *Modelo en forma de cebolla de un sistema informático.*

Finalmente, los SGBDs son los que soportan las aplicaciones de usuario final.

Que en la Figura 2.2 una capa se soporte sobre otra significa que la interna ofrece operaciones primitivas o elementales a la externa.

En definitiva, es como si la base de datos fuese una clase de un lenguaje de programación, que ofrece un conjunto de operaciones públicas como añadir, modificar, consultar o borrar cualquier objeto del dominio, además de aquellas consultas que se hayan formulado en la definición de requerimientos.



### 2.2.1 Arquitectura a Tres Niveles

La arquitectura cliente servidor tiene un vínculo directo con lo material, es una arquitectura física. También podemos hablar de una arquitectura lógica, o patrón arquitectónico al cual conviene ajustarse para implementar una base de datos.

Esta forma de segmentar los módulos que integran un SGBD reside en el lado del servidor. Los tres niveles se llaman nivel físico, lógico, y de aplicación, tal como se puede contemplar en la Figura 2.3. En el nivel físico hay todo lo que tiene que ver con la interacción entre el SGBD y el sistema operativo, así como la parte del SGBD que soporta las bases de datos que finalmente utilizarán las aplicaciones. Eso incluye los metadatos, el intérprete del lenguaje estructurado de consultas que normalmente dispone también de un optimizador de consultas, y yendo más allá podríamos hablar de gestores de concurrencia, de replicación, de seguridad, así como muchos otros módulos que en la Figura 2.3 no aparecen porque quedan fuera del foco que se pretende exponer en este libro.

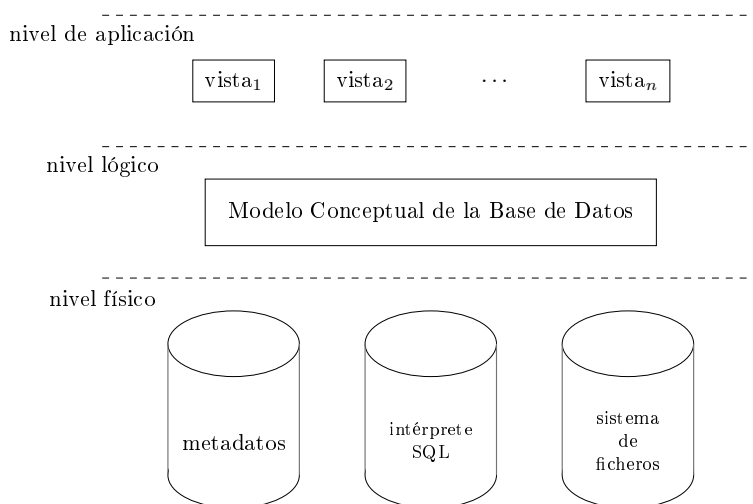


Figura 2.3: *Arquitectura a tres niveles.*

Una característica importante del nivel físico es el hecho de ser transparente a los desarrolladores de las bases de datos. No tienen ninguna interacción. Y aquí se da a conocer porque siempre conviene saber qué hay tras las herramientas que se están utilizando, pero no tiene más impacto en el diseño ni en la implementación de las bases de datos.

El modelo conceptual de una base de datos forma el nivel lógico en una arquitectura a tres niveles, tal como se ilustra en la Figura 2.3. El nivel lógico es la base de datos que implementamos los usuarios de los SGBDs. Este nivel materializa la independencia de las aplicaciones respecto al soporte físico. El modelo conceptual

debe ser capaz de absorber cambios en la forma física de almacenar los datos sin que trasciendan al nivel de aplicación.

El nivel de vistas, o de aplicación representa el compendio de utilidades que la base de datos proporciona a las aplicaciones de interfaz, que son las que en última instancia utilizará el usuario final.

## 2.3 Inmersión en el Dominio

Casi se podría decir que es imposible hacer una base de datos sin aprender alguna cosa. En los primeros instantes, cuando el proyecto empieza a concebirse, antes que nada, es importante acumular el máximo conocimiento de todo lo que pueda afectarlo.

Forma parte del dominio del problema toda aquella información externa que pueda condicionar el contenido de la base de datos. En particular, datos relativos a la legislación vigente sobre el tema. El diseñador de una base de datos, que en la fase inicial debe hacer de analista, debe ser una persona que tenga interés en todo lo que pueda afectar las restricciones que impone la base sobre los datos.

Llevado al extremo, si hablásemos de una base de datos para gestionar la liga de fútbol convendría que de alguna forma tuviera conocimiento del reglamento oficial. Así nos podría prohibir que en un partido jugasen doce jugadores, por ejemplo. Eso nos ayudaría a mantener la integridad de la base.

En el momento de realizar el modelo de dominio, quien diseña una base de datos debe estar dispuesto a aprender nueva terminología del entorno al que va a involucrarse. Cómo se utilizan las palabras, y qué conceptos van asociados a qué otros. Una curiosidad que se produce es que en ámbitos concretos, los adjetivos se utilizan como sustantivos. Hay que hacerlo. Por ejemplo, "central" en el entorno del fútbol significa una cosa muy diferente que en el entorno de la carpintería. A primera vista, desconfiaremos de un analista que le cuesten este tipo de asimilaciones.

Quien diseña una base de datos se debe preguntar qué depende de qué. Por ejemplo, el apellido de una persona no depende de la población donde vive. En cambio, la provincia donde vive sí que depende de la población. Dicho así, parece extraño... que la provincia depende de la población... Pero si tenemos en mente los diferentes valores de poblaciones y provincias que pueda haber, entonces está claro que siempre que la población sea por ejemplo *tena*, la provincia será *Napo*. Visto desde esta perspectiva, o sea, entendido como valores que pueden tomar unas variables, sí que tiene sentido decir que la provincia depende de la población. En cambio, al revés diríamos que la población viene restringida por la provincia. Estas relaciones de pertenencia son fundamentales en el diseño de bases de datos.

Un dato no depende de otro si cualquier combinación de valores posibles entre ellos tiene sentido. Dependencia significa que un valor condiciona el otro. Exactamente igual que entre proposiciones en el capítulo anterior.

Para llegar tan lejos como sea posible en el estudio del entorno, es muy conveniente comportarse en una actitud humilde en el diálogo con el profesional que después de todo será el usuario de la aplicación. Hay que observar como dice las palabras, y las preposiciones. Hacerle preguntas como si cualquier elemento de un conjunto puede estar relacionado con cualquiera de otro.

A veces lo tendrá clarísimo. Y otras veces no. Cuando le preguntemos si es necesario disponer de una información secundaria para poder existir una de primaria no sabrá qué responder y dirá que nunca se lo había planteado. A veces estas dudas se propagan hasta la interfaz de la aplicación y se acaban resolviendo con casillas de activación colocadas en pestañas de opciones de algún botón de propiedades o ajustes.

De todo ello se extrae un documento al que se acostumbra a llamar *definición de requerimientos*.

## 2.4 Definición de Requerimientos

Una definición de requerimientos es el objetivo principal que debe conseguir trenzar el analista que tiene el encargo de realizar una aplicación con una base de datos para gestionar un proyecto. Como es lógico, sin embargo, los profesionales del ámbito que corresponda no han escuchado hablar de ese documento, de manera que el analista antes de pedirlo, cosa que debe hacer incuestionablemente, debe explicar en qué consiste. Eso desemboca en un documento con conocimientos emitidos por el profesional usuario, o cliente, pero en cambio estructurado bajo las directrices del analista.

Todo ello provoca que el diálogo entre cliente y analista en los momentos iniciales sea intenso. El analista debe obtener el máximo provecho.

La estructura de una buena definición de requerimientos debería considerar los siguientes puntos.

- Cuál es la organización interesada en el desarrollo del proyecto. Cuál es su actividad. Cuántos años lleva haciendo esa tarea. También interesa mucho si la aplicación que se propone tiene alguna predecesora, y en caso negativo, como se realizaba el trabajo que ahora se pretende automatizar.
- Conceptos que debe manejar la base de datos. Cómo se identifican.
- Impacto del tiempo en los diferentes objetos. Distingir información histórica de

la funcional.

- Tipos de usuarios y intereses de cada uno de ellos.
- Aplicaciones que se le pretenden dar. Informes que se desea obtener. Acciones que se puedan realizar.

### 2.4.1 Información Estadística

A pesar de ser más complicado obtenerlas, las estadísticas nos permiten prever cuán a menudo se producirán casos que requieran informaciones adicionales, y son extremadamente importantes de cara al diseño.

Eso es, imaginemos que queremos guardar los datos de un conjunto de personas, y en el caso que la persona sea una mujer nos interesaría registrar si ha tenido hijos, y quiénes son, si también los tenemos guardados entre las personas de la base de datos. Entonces, la proporción de personas insertadas en la base que sean madres condiciona la estructura.

Si casi todas las personas son madres, podemos guardar la información juntamente con la que guardamos para las personas en general, dejando los datos correspondientes vacíos cuando la persona no sea una madre. Y como estamos suponiendo que la mayor parte lo serán, desperdiciaríamos poco espacio, y por tanto sería una solución admisible.

En cambio, si en la base de personas se insertan madres solo en casos excepcionales, entonces sería mejor guardarse la información relativa a ser madre en un lugar aparte, de manera que no desperdiciásemos espacio en las personas que no lo fuesen.

Está bien claro que la proporción de registros que requieran información adicional determina el lugar donde guardarla.

## 2.5 Trayectoria del Error

No es una buena cosa resolver confusiones que el usuario final pueda tener. Por ejemplo, imaginemos una base de datos de recetas de cocina. Las recetas de cocina tienen unos identificadores confusos. Hay más de una receta con el mismo nombre, como *macarrones a la boloñesa*. Entonces, no es misión del diseñador discernirlas. La actitud más adecuada que debería realizar una aplicación es la de retornar el error en la identificación al usuario. Una base de datos tiene mecanismos para distinguir cualquier tipo de objeto. No obstante, no es conveniente solucionar problemas tomando decisiones por iniciativa propia. Es decir, la base de datos debe reproducir las

confusiones del usuario final. Esto hace que este entienda mejor el sistema. Cualquier usuario entiende perfectamente que no puede llamar igual a cosas distintas. Hay que dejar que sea él quien decida la manera de identificar los objetos. Que sea él quien, a falta de más imaginación, acabe llamando *marcarrones a la boloñesa 1* y *macarrones a la boloñesa 2* a sus recetas.

En la mayoría de casos, y no solo en el entorno de bases de datos sino en cualquier tipo de aplicación, el tratamiento más seguro frente a un error es comunicarlo. Sin tomar ninguna decisión para solventarlo.

## 2.6 Ejemplo para un Club Deportivo

Como hilo conductor a lo largo de los próximos capítulos se propone desarrollar una pequeña aplicación para gestionar un club deportivo. La finalidad es más ilustrativa que funcional. Esto hace que muchos datos se hayan obviado.

Se trata de mantener la información sobre qué deportes hace cada socio, así como de contabilidad general para la empresa, o sea información de los trabajadores y de sus jefes.

### 2.6.1 Requerimientos del Club Deportivo

A un tipo de usuario le llamamos rol. De todos los usuarios se registra el número de pasaporte, nombres, apellidos, una dirección de correo electrónico, y su ciudad. Además, para cada persona podemos guardar una cantidad indeterminada de números de teléfono. Para informes estadísticos, y para poder hacer estudios de ampliación del club, conviene guardarse el número de habitantes de las ciudades de los socios, y su provincia.

En la Sección 7.3 se verá la manera de crearlos y darles permisos para poder hacer las acciones que se enumeran a continuación.

#### **Soci**

De cada socio se almacena su fecha de registro al club, que interesará en el momento de hacer descuentos para los socios más antiguos. Las acciones propias de los socios se enumeran seguidamente.

1. Consultar la información de los deportes, que integra:
  - Precio mensual que cuesta la práctica del deporte.
  - Cuota que pagan ellos en concreto si están inscritos, ya que el club puede hacer ofertas a determinados socios.
  - Cuantos jugadores faltan para formar un nuevo equipo, si es un deporte con varios participantes por equipo.
2. Ponerse en contacto con otros socios. Esto significa que el club mantendrá una relación de qué socios conocen a qué otros.
3. Inscribirse en cualquier deporte, o borrarse.

## **Trabajador**

En la empresa hay tres departamentos. Los trabajadores pueden ser comerciales, administrativos, o bien preparadores físicos que llamaremos entrenadores.

Los comerciales pueden consultar las direcciones de los correos electrónicos de los socios.

Los administrativos pueden registrar el pago de roles de los trabajadores. Y también pueden registrar o eliminar socios, y modificar sus datos.

Los entrenadores pueden ver la cantidad de socios que practican cada deporte.

## **Administrador**

El administrador puede registrar, eliminar y modificar todos los objetos de la base de datos. Eso contempla los socios, trabajadores, ciudades, provincias y deportes. Para los socios y trabajadores deberá crear un usuario con los permisos correspondientes para poder hacer lo que se requiere más arriba.

---

*En este capítulo se han visto cuestiones diversas relativas a como se debe plantear un proyecto para una aplicación de gestión de una base de datos. En particular, se han mostrado conceptos clásicos de la arquitectura de la aplicación. Se ha hecho una breve reflexión del impacto que puede tener la estadística en el modelo, y se ha presentado el concepto de definición de requerimientos. Finalmente, se ha introducido un proyecto sencillo que se utilizará en adelante como ejemplo en cada fase de la realización.*





## Capítulo 3

# Diseño de Bases de Datos

El objetivo final del proceso de diseño de una base de datos consiste en la elaboración de un diagrama, con la documentación asociada, que llamamos *modelo Entidad Relación*, modelo ER en adelante. Por la extensión que requiere exponer todo lo que involucra los modelos ER, el Capítulo 4 completo se dedica a detallar los conceptos.

Que quede claro. Hacer un diseño de una base de datos significa hacer un modelo ER, con su documentación.

Hacemos un inciso para hablar del término *diagrama*. Utilizamos esta palabra para indicar que tiene una estructura no secuencial. La palabra diagrama debe invocar a descripción en más de una dimensión. Empieza como *diagonal*, como si tuvieran alguna cosa en común. Diagrama significa una información plana, más que lineal. Se requieren dos dimensiones como mínimo para poder describir un diagrama. O sea, una representación que se desarrolla tanto en horizontal como en vertical, como un mapa.

No estaría de más que os paraseis a pensar en las banderas de los países del mundo. Clasificarlas en lineales y planas. Las banderas lineales nomás varían en una dirección. Es el caso de la bandera ecuatoriana, la catalana, la alemana o la francesa. En cambio, la bandera inglesa es una bandera plana, ya que se define en dos dimensiones. Como la australiana, la japonesa o la ikurriña vasca. Observa que podemos hacer banderas lineales tan largas como queramos, en cambio, con las planas no, cosa que tiene más trascendencia de lo que puede parecer, y que no queda tan solo en una simple anécdota.

Pero bien, volvamos al tema. Usamos diagramas para explicar el diseño de una base de datos y así no comprometernos con ningún orden entre los elementos que componen la idea que queremos expresar. Aparece así la noción de grafo. Por eso

convendría en este punto aclarar algunos términos de esta terminología.

Utilizamos la palabra *grafo* para indicar un conjunto de puntos, o *nodos*, que pueden relacionarse de alguna manera que indicamos mediante líneas que los unen. Sería bueno que a partir de esta indicación, os hicierais una idea de la abstracción que representa esta estructura mental.

Si las líneas que unen los nodos de un grafo tienen dirección, o sea son flechas, entonces se llaman *arcos*. Y si no, *aristas*. Podemos decir que los arcos o las aristas unen, conectan, asocian, vinculan, atan o enlazan sus dos nodos. El *grado* de un nodo en un grafo es el número de vecinos que tiene.

En la teoría de grafos, se distingue entre dos tipos. Grafos *dirigidos*, y grafos *no dirigidos*. Los dirigidos son los que las líneas que unen parejas de nodos son flechas, o sea los nodos son unidos por arcos. A veces, para diferenciarlos de los no dirigidos, a estos se les llama *dígrafos*. Así se indica que las conexiones tienen dirección. Los grafos dirigidos son los que nos interesan aquí, o sea los dígrafos, aunque los llamaremos grafos porque no induce a confusión. Todos los que utilizaremos serán dirigidos. Los que los arcos conectan un nodo predecesor con un nodo sucesor, es decir, los que la pareja de nodos que forman cada unión es una pareja ordenada y en la que diremos que un nodo *apunta* a otro. En la Figura 3.1 se muestra la idea de grafo que hay que retener.

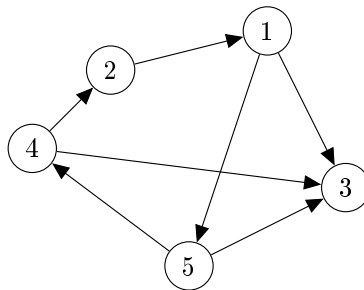


Figura 3.1: Grafo de 5 nodos y 7 arcos

A partir de estas definiciones no debería ser difícil imaginarse la definición de *camino en un grafo* como una secuencia de nodos tales que existen las aristas que los unen, consecutivamente. Por ejemplo, en el grafo de la Figura 3.1 un camino podría ser 1-5-3.

En los grafos dirigidos, como el de la Figura 3.1, está poco claro si el concepto de camino requiere que la secuencia de arcos tengan la misma dirección que la secuencia de nodos que visita. El solo hecho de hablar de grafos dirigidos hace que las definiciones se compliquen. Por ejemplo, en un grafo no dirigido podemos decir fácilmente si es *conexo* o no. Un grafo no dirigido conexo es aquel que para cualquier pareja de nodos hay algún camino que lleve de uno al otro. Bien, eso respecto a grafos no di-

rigidos... tiene gracia que definamos la conectividad de grafos cuando en la vida real, a un grafo no conexo, o sea por ejemplo con dos componentes conexas lo llamaríamos dos grafos.

En cambio, en el caso de grafos dirigidos hay que distinguir el concepto de conectividad en tres posibilidades. Un grafo dirigido puede ser *fuertemente* conexo, *unilateralmente* conexo, o *débilmente* conexo. Eso significa, en el primer caso que para cualquier pareja de nodos hay un camino para ir de un al otro y también del otro al uno. En los grafos unilateralmente conexos para cualquier pareja de nodos existe un camino que va de uno al otro, o bien del otro al uno. Y un grafo dirigido débilmente conexo es aquel que si ignoramos las direcciones de las flechas, es conexo.

En la Figura 3.2 hay ejemplos de las tres posibilidades. Observa como la conectividad más fuerte se produce en el caso (a), seguido del caso (b) donde podemos imaginarnos la noción de nodo *fuente* y nodo *pozo*, y el caso (c) donde queda bien claro que el nodo 1 es un nodo fuente, y los otros dos son nodos pozo.

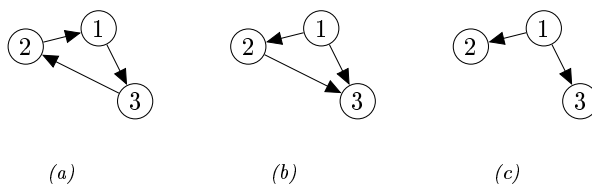


Figura 3.2: (a) *Fuertemente conexo*; (b) *Unilateralmente conexo*; (c) *Débilmente conexo*.

Y presionando una poco más, podemos adivinar la definición de *ciclo*, como camino que acaba en el mismo nodo que empieza. Y mira qué cosa, los ciclos empiezan y acaban en cualquiera de sus nodos. Debería verse muy claro que un ciclo en el grafo de la Figura 3.1 podría ser 5-4-2-1-5.

Un *árbol* es un grafo no dirigido, conexo y acíclico, o sea, que no tiene ningún ciclo. Este término, a pesar de estar definido en grafos no dirigidos se utiliza frecuentemente en dirigidos ignorando las direcciones de sus arcos. En la Figura 3.3, se muestra el aspecto que tienen.

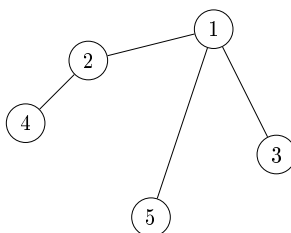


Figura 3.3: *Grafo acíclico conexo, o sea árbol*.

En la teoría de grafos, precisamente porque no los sabemos gestionar fácilmente, existen una enorme cantidad de definiciones. Piensa que ni siquiera tenemos ningún sistema de enumeración que nos permita asociar un grafo a un número. Piensa en eso e intenta inventarlo. Alguna forma de identificar los grafos en orden. Por ejemplo, podríamos decir que el grafo número 1 es el grafo formado por un nodo y ninguna arista. El 2, podría ser el de dos nodos aislados, es decir, también sin ninguna arista. El grafo número 3 podría ser el que tiene dos nodos y una arista que los une, pero entonces, el grafo con tres nodos aislados no está claro qué posición ocuparía en la secuencia...

Finalmente, para afrontar el diseño de bases de datos se requiere el concepto de nodo *terminal*. Es decir, el que tan solo tiene un vecino. Teniendo en cuenta que, como se ha dicho, estamos hablando de grafos dirigidos, los nodos terminales pueden ser fuentes o pozos, según si el único arco que los conecta al resto del grafo va hacia ellos, o sale de ellos. En los dos casos hablaremos de nodos terminales, aunque las funciones que ejercen son muy distintas.

Un vez se da por correcto un diseño, el modelo ER debe ser transformado o traducido al modelo relacional para empezar la implementación.

## 3.1 Modelos de Datos

Los modelos de datos se acostumbran a presentar como grafos. Cada nodo simboliza un concepto del universo de la aplicación. Un concepto que en definitiva unifica un grupo heterogéneo de datos relativos a un mismo tipo de objeto.

No estamos en un contexto demasiado riguroso. De modelos de datos hay de muchos tipos. Cualquiera se puede inventar el modelo que le resulte conveniente para lo que pretende expresar. Un diagrama de clases, en el entorno de la programación orientada a objetos, o el mismo modelo entidad relación en el que nos disponemos a sumergir, son diferentes expresiones de modelos de datos.

Los conceptos centrales que estructuran el dominio acostumbran a ser estables. Los terminales, en cambio, es más probable a contengan información histórica. Es importante comprender profundamente la diferencia entre datos que guardan tiempo, y datos que se mantienen constantes. Grandes bases de datos tienen como concepto central de referencia el reloj o el calendario. Es en esos escenarios donde interesa considerar los índices de actividad y volatilidad de cada objeto. El índice de actividad de una base de datos nos explica cuán frecuentemente es consultada. El índice de volatilidad, cuán frecuentemente nacen y mueren registros. Todo ello condiciona de manera directa las estrategias para el subministro de información que finalmente es el motivo por el que queremos la base de datos. Un valor que varía muy a menudo, mucho, no conviene calcularlo cuando se solicita, hay que tenerlo actualizado en todo

momento.

De hecho, tan importantes son los índices de actividad y volatilidad de cada una de las tablas de las bases de datos, que si los conociéramos antes de empezar a implementarla condicionarían directamente el diseño.

## 3.2 Dominio del Problema

En general, todos los modelos de datos tienen por finalidad ilustrar los conceptos básicos del dominio de la aplicación, así como las relaciones que puedan haber entre ellos mediante referencias que se puedan producir. De entre todos los diagramas que pueden incluirse en la especificación técnica de una aplicación el más abstracto es el *modelo de dominio*. Es un diagrama en el que aparecen las nociones esenciales que el proyecto debe gestionar. Es básicamente introductorio. Introduce el proyecto de manera vertical, como el índice de un libro, o el mapa de un territorio.

En la Figura 3.4 se muestra un ejemplo de modelo de dominio para el club de deportes.

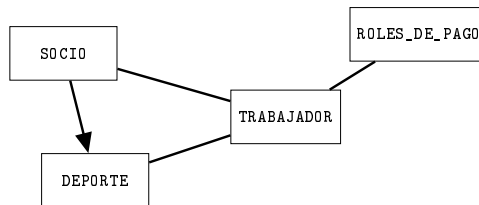


Figura 3.4: *Ejemplo de modelo de dominio para el club de deportes.*

Fíjate que nomás aparecen los conceptos fundamentales. El arco de SOCIO a DEPORTE podría representar que un socio está inscrito en un deporte. Estas cosas que no quedan claras se deben documentar con los diagramas. Es necesario acostumbrarse a pensar que un diagrama va siempre con una documentación asociada.

A pesar de que los modelos de dominio son unos diagramas utilizados en otras disciplinas, como la ingeniería de proyectos, encajan perfectamente en el desarrollo de bases de datos. Se trata del punto de inicio a partir del cual más adelante construiremos el diseño.

Un vez asumido un concepto del dominio, entonces utilizamos el verbo ser cuando enumeramos los atributos de los objetos. Decimos, por ejemplo, un socio *es* unos nombres, unos apellidos, una fecha de registro y una ciudad. Y por tanto, podremos

responder a preguntas como de qué ciudad es un socio, o bien qué deportes hace Slendy Rosales.

### 3.3 Máximas de Calidad de un Diseño

Es importante que un diseñador o administrador de bases de datos esté familiarizado con la terminología de estas máximas.

#### 3.3.1 Prohibir la Redundancia

Prohibir la redundancia significa que de ninguna manera se debe guardar una misma información en lugares diferentes, a no ser que sea por motivos rigurosamente estructurales.

La consecuencia de no respetar esa máxima es dar pie a la inconsistencia, y por tanto a la falta de integridad. Que una base de datos tenga errores de integridad es como decir que un barco es el titánic.

Y atención, no solo hay que evitar la redundancia en los datos, sino que también en las relaciones entre ellos. Esto es importante, y marca una diferencia entre un buen diseñador y otro no tan bueno. Lee con atención el párrafo siguiente.

Si un concepto del dominio, sea  $C_1$ , se relaciona con otro objeto del dominio,  $C_2$ , que está fragmentado en partes, o sea que contiene una colección de elementos de otro concepto del dominio,  $C_3$ , entonces a partir del momento que  $C_1$  se relacione con  $C_3$ , en caso que fuera necesario, quitaremos la relación entre  $C_1$  y  $C_2$  del diseño.

Por ejemplo, si  $C_1$  representa un estudiante,  $C_2$  representa un curso, o una asignatura, y  $C_3$  las clases, o sea las sesiones del curso  $C_2$ , entonces en principio podríamos pensar en relacionar un estudiante con un curso cuando se matricula. Pero a partir del momento que queramos tener alguna información del estudiante respecto a cada sesión, pongamos por caso querer saber si ha asistido o no, entonces ya no hay que crear la relación de cuando se matricula, ya que podemos actuar pensando que si un estudiante va a una clase de una asignatura es que está matriculado en ella. No es una ley en firme. Puede ser que nos interesen los estudiantes matriculados en un curso aunque no asistan a ninguna sesión. En cualquier caso, inclusive en el caso en que aceptemos que pueda estar relacionado con un curso, y además con cada una de sus clases, hay que ser conscientes que lo hacemos por motivos de corrección de posibles errores.

La única ventaja que nos proporciona la redundancia es el hecho de poder verificar

la coherencia que debe llevar asociada. En definitiva, tan solo es útil para el control de errores.

Reflexionando sobre el impacto que tiene esta máxima en el diseño de bases de datos, se concluye que siempre que se pueda se deben evitar los ciclos en el grafo del modelo. Y en cualquier caso, hay que analizar muy bien los ciclos en un modelo ER. En definitiva, los modelos ER en forma de árbol preservan la unicidad de los conceptos evitando las redundancias, cosa que los convierte en modelos fuertemente recomendables.

### 3.3.2 Prohibir Nulos Estructurales

No es correcto que sistemáticamente algún atributo de alguna entidad valga nulo. Por ejemplo, si nos guardásemos de cada persona qué deporte hace, y la cantidad de personas que no hacen ningún deporte fuese considerable, entonces utilizar un atributo dejándolo nulo cuando la persona no hace deporte sería incorrecto. Por eso es interesante, en el análisis, tener toda la información posible respecto a las cuestiones estadísticas.

### 3.3.3 Unificar Conceptos

Siempre que en una base de datos tengamos algún tipo de objeto que represente un conjunto de datos lógicamente relacionados será necesario referenciarlo cuando en alguna otra parte de la base de datos se utilice el concepto. Por ejemplo, imaginemos que en una base de datos tenemos una tabla con informaciones de carros, como puede ser el fabricante, el modelo, la cilindrada del motor, etcétera. Además también tenemos personas y cada persona puede tener un carro nomás. entonces, agregar un dato a la persona con el modelo de su carro, por ejemplo, es incorrecto. Hay que darse de cuenta que por el mismo precio (o sea el espacio que ocupa este dato) podemos tener guardado para cada persona un identificador del carro que tiene, y por tanto, no solo tener el modelo del carro, sino toda el resto de información.

Este error es grave. Y muy fácil de detectar porque en el modelo ER se puede ver que un atributo de una entidad se llama igual que otra entidad. Y eso es inadmisibile.

*En esta parte, de diseño, se ha hecho una introducción al que en el próximo capítulo se entrará en profundidad. Se han establecido criterios de calidad de los diseños, y en definitiva, se ha mencionado lo que formando parte del diseño de bases de datos no tiene un impacto directo en el modelo entidad relación.*



# Capítulo 4

## Modelo Entidad Relación

Los diagramas o modelos entidad relación son seguramente los modelos de datos más extendidos. Son grafos dirigidos en el sentido que se explica en la introducción del Capítulo 3. Las entidades son los nodos y las relaciones los arcos.

Los dos términos que le dan nombre son términos que deben hacer reflexionar.

Una entidad en el modelo ER representa un conjunto de elementos de los que nos interesa mantener una misma agrupación de datos. No es sencillo, estamos hablando de un conjunto de agrupaciones. Cada elemento de una entidad, pues, es una agrupación de datos heterogéneos en el sentido de que tienen diferentes semánticas, y en consecuencia pueden ser de diferentes tipos.

Es una definición muy parecida a la de las antiguas estructuras en lenguajes de programación de alto nivel como fortran, pascal, o c. Y una definición que se diferencia del actual concepto de clase en java por la carencia de métodos procedimentales. Representa lo mismo una clase de java sin ningún método que una antigua estructura en lenguaje fortran, por ejemplo, o que una entidad en el diseño de una base de datos.

Una relación es un predicado sobre una pareja de entidades, o más, tal como se ha descrito en la Sección 1.4.

O sea, la construcción de un modelo ER es como la de un grafo, primero son las entidades y después las relaciones.

Cuando un modelo se considera satisfactorio, es decir, se prevé cómo resolver cualquier demanda formulada en la definición de requerimientos, se pasa a la implementación, para lo cual debemos traducir el modelo ER a un modelo relacional,

procedimiento que se ve en detalle en la Sección 5.3.4.

Los destinatarios de los modelos ER somos los humanos. Los modelos ER los hacemos los humanos, y para los humanos. Tienen una finalidad que en última instancia es documental. El modelo ER debe formar parte de la documentación técnica de cualquier base de datos.

Por otra parte, por el hecho de que los destinatarios seamos los mismos humanos, hay partes de lo que se expresa en estos modelos que después no tienen traducción al modelo relacional, es decir a la implementación. En un diseño se pretende expresar las ideas sin preocuparse de la implementación.

Los diagramas ER tienen su propio lenguaje. De hecho, son un lenguaje. Por eso, disponen de un compendio de símbolos gráficos cada uno de los cuáles va asociado a una semántica para lo que modelan, y al mismo tiempo a una función que a veces es meramente descriptiva, y otras, estructural. Con esta colección de símbolos se establece entonces una gramática con la que se expresa la planificación de los espacios que finalmente alojarán los datos.

Los elementos que constituyen un modelo ER se presentan seguidamente.

## 4.1 Entidades

Las entidades representan el concepto más básico del diseño. Cada entidad tiene un nombre, que indica qué tipo de elementos la componen. Es decir, una entidad representa un conjunto de elementos, todos ellos caracterizados con distintos valores de unos mismos atributos.

Los elementos de una entidad comparten sus atributos, no los valores en cada uno de ellos. Hay que entender pues, que una entidad es una agrupación de datos que nos interesan, que van atados entre ellos, y no dependen de ninguna otra consideración.

Su representación gráfica es un rectángulo, como se muestra en la Figura 4.1.



Figura 4.1: *Representación de una entidad en un modelo ER.*

Todas las bases de datos parten de un diseño en el que hay una o más entidades.

Probablemente la entidad que aparece en más bases de datos es la entidad **PERSONA**.

En la mayor parte de las bases que nos rodean hay presente una entidad que guarda datos relativos a personas.

Así pues, un ejemplo de uso de la representación de la Figura 4.1 podría ser la que se muestra en el Modelo 4.1.



Modelo 4.1. *Ejemplo de una entidad en un modelo ER.*

Se entiende que el Modelo 4.1 representa un conjunto de personas.

Las pautas de estilo que se utilizan en este libro se encuentran resumidas en el Apéndice A. Por lo que respecta a los nombres o identificadores de entidades en los modelos ER utilizaremos sustantivos en singular. Es decir, una sola palabra, que escribiremos en mayúsculas. Excepcionalmente si necesitamos más palabras para designar una entidad, podemos usar guiones\_bajos. Hacer caso de estas directrices nos ayudará a interpretar los modelos. Al leer una entidad en un diagrama utilizamos el artículo indeterminado. El caso del Modelo 4.1 representa una persona, que puede ser cualquiera. Esta es la interpretación correcta.

## 4.2 Atributos

Cada uno de los datos que convenga agregar a una entidad será un atributo, que también podemos llamar *campo*.

Los atributos son las partículas elementales de las bases de datos, en el sentido que es la unidad mínima de transferencia, tanto de lectura como de escritura.

En principio, es razonable pensar que dos entidades con los mismos atributos deberían ser consideradas una misma entidad, aunque este es un aspecto que se verá más adelante. Una entidad es una agrupación lógica de atributos, entendiendo lógica como natural, dictada por el sentido común.

No es correcto decir que un atributo pertenece a una entidad. Es correcto decir que un atributo estructura una entidad, ya que los elementos de una entidad no son sus atributos. Una entidad es un conjunto de elementos que comparten unos mismos atributos.

Podemos suponer que si no fuese por los atributos una entidad no tendría razón

de ser. Más adelante nos encontraremos otros factores que conducirán a la existencia de entidades sin atributos, pero de momento, resulta útil entender que los atributos justifican la existencia de las entidades.

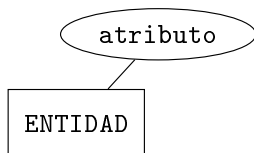


Figura 4.2: *Representación de un atributo en una entidad de un modelo ER.*

En el diagrama se expresan con una elipse que se une a la entidad que estructuran, tal como se puede ver a la Figura 4.2.

En general los nombres de los atributos serán sustantivos o, en el caso de atributos booleanos, a menudo nos encontraremos con adjetivos. Si queremos llamar un atributo con más de una palabra utilizaremos guiones bajos como con las entidades. Escribiremos los nombres de los atributos en minúsculas.

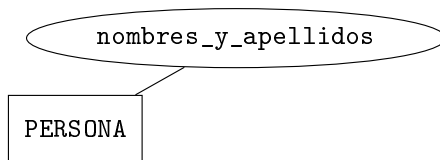
Un ejercicio frecuente en el diseño de bases de datos es encontrar un término que englobe un conjunto de posibilidades, o sea la abstracción. Frecuente y muchas veces difícil. Esto está relacionado con la capacidad de considerar el caso que tenemos delante como un caso particular de un conjunto más amplio. A una variable que pueda valer *moto*, *carro*, o *camión*, no hay que tener demasiada imaginación para llamarla *vehículo*. Pero en otros casos esto no es tan sencillo. Demasiado a menudo hay atributos que se llaman *tipo*, cosa mala, porque no aporta ningún contenido semántico y se presta a confusión. Siempre que un atributo se llame tipo entenderemos que se refiere al tipo de lo que sea el nombre de la entidad. Es decir, si una entidad se llama *E*, y tiene un atributo que se llama tipo, entonces mentalmente para leerlo diremos "tipo de *E*". Y aceptaremos ese nombre para el atributo si eso refleja su semántica en el dominio en que nos hallamos. En cualquier caso, los atributos llamados tipo hay que documentarlos por regla general.

A continuación, recuperando el ejemplo del Modelo 4.1, nos planteamos un dilema que nos ayudará a establecer la existencia de atributos juntos, o separados.

Si para el caso quisiéramos guardar los nombres y los apellidos de las personas, una aproximación inicial en el ejemplo del Modelo 4.1 sería añadiendo un atributo `nombres_y_apellidos`, como en el Modelo 4.2.

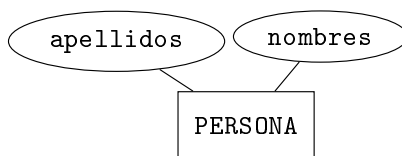
#### Modelo 4.2. *Ejemplo del uso de atributos en la entidad del Modelo 4.1.*

Eso bastaría para poder guardar los nombres y apellidos de las personas. Requer-



imiento satisfecho. No obstante, el Modelo 4.2 tiene graves limitaciones. Si se pide un listado de los que se llamen *García Pérez* de apellidos, costaría obtenerlo. En general, no lo tendríamos fácil para realizar ningún tratamiento que se basase en los apellidos. Ni siquiera obtener las personas ordenadas por apellidos. Por una simple razón. Los valores de los atributos son unidades de comparación, y por tanto, los tratamientos que necesiten exclusivamente una parte del valor ralentizarán la respuesta, perdiendo eficiencia.

Muy probablemente resulte mejor la solución de guardar los nombres y los apellidos como atributos independientes. Es el caso del Modelo 4.3.



Modelo 4.3. *Ejemplo más preciso del uso de atributos en la entidad del Modelo 4.1.*

La ventaja que tiene el Modelo 4.3 respecto al 4.2 es que resultará más sencillo obtener una lista de las personas ordenadas por apellidos, por ejemplo. En general, el Modelo 4.3 ofrece más posibilidades operativas a partir de los apellidos. No solo ordenarlos, sino cualquier otro tratamiento. Para la mayor parte de las aplicaciones, el Modelo 4.3 es mejor que el 4.2.

En conclusión, el Modelo 4.2 es más sencillo, pero no tan potente. Aún así puede ser útil, y muchas veces basta. Es más. Muchas bases de datos se guardan los datos fragmentados de maneras injustificables, como separar el bloque, la escala, el piso y la puerta de una dirección postal. Esto no se requiere, y complica el modelo. Como mucho, una dirección debe discernir entre la calle y el número. No porque nos interese el número (sería muy extraño una consulta a la base de datos que afectase a todas las personas que vivan en el número 30 de su calle). No es eso. Lo que efectivamente sí que nos puede interesar es el nombre de la calle, ya que puede ser que queramos enviar alguna cosa a todo el mundo que viva en una calle concreta, por ejemplo.

Por otra parte, para evaluar si vale la pena la introducción de un nuevo atributo, un criterio de calidad útil es: Si no somos capaces de encontrar valores de ejemplo de un atributo en menos de un segundo de tiempo significa que el atributo no tiene

sentido. Los atributos de una entidad son análogos a las variables de un lenguaje de programación. No respetar esa regla es un error que cometen muchos aprendices.

Por ejemplo, si tuviésemos una entidad que se llamase *VIVIENDA* que pudiera ser un *piso*, una *casa* o una *torre*, entonces en ningún caso deberíamos poner tres atributos que se llamaran *piso*, *casa*, y *torre*. Imagínatelo. ¿Qué valores podrían tomar? Si piensas en variables booleanas, mal, por la segunda máxima de la Sección 3.3, nulos estructurales. Como mucho, un atributo que se llamase *tipo*, y que pudiese tomar esos tres posibles valores. Efectivamente se referiría al tipo de vivienda, que es el nombre de la entidad. De todas maneras, si el valor de este atributo condicionara la existencia de otros datos adicionales, entonces deberíamos pensar en una especialización. Este concepto se trata en la Sección 4.14.1.

### 4.3 Atributos Clave

Todas las entidades deben tener un conjunto de atributos clave, aunque casi siempre con un solo atributo hay bastante.

Teniendo en cuenta que una entidad representa un conjunto de elementos, y que los elementos de un conjunto deben ser todos distintos, para cada entidad de un modelo ER deberemos garantizar que todos los elementos sean diferentes. Eso lo hacemos estableciendo una clave. Una clave de una entidad es un conjunto de atributos que por definición tendrá una combinación de valores diferentes en cada elemento de la entidad.

O sea, si la clave consta de un solo atributo, entonces los valores de ese atributo deben ser distintos para cada elemento de la entidad, es decir, no se pueden repetir. Y en cambio, si la clave consta de más de un atributo, entonces los valores de cualquiera de los que formen la clave se puede repetir sin ningún problema. Lo que no se puede repetir en este segundo caso es la combinación de valores de los atributos que forman la clave. La combinación.

Con esta definición aseguramos que cada elemento de una entidad sea diferente de los demás, y por tanto, que la entidad misma siga siendo un conjunto y no un multiconjunto cuando se produzcan nuevas inserciones. La clave sirve para identificar el elemento en la entidad. Esto marca la diferencia entre los atributos clave y los demás, no es el hecho que sean atributos con valores no repetidos, es el hecho que los usamos para identificar.

Los atributos que forman parte de la clave se representan subrayados en un modelo ER, Figura 4.3.

En adelante llamaremos atributos *estructurales* a los que sus valores vengán condi-

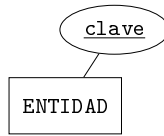


Figura 4.3: Representación de un atributo clave en una entidad de un modelo ER.

cionados por otros contenidos de la base de datos, como es el caso de los atributos clave, ya que no pueden tener un valor que ya exista en la misma entidad, y por tanto el valor de la clave viene condicionado por algún contenido de la base de datos. A los otros atributos los llamaremos *descriptivos*. Esta clasificación de los atributos en estructurales y descriptivos se ve en más detalle en la Sección 5.3.

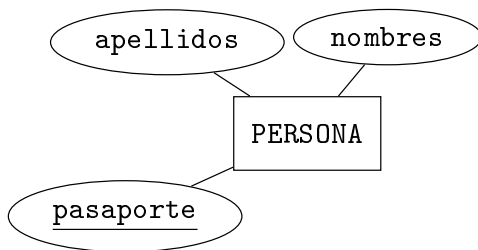
Bien, con el Modelo 4.3 de la página 55 tendríamos bastante para hacer una base de datos que guardase los nombres y los apellidos de un conjunto de personas. De todas maneras deberíamos tratar el problema de cómo distinguir colisiones, es decir, distintas personas que tuviesen los mismos nombres y apellidos.

Una posible solución sería establecer los nombres como atributo clave, o los apellidos, o la combinación de los dos valores. Pero está claro que si pusiéramos los nombres como clave, no admitiríamos personas con los mismos nombres, y con los apellidos o con la combinación de ambos tampoco admitiríamos las repeticiones correspondientes. Lo que debe hacer un diseñador de bases de datos es observar la realidad, y ser capaz de darse cuenta de qué criterio se utiliza en el mundo real para distinguir las personas de esa base de datos.

Ahora, como atributo clave, añadimos el número de pasaporte, ya que, por una parte, parece la manera más natural de asegurarnos que todos serán distintos y por otra, porque los usuarios de la base de datos considerarán natural esa manera de identificar las personas. Y por tanto, no se extrañarán cuando, si se diera el caso, no permitiésemos registrar un número de pasaporte que ya existe. Asumirán el error, e inntentarán arreglarlo como sea.

Una de las características que debe tener una buena base de datos es hacer comprender los errores que se puedan producir al usuario final. Y lo más deseable, es que esos usuarios asuman los errores como suyos.

Modelo 4.4. Representación de una entidad completa en un modelo ER.



## 4.4 Atributos Multivalorados

A veces interesa poder guardar más de un valor bajo un mismo concepto de atributo. En otras palabras, para cada elemento de una entidad, en lugar de tener tan solo un valor en cada atributo, para algún atributo concreto tenemos varios. Esta es la idea de lo que representa un atributo multivalorado. El símbolo que se utiliza en un modelo ER para expresar que un atributo es multivalorado es trazando la elipse en doble línea, tal como se ve en la Figura 4.4.

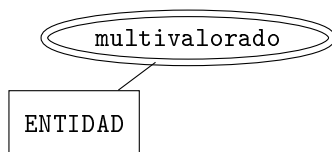


Figura 4.4: *Atributo multivalorado en un modelo ER.*

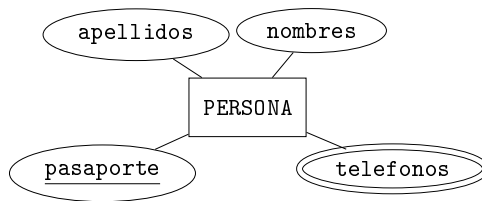
Bien, en el último párrafo de la introducción del Capítulo 3, precisamente cuando se definen los nodos terminales, se dice que según si son fuentes o pozos ejercen funciones muy distintas. Los atributos multivalorados son nodos terminales fuente en el grafo dirigido de un modelo ER. Así pues cuidado, aquí se presenta el caso en que son fuentes. O sea, la flecha sale de ellos, aunque al ser atributos, esa flecha no se dibuja. Nunca llevan flechas las líneas que unen atributos con sus entidades. De todas formas, para los atributos multivalorados, conviene imaginársela. De hecho, al interpretar que un atributo es un nodo del grafo ya estamos haciendo una excepción.

Para la aplicación del club, dicen los requerimientos de la Sección 2.6.1 que hay que admitir varios teléfonos por persona. Entonces, un modelo como el 4.5 podría valer.

Modelo 4.5. *Ejemplo de atributo multivalorado en el Modelo 4.4.*

Fíjate que no hacemos ninguna distinción entre los teléfonos asociados a una misma





persona. Lo que expresamos precisamente es que no queremos guardar ninguna información adicional con cada número de teléfono. Si se quisiera guardar la categoría, trabajo, casa, etc... por ejemplo, entonces no estaríamos hablando de un atributo multivalorado. De hecho, un atributo multivalorado se caracteriza por eso, porque no distingue entre los valores asociados a un mismo elemento de su entidad.

Un atributo multivalorado asocia colecciones a elementos de entidades, no les asocia secuencias. Estas colecciones de elementos acostumbran a ser disjuntas. El Modelo 4.5 significa que cada persona puede tener muchos teléfonos, pero por regla general un teléfono pertenece a una persona nomás. Esto es así porque con atributos multivalorados no tenemos ningún control de las repeticiones. Por tanto, si algún día un número compartido fuese modificado por las razones que fuere, al ser un atributo multivalorado se requeriría actualizar las apariciones de una en una. Si la cantidad de números de teléfono compartidos entre varias personas fuese bastante, por ejemplo de más del cincuenta por ciento, entonces moderarlo con un atributo multivalorado no sería lo correcto. Convendría hacerlo de alguna forma que el SGBD pudiese tener constancia de que un mismo número es compartido entre varias personas.

Para los nombres de los atributos multivalorados utilizaremos el plural, igual que hacemos con los atributos que son números enteros. La diferencia resultará evidente en cualquier caso por la doble elipse. Es fácil comprender que para los atributos enteros estamos diciendo cuántos, y para los multivalorados, cuáles.

## 4.5 Atributos Compuestos

En muchos casos puede interesar tratar los atributos de forma jerárquica. Esto significa poder disponer de los valores de un solo atributo concebido como un todo, aunque realmente esté formado de partes, de las que también nos interesa poder disponer. Los atributos compuestos son los que contienen atributos más pequeños.

Para entendernos, tratar o disponer de un atributo significa poder filtrar todos los elementos de la entidad según el valor de ese atributo, por ejemplo, o también poder ordenar según ese campo.

Los atributos más pequeños de todos son los atributos atómicos. Un atributo es

*atómico* si se corresponde con un tipo de datos primitivo, como son las cadenas de caracteres, enteros, booleanos, o números con decimales.

O sea, que de la misma manera que cuando hablamos de subconjunto nos imaginamos un subconjunto propio, cuando hablamos de un atributo nos imaginamos un atributo atómico. Los normales son los atómicos. Los nombres, los apellidos,...

En la Figura 4.5 se puede ver la forma que adopta un atributo compuesto en el diagrama.

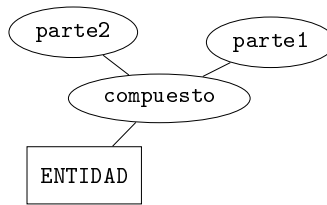


Figura 4.5: *Atributo compuesto en una entidad de un modelo ER.*

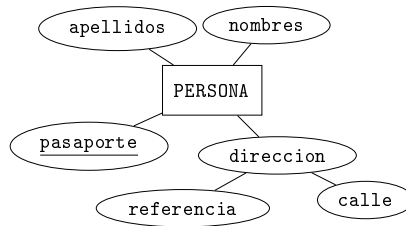
Anticipamos aquí que no tendremos forma alguna para expresar la idea de atributo compuesto en la implementación. Y a pesar de todo, sí que existe en el diseño. Quizás hay que recordar que como se ha dicho en la introducción de este capítulo, el modelo ER debe ser un documento hecho para y por los humanos, o sea que cuando diseñamos procuramos abstraernos de como se deberá implementar lo que estamos planificando, Aunque honestamente, cuando se dominan los conceptos de la implementación se produce una interacción incontrolable que los diseñadores difícilmente podemos evitar. Diseñamos pensando en la implementación... pero bueno, tampoco está claro que eso represente un inconveniente.

En cualquier caso, los atributos compuestos son un concepto. Y si hay que implementar un comportamiento que refleje fielmente esta idea siempre queda el recurso de incorporar procedimientos y vistas para poder ver la entidad según los diferentes niveles de los atributos compuestos que la integren.

Pongamos por caso que interesa poder tratar las direcciones de las personas, pero también se necesita hacer tratamientos según las calles, por cuestiones de distribución logística, o para saber en qué calle conviene poner una valla publicitaria...

El Modelo 4.6 responde a este requerimiento.

Modelo 4.6. *Ejemplo de atributo compuesto en el Modelo 4.4.*



## 4.6 Atributos Calculados

Los atributos calculados, o *derivados*, son atributos cuyos valores son autogestionados. No son datos que deba introducir el usuario, sino valores que el SGBD puede obtener a partir de otros valores presentes en su espacio de visibilidad. Normalmente sirven para mantener informaciones estadísticas.

Los atributos calculados tienen una importancia emergente. En la actualidad del año 2017 está aumentando espectacularmente la cantidad de recursos que se dedican a la minería de datos, en [9, 4] se habla de ello en profundidad. Claro, de alguna manera se puede entender que las bases de datos "saben" cosas que nosotros los humanos no sabemos.

Imagina una base de datos de una peluquería canina en la que resulta que todos los caniches no vuelven después de la primera visita. Esa es una información relevante para el dueño de la peluquería, y de hecho está contenida en la base de datos, pero en cambio no tenemos herramientas para hacerla aflorar.

U otro ejemplo. El ejemplo típico que se pone cuando se explica lo que es la minería de datos. Una cadena de supermercados norteamericana descubrió que los viernes se vendían cantidades excepcionales de cervezas y de pañales para bebés. Entonces pusieron de lado ambos productos y las ventas se dispararon. De todo ello se concluyó que los causantes eran padres separados que les tocaba el hijo o la hija el fin de semana.

Los atributos derivados son el origen de la minería de datos, es decir, de la explotación de los contenidos de las bases de datos.

Se representan con línea discontinua tal como se expone en la Figura 4.6.

En este momento no nos cuestionamos cómo se calcularán esos valores. Tan solo establecemos la intención de que alguien lo hará en algún momento. Estamos en la etapa de diseño.

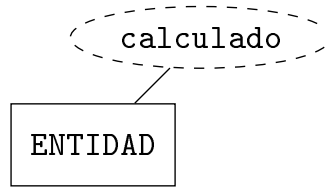


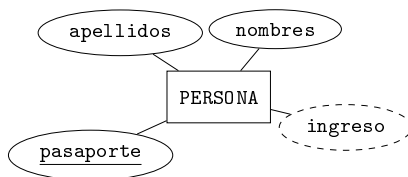
Figura 4.6: *Atributo calculado en una entidad de un modelo ER.*

Al definir un atributo como calculado se está haciendo una gestión muy meticulosa del tiempo. Por eso conviene tener en cuenta cuatro consideraciones que nos ayudarán en la decisión.

- La eficiencia del procedimiento de cálculo es fundamental. Cuando el valor de un atributo derivado se pueda calcular en  $O(1)$ , o sea que tarde como una asignación, una suma o una resta, entonces no vale la pena malgastar el espacio para guardarlo. Simplemente se puede calcular cada vez que se requiera. Esta tarea la puede hacer el mismo programa cliente, o se puede encapsular en el SGBD.
- Partimos pues de que la eficiencia del algoritmo es como mínimo  $O(n)$ , siendo  $n$  el número de elementos en la base de datos. Es decir, suponemos que para actualizar el valor de un atributo derivado hay que recorrer completamente los elementos de alguna entidad. Entonces, nuestra administración del tiempo es crítica. ¿Qué preferimos?
  - Que el SGBD tarde mucho cuando nos interese obtener el valor del atributo, o bien
  - Que cada vez que se modifique algún dato que impacte en el valor del atributo se tarde una poquito casi inapreciable más de tiempo en actualizar el valor derivado.

La decisión viene condicionada por la estadística. Si tenemos que consultar el valor muy de vez en cuando escogemos la primera opción, y por tanto no se necesitará ningún atributo derivado. En cambio si lo queremos tener disponible a menudo, entonces la segunda, que sí que requiere un atributo calculado.

- Se dice que una entidad de una base de datos tiene un alto índice de volatilidad si tiene muchos nuevos registros y eliminaciones de elementos. Otro caso en el que un atributo derivado queda totalmente justificado es cuando depende de valores que varían todo el tiempo. Conviene tener atributos derivados si los operandos para calcularlos residen en entidades de este tipo.
- Un atributo derivado está absolutamente justificado cuando sin el no podríamos obtener alguna información. Un caso muy frecuente es el de guardar las fechas de inserción de los elementos en las entidades.



Modelo 4.7. Ejemplo de atributo calculado en el Modelo 4.4.

En Modelo 4.7 se muestra el atributo calculado *ingreso*. Podrías pensar en llamarle *fecha\_ingreso*, y no, porqué por norma de estilo no ponemos los tipos de datos en los nombres de los atributos. Y en SQL, *fecha* es un tipo de datos. Sería como llamar al otro *texto\_nombres* o *numero\_pasaporte*.

Un ejemplo no tan típico sería, recuperando el Modelo 4.6 donde aparecía la dirección de las personas, añadir un campo entero calculado que contase los cambios de dirección que ha efectuado cada persona desde que ingresó en la base de datos. Fíjate que si no fuese por la existencia de un atributo calculado, no podríamos saberlo de ninguna manera.

## 4.7 Relaciones

Las relaciones en un modelo ER unen dos o más entidades.

Toda la estructuración de las bases de datos proviene de las relaciones entre entidades. Por eso son tan importantes.

El grado de una relación es el número de entidades que relaciona, cosa que coincide con la definición de grado de un nodo en un grafo. Llamamos relación *binaria* la de grado dos, o sea que relaciona dos entidades. Si relaciona tres, relación *ternaria*. Y si relaciona más de tres, aunque podríamos decir cuaternaria o lo que hiciera falta, más bien podríamos asegurar que la base de datos está mal diseñada. No es una ley en firme, pero sí que muy habitual. Es excepcional encontrar una relación con más de tres entidades en un modelo ER. Y cuando la hay, fácilmente es incorrecta.

A lo largo de esta sección suponemos que tenemos una relación de nombre  $R$ , que relaciona las entidades  $E_1$  y  $E_2$ .

### 4.7.1 Participaciones de las entidades en una relación

Cuando se establece una relación entre dos entidades es conveniente pensar bien en qué tipo de participación debe tener la entidad en la relación. La participación de una entidad en una relación puede ser parcial o total. Que una entidad  $E_1$  tenga *participación parcial* en una relación  $R$  significa que no todos los elementos de  $E_1$  están relacionados con algún elemento de  $E_2$  necesariamente. En cambio, *participación total* significa que todos los elementos de  $E_1$  deben estar relacionados con algún elemento de  $E_2$  forzosamente.

La participación de una entidad en una relación es una cuestión de tercer orden en el momento del diseño. De lo que se preocupa quien diseña es de establecer las relaciones como tema primordial. Después analizar cardinalidades, que veremos seguidamente. Y en tercera posición, por orden de prioridades, se analizan y se establecen las participaciones... si se hace. Eso conduce a que en los modelos ER se le dé muy poca importancia al tema de las participaciones, y en muchos casos no se expresa. se considera una cuestión menor. Y es una lástima, porque la información que aportan es muy útil.

La participación parcial de una entidad en una relación se expresa con línea simple en el diagrama. La total con línea doble. En la Figura 4.7 la línea simple de  $E_1$  a  $R$  significa que puede haber elementos de  $E_1$  que no estén relacionados con ningún elemento de  $E_2$ . En cambio, la línea doble de  $E_2$  a  $R$  significa que todos los elementos de  $E_2$  están relacionados con algún elemento de  $E_1$ .

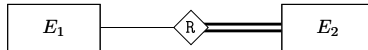


Figura 4.7: *Participación parcial de  $E_1$  y total de  $E_2$  en la relación  $R$ .*

Para entendernos, si  $E_1$  fuese PAÍS,  $E_2$  fuese PERSONA, y  $R$  fuese ES, el modelo de la Figura 4.7 permitiría añadir un país aunque no hubiera ninguna persona de ese país. En cambio, no se podría añadir una persona si no se supiera de qué país es.

La doble línea, que en un modelo ER se utiliza para indicar que un atributo es multivalorado, también se usa para indicar participación total de una entidad en un atributo. Aún así hay una ligera diferencia. En un atributo multivalorado la doble línea significa "muchos". En una participación total significa "todos".

### 4.7.2 Dependencias de existencia

Cuando una entidad tiene participación total en una relación se produce una *dependencia de existencia*. Es decir, la entidad no puede contener ningún elemento mientras

no exista el correspondiente elemento relacionado en la otra entidad de la relación. Eso significa que sólo en casos patológicos puede suceder que las dos entidades de una relación tengan participación total en la relación. Piénsalo. Parece imposible. Si tanto  $E_1$  como  $E_2$  tienen participación total en  $R$ , entonces no puede existir ningún elemento a  $E_1$ , porque para poder existir debería tener un elemento ligado con él por la relación. Pero como  $E_2$  tampoco podría tener ningún elemento mientras no existiese el correspondiente en  $E_1$ , nos encontramos ante un pez que se muerde la cola.

En los SGBDs hay mecanismos para poder romper este tipo de abrazos mortales, o death locks. Son mecanismos sencillos que se limitan a inhibir la verificación de restricciones hasta nueva orden. Es decir, inhibiríamos el control, rellenaríamos los datos, y restauraríamos la verificación de restricciones otra vez. De todas formas, eso son casos excepcionales que no suceden casi nunca.

Como consecuencia, acostumbramos a entender que cuando una entidad participa totalmente en una relación, la otra no.

Cuando una entidad  $E_2$  participa totalmente en una relación  $R$  significa que no aceptamos elementos a  $E_2$  si no están relacionados con al menos un elemento de  $E_1$ , y en este caso decimos que  $E_2$  tiene una dependencia de existencia de  $E_1$ . Los SGBDs también tienen mecanismos para decir que cuando se borre, o se cambie, un elemento de  $E_1$  se borren, o se actualicen, automáticamente todos los elementos de  $E_2$  relacionados con él.

### 4.7.3 Cardinalidad de una relación binaria

La *cardinalidad* de una relación binaria en un modelo ER se define como el número de elementos de una entidad que pueden estar asociados a un cierto elemento de la otra entidad de la relación. El término cardinalidad es nuevo. En la Sección 1.2 se vió el cardinal de un conjunto. Ahora hablamos de la cardinalidad de una relación. Son cosas distintas. La noción de cardinalidad refleja relaciones de pertenencia de elementos de una entidad a elementos de la otra.

Hay tres tipos de cardinalidades para relaciones binarias, 1:1, 1:N, y M:N, y se pronuncian simplemente con las palabras "uno uno", "uno ene", y "eme ene". Tan importante es la cardinalidad de una relación que diremos directamente relación 1:1, o 1:N, o M:N, en vez de decir relación de cardinalidad 1:1, o 1:N, o M:N.

Todas aquellas palabras que en la transformación del modelo ER al modelo relacional vayan a desaparecer nos las ahorraremos, tal como se indica en las normas de estilo del Apéndice A. Eso es, la mayoría de los nombres de las relaciones 1:1 o 1:N desaparecerán. La trascendencia de los nombres de las relaciones en un modelo ER depende de su cardinalidad.

En las tres próximas secciones se explica con más detalle cada una de las tres cardinalidades.

## 4.8 Relaciones 1:1

Las relaciones binarias con cardinalidad 1:1 permiten cada elemento de  $E_1$  asociarse con un único elemento de  $E_2$ , o ninguno si la participación es parcial, y cada elemento de  $E_2$  asociarse con un solo elemento de  $E_1$ , o ninguno.

Tal y como omo se muestra en la Figura 4.8, las flechas siempre salen de la relación hacia fuera, y significan "solo uno, o ninguno".

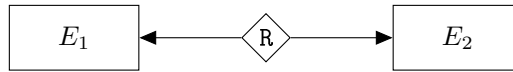


Figura 4.8: *Relación 1:1 en un modelo ER.*

Se puede evitar dibujar el rombo, y el nombre de la relación. Entonces, según  $E_1$  la relación se llamaría " $E_2$ ", y viceversa. Por tanto, no se introduciría ninguna ambigüedad.

No obstante, las relaciones 1:1 son excepcionales. Su rareza es fruto de que casi siempre son supérfluas y pueden suprimirse del diagrama fusionando los atributos de las dos entidades en una sólo que puede llamarse con cualquiera de los nombres de las dos iniciales.

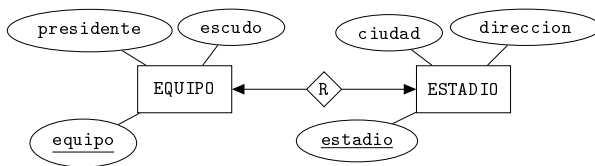
La justificación de que en un modelo ER aparezca una relación 1:1 pasa por la estadística. Tiene sentido fragmentar la información que lógicamente debería ir agregada por uno de los dos motivos siguientes:

- Cuando una parte de los dos fragmentos tenga un índice de actividad muy distinto al de la otra parte. Es decir, cuando se tengan que producir muchas transferencias de datos de solo una de las dos entidades.
- Cuando las dos entidades son grandes, es decir tienen muchos atributos, y la asociación no es indispensable.

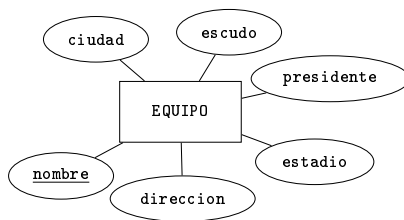
En el Modelo 4.8 hay un ejemplo de relación entre equipos de fútbol y estadios.

Modelo 4.8. *Ejemplo de relación 1:1 en un modelo ER.*





Como se ha mencionado, ante de un modelo como el 4.8, lo que debemos hacer como diseñadores, es aplicarle la simplificación que se muestra en el Modelo 4.9.



Modelo 4.9. *Simplificación del Modelo 4.8.*

Observa que fusionando las dos entidades estamos relajando una restricción del Modelo 4.8, que en el Modelo 4.9 ha desaparecido. Se trata del atributo clave de la entidad ESTADIO. Los nombres de los estadios en el Modelo 4.8 no pueden repetirse, porque son clave. Y en cambio en el modelo simplificado 4.9, sí. Eso no nos debe preocupar, ya que podremos imponer unicidad al estadio.

## 4.9 Relaciones 1:N

Las relaciones 1:N son el reflejo gráfico más claro de lo que significa una referencia en una base de datos. Este tipo de cardinalidad acostumbra a representar una relación de pertenencia. Se diría que hay varios elementos de  $E_2$  que pertenecen a un único elemento de  $E_1$ . Por tanto, la entidad del lado 1 siempre tendrá participación parcial en la relación.

Las relaciones 1:N permiten a cada elemento de  $E_1$  asociarse con varios elementos de  $E_2$ , pero para cada elemento de  $E_2$  se permite asociar un único elemento de  $E_1$  o ninguno. Como en el caso 1:1, a veces el 1 de la relación puede ser cero, cosa que podemos impedir con la participación total del costado N, doble trazo en el diagrama. Por otro lado, ña participación de  $E_1$  siempre será parcial en la relación, es decir, siempre puede haber elementos de  $E_1$  no apuntados por  $E_2$ .

En un modelo ER, las relaciones 1:N se representan como en la Figura 4.9.

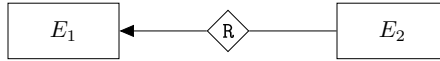


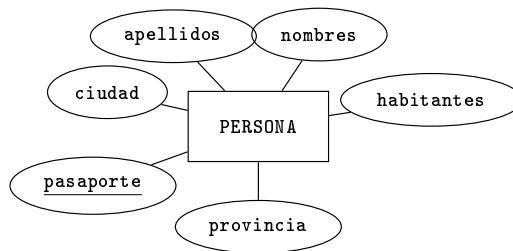
Figura 4.9: *Relación 1:N en un modelo ER.*

Que quede claro, la flecha apunta hacia la que solo puede ser uno.

No distinguimos ningún orden en las entidades relacionadas. Es decir, no definimos las relaciones N:1, porque como se dice en la introducción de este capítulo, expresar un modelo en un diagrama sirve para no comprometerse con ningún orden entre las entidades. Los nodos de un grafo no tienen ninguna posición relativa preestablecida, y por tanto, si nos encontramos con una relación N:1, girando la hoja de papel ya la tenemos 1:N.

Tanto el rombo como el nombre de la relación se puede suprimir sin ningún problema, siempre que solo haya una relación 1:N entre esta pareja de entidades. Al no dibujar ni el símbolo ni el nombre de la relación se entiende que el nombre de la relación es " $E_1$ ", que es la entidad apuntada, o sea la del costado 1 de la cardinalidad.

Ahora debería venir aquí un ejemplo de relación 1:N. Aprovechamos la ocasión para ilustrar una forma en la que se procede para decidir crear nuevas entidades. Para observar que el establecimiento de una nueva entidad no siempre es fruto del análisis, sino que a veces es automática. Hay que imaginar que nos equivocamos al asignar atributos a una entidad, y concluiremos en la creación de la nueva. Para hacerlo, recuperamos el ejemplo del Modelo 4.4 de la página 57 donde había solamente personas con nombres, apellidos y el campo clave con el pasaporte. Suponemos que queremos guardar más atributos para cada persona. Añadimos su ciudad, el número de habitantes, y la provincia. Nos quedaría un diseño parecido al del Modelo 4.10.



Modelo 4.10. *Entidad con atributos incorrectos en un modelo ER.*

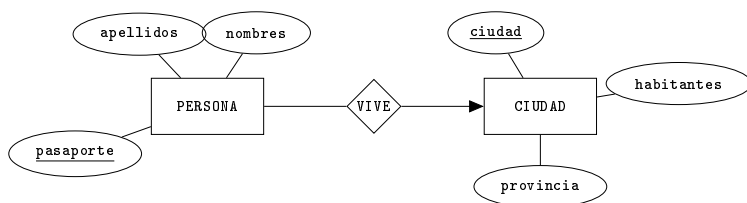
El plural del atributo **habitantes** es para indicar que es una cantidad. Un número entero. El nombre del atributo podría ser **numero\_habitantes** pero por cuestiones de estilo no ponemos el nombre de los tipos de los atributos, por tanto no ponemos *numero*.

Fíjate que añadiendo los atributos tal como se indica en el Modelo 4.10, entonces para cada persona de *tena* que se inserte se repetiría toda la información que solo depende de la ciudad. O sea *provincia* valdría *Napo*, y *habitantes*, 23307. Esto contradice la primera máxima de la Sección 3.3, ya que estaríamos introduciendo redundancia.

Así pues, el Modelo 4.10 es incorrecto. Y la explicación del por qué, es que nos encontramos frente de lo que se define como una *dependencia funcional*. Es decir, que en una entidad el valor de un atributo dependa del valor de otro atributo de la misma entidad. Es un error grave de diseño, que hay que evitar a toda costa.

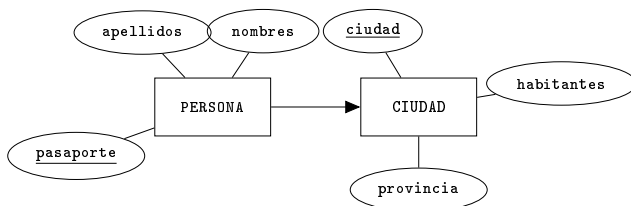
Precisamente en este punto se pone en práctica lo que se decía en la introducción del Capítulo 2. Qué depende de qué.

La forma correcta de proceder es agregando los atributos correlacionados en una nueva entidad. Eso, en el caso del ejemplo, nos conduce a la propuesta ilustrada en el diagrama del Modelo 4.11 en el que se ha añadido la entidad CIUDAD con los atributos que lógicamente dependen de ella, *ciudad*, *habitantes* y *provincia*. La solución del Modelo 4.11 corrige el error mostrado en el Modelo 4.10.



Modelo 4.11. *Modelo correcto para el ejemplo del Modelo 4.10.*

Y tal como se ha indicado más arriba, en este caso se puede suprimir el símbolo y el nombre de la relación. Entonces, en lugar de llamarla *VIVE* la estaríamos llamando *CIUDAD*, asumiendo siempre que la ciudad de una persona significa donde vive, extremo que debería quedar documentado. El nuevo diseño es el del Modelo 4.12.



Modelo 4.12. *Modelo correcto simplificado para el ejemplo del Modelo 4.10.*

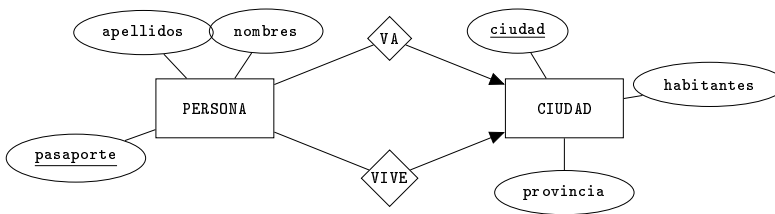
Conviene estas simplificaciones porque nos recuerdan que una relación 1:N vista desde el lado N es como un meta atributo, cosa que encaja en la filosofía.

Pero cuidado. Esta simplificación es posible porque no introduce ambigüedad. Esto es gracias a que entre las relaciones PERSONA y CIUDAD hay una relación nomás. Si hubiera dos relaciones 1:N entre el mismo par de entidades, entonces sería necesario un nombre distinto para cada una.

Como es importante, vamos a analizar un nuevo caso. Supongamos por ejemplo que en la misma base de datos de las figuras anteriores surgiera la necesidad de guardar adicionalmente la ciudad donde va a trabajar cada persona.

Fíjate pues que entre personas y ciudades tendremos dos relaciones, una para saber donde vive y la otra, donde trabaja. Ergo, no podremos eliminar los rombos con sus identificadores del diseño.

En el Modelo 4.13 se ha añadido la nueva relación VA para cumplir con ese nuevo requerimiento.



Modelo 4.13. Ejemplo del Modelo 4.11 con una relación nueva.

### 4.9.1 Entidades de soporte a la interfaz

La definición de una entidad como *de soporte a la interfaz* está relacionada con la entrada y salida de datos, es decir con la interfaz y por tanto con el programa cliente. Por eso no forma parte esencial del diseño de bases de datos. De todas formas se introducen aquí porque son un caso muy habitual de relaciones 1:N.

Las entidades de soporte son fruto de atributos descriptivos, no estructurales, con los que queremos mantener un control de posibles errores en el formato de los valores. En otras palabras, son nuevas entidades nacidas a partir del deseo de uniformar los valores de un atributo descriptivo. Por eso es correcto decir que el atributo *se segrega* en una entidad de soporte.

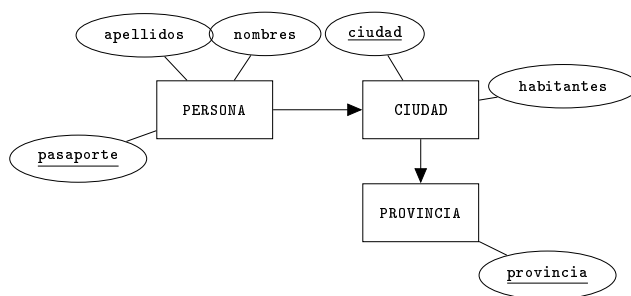
Estas entidades satisfacen dos condiciones.

- Tan solo tienen un campo clave. No tienen atributos descriptivos.
- Son nodos pozo en el modelo ER, es decir, nodos terminales que la flecha apunta hacia ellos.

Al principio del Capítulo 3, al final de la introducción, se definen los nodos terminales en un grafo dirigido como es un modelo ER. Y también se dice que los nodos del grafo son las entidades del modelo. Se ve la diferencia entre nodos terminales fuente, y pozo y también se anticipa que tienen funciones distintas. Después, en la Sección 4.4 se ve como los atributos multivalorados, aun siendo atributos y no entidades, son fuentes en un modelo ER. Y ahora aquí, se cierra la explicación con las entidades de soporte haciendo de nodos terminales pozo.

Volvamos al ejemplo del Modelo 4.12, de personas que viven en ciudades. Como el nombre de la ciudades es clave, confiamos que no se producirán errores de deletreo. Ahora bien, la provincia no es clave, y es probable que el usuario cometa errores, como poner "Chimboraso", con con "s", en lugar de "Chimborazo". Esos errores ortográficos o de deletreo en la introducción de datos son muy peligrosos porque no hay ningún nivel en la arquitectura de la aplicación que se responsabilice de ellos. Es decir, nadie nos avisará nunca de que existe un error de ese tipo. Eso provocará que el día que se haga una consulta para saber cuantas personas hay de la provincia del *Chimborazo* se responderá una cantidad equivocada. Y eso tampoco lo detectará nadie. En definitiva, hay que tomar medidas para evitar ese tipo de errores. Ante casos así, las entidades de soporte no es que los resuelvan, pero sí que sirven para tener un mayor control, y es muy habitual su uso.

En el Modelo 4.14 se puede ver el ejemplo modificado habiendo segregado la entidad de soporte **PROVINCIA**.



Modelo 4.14. *Entidad de soporte* **PROVINCIA**.

Con el diseño del Modelo 4.14 se tiene un mayor control porque existiendo esa entidad, es fácil imaginar que la introducción de los valores de provincias se hará vía menú desplegable. De manera que para el usuario no solo le resulte más cómodo

seleccionar el valor cuando ya existe, sinó que además, se lo tenga que pensar dos veces al registrar un nuevo nombre de provincia, cuando tenga el menú frente a él y cuando vea que no existe la provincia que quiere ingresar.

Esta es una buena práctica y se recomienda usarla. De todas formas, si eso nos interfiere el modelo con un exceso de entidades, puede quedar documentado de manera textual, tan solo informando que tal atributo se segregará en una entidad de soporte.

Es normal que aparte de las tablas que se infieren a partir de un modelo ER las bases de datos utilicen muchas otras tablas independientes del diseño para otras tantas utilidades. Por ejemplo, para guardar opciones de configuración del programa cliente puede haber una tabla cuyos registros guarden parejas en forma `clave = valor`. Además, cuando las opciones son personalizables, entonces es común guardar también una tabla adicional de usuarios. En fin, en este aspecto no nos debemos preocupar. No hay ningún problema en crear tablas auxiliares para hacer cálculos y para la gestión de la aplicación.

#### 4.9.2 Entidades débiles

Para introducir el concepto de entidad débil hay que recuperar el espíritu analítico y la vocación de hacer de espejo de la realidad. Una entidad débil es rigurosamente fruto de la forma de identificar los elementos de las entidades en la vida real.

Esta es precisamente la característica más importante que tienen las entidades débiles. Para identificar sus elementos hay que identificar previamente al elemento relacionado. Esto se hace mediante una relación que recibe el adjetivo de relación *identificadora*. Y la entidad relacionada vía relación identificadora se llama entidad *fuerte*.

Así pues, una entidad débil debe tener forzosamente una única entidad fuerte, que le proporciona parte de la identificación de sus elementos mediante la relación identificadora. Sólo una parte. No obstante, al ser una entidad, debe poderse identificar sus elementos completamente, a diferencia de los atributos multivalorados que por eso mismo no son entidades.

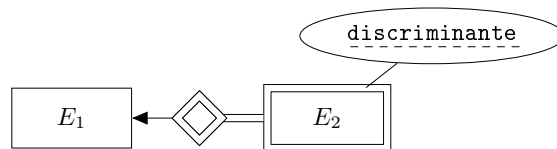


Figura 4.10: Entidad débil  $E_2$  identificada vía  $E_1$ .

Por todo ello añadimos un atributo que se llama *discriminante* a la entidad débil,

para distinguir entre los elementos relacionados con un mismo elemento de la entidad fuerte. En un modelo ER, el discriminante se subraya en línea discontinua, cosa que tiene su razón de ser, ya que un discriminante se puede entender como media clave.

El hecho de ser entidades, habilita las entidades débiles para poder relacionar sus elementos con otras entidades aparte de la identificadora, capacidad que no tienen los elementos de un atributo multivalorado.

Una entidad débil es una extensión de un atributo multivalorado. Le añade una distinción entre los valores. Por eso el símbolo gráfico utilizado para representar una entidad débil en un modelo ER es el mismo que para los atributos multivalorados, como se muestra en la Figura 4.10.

Fíjate que la relación identificadora de una entidad débil viene indicada por el doble rombo que forma parte del mismo símbolo que expresa que una entidad es débil. Y no requiere nombre. Es importante que siempre que se dibuje un rectángulo en doble línea, una entidad débil, se dibuje automáticamente la doble línea que indica participación total, y el doble rombo que indica la entidad identificadora. Estas tres partes forman un solo símbolo. No tiene sentido el doble rombo sin el doble rectángulo, ni el doble rectángulo sin el doble rombo. Y precisamente por eso, porque una entidad débil siempre debe tener una entidad fuerte identificadora, mejor recordar toda la estructura de doble línea como un símbolo monolítico.

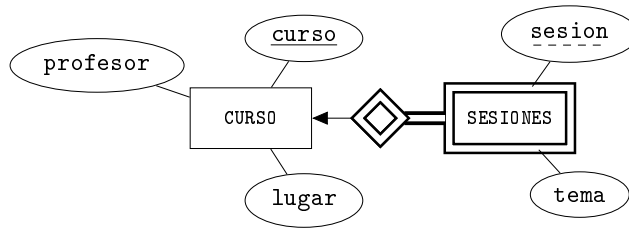
Al decidir si en un modelo introducimos una entidad débil hay una regla de oro. Y hay que serle fieles. Si para identificar los elementos de la entidad en la vida real utilizamos la identificación de alguna otra entidad, entonces conviene introducirla en el diseño como entidad débil. Para responder esta cuestión no hay que ser ni analista ni diseñador ni nada. Simple observación.

Que quede claro. Utilizaremos una entidad débil para disponer de las partes de un todo siempre que para identificar cada una de esas partes se requiera la identificación del todo que las contiene. Esa es una regla de diseño que distingue un buen diseño de otro que no lo es.

El ejemplo del Modelo 4.15 es de un diseño con una entidad que representa cursos, o asignaturas, formados por un conjunto de sesiones, o clases, que se identifican enumerándolas del 1 en adelante.

Modelo 4.15. *Ejemplo de uso de una entidad débil.*

También se podrían identificar con el día, si en ningún caso se pudieran dar dos clases el mismo día. Observándolo con espíritu crítico hay que preguntarse si realmente para poder hablar de una sesión de un curso se requiere necesariamente decir



de qué curso se trata. La respuesta parece ser que efectivamente sí. Y por tanto el modelo parece ser correcto. El atributo discriminante *sesion* distingue las sesiones de un mismo curso. Aunque cueste recordarlo, en la entidad *SESIONES* hay mezcladas sin ningún orden todas las sesiones de todos los cursos. Una entidad débil es un conjunto de conjuntos de elementos, como los atributos multivalorados.

Y por eso, al identificar un elemento de una entidad débil, en primer lugar hay que identificar el conjunto, es decir el elemento de la identificadora que lo contiene.

Otros ejemplos son los pasajes para un vuelo de avión. La entidad fuerte sería *VUELO*, y la débil *PASAJES*. Efectivamente, no se puede identificar un pasaje de avión sin decir de qué vuelo se trata. O bien el típico ejemplo en el que la entidad fuerte es *FACTURA*, y la débil *LINEAS*, entendiendo que una línea de una factura no puede ser identificada si no sabemos de qué factura hablamos.

Llamaremos a las entidades débiles con sustantivos en plural. Y por descontado, la participación de una entidad débil en la relación identificadora es total. Es más, entre una entidad débil y la correspondiente identificadora hay una dependencia de existencia. Sólo faltaría! Si no fuera necesario que un elemento de la entidad débil estuviera relacionado con alguno de la identificadora, entonces no lo podríamos identificar. Y en una base de datos, los elementos de todas las entidades deben poder ser identificados.

### 4.9.3 Clasificación de las Relaciones 1:N

Cerramos el tema de las relaciones 1:N con una sinopsis que nos ayudará a comprender los distintos niveles de compromiso que puede haber entre las dos entidades relacionadas en una relación de cardinalidad 1:N.

Recordemos que el hecho de que una relación 1:N sea con dependencia de existencia es una decisión del diseño. Depende de si nos interesa mantener los elementos del lado N cuando se borra el elemento asociado al lado 1. O, visto de desde otro ángulo, depende de si aceptamos elementos en la entidad del lado N sin tener información del elemento que le corresponde en el lado 1, o no. En definitiva, es el mismo concepto el de dependencia de existencia que el de participación total.



Llevado al ejemplo del modelo de personas que viven en ciudades. Observa que según qué uso se le tenga que dar a la base de datos puede interesar que cuando se borre una ciudad, las personas que viven en ella sigan presentes (sin tener información de donde viven), o no. Es decir, que **PERSONA** participe total o parcialmente en la relación **CIUDAD**. La participación total significa dependencia de existencia. O sea, que nos interesa tanto saber de donde es una persona, que si se borra la ciudad donde vive, también eliminamos la persona. O, en otras palabras, si no sabemos donde vive, prohibimos el registro de la nueva persona.

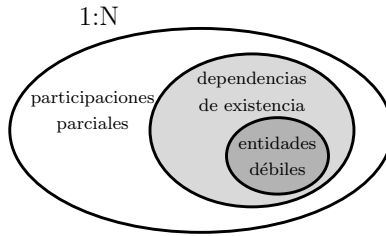


Figura 4.11: *Relación jerárquica entre relaciones 1:N.*

Un vez introducidas las diferencias entre los niveles de compromiso entre las relaciones 1:N se debe comprender la Figura 4.11, donde se muestra una relación jerárquica entre las distintas categorías de las relaciones 1:N.

De la imagen de la Figura 4.11 se desprenden estas conclusiones.

- Una entidad débil depende existencialmente de su entidad fuerte, ya que si no existe quien pueda identificar un elemento de una entidad, no puede existir ese elemento.
- Una relación identificadora es una relación 1:N.

Cuando la entidad del lado N participa parcialmente en una relación, puede originar nulos estructurales.

## 4.10 Relaciones M:N

Las relaciones binarias con cardinalidad M:N permiten a cada elemento de  $E_1$  asociarse con muchos elementos de  $E_2$ , y a cada elemento de  $E_2$  asociarse con muchos de  $E_1$ .

Como analistas, la primera cuestión que nos viene en mente es por qué no hacemos todas las relaciones M:N ya que este tipo de relaciones engloban las otras. Sin

embargo, eso supondría una complicación innecesaria tanto del diseño como de la implementación.

Dicho eso, recuperamos con gran interés el producto cartesiano explicado en la Sección 1.3.4. Porqué es de lo que estamos hablando. Quizás más que bases de datos relacionales deberíamos llamarlas bases de datos cartesianas. Podemos entender una relación M:N entre dos entidades  $E_1$  y  $E_2$  como una matriz con los elementos de  $E_1$  uno por fila, y los de  $E_2$  uno por columna. El contenido de la matriz entonces serían booleanos, que nos dirían si el elemento de  $E_1$  de la fila está relacionado con el elemento de  $E_2$  correspondiente a la columna.

Estas relaciones se representan como se muestra en la Figura 4.12.

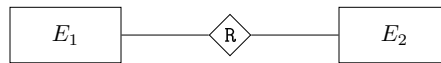


Figura 4.12: *Relación M:N en un modelo ER.*

Jamás podremos prescindir del nombre de una relación M:N. Es más, una relación con esta cardinalidad deberá ser considerada como un nuevo nodo del grafo que es el modelo. La razón es filosófica. Así como para las relaciones de cardinalidad inferior no se produce un efecto multiplicativo, ni por tanto de aumento del espacio necesario para guardar los valores relacionados, aquí eso sí que efectivamente sucede.

Si  $E_1$  tiene  $n = |E_1|$  elementos y  $E_2$  tiene  $m = |E_2|$ , entonces el número de elementos que puede haber relacionados mediante una relación M:N es, lógicamente,  $|E_1 \times E_2| = nm$ . Esta es la cantidad de parejas de elementos que pueden estar relacionados. Y normalmente es superior a  $n$  y a  $m$ , y por tanto, para guardar las parejas de elementos relacionados requeriremos espacio adicional. Y eso lo conseguimos considerando la relación como un nuevo nodo del grafo, cosa que hace que se parezcan a las entidades.

Además, estas relaciones pueden tener atributos. O sea que todavía se parecen más a las entidades. Y si hay que entender una relación así como un nuevo nodo en el grafo y además puede tener atributos, la cuestión que nos debería interesar es qué diferencia hay entre una relación M:N y una entidad.

La respuesta es la existencia de clave primaria. La diferencia entre una relación M:N y una entidad es que una relación M:N no tiene clave por naturaleza. Es decir, el usuario final no nos podrá decir como identifica los elementos de una relación, porque no lo hace. No los identifica. Es como preguntarse como se llama la amistad que hay entre dos amigos.

Posterior al diseño, en la implementación, a menudo se define como clave de una relación M:N la pareja de claves de los elementos relacionados. No obstante, la finalidad con la que se establece esta definición más que para identificar cada elemento de la

relación es para significar que no se admitirán repetidos, simplemente. En cualquier caso, en un diseño no se admite establecer identificación en las relaciones. Es más, en el momento que ponemos un campo clave efectivamente identificador en una relación la estamos convirtiendo en entidad. Ese es un extremo en el que se demuestra el ingenio del diseñador. Encontrar palabras para transformar en entidades las relaciones M:N requiere cierta habilidad lexicográfica.

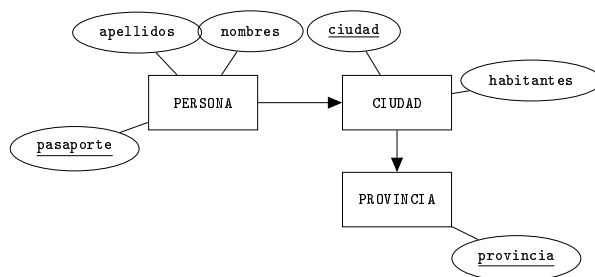
Cuando nos preguntemos si una relación es M:N, hay que decir dos frases, en voz alta mejor,

*Un  $E_1$  puede relacionarse con muchos  $E_2$ ...*  
*...y un  $E_2$  puede relacionarse con muchos  $E_1$ .*

Aunque parece fácil, seguro que tardaremos más de lo que parece en formularlas. Porque para decir las, hay que concentrarse dos veces. Se está expresando una dualidad, y eso cuesta trabajo mental.

Visto desde el foco del análisis matemático, una relación binaria M:N representa una función booleana de dos variables independientes.  $f = f(x, y)$ , siendo  $f \in \{0, 1\}$ , y definida sobre el conjunto que resulta del producto cartesiano entre las entidades, es decir sobre cualquier pareja tal que  $x \in E_1$  e  $y \in E_2$ . La respuesta puede ser uno o cero, significando que los elementos  $x$  e  $y$  están relacionados, o no.

Tomaremos el diseño del Modelo 4.16 como punto de partida para la aplicación del club deportivo. En la Sección 2.6.1 se ha dado una definición de requerimientos breve pero lo bastante clara de lo que debe soportar esa aplicación.



Modelo 4.16. *Modelo inicial antes de introducir una relación M:N.*

Resulta que hay que guardar qué deportes practica o hace cada persona, los socios del club. Por eso, pagan un precio mensual para cada deporte que deseen practicar. También interesa poder formar equipos entre los socios, y para ello, disponer de cuántos jugadores forman un equipo, de cada deporte.

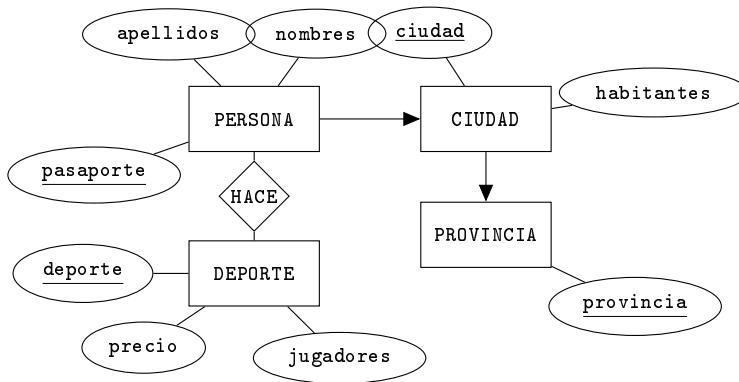
Quien diseña la base de datos piensa, de entrada, como se identifica cada deporte

en el club. Pregunta, y le dicen que con el nombre. O sea, seguro que no puede haber dos deportes que se llamen igual. Después, destila de entre los requerimientos los datos que van atados a cada deporte, y observa que son el precio, y el número de jugadores que componen un equipo. La entidad deporte está clara. Ahora nos planteamos la relación que tiene con las personas y pensamos, con cuidado, formulando estas dos frases en voz alta.

*Una persona puede hacer muchos deportes...  
...y un deporte puede ser hecho por muchas personas.*

Siempre resulta conveniente preguntarle al usuario final de la aplicación si está de acuerdo. Y otra vez, es muy probable que se sorprenda de las preguntas diciendo que nunca se lo había planteado. Y muy probablemente también, responderá mal, y tendremos que preguntarle si está seguro de su respuesta y rectificará.

En el Modelo 4.17, la relación **HACE** guarda información de cuáles deportes hace cada persona, así como de cuáles personas hacen cada deporte. Fíjate que esta última afirmación viene estructurada por las palabras "cuáles" y "cada". O sea, la relación  $R$  guarda *cuáles*  $E_1$  para *cada*  $E_2$ , y por consiguiente *cuáles*  $E_2$  para *cada*  $E_1$ .



Modelo 4.17. *Ejemplo de relación M:N en un modelo ER.*

En la documentación que acompañase este modelo ER debería figurar la explicación de que el atributo **jugadores** es el número de jugadores por equipo, plural que significa número entero. Y también que el **precio** es un número con decimales, para los centavos. Y observa que los atributos de la nueva entidad deporte son los datos que dependen exclusivamente del deporte.

### 4.10.1 Atributos en relaciones M:N

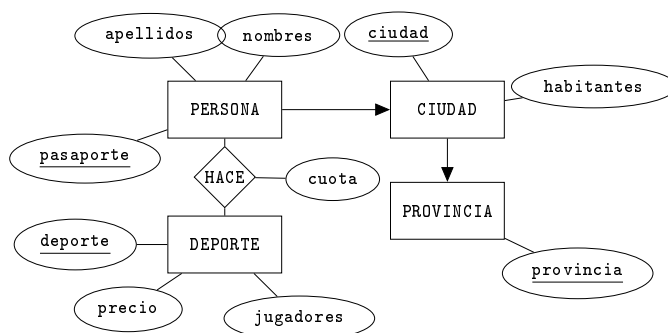
Tal como se dice en los requerimientos, el precio de los deportes debe depender de cada persona porque el club quiere hacer ofertas como que el primer mes es gratuito, o que lo que se paga depende de la edad del socio. Entonces con guardar el precio no es suficiente. Se requiere un nuevo atributo, lo llamaremos *cuota*, que depende además del deporte, de la persona. Así nos reservamos la posibilidad de que por un mismo deporte, distintas personas paguen cantidades diferentes. Si después resultase que todos pagan lo mismo podríamos poner el mismo valor en todos los registros, que coincidiría con el precio del deporte. Pero si en algún momento eso se convirtiese en una regla, entonces estaríamos introduciendo redundancia.

La cuestión que hay que formularse para darse cuenta de estas situaciones está relacionada con la interpretación del análisis matemático expuesta más arriba.

*¿Qué necesitamos para poder hablar de una cuota?*

Y observamos que necesitamos una persona y un deporte.

Un vez convencidos de que el razonamiento es correcto, entonces podemos añadir atributos a la relación, tal como se muestra en el Modelo 4.18.



Modelo 4.18. *Ejemplo de atributos en una relación M:N de un modelo ER.*

Queda claro, el atributo *cuota* de la relación *HACE* significa que cada persona paga una cantidad concreta para cada deporte.

Este tipo de atributos asociados a relaciones M:N son muy habituales. Si el número de atributos de una relación M:N aumenta excesivamente, entonces hay que considerar la posibilidad de hacer una agregación, dando clave primaria a la relación *hace*, y con un poco de imaginación, encontrando un sustantivo que le dé nombre.

## 4.11 Autorelaciones

Llamamos *autorelación* la relación de una entidad con ella misma.

Con las autorelaciones se introduce también el concepto de rol. Un *rol* en un diseño es una palabra que nos dice qué papel juega una entidad en una relación. En el modelo se representan con el nombre del rol al lado de la línea que une la entidad con la relación. Y no tiene nada que ver con los roles de la base de datos que son los tipos de usuario.

Lo dicho de las relaciones binarias entre entidades en un modelo ER es igualmente válido cuando las dos entidades de la relación son la misma, excepto por el hecho de que ahora hay que aumentar un nivel los objetos identificados. Eso es, tanto las autorelaciones 1:1 como las 1:N ahora sí necesitarán identificador. Y para el caso de las M:N, aun más. Para esas hay que usar roles forzosamente.

En adelante, por lo que resta de sección, vamos a ver en detalle las autorelaciones y sus usos habituales para cada tipo de cardinalidad.

### 4.11.1 Autorelaciones 1:1

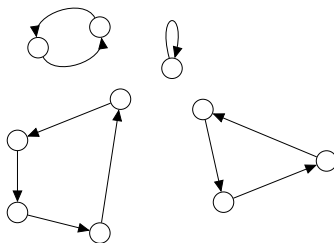
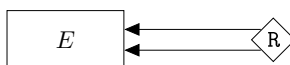
Con una autorelación 1:1 podemos representar desde una simple secuencia, como representa la relación sucesor de números naturales, hasta una relación de equivalencia que particione los elementos de una entidad en clases de equivalencia. Para comprender la capacidad expresiva que tienen este tipo de relaciones, la mejor forma es imaginando un grafo dirigido donde todos los nodos tienen un arco entrante y uno saliente, que pueden ser el mismo.

En la Figura 4.13 los nodos del grafo representan los elementos de una entidad con participación total en la relación indicada por los arcos. Fíjate que esa figura presenta tan solo una posibilidad entre una enorme cantidad que van desde un ciclo que conectase todos los elementos, hasta una relación que los agrupase por parejas. Todas estas posibilidades manteniendo la participación total.

Caso que la entidad tuviera nomás participación parcial en la autorelación 1:1, entonces abrimos todavía más el abanico de posibilidades de representación incluyendo nodos terminales y nodos aislados.

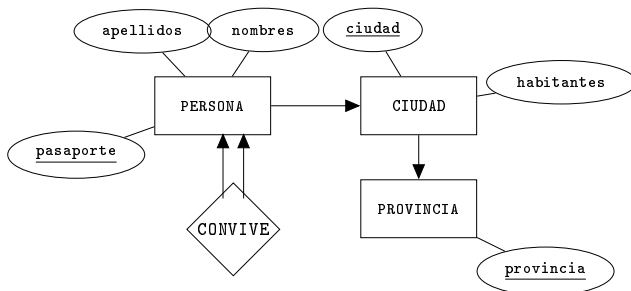
El símbolo de las autorelaciones 1:1 es el de la Figura 4.14.

Veamos un ejemplo de este tipo de relaciones a partir del diseño del Modelo 4.16 de la página 77, donde hay personas que viven en ciudades que están en provincias.

Figura 4.13: *Potencia expresiva de las autorrelaciones 1:1.*Figura 4.14: *Autorrelación 1:1 en un modelo ER.*

Pongamos por ejemplo que interesa guardar para cada persona quien es su pareja. Es decir, para cada persona saber los datos de la persona con quien convive, siempre que sea otra persona de la base de datos. Conste que no se pretende modelar la realidad sino ejemplificar una autorrelación 1:1. Para almacenar esos datos nos convendría una autorrelación M:N.

El aspecto que tendría en el diagrama la relación CONVIVE se puede observar en el Modelo 4.19.

Modelo 4.19. *Ejemplo de autorrelación 1:1 en el Modelo 4.16.*

Por supuesto que la participación de la entidad **PERSONA** en la relación **CONVIVE** es parcial, porque queremos poder insertar personas que vivan solas.

Hacemos un análisis breve para darnos cuenta que si fuese una participación total, se produciría un caso de dependencias de existencia cruzadas, tal como se ha visto en la Sección 4.7.2. Es decir, no podríamos insertar personas porque debería existir su pareja, pero tampoco podríamos insertar la pareja de la persona, porque debería

existir su pareja, que es la persona que estamos pretendiendo insertar. Un pez que se muerde la cola. Death lock.

Ya se ha dicho que los mecanismos para romper ese tipo de ciclos en los SGBDs, posponen la verificación de las restricciones de integridad hasta nuevo aviso. O sea, permiten violar las restricciones transitoriamente.

### 4.11.2 Autorelaciones 1:N

A diferencia de las relaciones entre entidades distintas, con las autorelaciones sucede que cualquier fenómeno que pueda representarse con una autorelación 1:1 también puede ser representado con una autorelación 1:N sin más esfuerzo que restringir repetidos. En otras palabras, no habría ningún problema en utilizar una autorelación 1:N para explicar las relaciones de la Figura 4.13. Como consecuencia las autorelaciones 1:1 quedan arrinconadas, y no se utilizan casi nunca, ya que la diferencia con las autorelaciones 1:N se limita a establecer una restricción de unicidad.

Las autorelaciones 1:N amplían la capacidad expresiva de las 1:1. Sirven particularmente para describir jerarquías entre los elementos de una entidad, cosa que no puede representarse con autorelaciones 1:1, debido precisamente a la restricción de unicidad. Ahí, dos no pueden apuntar al mismo.

Una jerarquía se puede representar en un *vector de predecesores*, cosa que se ilustra en la Figura 4.15. En la parte izquierda se muestra una estructura jerárquica, o sea un árbol. En la parte central, hay una secuenciación por niveles de los nodos manteniendo la estructura arborescente, imagen que conviene retener como concepto de vector de predecesores. Y en la parte derecha, se puede ver el contenido del vector de predecesores propiamente dicho, donde cada casilla contiene el índice del nodo padre. Fíjate que esta estructura tiene un -1 como padre del nodo raíz. De la misma manera podrían haber varias raíces convirtiendo la representación en un conjunto de árboles disjuntos. Es decir, un bosque.

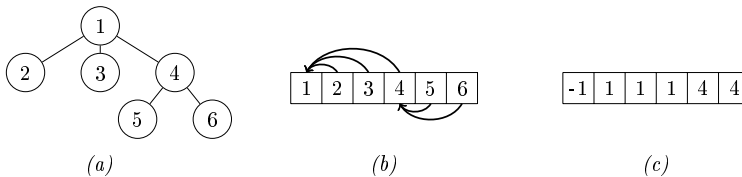


Figura 4.15: Vectorización de un árbol. (a) Árbol; (b) Redisposición secuencial; (c) Vector de predecesores.

Los tratamientos para recorrer la ruta que hay por encima de un nodo hay que plantearlos recursivamente. Una jerarquía es una relación 1:N entre los elementos de un mismo conjunto, ya que cada nodo puede tener muchos hijos, pero un padre



nomás.

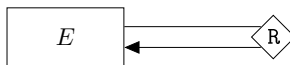
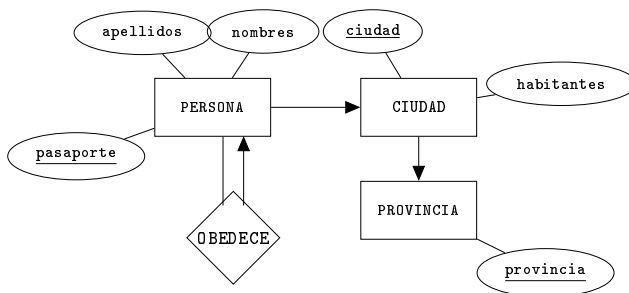


Figura 4.16: Autorelación 1:N en un modelo ER.

El símbolo gráfico correspondiente a una autorelación 1:N se puede observar en la Figura 4.16.

Si en el Modelo 4.16 del diseño con personas, ciudades y provincias suponemos que las personas son trabajadores, entonces tenemos la relación OBEDECE como se muestra en el Modelo 4.20.



Modelo 4.20. Ejemplo de autorelación 1:N en el Modelo 4.16.

La autorelación del Modelo 4.20 se lee como que cada persona obedece a una persona como máximo, pero puede mandar a muchas. Dejémoslo aquí.

### 4.11.3 Autorelaciones M:N

Las autorelaciones M:N expresan dependencias entre elementos de una misma entidad, de manera que un elemento puede ser relacionado con muchos, y al mismo tiempo, muchos elementos se pueden relacionar con uno. Esto introduce una confusión que hay que desambigüizar con los roles.

Se puede intuir que la potencia expresiva de esta herramienta es grande. De hecho, estamos hablando del tipo de relación que hay entre los nodos de un grafo, expresada mediante las aristas que los unen. La capacidad de representación de una autorelación M:N es la misma que la de un grafo como estructura. O sea, extremadamente abstracta. Tanto es así, que como ejemplo podemos imaginar una entidad NODO con una autorelación que se llamara CONNECTA que vendría a representar el conjunto de aristas de un grafo.

Siguiendo en esa línea, si el grafo fuese dirigido, los roles podrían ser *origen* y *destino*. Y si fuese un grafo no dirigido, entonces tan solo podríamos llamar los nodos con nombres de roles que no tendrían demasiado contenido semántico, *nodo<sub>1</sub>* y *nodo<sub>2</sub>*, de manera que reflejaríamos la ambigüedad de un grafo no dirigido en los roles. No es trabajo del diseñador distinguir entre conceptos que en la realidad se confunden.

De todas formas, es una norma obligatoria en un modelo ER que contenga una autorelación M:N la presencia de roles que etiqueten las líneas que hay entre la entidad y la autorelación.

Gráficamente se representan como se ilustra en la Figura 4.17.

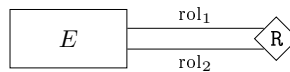
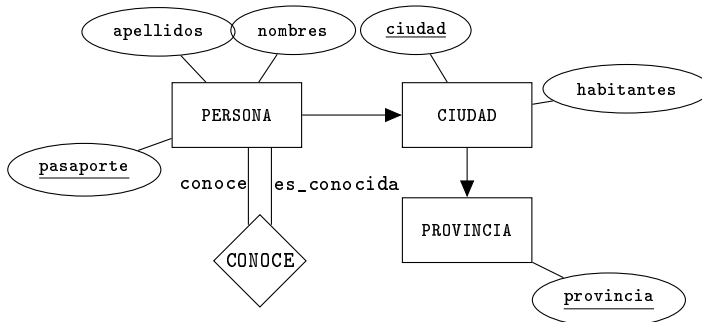


Figura 4.17: Autorelación M:N en un modelo ER.

Seguimos el hilo de los ejemplos que se han ido dando en estas últimas secciones. En el club se desea saber qué personas conocen a qué otras. Viene a ser una especie de red social interna, con el objetivo específico de formar equipos. Esto nos conduce a un modelo más realista que en el ejemplo de las autorelaciones 1:1, donde se suponía que cada persona solo podía convivir con una sola persona.

Hay que modelar el hecho de que una persona puede conocer muchas. Y también puede ser conocida por muchas otras.



Modelo 4.21. Ejemplo de autorelación M:N en el Modelo 4.16.

En el diseño del Modelo 4.21 se muestra la autorelación M:N que explica que unas personas conocen otras personas. Los roles *conoce* y *es\_conocida* son necesarios en el modelo, ya que trascienden al modelo relacional una vez aceptado el diseño, a la hora de implementarlo.

## 4.12 Cálculo Automático de las Cardinalidades

Fácilmente se puede automatizar la tarea de establecer la cardinalidades de las relaciones. O sea, es una cosa simple, pero se debe tener claro. Más que tener utilidad funcional, el código que se muestra en esta sección sirve para aclarar la manera de proceder cuando hay que establecer las cardinalidades de un modelo ER.

El algoritmo de la Caja 4.1 sirve para bases sencillas nomás, ya que considera que entre entidades puede haber una relación como mucho, no más de una. Es decir, el pseudocódigo que se muestra seguidamente no serviría para el diseño del Modelo 4.13, donde entre personas y ciudades hay dos relaciones.

```

funcio establecer_cardinalidad( $E_1, E_2$ ) retorna string {
    cuantos12 = leer("¿Cuántos " +  $E_1$  + " tiene " +
                    "un único " +  $E_2$  + "(NINGUNO, 1, o N) ?");
    si (cuantos12 = "NINGUNO") retorna nulo;
    si (cuantos12 = "1") {
        cuantos21 = leer("¿Y cuántos " +  $E_2$  + " tiene " +
                        "un único " +  $E_1$  + " (1, o N) ?");
    }
    sino {
        cuantos21 = leer("¿Y cuántos " +  $E_2$  + " tiene " +
                        "un único " +  $E_1$  + " (1, o M) ?");
    }
    retorna cuantos12 + ":" + cuantos21;
}

procedimiento establecer_cardinalidades( $BD$ ) {
     $n = |BD|$ ;
    para cada  $i$  en 1.. $n$  {
        para cada  $j$  en 1.. $i$  {
            cardinalidad = establecer_cardinalidad( $BD[i], BD[j]$ );
            si (cardinalidad) {
                escribir("La relación entre " +  $i$  + " y " +  $j$  +
                        " tiene cardinalidad " + cardinalidad);
            }
        }
    }
}

```

Caja 4.1. Pseudocódigo para establecer las cardinalidades de una base de datos.

El procedimiento en pseudocódigo `establecer_cardinalidades`, en plural, de la Caja 4.1 recibe como parámetro de entrada una colección de entidades que llama  $BD$ . No retorna nada, ya que es un procedimiento y no una función. El resultado queda en el canal de salida. Realiza la mitad del recorrido del conjunto  $BD \times BD$ ,

es decir, recorre la mitad del producto cartesiano del conjunto de entrada por sí mismo. Eso es, trata cada pareja posible de entidades una sola vez, incluyendo las formadas por una entidad y ella misma. Observa que como el bucle interno viene limitado por la variable del bucle externo, en el cuerpo interior a los dos bucles se tratan todas las parejas  $(i, j)$  tales que  $i \geq j$ . Así nos aseguramos que solo trate una vez cada pareja. Entonces, para cada pareja posible de entidades hace una llamada a la función `establecer_cardinalidad`, en singular, que recibe por parámetros dos entidades, y retorna un string que puede valer "1:1", "1:N", o "M:N", o bien puede ser nulo cuando entre ambas entidades no haya relación alguna.

La función `establecer_cardinalidad` anima al diseñador a realizar la reflexión preguntándole qué relación de cardinalidad hay entre las entidades que se le han pasado como parámetros. Y ya que en el código de la Caja 4.1 no hay ningún control de errores, al menos en la mismas preguntas se le muestran las respuestas posibles admitidas.

La primera pregunta es cuántos elementos de la primera entidad se pueden relacionar con un único elemento de la segunda. La respuesta del usuario se recoge en la variable local `cuantos12`. Hay tres posibilidades para esa primera respuesta. "NINGUNA", "1", o "N".

- Si el usuario dice que ninguna, entonces no hace falta decir nada más. No hay relación entre las entidades, y se retorna nulo.
- Y si no, se requiere una segunda pregunta que es la inversa de la primera. Es decir, cuántos elementos de la segunda entidad se pueden relacionar con un único elemento de la primera. Las respuestas posibles son dos, que también dependen de la primera, como se puede ver. En cualquiera de esos dos casos, las respuestas admisibles también se indican entre paréntesis.

El pseudocódigo de la Caja 4.1 es orientativo. Básicamente se pretende dar a entender que una vez decididos los conceptos que formarán entidades en el modelo ER, entonces para cada pareja posible se debe analizar si hay relación, y en caso que sí, de qué cardinalidad. No se trata de una panacea, ya que resultarán relaciones redundantes, porque las dirá todas. Por ejemplo, para el diseño del Modelo 4.16 nos diría que entre `PERSONA` y `PROVINCIA` hay una relación 1:N, bien, en rigor nos diría N:1. Y realmente esta relación no debe estar en el modelo porque es absorbida por las otras dos. Pero bueno, este último razonamiento también debe formar parte del análisis, y por tanto, es cosa buena que nos lo planteemos.

## 4.13 Relaciones Ternarias

Dadas tres entidades  $E_1$ ,  $E_2$ , y  $E_3$ , y una circunstancia que las involucra, podemos establecer una relación ternaria que las una. Si las relaciones M:N en un modelo ER provocan la existencia de nuevos nodos en el grafo, aún con más razón las relaciones ternarias. Ese tipo de relaciones son extremadamente concisas, y dan mucha información con toda la coherencia necesaria, aunque mal usadas suponen la generación de nulos estructurales.

Desde el punto de vista del análisis matemático, una relación ternaria vendría a ser una función lógica que dependiese de tres variables independientes. Es decir  $f = f(x, y, z)$ , siendo  $x$  un elemento de  $E_1$ ,  $y$  uno de  $E_2$ , y  $z$  de  $E_3$ . Así pues, el dominio de  $f$  es un espacio tridimensional, o sea como un cubo en el que cada uno de los tres ejes corresponde a una disposición secuencial de los elementos de cada una de las tres entidades. Es decir, de lo que estamos hablando es del conjunto resultante del producto cartesiano entre las tres entidades,  $E_1 \times E_2 \times E_3$ .

O en otras palabras, eso significa que cada elemento de la relación ternaria es un trío o terna posible formado por un elemento de cada entidad.

Las frases que debemos pronunciar, en voz alta, antes de establecer una relación ternaria entre tres entidades debe tener tres veces la palabra "cualquier".

*Cualquier  $E_1$  puede estar relacionado con cualquier  $E_2$  y cualquier  $E_3$ .*

Conviene además, repetir esta afirmación con los distintos órdenes entre las entidades. O sea, añadir

*Y cualquier  $E_2$  puede estar relacionado con cualquier  $E_3$  y cualquier  $E_1$ , y también cualquier  $E_3$  puede estar relacionado con cualquier  $E_1$  y cualquier  $E_2$ .*

Formular estas consideraciones en voz alta provocará una reflexión sobre si efectivamente el fenómeno que pretendemos modelar se ajusta a una relación ternaria.

La representación de una relación ternaria en un modelo ER es la que se puede ver en la Figura 4.18.

Retomamos el ejemplo del Modelo 4.17 del club deportivo donde había personas que vivían en ciudades que estaban en provincias, y que podían hacer diversos deportes. Bien, ahora entran a escena los distintos pavellones del club. Son pavellones

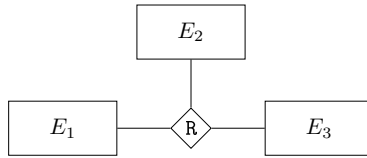


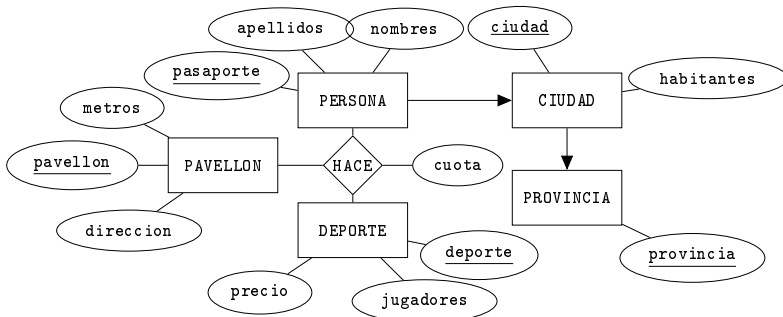
Figura 4.18: *Relación ternaria en un modelo ER.*

polivalentes que en cualquiera de ellos se puede hacer cualquier deporte. Y suponemos que resulta interesante saber en cuáles pavellones hace cada deporte cada persona. Fíjate en la estructura *cuáles...cada...cada*.

El razonamiento que nos conduce a establecer una relación ternaria pasa por las siguientes reflexiones.

*Cualquier persona puede hacer cualquier deporte en cualquier pavellón, cualquier deporte se puede hacer en cualquier pavellón por cualquier persona, y cualquier pavellón puede ser usado por cualquier persona para hacer cualquier deporte.*

Y si el usuario final lo aprueba, podemos proseguir adelante con el Modelo 4.22.



Modelo 4.22. *Ejemplo de relación ternaria añadido al Modelo 4.18.*

### 4.13.1 Cardinalidades en las relaciones ternarias

La relación ternaria del Modelo 4.22 se dice que tiene una cardinalidad M:N:P. En las relaciones binarias las cardinalidades condicionan el modelo. A lo largo de las secciones anteriores se ha visto cómo podíamos fusionar las relaciones 1:1, o cómo hacer

desaparecer el nombre y el rombo de una relación 1:N del diagrama. Y también cómo una M:N causaba la introducción de un nuevo nodo en el modelo. En las relaciones ternarias todo eso es muy distinto. La capacidad expresiva de una relación ternaria es tal que en cualquier caso generará un nuevo nodo en el grafo que es el modelo ER. Y por tanto, todo lo que sea cardinalidades dejará de condicionar el espacio usado.

Las cardinalidades en las relaciones ternarias no trascienden a la implementación. Es decir, cualquier restricción de participación se deberá controlar a partir de la lógica que finalmente, después de haber implementado el modelo, añadimos a la base de datos en forma de procedimientos mediante los lenguajes que el SGBD nos ponga a disposición. Por eso, no se le presta tanta atención a las cardinalidades de las relaciones ternarias, ya que siempre se pueden modificar en última instancia. Con una finalidad estrictamente documental, admitiremos sin embargo que aparezca una flecha nomás en una relación ternaria, tal como se dibuja en la Figura 4.19.

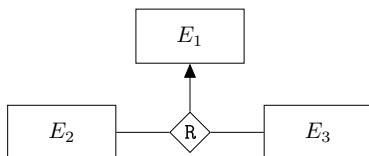


Figura 4.19: *Relación ternaria con cardinalidad 1:M:N.*

El modelo de la Figura 4.19 significa que cada pareja formada por un elemento de la entidad  $E_2$  y uno de la  $E_3$ , puede ser relacionada con un único elemento de la entidad  $E_1$ . Es decir, que no se admiten repeticiones de parejas formadas por un elemento de  $E_2$  y uno de  $E_3$ . Eso hay que documentarlo necesariamente.

Más de una flecha en una relación ternaria en un modelo ER está prohibido. Debido a la confusión semántica que podría querer indicar no se considera conveniente usarlo en un modelo. Por eso, si en una relación ternaria hay más de una flecha se trata de un error. No hay ninguna interpretación oficial.

### 4.13.2 Relación ternaria con tres relaciones binarias

A menudo hay quien se plantea tres relaciones binarias M:N en lugar de una sola relación ternaria. Es decir, el modelo de la Figura 4.20 en lugar del de la 4.18.

Son cosas muy distintas.

El modelo de la Figura 4.18 es mucho más conciso, ya que entre otras cosas pide que exista un tercer elemento para poder relacionar dos cualesquiera.

El 4.20 es mucho menos restrictivo, y en muchas ocasiones, cuando lo utilizamos

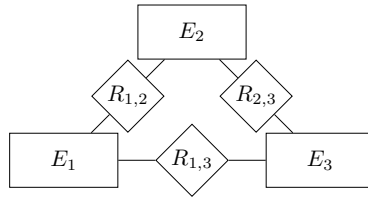


Figura 4.20: *Alternativa incorrecta a una relación ternaria.*

en vez de una ternaria, introduciremos confusión en la base de datos.

La diferencia entre ambos modelos es considerable, y hay que saber qué tipo de relación se necesita en cada caso. Esencialmente, el contraste está en que en el modelo 4.18 hay una sola relación, y por tanto todo va atado. En el 4.20 las relaciones son independientes. O sea, si la posibilidad de que  $E_1$  se relacione con  $E_2$  no tiene nada a ver con la existencia de un elemento de  $E_3$  que participe, entonces el modelo 4.20 está utilizado correctamente. De hecho, eso no tiene nada que ver con las relaciones ternarias.

## 4.14 Modelo Entidad Relación Extendido

La antigüedad que tiene toda la teoría de bases de datos es una garantía de su robustez. El modelo ER como método de diseño de bases de datos es igual hoy en fecha de 2017 que hace cuarenta años. Y eso es fantástico, porque significa que a lo largo de todos esos años ha demostrado reiteradamente su efectividad.

Es más. El éxito de los diagramas entidad relación en las aplicaciones de datos ha sido tan rotundo que ha creado escuela. Los diagramas de clases que se utilizan intensamente en la programación se han servido de algunas de las propuestas del modelo ER. Especialmente por lo que respecta a las cardinalidades. También, algunos de los artefactos que se definen en ingeniería de proyectos en UML tienen su raíz en el modelo ER, en especial todos aquellos que no tienen relación con el tiempo ni sincronicidad entre módulos.

Sin embargo, ha habido un momento de la historia, el único, en el cual el diseño de bases de datos se ha alimentado de una nueva tecnología. La programación orientada a objetos, que aún no tiene treinta años. Y la cosa buena que tiene la programación orientada objetos, y que en un principio era una carencia de los modelos ER y por eso finalmente lo incorporó, es la capacidad de alojar datos de manera condicional. Condicionados a un valor de algún dato existente.

Es decir, estructuras que permitan guardar si una persona es un bombero o un



taxista, y que además, caso de ser bombero, se pueda guardar si es cabo, o raso, es decir el cargo. Y si es taxista, guardar el número de licencia del taxi.

En un modelo ER la solución antigua establecería las dos entidades **BOMBERO** y **TAXISTA**, y en cada una de ellas colocaría los atributos necesarios para guardar lo que se pide. El cargo para los bomberos y la licencia para los taxistas. No obstante, aparecen una serie de datos como los nombres, los apellidos, o el número de pasaporte, que estarían en las dos entidades, y que, por cuestiones de diseño se guardarían separados a pesar de que lógicamente hacen referencia a un mismo tipo de concepto, las personas.

Otras veces interesaría definir restricciones de unicidad entre todos los valores no tan solo de un atributo de una tabla, sino también entre los de dos atributos de dos tablas. Por ejemplo, suponiendo que tenemos la forma antigua de dos tablas, **BOMBERO** y **TAXISTA**, y que además no se permitiese la existencia de bomberos que fuesen también taxistas, entonces nos agradecería poder decir que el número de pasaporte debe ser diferente no solo entre todos los bomberos, sino también entre los bomberos y los taxistas.

Con las herramientas que se han mostrado hasta ahora, eso no es posible. Por eso, incorporamos conceptos procedentes de la programación en clases o lo que es lo mismo, programación orientada a objetos.

#### 4.14.1 Especialización y Generalización de Entidades

Así pues, una aportación muy importante que la programación orientada a objetos hizo al diseño de bases de datos es la capacidad de considerar un tipo de objeto como una extensión de otro. Las herencias.

Se representan en una estructura como la de la Figura 4.21.

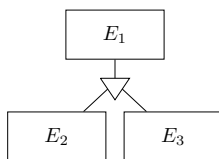
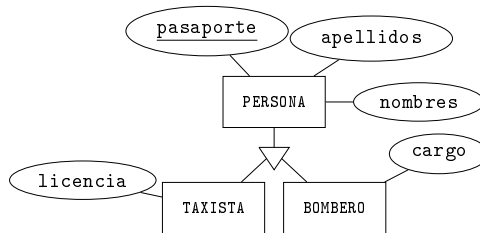


Figura 4.21: *Especialización o generalización en un modelo ER.*

A primera vista, se observa que el modelo 4.21 es el primero en el que las líneas que unen entidades no representan estrictamente relaciones. En la Figura 4.21 hay tres entidades.  $E_1$  se llama entidad *genérica*, o *madre* si se desea usar la terminología de la programación orientada a objetos. Tanto  $E_2$  como  $E_3$  son las entidades *especializadas*, o *derivadas* en aquel vocabulario.

El símbolo de la Figura 4.21 significa que cualquier atributo que se añada a  $E_1$ , es como si se añadiese automáticamente a  $E_2$  y a  $E_3$ . A esos atributos, los de  $E_1$ , los llamamos atributos *genéricos*. Y llamamos atributos *específicos* a los que no son genéricos, es decir, los que se encuentran exclusivamente en las entidades especializadas,  $E_2$  y  $E_3$ . A partir de esta estructura cada vez que se inserte un elemento en cualquier entidad especializada también se estará añadiendo de retruque a la genérica. Pero no al contrario.

Un ejemplo sencillo que ilustra esos conceptos se muestra en el Modelo 4.23.



Modelo 4.23. *Ejemplo de especialización o generalización en un modelo ER.*

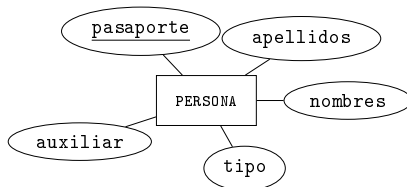
Según este modelo, de las personas que no sean taxistas ni bomberos nos guardaremos nombres, apellidos, y pasaporte. De los taxistas, además nos guardaremos la licencia. Observa que en ningún lugar nos guardamos el hecho que la persona sea taxista. Lo sabremos porque tiene licencia. Y de los bomberos además de los datos de cualquier persona, tendremos el cargo.

Fíjate que en el ámbito del diseño de bases de datos se distingue entre dos conceptos que en última instancia van a parar a una misma representación. Por un lado, se considera una especialización cuando una entidad puede ser de distintos tipos. Y según de qué tipo sea desearíamos guardar otros datos adicionales, los atributos específicos. Para decidir crear una especialización, además debe suceder que los tipos específicos de entidades se presenten en proporciones parecidas. Es decir, siguiendo con el ejemplo, si de las personas que nos interesa guardar hubiera un 95% de taxistas, entonces casi merecería la pena, rompiendo la máxima de nullos estructurales, guardar el número de licencia para todas las personas dejándolo nulo en el 5% de las personas restantes.

Y por otro lado, el proceso de creación de una generalización es a la inversa. Y más sencillo. De entrada a partir del modelo que tenemos, observamos que hay un conjunto de atributos presentes en varias entidades. Y entonces decidimos crear una nueva entidad con esos atributos comunes. Eso requiere inventarse un nombre para esa nueva entidad.

Hay malos diseñadores que, creiéndose muy listos, piensan que la solución que se muestra en el Modelo 4.24 es buena, inclusive mejor que la del Modelo 4.23 ya que se

ahorra una entidad. Para entender esta propuesta se explican diciendo que el atributo **tipo** guardará uno de los valores *bombero* o *taxista*. Y entonces, si **tipo** vale *bombero*, el atributo **auxiliar** contendrá el valor del **cargo**. Y si vale *taxista*, entonces el **auxiliar** guardará la **licencia**. Y lo explican orgullosos, con una sonrisa.



Modelo 4.24. *Buñuelo inadmisibile en un modelo ER.*

La existencia del atributo **tipo** aún. Se puede justificar en los casos que haya más de un tipo de entidades específicas de las que no nos interesa guardar información adicional. Cuando eso ocurre, se puede poner este atributo a pesar de introducir redundancia en las entidades específicas que sí tienen atributos específicos.

Ahora bien, quede claro que la idea "inteligente" de guardar innecesariamente el tipo de persona de manera explícita en un atributo, y, además, un atributo auxiliar que según el tipo guarde la licencia o el cargo, es un error grave que supone un insulto al modelo introduciendo una dependencia funcional palmaria.

Recuerda de la Sección 4.9 que una dependencia funcional en una entidad se produce cuando el valor de un atributo condiciona el valor de otro de la misma entidad. Todo ello resulta en una complicación incómoda que forzaría cualquier tratamiento de la entidad a considerar casos. Es por tanto un agravio a la legibilidad, además. Las dependencias funcionales violan las formas normales para las tablas relacionales. En este libro no se describe la teoría de las formas normales, pero se respetan. Quien esté interesado puede encontrar la descripción minuciosa en [10].

Hay que quitarse de la cabeza ideas como que si este atributo vale no se qué, entonces este otro contendrá eso, y si no aquello. Eso son dependencias funcionales y por tanto diseños tan inadmisibles como el del Modelo 4.10.

### Especializaciones o generalizaciones completas

Cuando una especialización, o generalización, establece que cualquier elemento de la entidad genérica debe estar también en alguna de las entidades especializadas recibe el distintivo de *completa*. O sea, una especialización completa no permite que haya elementos en la entidad de arriba que no aparezcan en ninguna de las de abajo. Esto se puede indicar en el diagrama haciendo doble la línea de la conexión superior del

triángulo, que vendría a decir que la participación de la entidad genérica en el conjunto de las especializadas es total. Hay cierta analogía con los lenguajes de programación orientados a objetos donde la entidad genérica se declararía como clase abstracta, de forma que no pudiese ser instanciada directamente.

### Especializaciones o generalizaciones disjuntas

Que una especialización, o generalización, sea completa no impide que un elemento de la entidad genérica pueda estar en todas las especializadas. Es decir, es posible que haya un taxista bombero, en el caso del ejemplo. Cuando se desea impedir que un mismo elemento de la entidad superior aparezca en más de una de las entidades inferiores, entonces se dice que la especialización, o generalización, es *disjunta*, y se indica en el diagrama con el término "disjunta" tocando a la línea de la conexión superior del triángulo, es decir abajo a la derecha de la entidad genérica.

#### 4.14.2 Agregación de Entidades

A lo largo de todo este capítulo se ha insistido repetidamente que la diferencia entre una entidad y una relación M:N es la existencia o no de atributos clave primaria que identifican cada elemento. Los atributos clave no solo tienen valores distintos en cada elemento de una entidad, sino que precisamente porque identifican esos elementos, también son utilizados en las relaciones donde participa la entidad para ser aparejados con elementos clave de entidades relacionadas. Análogamente, es lo que en programación orientada a objetos se declararía como variable miembro pública de una clase. Lo que se conoce desde fuera. Eso significa que la esencia que realmente caracteriza una entidad es la capacidad de ser relacionada con otras. Una relación que relacione relaciones es un concepto absolutamente prohibido, que tumba cualquier diseño. No puede ser.

Sin embargo, a veces resulta que nos convendría hacer alguna cosa como una relación entre relaciones. Básicamente, nuestra intención es poder relacionar cualquier elemento de una relación con alguna otra cosa. Y como eso no puede ser, precisamente porque las relaciones no tienen clave, existe el concepto de agregación. Hacer una agregación es introducir una clave en una relación para transformarla en entidad, para lo cual conviene cambiarle el nombre, del verbo que pudiese tener al sustantivo propio de las entidades. Eso se hace en relaciones M:N y relaciones ternarias.

Las agregaciones no tienen ningún símbolo para representarse en un modelo ER, aunque de todas formas, los diseñadores experimentados adquieren la capacidad de reconocerlas.

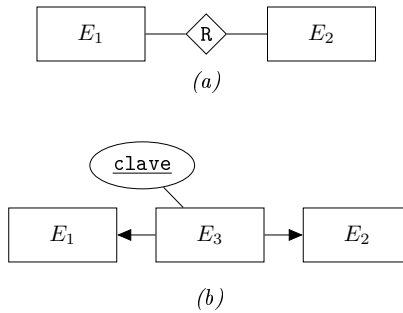


Figura 4.22: *Agregación de una relación. (a) Relación M:N. (b) Entidad agregada.*

En la Figura 4.22 se puede ver como se hace una agregación en el caso de una relación binaria M:N. En la parte superior hay la relación antes de ser agregada. Es decir, antes de descubrir la necesidad de relacionarla con otras entidades del modelo. En la inferior se ha transformado la relación  $R$  en una nueva entidad  $E_3$  añadiendo una clave primaria.

En definitiva, hacer una agregación significa transformar una relación M:N en una entidad en dos relaciones 1:N, en las cuales la nueva entidad está al lado N de las dos.

*En este capítulo se ha introducido un lenguaje gráfico que sirve para modelar diseños de bases de datos. Los conceptos primordiales de este lenguaje son el de entidad y el de relación. Una entidad viene caracterizada por una colección de atributos que le dan estructura entre los cuales hay alguno o algunos que forman la clave. Las entidades tienen clave. Las relaciones no. Una relación viene caracterizada por las agrupaciones, sean parejas o tríos, de claves de las distintas entidades que relaciona.*

## Capítulo 5

# Álgebra Relacional

Como dice el preámbulo, soportar contenidos teóricos en el método matemático hace más robustos los contenidos. Bien, pues el álgebra relacional es una teoría matemática que nació juntamente con las bases de datos, y rige su comportamiento.

El propósito de este capítulo es adquirir la capacidad de actuar, con papel y lapicero, de la misma manera que lo hace un SGBD de manera automática. Eso significa saber describir los datos, y saber calcular con ellos.

Este capítulo empieza con una introducción a la noción de álgebra en la Sección 5.1. Después, a pesar de formar parte de un mismo contenido teórico, el álgebra relacional se fragmenta en dos partes. Primero, en la Sección 5.2, la que tiene que ver con el espacio de datos, o sea los objetos, que a partir de aquí en adelante serán las relaciones. Y, antes de entrar en la segunda, se describe la transformación del diseño de la base de datos para el club deportivo a su modelo relacional en la Sección 5.3. Es decir, cerramos definitivamente todo lo que afecta al espacio antes de entrar en las operaciones del álgebra, en la Sección 5.4. Las operaciones viven en el mundo del tiempo.

Con todo, lo que en este capítulo se presenta son unas nociones básicas del álgebra relacional, pero no es el propósito aquí extenderse demasiado en la dirección más teórica. Particularmente, se echa mucho de menos la parte de formas normales de las relaciones.

Para quien esté interesado en los fundamentos y desee profundizar, hay la referencia ineludible de Silberschatz, [10]. Tanto la sintaxis como todo lo que afecta la teoría del álgebra relacional de este capítulo fue introducido en esa obra.

## 5.1 Álgebras

En su sentido más genérico el Álgebra estudia los comportamientos y el potencial expresivo de unas estructuras en las que básicamente hay definidos unos valores, así como unas operaciones internas para trabajar con esos valores.

Una operación es *interna* a un conjunto de valores cuando tanto los operandos de entrada como el valor resultante de salida de la operación pertenecen al conjunto. Hay muchos ejemplos, como la suma, que es una operación interna para los números naturales, y también para los números pares, pero no para los impares. La comparación, en cambio, no es una operación interna ni para los naturales ni para los enteros, ni para ningún conjunto de números, porque aunque de entrada le demos los números que sí son del conjunto en cuestión, nos retorna un valor lógico, es decir cierto o falso, cosa que no es un número.

El ejemplo más conciso de lo que es un álgebra lo tenemos en el Álgebra de Boole, donde los valores posibles son el cero y el uno, y las operaciones la negación, el producto y la suma lógica, en la cual uno más uno da uno. Es decir, la suma, en el álgebra booleana se define como la unión de conjuntos de la Sección 1.3.1, o la operación disyuntiva del cálculo de predicados que se ha visto en la Sección 1.4.1.

En el álgebra relacional, en cambio, los valores en lugar del cero y el uno, son relaciones, cosa que las convierte en el concepto central de la teoría.

## 5.2 Relaciones

Uno de los aspectos que seguramente introduce más confusión a la hora de adquirir la terminología propia de las bases de datos es el cambio semántico que va sufriendo el término *relación* a lo largo de los distintos capítulos. En los Capítulos 3 y 4, se ha utilizado básicamente como un arco del grafo del modelo ER, aunque también, para cardinalidades complicadas, se han visto relaciones que generan nodos.

Aquí es distinto. En el álgebra relacional retomamos el concepto de conjunto visto en el Capítulo 1.1 para las relaciones. Las relaciones son conjuntos, o si quieres listas, por imaginar una cosa más concreta. Y recuerda que ya en la Sección 1.2 se insiste en que por definición un conjunto no tiene elementos repetidos. Es decir, en el álgebra relacional por definición las relaciones no tienen tuplas iguales.

El núcleo de una base de datos es un conjunto de relaciones. Lo más importante y más independiente de cualquier otra cosa. Las bases de datos empiezan declarando el conjunto de relaciones que las constituyen.



Para las personas más jóvenes puede resultar sorprendente escuchar utilizar la palabra relación como sinónimo de lista. El primer significado que le damos, en la infancia o la adolescencia, es el de que una persona está relacionada con el lugar donde vive, o los hermanos están relacionados porque son hijos de una misma madre... una semántica cercana al concepto de dependencia. Después, llega un día que escuchamos a alguien diciendo que tiene la relación de todas las personas de una comida o de una reunión. Y pensamos, ¿Cómo?, ¿Una relación también es una lista?

Cuando vas viendo que efectivamente el término tiene un sentido que hasta entonces no habías descubierto, te preguntas si realmente es un sentido nuevo, o el antiguo ya era un concepto lo bastante genérico como para incorporar este nuevo uso del término. Porque claro, las cosas que aparecen en una lista están relacionadas por el mero hecho de estar en esa lista...

De todas maneras, en los diccionarios se diferencian estas dos acepciones. Mirando [8] el sentido de lista aparece en el segundo lugar. La entrada tiene muchas acepciones y usos. Las tres primeras dicen así,

**relación** [s. XIV; del ll. *relatio*, *-ōnis*, *íd.*]

*f 1 1 Acción de relatar o referir aquello que uno ha oído, ha visto, etc. Una relación detallada de los hechos. Relación oral, escrita. (...)*

*2 Lista o enumeración de nombres, direcciones u otras indicaciones especialmente ordenadas.*

*3 1 Vínculo, referencia, conexión, que uno percibe o imagina entre dos o más cosas. Una relación de similitud. La filosofía y la teología tienen una relación muy estrecha. Entre estos hechos no hay ninguna relación.*

...y sigue con varios usos de distintos ámbitos.

Respecto a esta entrada del diccionario, parece que la primera acepción sea un caso particular de la segunda. En cualquier caso, la primera es útil por el hecho de mostrar que la raíz de relación es la misma que la de relatar. Interesante. Respecto las otras dos, claramente la segunda es la que permite considerarla un sinónimo de lista, y es el uso que se hace en el álgebra relacional. Y observa que curiosamente, la tercera acepción es la que utilizábamos en el capítulo de diseño con los modelos ER.

Bien. Dejamos atrás el circunloquio sobre las relaciones para poner los puntos sobre las íes. A partir de aquí, de este capítulo en adelante, nos encontraremos en un entorno más categórico que en capítulos anteriores. Por ese motivo, hay que describir con rigor la definición de relación.

### 5.2.1 Relación

Una relación es un conjunto de unos elementos que llamamos *tuplas*. Un tupla es un elemento con un número fijo de *atributos*. El valor de una tupla en un atributo es un dato, o una información elemental.

El término tupla, no se utiliza demasiado en la práctica. Normalmente se habla de registros o filas, pero no de tuplas. Sin embargo, tiene una razón de ser. Una tupla marca diferencias con un registro en el sentido que un registro tiene un enfoque más tecnológico. Más físico. Y la definición del término fila hace referencia a la relación a la cual pertenece. Así pues, la palabra tupla nos debe servir para la parte estrictamente formal del álgebra. El sujeto al que pretendemos referirnos es un elemento de la relación entendida como conjunto. Eso permite el desarrollo del álgebra relacional, y más aún del cálculo relacional por tuplas.

Así pues, una relación se puede representar como una matriz de datos. Una matriz plana, o sea de dos dimensiones. Filas por columnas, vaya. A las filas, que son los elementos del conjunto, se las llama tuplas dentro del ámbito del álgebra relacional, y registros dentro del ámbito de la programación de bases de datos. A las columnas se las llama atributos o campos, en todos los ámbitos.

En la Figura 5.1 se muestra la terminología propia del álgebra relacional.

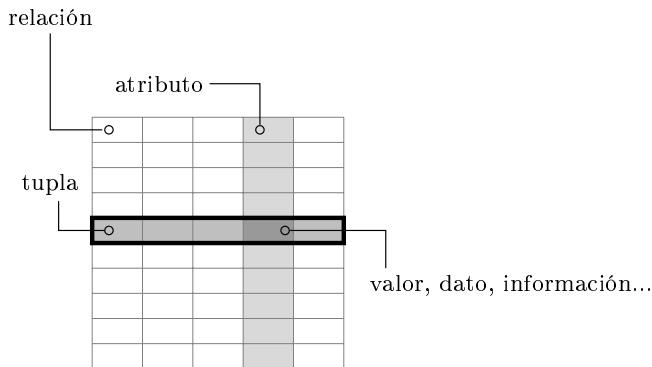


Figura 5.1: Una relación es un conjunto de tuplas.

#### Tuplas

Una tupla es una agrupación de valores de tipos heterogéneos definidos en un orden concreto, cosa que permite que asociemos esta noción a las estructuras de datos `RECORD` del fortran, o `struct` del lenguaje c. En java, en cambio, una tupla vendría a ser una clase sin ningún método. Es decir, un secuencia de atributos de diversos tipos.

**Regla 1 (de las bases de datos relacionales)**

*Las tuplas de una relación tienen un número fijo de atributos, cada uno de los cuáles tiene un tamaño constante.*

Un tupla es un elemento con un número fijo de atributos. Insistimos, porque todo arranca de aquí. Un número fijo para que podamos mover grandes cantidades de tuplas bajo control riguroso. Un número fijo para poder acceder rápidamente a la  $n$ -ésima tupla multiplicando  $n$  por el tamaño de la tupla. Un número fijo porque a pesar de que los humanos representamos los datos en superficies planas de dos dimensiones, a la hora de añadir y quitar datos, resulta más sencillo utilizar solo una.

Es tal la trascendencia de esta limitación, que condiciona la teoría completa de las bases de datos relacionales, inclusive el álgebra relacional que aquí ahora empezamos.

El número de columnas es fijo. En cambio, el de filas puede variar muy a menudo, ya que los elementos del conjunto que es la relación son las filas. Y recuerda de la Sección 1.3 que podemos hacer uniones y diferencias para añadir y quitar elementos de los conjuntos.

Demos una vuelta más a la cuestión, desde una visión más tecnológica. Tenemos un conjunto de registros, o filas, que todos tienen la misma anchura. Eso lo hacemos así para que nos resulte más fácil poder acceder a su contenido. Como ya se ha dicho, si cada fila ocupa un espacio fijo, entonces podemos acceder rápidamente a la fila  $n$  multiplicando  $n$  por el tamaño de una fila. Y es una prioridad poder acceder rápidamente a los registros.

Eso resulta interesante. Por una parte, entre líneas se puede leer una regla que parece natural. Se está diciendo que los archivos con registros de longitud variable no pueden tener acceso directo. Y no es verdad. Existen archivos con longitud de registro variable y acceso directo, los ficheros indexados o calculados. Pero soportar los SGBDs en estos tipos de ficheros complicaría excesivamente la gestión de los datos. En fin, parece razonable asociar registros de longitud variable con acceso secuencial, como los xml o los html.

Y por otro lado, también entre líneas se vuelve a hablar de aquello de las banderas lineales y las banderas planas de la introducción del Capítulo 3. Allí se decía que podemos hacer las banderas lineales tan largas como queramos, y las planas no. Y aquí tenemos la prueba. Pon atención. Como se ha dicho más arriba, guardamos la información en tablas de dos dimensiones, pero la estructuramos de forma que tan solo le permitimos crecer indefinidamente en una de las dos. Es como las banderas. Una relación es como una bandera lineal, en vertical.

Bien, ahora vamos al detalle. Hasta aquí debería estar claro que el número de datos que tiene una relación es el número de atributos multiplicado por el número de tuplas. Para poder trabajar con estos datos, lo primero es identificar cada celda de

la matriz. Y para identificar una celda hay que dar columna y fila.

La identificación de las columnas es fácil. Se llaman con una palabra diferente a cada una y ya está, identificadas. Ergo, cada relación que es una matriz, además de tener un nombre como tal, también debe tener un nombre diferente para cada columna. Fíjate que una de las dos dimensiones que tiene una relación, la horizontal, se establece en el momento de su definición. En cambio, el nombre de filas es variable, siempre se pueden añadir o quitar tuplas de una relación. Para las filas, pues, como no sabemos qué nuevos valores pueden venir, la única cosa que podemos hacer es establecer una restricción que fuerce que, al menos en los valores de algunas columnas concretas, todas las filas tengan valores o combinaciones de valores distintos.

De la capacidad de establecer esta restricción nace la primera diferencia entre un SGBD y un sistema de archivos de un sistema operativo. Tener conocimiento del contenido de los datos es la primera característica distintiva de los SGBDs, cosa necesaria para poder dictar una restricción que impida la repetición de según qué contenidos.

En síntesis, observa que la identificación de las filas es un orden de abstracción más complicado que la de las columnas, ya que, en filas, la única cosa que podemos hacer en el momento de la creación de la relación es establecer que para algún atributo concreto todas las tuplas que se añadan a la relación por los siglos de los siglos sean distintos. Eso es una norma, una restricción, una ley... En cambio para las columnas es más fácil. Somos nosotros, los diseñadores en el momento de la creación, que establecemos los títulos. Y somos nosotros que sabemos que deben ser todos distintos.

## Cálculo relacional por tuplas

Del cálculo relacional por tuplas no hablaremos demasiado. Simplemente enunciamos que es una herramienta alternativa al álgebra relacional, un poco más teórica, y que lo que diferencia los dos métodos es que, así como el álgebra relacional explica el procedimiento que hay que seguir a partir de los datos iniciales para obtener los resultados, el cálculo relacional por tuplas se limita a definir los conjuntos de tuplas que se pretenden conseguir en el resultado por medio de la lógica de predicados vista en la Sección 1.4. Estos predicados se expresan a partir de los atributos de las tuplas, constantes, y operadores de comparación. De hecho, ya se ve que el cálculo relacional por tuplas es más abstracto, ya que se limita a tratar las tuplas como elementos de los conjuntos que son las relaciones. El álgebra relacional es más cercana a la computación, ya que explica el procedimiento en su formulación.

## Dominios

Un dominio es el conjunto de valores que puede tomar un atributo.

Fíjate que los dominios no dependen de nada. En cierto sentido, matizamos aquí lo dicho en el tercer párrafo de esta misma Sección 5.2. Allá dice que las bases de datos empiezan declarando las relaciones que contienen, y para ser más finos, se debería decir que antes de crear una relación se deben crear los dominios de sus atributos. No obstante, en la mayor parte de los casos las relaciones usan dominios predefinidos.

De todas formas, queda claro que un dominio no depende de nada, y una relación depende de sus dominios. Eso convierte la definición de dominios, cuando se usan, en la primera etapa de la definición de una base de datos.

La inferencia del dominio a partir de un atributo dado no es una cuestión demasiado estricta, ya que un mismo atributo puede tener dominios más o menos restrictivos. Un nombre de persona puede tener por dominio las cadenas de caracteres, o bien puede definirse restringido a los nombres que figuren en una relación concreta. Igualmente, la edad de las personas puede implementarse en un entero, pero también en un entero restringido a ser inferior a doscientos.

Los tipos de datos primitivos de cualquier lenguaje de programación sirven para implementar cualquier dominio. Y a partir de aquí, establecer restricciones. El valor especial *nulo* pertenece a todos los dominios, y significa ausencia de valor.

Estrictamente, dentro del álgebra relacional, los tipos de los atributos no nos afectan. Aquí se parte de la existencia de conjuntos de valores que llamamos dominios,  $D_i$  para  $i = 1, \dots, n$ , siendo  $n$  el número de atributos de la relación. Cada dominio contiene los valores posibles que puede tomar cada atributo. Así pues, si llamamos  $A_i$  para  $i = 1, \dots, n$  a los atributos de una relación, entonces  $A_i \subseteq D_i$ , para  $i = 1, \dots, n$ . Nos referimos al atributo como al conjunto de valores que toma en las tuplas de la relación.

### Definición formal de relación

Llegados a este punto, se supone que se tiene una idea bien clara de lo que significa una relación en la estructura de una base de datos. Ahora bien, para poder desarrollar un álgebra se requiere rigor en la definición. Y eso nos lleva a establecer una definición de relación que se soporta única y exclusivamente en la definición de conjunto.

La definición formal de una relación utiliza la noción de esquema de la relación.

El *esquema de una relación* en el álgebra relacional es la lista ordenada de los nombres de sus atributos, aunque a menudo ponemos el nombre de la relación antes de la lista y a la expresión completa le llamamos también esquema de la relación. Como la lista de atributos se supone ordenada, debe ir entre paréntesis, tal como muestra la Caja 5.1.

$$r(A_1, A_2, \dots, A_n)$$

Caja 5.1. *Esquema de una relación en el álgebra relacional.*

Aunque sean conjuntos, a las relaciones las designaremos con minúsculas.

Los esquemas de relación son fundamentales para comprender las operaciones del álgebra relacional. Aunque parezca mentira, una relación es a un número entero lo mismo que los atributos de la relación a cada una de las cifras. Esta percepción tan desconcertante se consolidará más adelante con la operación del producto cartesiano.

En la Caja 5.2 se muestra un ejemplo de esquema de relación.

$$\text{persona}(\text{ciudad}, \text{apellidos}, \text{nombres}, \text{pasaporte})$$

Caja 5.2. *Ejemplo de esquema de relación en el álgebra relacional.*

Estrictamente, una relación es un subconjunto del producto cartesiano entre los dominios de sus atributos. No es una definición complicada. Es una forma de hablar. Por eso, veámoslo poco a poco.

Probablemente recuerdes que en el análisis matemático hay funciones de variables reales. En un espacio euclídeo representamos  $x^2$ , la típica parábola. La Figura 5.2 muestra la curva  $f(x) = x^2$ .

Pues bien. Podemos definir cada función de variable real como un subconjunto de puntos del plano, o lo que es lo mismo, como un subconjunto del plano. Es decir, siendo  $\mathbb{R}$  el conjunto de los números reales, podemos hacer referencia a la curva de la Figura 5.2 como el subconjunto  $\{(x, y) \in \mathbb{R} \times \mathbb{R} \mid y = x^2\}$ . Es extraño, ¿no?

Por otra parte, el análisis matemático determina que una función real de variable real como la de la Figura 5.2 no puede tener más de una respuesta para una  $x$  determinada, condición que la parábola satisface por descontado. Eso significa que la función

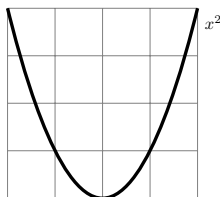


Figura 5.2: Representación de la parábola que describe la función  $f(x) = x^2$ .

representada en el plano solo puede cortar una línea vertical en un único punto, cosa que se puede comprobar desplazando horizontalmente un esfero sobre la Figura 5.2. De hecho, para representar una circunferencia hay que definir dos funciones en el plano. Una para la mitad de encima, y la otra para la de abajo.

Pero en cambio, si en rigor aceptásemos que por definición una función real de variable real es un subconjunto de puntos del plano, la restricción de que en vertical solo podemos cortar la curva en un punto no quedaría plasmada. Y por tanto, esta definición para el concepto de función real de variable real es insuficiente. O sea, podemos definir cada función con un subconjunto del plano, pero no la definición de función como concepto genérico, que hay que hacer de alguna otra manera.

Bien, en cualquier caso, traer a escena la definición de función como subconjunto de puntos del plano se justifica porque la definición formal de relación, con todo el rigor, tiene un aspecto muy parecido, tal como se puede ver a la Caja 5.3.

Dados los  $n$  dominios  $D_1, D_2, \dots, D_n$ ,

decimos que  $r = r(A_1, A_2, \dots, A_n)$  es una relación

$\Leftrightarrow$

$$r \subset D_1 \times D_2 \times \dots \times D_n,$$

siendo  $A_i \subseteq D_i$ , para  $i = 1, \dots, n$ .

Caja 5.3. *Definición formal de una relación.*

Dicho en palabras, una relación es un subconjunto del producto cartesiano de los dominios de los atributos que la componen.

Fíjate que para  $n = 1$  y  $D_1$  igual al conjunto de todos los números, la definición contempla que una relación sea tan solo un número.

### Compatibilidad entre relaciones

La compatibilidad entre relaciones hace referencia a la compatibilidad de sus esquemas. En la Sección 1.2 se dice que dos conjuntos son compatibles si uno de ellos es subconjunto del otro, y no necesariamente subconjunto propio. O sea, pueden ser iguales. La compatibilidad se denota con el símbolo  $X \sim Y$ . Es decir  $X \sim Y \Leftrightarrow X \subseteq Y \vee Y \subseteq X$ .

Los esquemas de dos relaciones son compatibles si tomando los atributos por parejas según su orden, los dominios correspondientes son compatibles.

La idea de compatibilidad de relaciones se expresa formalmente en la Caja 5.4.

Dadas dos relaciones,

$$r = r(A_1, A_2, \dots, A_n), \quad \text{siendo } A_i \subseteq D_i^A \text{ para } i = 1, \dots, n$$

$$s = s(B_1, B_2, \dots, B_m), \quad \text{siendo } B_j \subseteq D_j^B \text{ para } j = 1, \dots, m$$

decimos que  $r$  y  $s$  son compatibles

$\Leftrightarrow$

$$n = m, \text{ y } D_i^A \sim D_i^B, \text{ para } i = 1, \dots, n.$$

Caja 5.4. *Definición de compatibilidad entre las relaciones  $r$  y  $s$ .*

### 5.2.2 Visión Cartesiana de una Relación

Las coordenadas cartesianas son una buena herramienta para representar conceptos ortogonales. Ortogonal significa perpendicular. La raíz esencial de su utilidad tan extendida está en el hecho de que horizontal es perpendicular a vertical. O sea, moviéndonos en horizontal, jamás de la vida subiremos ni bajaremos. O sea, la altura en la que estamos no depende de qué movimiento horizontal hacemos, de todos los posibles. Ortogonal o perpendicular, pues, también significa independiente.

Y eso es importante. Podemos representar conceptos independientes en distintos ejes de coordenadas cartesianas, ya que la perpendicularidad de los ejes refleja la independencia de los conceptos.

Las coordenadas cartesianas tienen multitud de usos, a parte del de representar funciones. Seguidamente se muestra uno de muy interesante proveniente de la estadística, donde se utilizan los diagramas bivariantes para representar valores de variables



aleatorias que en principio son independientes. Por ejemplo, el nivel de inglés y la edad de una población.

La hipótesis que nos planteamos es,

*¿Hay alguna dependencia entre el nivel de inglés y la edad de una población?*

Entonces tomamos una muestra de 400 personas seleccionadas al azar y les preguntamos qué edad y nivel de inglés tienen con un examen que puntuamos del 0 al 100.

Todo ello lo representamos como en el Diagrama 5.1, en el cual cada persona de la muestra está representada con un punto marcado en el plano. Observa que en este momento, con esta representación, estamos haciendo uso de la supuesta independencia entre las dos variables. Si finalmente resultase que efectivamente una variable influyera en la otra, en el diagrama aparecerían distribuciones de puntos reconocibles, que en el caso más sencillo podría ser una distribución lineal.

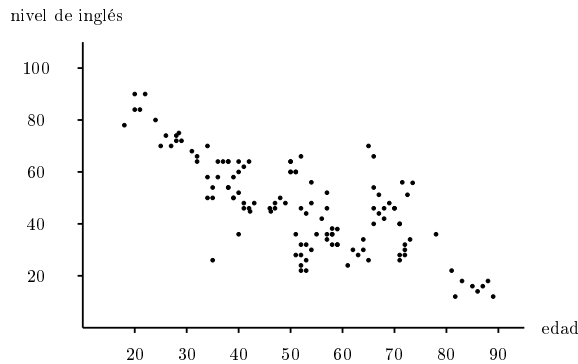


Diagrama 5.1. *Diagrama bivariante como ejemplo de uso de las coordenadas cartesianas.*

Si ahora llamamos  $x$  a la edad, e  $y$  al nivel de inglés, podríamos buscar la recta  $y = mx + n$  que ate las dos variables. Eso es encontrar las  $m$  y  $n$  tales que minimicen la desviación de las  $y_i$ 's estimadas, que valen  $mx_i + n$ , menos las  $y_i$ 's observadas. O sea, resolver

$$\min_{m,n} \sum_{i=1}^{400} (mx_i + n - y_i)^2.$$

Calculando los valores para  $m$  y  $n$  que minimicen la suma de esos cuadrados, tendremos en el valor de ese mínimo un indicador de la calidad del ajuste, de forma que si el mínimo fuera cero, entonces podríamos deducir que el nivel de inglés condiciona la edad, o viceversa. En cualquier caso, descubriríamos que las dos variables no son

independientes. Y eso es una conclusión muy importante. Es fundamental saber qué depende de qué. Por otra parte, fíjate bien que sumamos los cuadrados de las las restas (también llamadas aquí *desviaciones*) entre  $mx_i + n$  e  $y_i$ . Eso es para evitar el efecto que podría ocurrir en caso de no poner esos cuadrados. Y ese efecto es que un error podría ocultar otro de signo contrario.

En fin, a partir de regresiones como ésta, y no solo con comportamientos lineales sinó con cualquiera que plantee una hipótesis, podemos explotar el contenido de una base de datos infiriendo nuevos conocimientos, tarea que realiza la minería de datos usando métodos estadísticos, [9, 4].

Muy bien. El ejemplo de uso de coordenadas cartesianas se ha mostrado para un caso bidimensional, es decir, a partir de puntos del plano cartesiano. Y además, con dos variables cuantitativas continuas, cosa que hace que la representación sea más consistente por que cualquier punto del plano tiene una interpretación real.

De todas formas, la idea de independencia entre las variables no implica que tengan que ser cuantitativas. Y si en lugar de cualquier punto del plano solo tuvieran sentido los puntos pertenecientes a la malla de números enteros, la manera de representar la información seguiría siendo útil. Igual que en dos dimensiones, podemos representar tres en el espacio. Y para una relación sin campos numéricos, podemos utilizar el orden alfabético, como en el ejemplo que se muestra a continuación.

En la Sección 5.2.1 se ha visto que para definir una relación son necesarios previamente los dominios. La definición de la relación  $r$  se expone a la Caja 5.5.

Dados los tres dominios

$D_1 = \{A,B,\dots,Z\}$	mayúsculas del alfabeto inglés.
$D_2 = \{1,2,\dots,10\}$	enteros entre uno y diez.
$D_3 = \{\alpha,\beta,\dots,\omega\}$	minúsculas del alfabet griego.

definimos la relación

$$r = r(A_1, A_2, A_3),$$

siendo  $A_i \subseteq D_i$ , para  $i = 1, 2, 3$ .

Caja 5.5. *Definición de la relación para la visión cartesiana.*

En la Figura 5.3 se presenta una instancia de esta relación. Una instancia de una relación es el contenido que puede tener en un instante, pero también se puede entender como una traducción del inglés *instance*, que significa ejemplo.

En la Figura 5.3(a) se ha añadido una columna con un tono más claro con los nombres de las tuplas. Esta columna no forma parte de la relación. Por eso el tono claro. Sólomente se muestra para facilitar la identificación en el diagrama de la derecha.

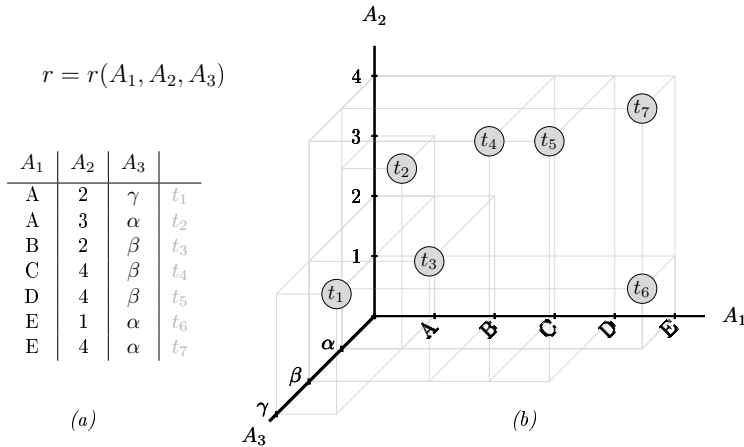


Figura 5.3: Instancia de la relación de la Caja 5.5: (a) Descripción tabular. (b) Descripción cartesiana.

Obsérvalo bien. A partir de las tuplas, que son los puntos etiquetados resigue con el dedo por las líneas hasta llegar a los ejes para comprobar que contiene la misma información que la relación.

Representaciones geométricas como la de la Figura 5.3 abren posibilidades a la hora de operar con relaciones. Eso es porque la analogía entre la tabla y la visión cartesiana es robusta. Tanto, que a partir de aquí disponemos de una herramienta con la cual reproducir gráficamente cada operación sobre relaciones que definamos en el álgebra relacional, cosa muy interesante.

Por otra parte, la limitación de no poder representar más de tres columnas de una relación no debe suponer ningún problema. Podemos comprender una distribución espacial de los datos tengan el número de componentes que tengan.

### 5.3 Modelo Relacional de una Base de Datos

El modelo relacional de una base de datos es la descripción textual más concisa de la estructura de una base de datos. La descripción de una base de datos en un modelo relacional es un punto de articulación en el proyecto. Eso significa que el camino que nos lleva de la idea inicial a la funcionalidad final debe pasar necesariamente por este

estadio. Se trata del primer enfoque automático del proyecto. Empezamos a tocar de pies en el suelo. Llegados a este punto, hay que saber hacerse una idea precisa de cómo resolver cada una de las solicitudes de la definición de requerimientos inicial mirando el modelo ER.

Venimos del modelo ER, y vamos a la implementación en lenguaje estructurado de consultas. Es decir, igual como un modelo ER se implementa en un modelo relacional en la transición de la primera a la segunda etapa del proyecto, lo que se describe en este capítulo sirve de origen a partir del cual implementaremos la base de datos en una segunda transición que irá del modelo relacional al SQL.

Tanto lo que eran entidades como relaciones M:N en el modelo ER, aquí son relaciones. Cerrada la fase de diseño, dejamos atrás el término entidad, y cambiamos el sentido del término relación.

El modelo relacional de una base de datos es, sencillamente, la lista de los esquemas de sus relaciones.

No se debe confundir con un diagrama de esquemas, que es un concepto ideado muchos años más tarde, y aquí se muestra en la Sección 5.3.5. Esta confusión es impresionantemente frecuente. A muchos alumnos les cuesta entender que un modelo sea una lista textual. Debe ser porque los estudiantes asocian modelo a diagrama, cuando es un concepto infinitamente más amplio.

El modelo relacional contribuye al desarrollo del proyecto en tres aspectos muy importantes.

- Establece orden entre los atributos de las entidades.
- Transforma tanto las entidades como las relaciones del modelo ER en un único concepto de relación del modelo relacional.
- Incorpora un orden en la colección de relaciones que constituyen la base de datos.

Eso significa que a parte de establecer un orden en los atributos de las entidades, transforma las flechas en alguna cosa textual, por un lado, y que secuencializa el diagrama por otro.

Escribiremos los nombres de las relaciones de un modelo relacional en minúsculas, como se indica en el Apéndice A. Eso nos ayudará a distinguir las de las entidades del modelo ER donde han sido originadas.

Si te fijas en el esquema de la Caja 5.2 de la página 104, observarás que no expresa la condición de clave del atributo **pasaporte**. Eso es porque la definición de esquema

de una relación pertenece al álgebra relacional, y no al modelo relacional. En el álgebra, las tuplas de las relaciones son distintas entre ellas por definición, y no saben nada de claves ni hay ningún problema de identificación. El álgebra relacional es una teoría matemática. El modelo relacional es la manera de poner en práctica la teoría. Ponemos en práctica la teoría para poder implementar bases de datos. Es en esta materialización del álgebra cuando introducimos el concepto de clave.

### 5.3.1 Atributos estructurales y atributos descriptivos

A partir del concepto de clave, dentro el modelo relacional tenemos que tanto las claves primarias como las claves foráneas son atributos *estructurales*. Los atributos que no son estructurales son atributos *descriptivos*. Es interesante discernir entre estos dos tipos de atributos porque la única redundancia que admitimos en la base de datos es con atributos estructurales. Esta redundancia no solo es admisible. Es necesaria porque es lo que efectivamente implementa la estructura, que se basa en repeticiones de valores en distintas relaciones.

Fíjate que por lo que respecta a la integridad, no hay ningún problema con eliminar de la base de datos cualquier atributo descriptivo. En cambio, según qué atributo estructural se elimine, se derrumba la base de datos completa.

### 5.3.2 Claves Primarias

Establecer una *restricción de existencia* a un atributo de una relación significa no permitir añadir tuplas a la relación si no tienen un valor en ese atributo. O sea, los atributos con restricción de existencia deben tener valores no nulos en todas sus tuplas. Y se llaman atributos *requeridos*, asumiendo que lo que es requerido es un valor en cada tupla de la relación para este atributo.

Establecer una *restricción de unicidad* a un atributo de una relación significa no permitir añadir tuplas a la relación si ya existe alguna tupla con el mismo valor en el atributo que el que se está pretendiendo insertar. Y se llaman atributos *únicos*, asumiendo que lo que es único es el valor de este atributo entre todas las tuplas de la relación.

Sobretudo cuidado. Que quede claro que no hay redundancia entre los dos conceptos. Restringir la unicidad no implica restringir la existencia. Los atributos únicos pueden contener valores nulos. Caso contrario, sería como considerar que dos valores nulos son el mismo valor. Este es un tema delicado que se ve en profundidad en la Sección 6.4.8. En cualquier caso, por lo que respecta a las restricciones de unicidad y existencia, se considera que dos valores nulos son distintos. Y por tanto, se puede establecer que en un atributo que admita valores nulos, todos los valores no nulos

sean distintos entre ellos.

Una vez definidos esos dos tipos de restricciones podemos afrontar la definición de una clave primaria, o principal, como subconjunto de los atributos de una relación sobre el cual se establecen las restricciones de existencia y de unicidad con la intención de identificar las tuplas de la relación donde residen. Obsérvese pues que esas restricciones se asocian a conjuntos de atributos, aunque en la mayoría de los casos se trate de uno solo.

Para escribir un modelo relacional a partir de un modelo ER, en los esquemas de las relaciones pondremos las claves primarias al principio de la lista de atributos, subrayadas, como en el Modelo 5.1. Haciendo eso estamos definiendo un orden parcial entre los atributos.

```
persona(pasaporte, ciudad, apellidos, nombres)
```

Modelo 5.1. *Ejemplo de esquema de relación.*

Cuando se subraya una clave formada por más de un atributo, hay que subrayar también las comas que los separan. Es una sola línea que subraya varios atributos. Eso refleja que es una sola clave primaria, como en el Modelo 5.2.

```
persona(nombres, apellidos, ciudad)
```

Modelo 5.2. *Ejemplo de esquema de relación con clave primaria binomial.*

Un atributo puede ser requerido y único, y no por esto ser clave primaria. Hace falta la intención. Fíjate que los adjetivos primaria o principal tienen cierta connotación subjetiva, de criterio. El diseñador debe sumirse al dominio del problema para establecer las claves primarias. Una submisión abyecta. El usuario final de la aplicación, que reside en el dominio del problema, sabe mejor que nadie el sistema que tiene para identificar los objetos de su dominio. Por eso, cuando el diseñador establece las claves primarias de las entidades, debe reflejar la manera como el usuario identifica los elementos, y si se da el caso, hacer aflorar las inconsistencias que pueda haber en esa identificación.

Mirándolo desde la perspectiva del análisis matemático, se entiende que la clave primaria es como una variable independiente común a una colección de funciones que son los otros atributos de la entidad. Visto así, se puede entender que los nombres

de una persona dependen del número de pasaporte, aunque parezca extraño. Realmente significa que hacemos depender los nombres del número de pasaporte, cosa más entendible. Es interesante la perspectiva analítica porque tiene continuidad en todo aquello que tiene que ver con la implementación de las relaciones del modelo ER.

En el ejemplo de la Tabla 5.1 se muestra una instancia de la entidad `PERSONA` que se verá en el Modelo 5.3 de la página 116 que se irá desarrollando para la aplicación del club deportivo introducida en la Sección 2.6.1. En esta etapa inicial, tan solo hay que observar que efectivamente los valores de la clave primaria, `pasaporte`, son todos ellos distintos. Fácil.

pasaporte	nombres	apellidos	mail
0127673812	Carmen Esmeralda	Peralta Gutiérrez	cperalta@ionos.gov
0123478937	Carlos Sayaro	Sanabria Oña	sayarona1998_2@hotmail.com
1047548338	Ana Estefanía	Sanabria Oña	anasteona1997@yahoo.ec
1145493393	Jesús Marcelo	Hortesa Orellana	rexstat143@rediris.es
CKUS01549	Michael	Bros	mbros1989@aol.com
FFJ904992	Klauss	Stallman	kstallman@dvw.tum.de

Tabla 5.1: *Instancia de la entidad PERSONA del ejemplo del Modelo 5.3.*

### 5.3.3 Claves Foráneas

En rigor, una clave foránea, o externa, es un subconjunto de atributos de una relación el dominio de los cuáles es el conjunto de valores que contiene una clave primaria de alguna otra relación de la misma base de datos.

En palabras más prosaicas: ¿Qué valores puede tomar este atributo? Los que haya allá.

En la Figura 5.4 se puede observar la idea de la implementación de una relación 1:N del modelo ER con claves foráneas en el modelo relacional.

Las claves foráneas constituyen sin duda el concepto más nucleico de la lógica de las bases de datos relacionales. Tienen la misma trascendencia a nivel estructural que las propias relaciones como concepto primogenio. Son las flechas del modelo ER implementadas en una máquina.

Es más aún, la razón por la que se definen claves primarias es para que puedan ser apuntadas desde claves foráneas. Y esto, precisamente, es lo que diferencia una clave primaria de un simple atributo requerido y único. Que la clave primaria, además de ser requerida y única, es apuntada desde otras relaciones.

Eso representa un paso de gigante en la implementación de las relaciones 1:N del

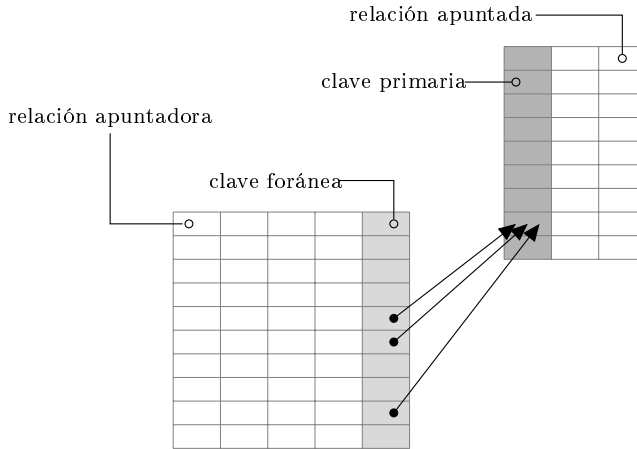


Figura 5.4: Implementación de una relación 1:N del modelo ER con claves foráneas en el modelo relacional.

modelo ER. Es importante.

## Regla 2 (de las bases de datos relacionales)

*Las claves foráneas implementan las relaciones 1:N de un modelo ER.*

Cuando una relación apunta a una otra, es decir, tiene una clave foránea hacia la otra, se dice que las dos columnas, clave primaria de una y clave foránea de la otra, son columnas *vinculadas*. Se recomienda fuertemente que se llamen igual para poder sacar el máximo provecho de las operaciones que más adelante se verán.

Observa que la Regla 2 es consecuencia inmediata de la Regla 1, que decía que las tuplas tienen una cantidad fija de columnas, ya que gracias al 1 de las relaciones 1:N no es necesario más que un atributo adicional a la entidad del lado N para representar las relaciones 1:N del modelo ER, y por tanto, la cantidad de atributos sigue siendo fija para todas las entidades. Y no solo esto, sino que jugando con las restricciones que imponemos al atributo que es clave foránea podemos traducir más de un concepto de los vistos en el Capítulo 4.

- Una restricción de existencia en una clave foránea implementa una participación total en la relación del modelo ER.
- Una restricción de unicidad en una clave foránea implementa una relación 1:1 del modelo ER en lugar de una 1:N.

Atención al hecho de que la entidad del lado 1 de la relación no resulta modificada en nada por implementar una relación 1:N. Es decir, ni se entera que alguien la está



apuntando. Igual que una, podrían estar apuntándola cincuenta relaciones que esto no modificaría su estructura. Com tú mismo, que tus nombres no varían por mucha gente que hable de ti.

En los esquemas de las relaciones, las claves foráneas se colocan al final de la lista de atributos, y siempre se llamarán con el mismo nombre que la clave primaria a la que apuntan.

Refinamos pues el orden parcial de los atributos en los esquemas que hemos empezado a establecer en la sección anterior. Así como las claves primarias las subrayamos, a las claves foráneas no. Por tanto, el aspecto de las claves foráneas en el esquema de la relación hace que se confundan con atributos descriptivos. No obstante, se pueden diferenciar porque el nombre de las claves foráneas en una relación coincide con el nombre de alguna clave primaria de otra relación de la base de datos.

Cuando en una relación no hay clave primaria, las claves foráneas se ponen al principio de su esquema.

Todo ello hace que la relación mostrada en la Tabla 5.1 no fuese completa. Faltaba la clave foránea que sí se muestra en la Tabla 5.2.

pasaporte	nombres	apellidos	mail	ciudad
0127673812	Carmen Esmeralda	Peralta Gutiérrez	cperalta@ionos.gov	Babahoyo
0123478937	Carlos Sayaro	Sanabria Oña	sayarona1998_2@hotmail.com	tena
1047548338	Ana Estefanía	Sanabria Oña	anasteona1997@yahoo.ec	tena
1145493393	Jesús Marcelo	Hortesa Orellana	rexstat143@rediris.es	Machala
CKUS01549	Michael	Bros	mbros1989@aol.com	San Francisco
FFJ904992	Klauss	Stallman	kstallman@dvw.tum.de	Berlín

Tabla 5.2: *Instancia completa de la entidad PERSONA del ejemplo del Modelo 5.3.*

### 5.3.4 Transformación del Modelo ER al Modelo Relacional

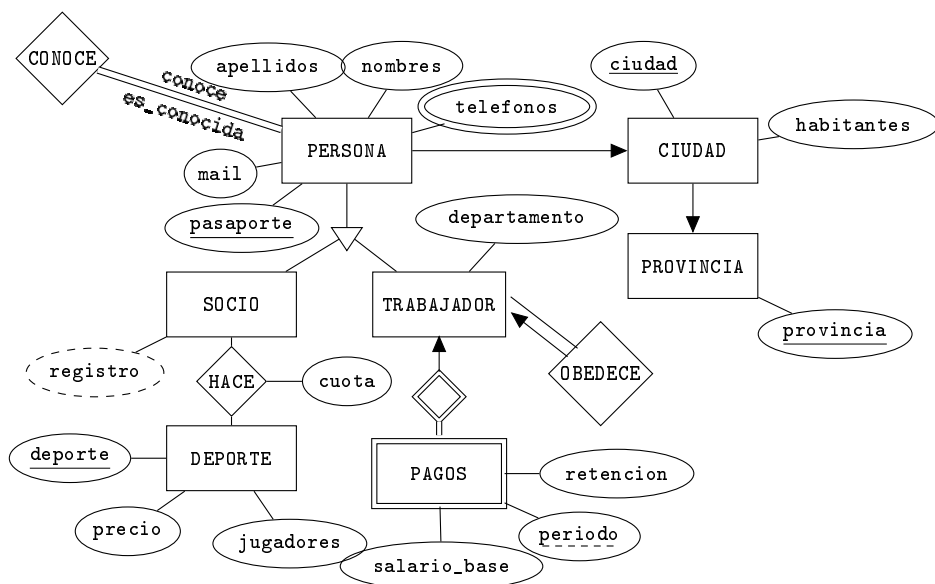
El orden que un modelo relacional impone en la colección de esquemas de relaciones que lo componen, es un orden parcial. Eso es, a partir de un mismo modelo ER, hay distintas ordenaciones válidas para listar las relaciones del modelo relacional correspondiente.

En términos estrictamente técnicos, hay que obtener alguna ordenación topológica del grafo dirigido que es el modelo ER. Una ordenación topológica de un grafo acíclico dirigido es una secuenciación de los nodos tal que cualquier predecesor aparece en la secuencia antes que sus sucesores. En general tenemos distintas posibilidades para un mismo grafo. Y si todo eso cuesta de entender, entonces solo hay que mantener el principio del orden parcial:

*Nomás se puede apuntar a relaciones que ya existan.*

Eso respecto al orden de declaración de las relaciones. Mantener el principio hará que antes definamos las entidades que las relaciones del modelo ER. Y primero, las entidades apuntadas que las apuntadoras. Por lo tanto, es razonable empezar por las entidades de soporte.

El diseño del Modelo 5.3 es para la base de datos del club deportivo introducida en la Sección 2.6.1. Gestiona los datos de una serie de personas que son socios del club o bien trabajadores. Es necesario el número de pasaporte de las personas para poderlas registrar en la base de datos. Además de los nombres, los apellidos, y la dirección de correo electrónico, de las personas también se guardan varios teléfonos para cada una de ellas. En qué ciudad viven, cuántos habitantes tiene esa ciudad, y la provincia donde se encuentre. Por otra parte, la base de datos también es capaz de establecer amistades entre las personas que guarda. Pueden ser socios, o bien trabajadores, o las dos cosas a la vez. Para los socios se guarda automáticamente la fecha de registro. Y para los trabajadores se guarda el departamento, que puede ser *administración*, *comercial*, o *entrenador*, así como los roles de pago que les efectúa el club.



Modelo 5.3. Modelo ER para la aplicación del club deportivo.

De los deportes se conoce el número de jugadores necesarios para formar un equipo y el precio mensual. De cada socio, se guardan sus deportes y qué cuota paga por cada uno, que puede variar y prevalece respecto el precio del deporte. Para cada trabajador se guardan los datos de su jefe, así como el historial de los roles de pago. A los roles de pago los llamaremos simplemente *pagos* asumiendo que una vez metidos

en el dominio de la aplicación, eso no introducirá confusión alguna. Un pago viene identificado por el trabajador a quien corresponde más un periodo que contiene el mes y el año. Los valores posibles de este atributo son fechas. La base de datos guarda la fecha de cada pago a cada trabajador. Un pago es un salario base y una retención, que es un porcentaje.

Muy bien. A continuación se hará la transformación del diseño del Modelo 5.3 a un modelo relacional.

Empezamos.

De entrada, los esquemas de las entidades que sean apuntadas pero no apunten a ninguna otra entidad. Después, las que apunten a las primeras, y así ir haciendo. Por tanto, podemos empezar perfectamente por la relación *provincia*, y luego *ciudad*, tal como se muestra en el Modelo 5.4.

```
provincia(provincia)
ciudad(ciudad,habitantes,provincia)
...
```

Modelo 5.4. *Primeras relaciones de la versión relacional del ejemplo 5.3*

El hecho de que el atributo *provincia* de la relación *ciudad* se llame como la clave primaria de otra relación significa que es una clave foránea que la apunta. Por tanto, las tuplas de la relación *ciudad* deben tener como valores posibles del atributo *provincia* los que existan en la clave primaria de la relación apuntada, es decir, los valores del atributo *provincia* de la relación *provincia*.

Ahora podemos seguir con la traducción de la entidad *PERSONA*, ya que no apunta a ninguna relación no definida ya.

```
provincia(provincia)
ciudad(ciudad,habitantes,provincia)
persona(pasaporte,nombres,apellidos,mail,ciudad)
...
```

Modelo 5.5. *Incorporación de la relación persona.*

Observa en el Modelo 5.5 dos ausencias notables. La autorelación *CONOCE*, que al ser M:N, provocará una nueva relación en el modelo relacional, y por eso no la tenemos en

cuenta en este momento. Luego, los atributos multivalorados como `telefonos` tampoco se consideran en el esquema de esa relación. Sin embargo, lo hacemos seguidamente, puesto que solo dependen de `PERSONA`.

```

provincia(provincia)
ciudad(ciudad,habitantes,provincia)
persona(pasaporte,nombres,apellidos,mail,ciudad)
telefonos(pasaporte,telefono)
...

```

Modelo 5.6. *Transformación de un atributo multivalorado.*

Las tuplas de la relación `telefonos` nos dirán que tal persona tiene tal teléfono. El número de pasaporte de una persona aparecerá tantas veces como teléfonos tenga.

Observando el Modelo 5.6 tenemos la primera relación que aparece en un modelo relacional y no proviene de una entidad del modelo ER. Cosa nueva. Por eso no tiene clave primaria, porque no es una entidad. En cambio, sí que tiene una clave foránea, `pasaporte` que puede tomar como valores los números de pasaporte de las personas que haya a la base de datos. Fíjate que cuando en una relación no hay clave primaria, las claves foráneas se ponen al principio de su esquema.

Como no se permitirá que el mismo teléfono pertenezca a dos personas, aunque en el Modelo 5.6 no se muestre, será necesaria una restricción de unicidad para este atributo. Además, un registro en la tabla `telefonos` que no contenga un valor en la columna `telefono` no tiene sentido. Así pues, `telefono` es un atributo requerido y no repetido, pero no por eso clave primaria, ya que no se apunta desde ningún lugar.

La transformación de la autorelación `CONOCE` sigue desarrollando el modelo relacional como se indica en el Modelo 5.7. Como en el caso anterior, la relación `conoce` tampoco incluye clave primaria porque no es una entidad.

```

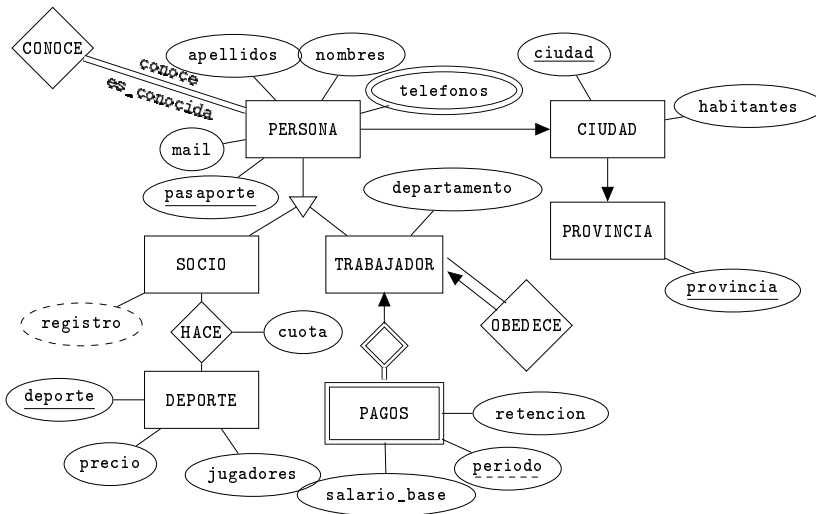
provincia(provincia)
ciudad(ciudad,habitantes,provincia)
persona(pasaporte,nombres,apellidos,mail,ciudad)
telefonos(pasaporte,telefono)
conoce(conoce,es_conocida)
...

```

Modelo 5.7. *Transformación de una autorelación M:N al modelo relacional.*

Atención, las claves foráneas de *conoce* violan la norma de que las claves foráneas se deben llamar con el nombre de la clave primaria de la relación a la que apuntan. Está justificado. Son roles. Para las autorelaciones M:N haremos esa excepción. Los dos atributos de la relación *conoce* son claves foráneas que apuntan a la relación *persona*. Cada tupla de esta relación nos dirá tal persona conoce tal otra. Y para decirlo utilizará los números de pasaporte. Una persona que aparezca muchas veces a la izquierda es una persona que conoce a mucha gente. Una persona que no aparezca ninguna vez a la derecha significa que es una persona no conocida por nadie, la pobre.

Llegados a este punto, ya se ha transformado buena parte del Modelo 5.3, que se repite en el Modelo 5.8 para agilizar el procedimiento en el que estamos inmersos.



Modelo 5.8. Modelo ER para los ejemplos, repetición del 5.3.

Ataquemos la especialización. Para transformarla al modelo relacional, añadimos a cada una de las entidades especializadas una clave foránea que apunte a la clave de la entidad genérica. Además, atención, esta clave foránea también es clave principal. Un fenómeno que no habíamos visto hasta ahora, y que es la manera de implementar una especialización o generalización. Total, que añadimos las dos entidades como se puede ver en el Modelo 5.9.

```

provincia(provincia)
ciudad(ciudad,habitantes,provincia)
persona(pasaporte,nombres,apellidos,mail,ciudad)
telefonos(pasaporte,telefono)
conoce(conoce,es_conocida)
socio(pasaporte,registro)
trabajador(pasaporte,departamento,obedece)
...

```

Modelo 5.9. *Transformación de entidades especializadas y de autorelación 1:N.*

De las dos incorporaciones del Modelo 5.9 se desprenden tres comentarios.

- Como se puede observar, el hecho de que el atributo **registro** de la relación **socio** sea calculado no impacta en el modelo relacional. Se trata como un atributo cualquiera. El hecho de ser calculado, pues, también debe constar en la documentación del modelo relacional.
- Por otra parte, el hecho de que los atributos **pasaporte** de las relaciones correspondientes a las entidades especializadas sean claves primarias, además de foráneas, garantiza la unicidad de estos atributos. Están pues implementando relaciones 1:1, como debe ser, en lugar de implementar relaciones 1:N.
- También es notable la aparición de la clave foránea **obedece**, apuntando a la misma relación **trabajador**. Contendrá pues el número de pasaporte del trabajador que sea el jefe del que representa la tupla donde se encuentre. Resulta gratificante que las autorelaciones 1:N, igual que hemos visto con las M:N, se implementan exactamente igual que las relaciones entre entidades distintas.

La relación **HACE** es M:N. O sea, como dos relaciones 1:N. Por tanto, apunta a **socio** y a **DEPORTE**. Eso significa que antes de describir el esquema de la relación **HACE** hay que escribir el de la entidad **DEPORTE** tal como dice el principio del orden parcial. Nomás se pueden apuntar a relaciones existentes. Total, que la transformación continua con dos relaciones más, tal como se muestra en el Modelo 5.10.

```

provincia(provincia)
ciudad(ciudad, habitantes, provincia)
persona(pasaporte, nombres, apellidos, mail, ciudad)
telefonos(pasaporte, telefono)
conoce(conoce, es_conocida)
socio(pasaporte, registro)
trabajador(pasaporte, departamento, obedece)
deporte(deporte, precio, jugadores)
hace(pasaporte, deporte, cuota)
...

```

Modelo 5.10. *Transformación de relaciones M:N previa descripción de las entidades relacionadas.*

Por definición de la estructura, las dos claves foráneas de la relación **hace** deberían ser requeridas, ya que no tiene sentido un elemento en esta relación si no apunta a una pareja de valores existentes. Es normal cuando dos elementos de dos entidades con una relación M:N se relacionan, que no tenga sentido la duplicidad del elemento de la relación. O sea, el dato correspondiente a que un socio hace un deporte queda registrado solo una vez. Eso es especialmente cierto cuando no hay atributos en la relación. Entonces, a menudo se declara la pareja de claves foráneas de la relación **hace** como una clave primaria binomial. Estrictamente no es correcto, ya que tan solo se hace así para restringir la unicidad y la existencia. Y para conseguir eso no hay que declararlas como clave primaria. Por otra parte, como haciéndolo así no es menos eficiente ni ocupa más espacio, también es admisible. Pero entonces, hay que tener muy claro que se está haciendo este uso alternativo del concepto de clave primaria. El uso genuino de las claves primarias es el de ser apuntadas.

La relación **deporte** no requiere ningún comentario adicional. Fíjate que al no apuntar ninguna otra relación en el diagrama se hubiera podido poner al principio del modelo relacional sin ningún problema. La relación **hace** nos viene a decir, para cada tupla, el socio tal paga tal cuota por tal deporte. A partir de eso supondremos que el socio practica el deporte, aunque la realidad sea tan distinta. No tiene clave principal porque no proviene de ninguna entidad, y contiene dos claves foráneas a socio y deporte.

Atención al tema de las participaciones totales. Así como implementar una participación total de una entidad en una relación 1:N es tan sencillo como hacer requerida la clave foránea, para el caso de las M:N no es tan fácil. La única manera que tenemos de garantizar una participación total de una entidad en una relación M:N es por medio de lógica. O sea, añadiendo programas especializados en la base de datos que no permitan insertar un elemento en la entidad si no se inserta también algún elemento en la relación que lo involucre.

Finalmente añadimos la entidad débil `PAGOS`. El hecho de que la entidad identificadora sea `TRABAJADOR` significa que no podemos hablar de un pago si no sabemos de qué trabajador es. Y el discriminante, `periodo`, servirá para distinguir entre los pagos de un mismo trabajador. Observa que para implementar entidades débiles se produce un fenómeno que hasta ahora no había aparecido. Una clave foránea forma parte de una clave primaria. Es el caso del atributo `pasaporte` de la tabla `pagos`.

En el Modelo 5.11 se muestra la versión relacional del diseño del Modelo 5.3.

```

provincia(provincia)
ciudad(ciudad,habitantes,provincia)
persona(pasaporte,nombres,apellidos,mail,ciudad)
telefonos(pasaporte,telefono)
conoce(conoce,es_conocida)
socio(pasaporte,registro)
trabajador(pasaporte,departamento,obedece)
deporte(deporte,precio,jugadores)
hace(socio,deporte,cuota)
pagos(pasaporte,periodo,salario_base,retencion)

```

Modelo 5.11. *Modelo Relacional del diseño del Modelo Entidad Relación 5.3.*

La extensión de relaciones binarias M:N a relaciones ternarias no tiene más secreto. En lugar de dos, serían tres claves foráneas. Dependiendo de las restricciones de existencia o de unicidad de cada una de las claves controlaríamos las cardinalidades de la relación. Se ha dejado atrás la inclusión de pavellones para agilizar todo el procedimiento, considerando que a parte del hecho de ser tres claves foráneas, no hubiera introducido ningún concepto nuevo.

Igualmente se ha abandonado el concepto de atributos compuestos porque se implementan solo los atómicos ignorando el nombre del atributo compuesto.

Bien, una vez desmenuzado el procedimiento de traducción a partir del diseño hacia al modelo relacional, veamos una sinopsis de como han quedado las claves de las relaciones del modelo relacional dependiendo del tipo de entidades en el modelo ER que las han originado.



modelo ER	claves del modelo relacional
entidad	clave primaria en un solo atributo
relación 1:N	clave foránea
relación M:N	pareja de claves foráneas
atributo multivalorado	clave foránea y ninguna clave primaria
especialización	la clave primaria es clave foránea
entidad débil	clave primaria formada por dos atributos, el primero de los cuales es clave foránea

Tabla 5.3: *Sinopsis para la transformación del modelo ER al modelo relacional.*

### 5.3.5 Diagrama de Esquemas de una Base de Datos

Los diagramas de clase utilizados en la programación orientada a objetos resultan unas herramientas de gran utilidad. Por otra parte, hay un isomorfismo bastante intuitivo entre una relación en una base de datos y una clase de objetos en la programación con lenguajes de alto nivel.

Como en todo, para comprender el impacto de cada definición conviene conocer la historia. La programación orientada a objetos es de los años noventa. El modelo ER de los setentas. O sea, tanto el modelo ER como el modelo relacional son considerablemente anteriores a los diagramas de esquemas. Por eso, en la descripción de las etapas del proyecto, no consideramos el diagrama de esquemas como un paso imprescindible.

El diagrama de esquemas de una base de datos se sitúa en un punto intermedio entre el modelo ER y el modelo relacional. Este tipo de diagramas aparecieron con el Access de Microsoft a principios de los noventa, en lo que fue la primera interfaz gráfica de aplicación para bases de datos, como se dice en la Sección 2.1.1. A pesar de tener graves lagunas en la capacidad de representación su éxito ha sido notable. La prueba es que actualmente existen aplicaciones de interfaz gráfica para todos los SGBDs, a pesar que son programas absolutamente independientes del gestor.

Un ejemplo de un esquema de relación, como se describía en el Modelo 5.1, era `persona(pasaporte, nombres, apellidos, mail, ciudad)`. Bien, pues si tenemos claro lo que es un esquema de relación, entonces un diagrama de esquema de relación es lo mismo, pero puesto dentro una caja en vertical. El nombre de la relación se pone arriba del rectángulo como título, y debajo, en formato lista, los atributos empezando por las claves primarias en negrita. El mero hecho de utilizar negrita refleja que es una herramienta posterior a las otras. O sea, a pesar ser un instrumento para la documentación entre humanos, nació luego de las aplicaciones de interfaz. El aspecto de un diagrama de esquema de relación entendido de forma genérica se muestra en el Diagrama 5.2.

Diagrama 5.2. *Ejemplo de diagrama de esquema de relación.*

Con estos diagramas no hay una convención lo bastante extendida sobre la sintaxis

persona
<b>pasaporte</b>
nombres
apellidos
mail
ciudad

como para poder establecer las decisiones del diseño con el rigor que sí permite un modelo ER o el mismo modelo relacional. Es decir, que esto de la negrita tampoco es para poner las manos al fuego.

Hay que entender pues que un diagrama de esquemas de relación no es una herramienta estrictamente necesaria, ya que durante veinte años se hicieron bases de datos sin usarlos. Como se decía en los primeros párrafos de este capítulo, un modelo ER es una descripción orientada a los humanos, y un modelo relacional es el punto de partida para hacerle comprender a la computadora. Un diagrama de esquemas es una herramienta auxiliar que no tiene por qué aparecer en la documentación del proyecto, aunque cuando se pretenden exponer algunas condiciones sin demasiado rigor, efectivamente resulte una herramienta bastante útil.

Está bien claro que la nomenclatura utilizada en los modelos ER tiene el valor que tiene mayormente por el hecho de estar conveccionalmente admitida. Eso es una característica propia de cualquier lenguaje humano. Sin embargo, no hay una organización internacional que imponga el uso de ninguna simbología concreta. Eso ha provocado que todo aquello más complicado de representar gráficamente haya sido contaminado de alternativas que han tenido más o menos éxito. Aquéllos que alguna vez hayan programado aplicaciones de diseño gráfico, se habrán encontrado que la complejidad de dibujar una flecha es superior a la de dibujar un texto. Eso es así porque un texto siempre se escribe igual, en cambio la forma que tiene una flecha depende de la orientación de la línea sobre la que se soporte. Eso mismo pasa con el sistema de Microsoft Acces. De manera que los diseñadores de estas aplicaciones se toman la libertad de redefinir los símbolos que resulta complicado dibujar automáticamente introduciendo dialectos al modelo ER que luego requiere un esfuerzo, menor pero adicional, para poder interpretar.

Es frecuente en muchas aplicaciones de interfaz que en lugar de flechas se limiten a escribir textualmente las cardinalidades al lado de las líneas que unen las entidades.

Todo ello desemboca en una eclosión de distintas versiones de modelos ER. Hay quien para declarar una cardinalidad como 1:N, por ejemplo, no pone flecha en el lado 1, y en lugar de esto se pone una especie de tenedor al lado N.

Bien, tampoco es la intención de este libro adoctrinar sobre la manera de describir los diseños de bases de datos, de forma que las distintas simbologías que se utilizan se deben considerar todas correctas. La que aquí se ha mostrado es la primera que

se difundió. Ahora bien, tan extendidas están las formas alternativas, que no hay motivos para considerarlas incorrectas, ya que su proliferación es una demostración de su utilidad.

Sin más explicaciones, en el Diagrama 5.3 se puede ver el diagrama de esquemas de relación del diseño del Modelo 5.3. El Microsoft Acces lo haría parecido, pero con  $\infty$ 's en los lugares donde empiezan y 1's en los lugares donde hay las puntas de las flechas.

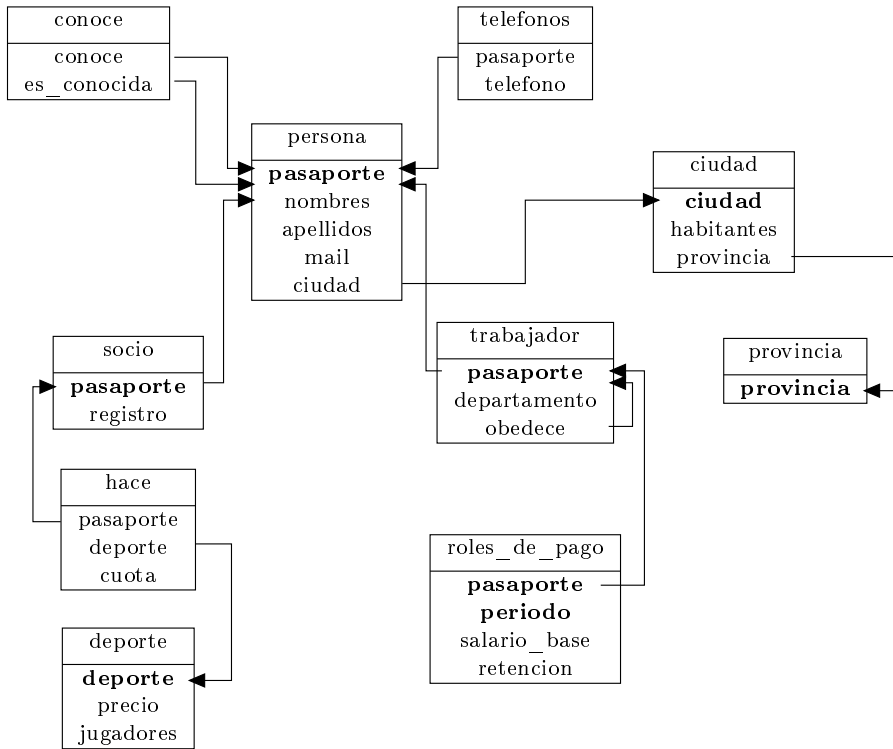


Diagrama 5.3. Diagrama de esquemas de relación del Modelo ER del ejemplo.

## 5.4 Operaciones Básicas con Relaciones

Hasta aquí se ha visto la manera de almacenar los datos. Con relaciones, claves primarias y claves foráneas.

De aquí en adelante se expone la manera de calcular las relaciones resultantes de las solicitudes que se puedan plantear. Dar respuestas, obtener expresiones relacionales.

Decimos que una operación es básica cuando el resultado que obtiene no se puede obtener por medio de ninguna otra operación. Son seis. La *selección*, que obtiene la relación formada por las tuplas de la relación dada que cumplen una condición. La *proyección* nos retorna una relación formada por las columnas de la relación de entrada que se le pidan. El *producto cartesiano*, que nos retorna una relación con todas las posibles parejas entre tuplas de las dos relaciones de entrada. La *unión*, que agrupa las tuplas de dos relaciones y nos devuelve una. La *diferencia*, que elimina todas las tuplas de la primera relación de entrada que estén en la segunda. Y el *renombramiento*, que permite cambiar el nombre y los nombres de los atributos a cualquier expresión relacional.

En la Figura 5.5 se presenta una nueva relación *persona* para los ejemplos que se verán en las próximas secciones.

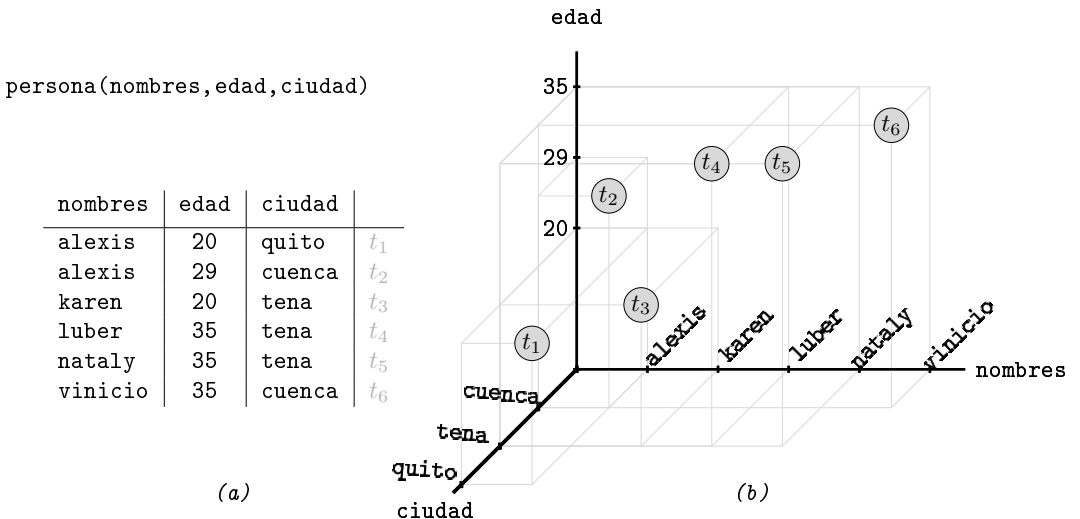


Figura 5.5: Relación *persona* de los ejemplos. (a) Descripción tabular. (b) Descripción cartesiana.

### 5.4.1 Operación de Selección

La nomenclatura utilizada para la operación de selección se indica en la Caja 5.6,

$$\sigma_p(r)$$

Caja 5.6. *Expresión relacional de una selección.*

donde  $p$  es un predicado que se le proporciona como argumento, y  $r$  es la relación de entrada. Esta expresión se pronuncia "sigma sub pe de erre" o simplemente "sigma pe de erre", y usa la letra ese griega como mnemotécnico de "selección".

Retorna la relación formada por las tuplas de la relación de entrada que satisfacen el predicado dado. El esquema de la selección es el mismo que el de la relación que se le da. Esta operación sirve para establecer criterios.

**ejemplo 5.1.** *Obtener la relación de personas que viven en el tena.*

**solución**  $\sigma_{\text{ciudad}='tena'}(\text{persona})$

En la Figura 5.6 se ilustra la solución del ejemplo 5.1. A la izquierda, Figura 5.6(a), se han dejado en tono más claro las tuplas excluidas de la solución.

Seguramente, lo más interesante de la visión cartesiana es la analogía entre el predicado  $\text{ciudad} = 'tena'$  y el subespacio formado por este mismo plano. De manera que las tuplas seleccionadas, y que por tanto forman parte de la relación resultante, son las que se ubican en el plano definido por el predicado. Eso es muy gráfico. Fíjate que el hecho que el predicado esté formado por tan solo una proposición, hace que reduzcamos el espacio de la relación en una dimensión.

A continuación, el caso de un predicado conjuntivo de dos proposiciones.

**ejemplo 5.2.** *Personas de 35 años que viven en el tena.*

**solución**  $\sigma_{\text{ciudad}='tena' \wedge \text{edad}=35}(\text{persona})$

En la Figura 5.7 se dibuja la solución del ejemplo 5.2

Estamos reduciendo mucho más el espacio de selección. De un plano pasamos a una línea. La línea de intersección entre los dos planos dados por las proposiciones, por tanto, en la selección resultante tan solo aparecerán las tuplas que se ubiquen en esa línea. Como se puede ver, el subconjunto de tuplas seleccionadas consta de las

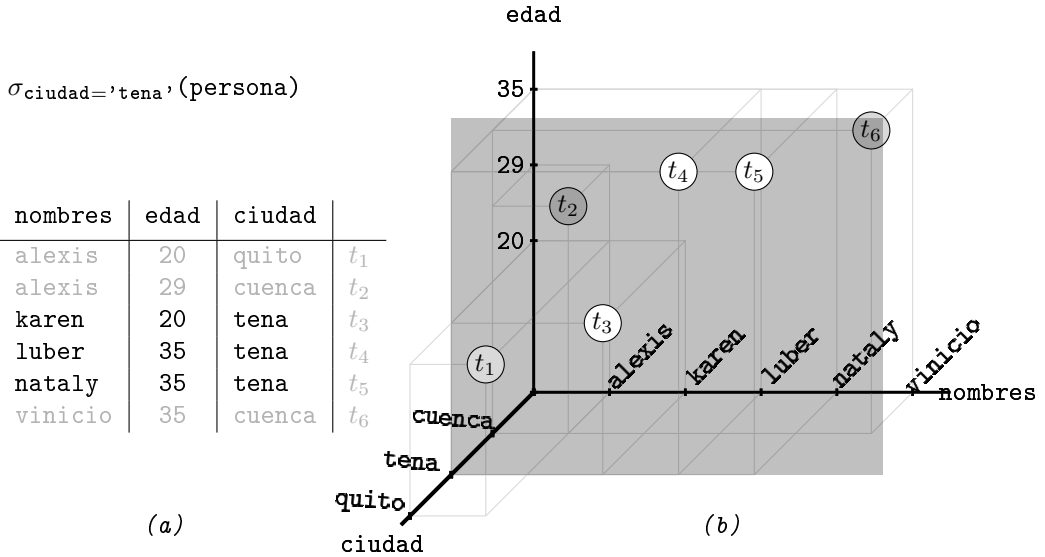


Figura 5.6: Selección de personas de tena. (a) Descripción tabular. (b) Descripción cartesiana.

dos tuplas,  $t_4$  y  $t_5$ .

Todo ello está muy relacionado con el análisis matemático y la resolución de ecuaciones lineales. Es fácil imaginarse predicados consistentes en relaciones vacías. Si los dos planos de la Figura 5.7(b) fueran paralelos, y no perpendiculares como ahora, entonces no habría intersección entre ellos, y por tanto el resultado de la selección sería el conjunto vacío, bien, una relación vacía. Pero claro, ¿Qué significa que los dos planos sean paralelos? Pues significa que pedimos una cosa imposible, como por ejemplo personas que tengan 35 y 29 años al mismo tiempo, o mejor dicho, en la misma tupla. Imposible.

Si en cambio consideramos el predicado disyuntivo, con la unión en lugar de la intersección, personas que tengan 35 años o que vivan en el *tena*, entonces las tuplas seleccionadas no serán solo las de la línea de intersección entre los dos planos, sino cualquier tupla que se ubique en uno u otro plano. Eso se muestra en la Figura 5.8, donde se puede observar que la relación resultante consta de cuatro tuplas. Además de las dos de la intersección han sido seleccionadas  $t_3$  por ser del *tena*, y  $t_6$  por tener 35 años.

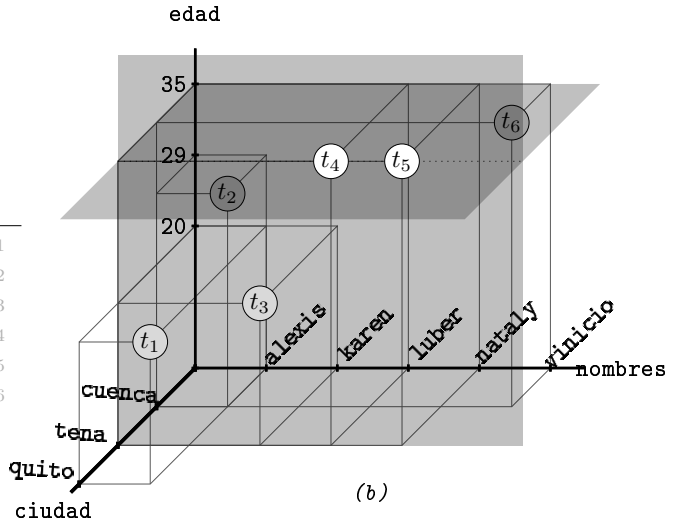
### 5.4.2 Operación de Proyección

El símbolo del operador de proyección se expone en la Caja 5.7,

$$\sigma_{\text{ciudad}='tena' \wedge \text{edad}=35}(\text{persona})$$

nombres	edad	ciudad	
alexis	20	quito	$t_1$
alexis	29	cuenca	$t_2$
karen	20	tena	$t_3$
luber	35	tena	$t_4$
nataly	35	tena	$t_5$
vinicio	35	cuenca	$t_6$

(a)



(b)

Figura 5.7: Selección de personas de 35 años, del tena. (a) Descripción tabular. (b) Descripción cartesiana.

$$\Pi_{A_1, A_2, \dots, A_k}(r)$$

Caja 5.7. Expresión relacional de una proyección.

siendo  $A_1, A_2, \dots, A_k$  un subconjunto de los atributos de  $r$  que es la relación de entrada a la operación. Esta expresión se pronuncia "pi sub a uno, a dos, a ca, de erre", y usa la letra pi griega como mnemotécnico de "proyección".

De la relación de entrada, el resultado de una proyección tan solo contiene los atributos presentes en el argumento que se le da. Por tanto, la proyección tiene por esquema exactamente el argumento. Esta operación sirve para especificar los atributos resultantes de una consulta.

En principio, la relación resultante tendrá tantas tuplas como tenga la relación dada. No obstante, hay la posibilidad de que las tuplas de la relación de entrada se diferencien entre ellas en campos que no son pedidos en el argumento. Entonces, en la relación de salida serían tuplas iguales. Si esto sucediera el nombre de elementos de la relación resultante se vería reducido.

**ejemplo 5.3.** *Obtener la edad y la ciudad de todas las personas.*

**solución**  $\Pi_{\text{edad}, \text{ciudad}}(\text{persona})$

En la Figura 5.9 se puede observar de manera gráfica la solución del ejemplo 5.3.

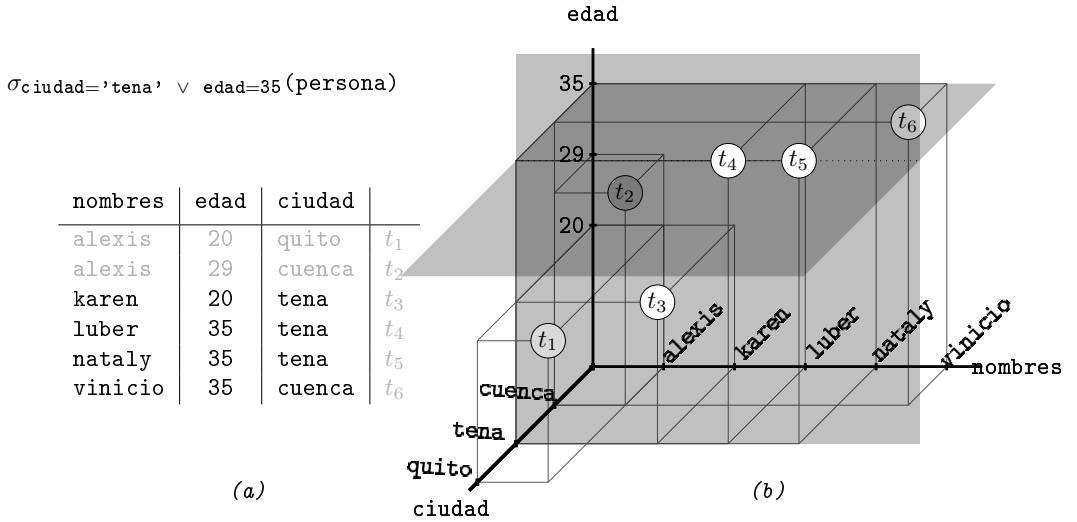


Figura 5.8: Selección de personas de 35 años, o de tena. (a) Descripción tabular. (b) Descripción cartesiana.

La tupla  $t_4$  ha desaparecido. El atributo que la diferenciaba de  $t_5$  no ha sido proyectado, y por tanto en la relación resultante  $t_4 = t_5$ . El ejemplo hubiera sido igual de válido si hubiera desaparecido  $t_5$ .

Fíjate también, a partir de la Figura 5.9(b), que para imaginar lo que hace la proyección conviene imaginarse la relación inicial de la Figura 5.5 enfocada desde la derecha, perpendicular al plano (edad,ciudad). Las sombras que dibujan las tuplas iniciales en esa pared son la relación resultante. De aquí el nombre de proyección.

Bien, todo esto habiendo proyectado dos atributos de los tres de la relación inicial. Desgranemos el caso de un solo atributo.

**ejemplo 5.4.** *Obtener las edades de todas las personas.*

**solución**  $\Pi_{\text{edad}}(\text{persona})$

En este caso, deberíamos imaginar que enfocamos desde delante un pelo a la izquierda la Figura 5.9(b), de manera que todo quedase concentrado contra el eje vertical. Las sombras finales marcarían los distintos valores que hay para la edad en la relación inicial.

En la Figura 5.10 se manifiesta este caso. Han desaparecido varias tuplas. Queda nomás un representante de cada edad.

Una característica importante de la proyección es que permite hacer reordenaciones en los esquemas de las relaciones. Por ejemplo, la relación



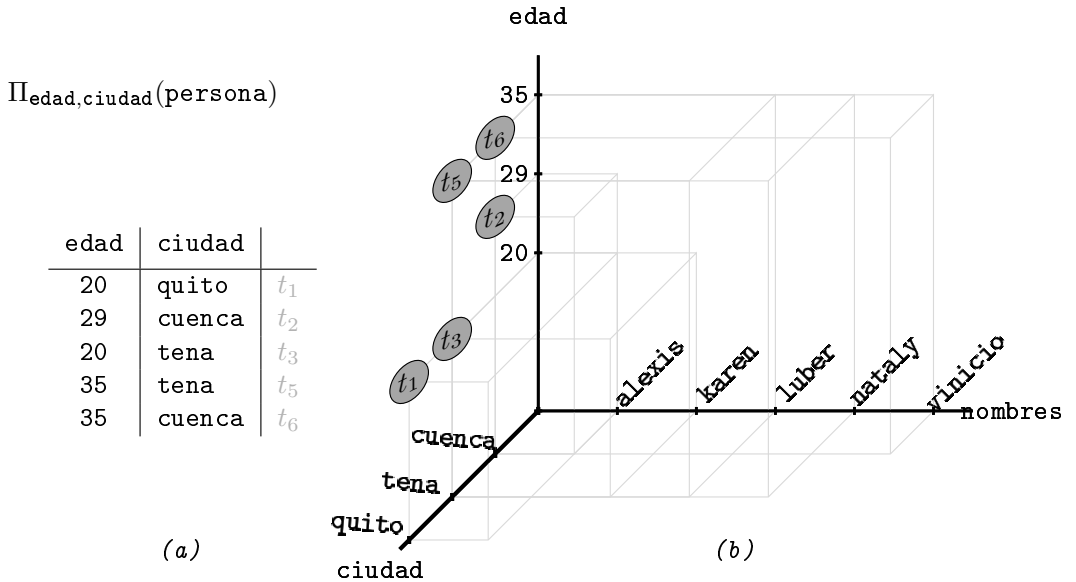


Figura 5.9: edad y ciudad de las personas. (a) Descripción tabular. (b) Descripción cartesiana.

$$\Pi_{\text{ciudad,nombres,edad}}(\text{persona})$$

es la misma que persona, pero con los atributos en otro orden.

### Composición de operaciones

Llegados a este punto podemos resolver consultas más complicadas. Eso nos particiona los datos en dos grupos. Los valores que sirven para establecer criterios, y los nombres de atributo que deben encabezar las columnas de la relación resultante, o sea su esquema.

**ejemplo 5.5.** *Obtener las edades y las ciudades de las personas que se llaman alexis.*

**solución**  $\Pi_{\text{edad,ciudad}}(\sigma_{\text{nombres}='alexis'}(\text{persona}))$

En la visión cartesiana del ejemplo 5.5 impresa en la Figura 5.11 hay dos etapas. Primeramente, se seleccionan las tuplas del plano  $\text{nombres} = \text{'alexis'}$ , que aparecen en tono más claro. Y luego, se proyectan al plano  $(\text{edad,ciudad})$ . En total, quedan las dos tuplas que aparecen en blanco, aunque sombreado por el plano, en la Figura 5.11(b), que son  $t_1$  y  $t_2$  tal como se indica en la descripción tabular de la Figura 5.11(a).

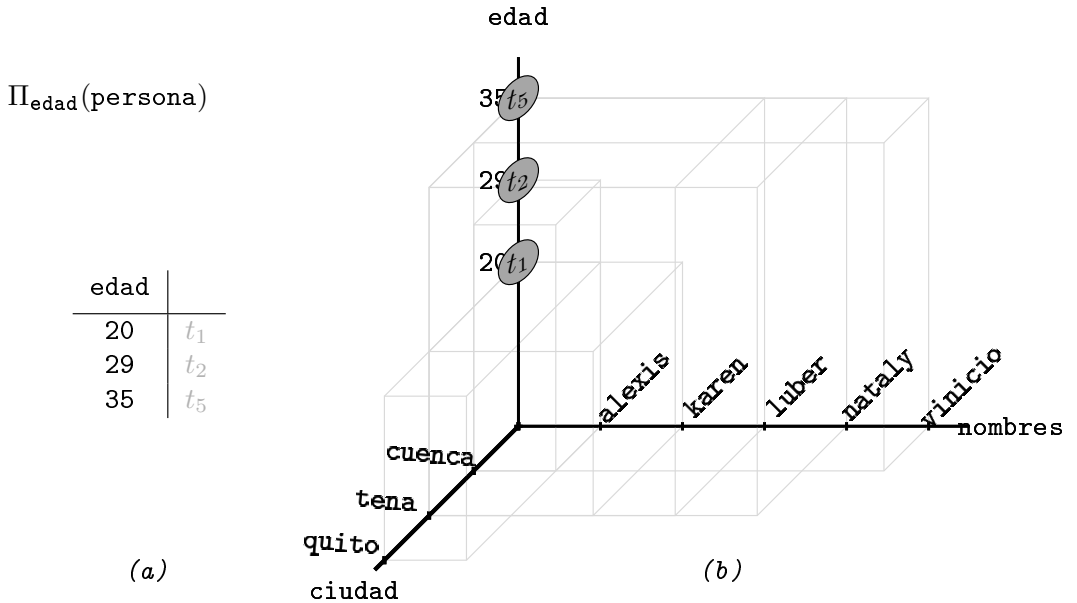


Figura 5.10: edad de todas las personas. (a) Descripción tabular. (b) Descripción cartesiana.

### 5.4.3 Producto Cartesiano de Relaciones

Para el producto cartesiano se usa el mismo operador que para los conjuntos en general, como se indica en la Caja 5.8,

$$r \times s$$

Caja 5.8. Expresión relacional del producto cartesiano entre dos relaciones.

siendo  $r$  y  $s$  dos relaciones con esquemas cualquiera.

Sin duda, esta es la operación más habitual en las bases de datos relacionales. El producto cartesiano entre dos relaciones es una nueva relación en la cuál cada tupla es una pareja posible formada por un elemento de cada una de las dos relaciones de entrada. Por tanto, hay tantos elementos como parejas posibles, el producto de cardinales. De hecho, esta definición no tiene nada de nuevo respecto a la misma operación definida entre conjuntos, salvo la forma de aparejar los elementos. Los elementos de  $r$  y  $s$  se aparejan definiendo las tuplas del producto con los atributos de  $r$  y de  $s$ . Ergo, el esquema del producto cartesiano es la unión de los esquemas de las relaciones de entrada. En caso que haya algún atributo que se llame igual

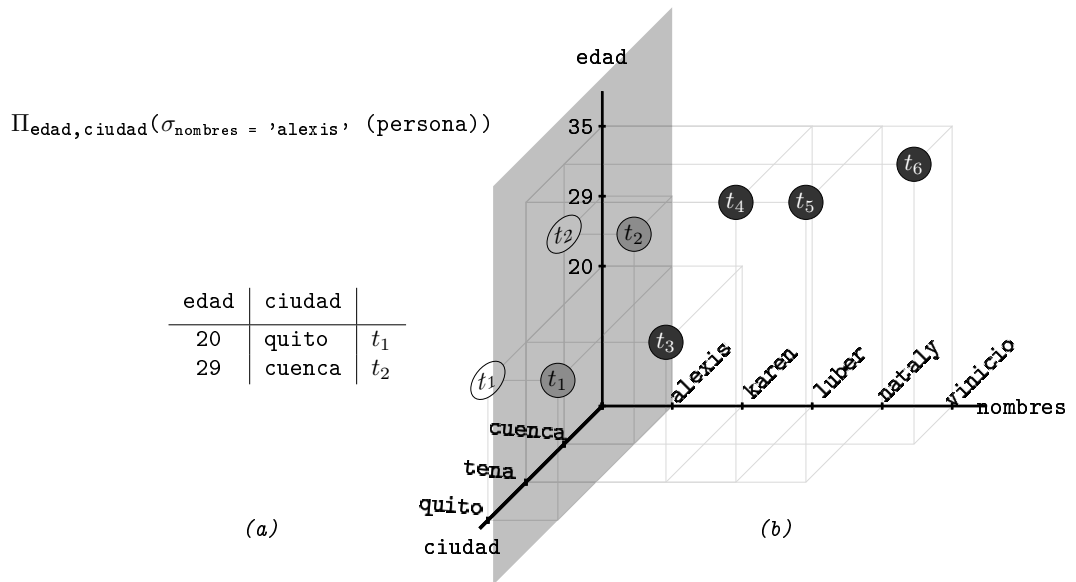


Figura 5.11: edad y ciudad de las personas que se llaman alexis. (a) Descripción tabular. (b) Descripción cartesiana.

en  $r$  y en  $s$ , habrá que poner por prefijo el nombre de la relación, llamando a los atributos homónimos  $r.A$  y  $s.A$ . Eso ocurrirá con las claves foráneas. O también se pueden utilizar operaciones de renombramiento para los atributos. La operación de renombramiento se ve en la Sección 5.4.6.

En el ejemplo de la Figura 5.12 se utilizan dos nuevas relaciones de entrada. De hecho, la primera, **persona**, coincide con las tres últimas tuplas de la relación de las secciones anteriores. La segunda es nueva, **ciudad(ciudad,provincia)**. Suponemos que la columna **ciudad** de la relación **persona** hace el papel de clave foránea a la relación **ciudad**. En la Figura 5.12(a) se presenta el contenido de las dos relaciones de entrada. Y abajo a la izquierda, Figura 5.12(b), la relación resultante, **persona**  $\times$  **ciudad**.

Filósóficamente, tal como se decía en la Sección 5.2.1, observa que hay cierto paralelismo entre el producto cartesiano de relaciones, donde el esquema es la unión de los esquemas de las relaciones iniciales, y la multiplicación de números enteros, que el resultado tiene tantas cifras como la suma de los números de cifras de los dos números que se multiplican. Estamos hablando de logaritmos, y el logaritmo del producto es la suma de logaritmos.

Esta analogía aún va más lejos. Fíjate que para rellenar la tabla de la Figura 5.12(b) se tratan los valores de los atributos de cada relación como si fueran cifras de un sistema de enumeración.

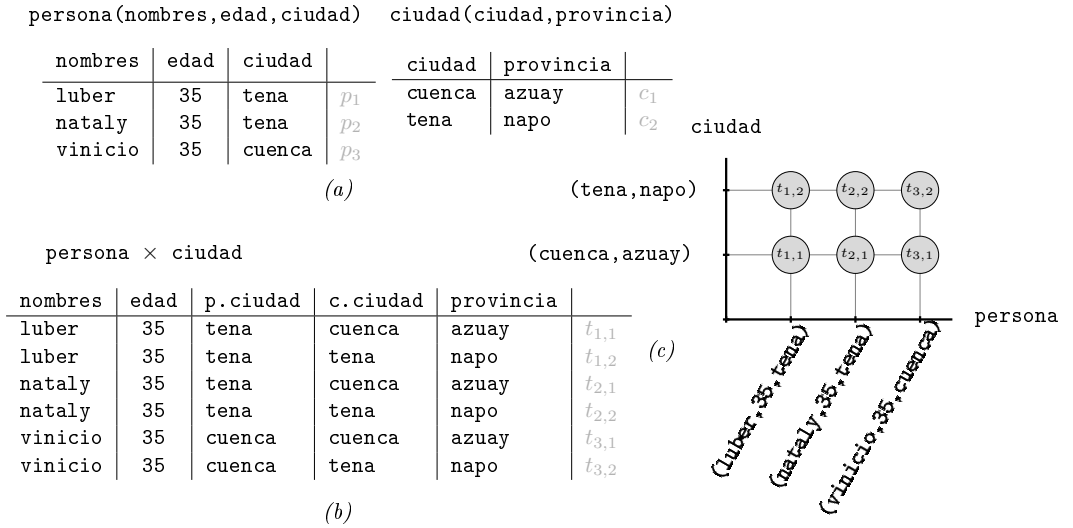


Figura 5.12: *Producto cartesiano.* (a) *Relaciones de entrada.* (b) *Descripción tabular.* (c) *Descripción cartesiana.*

Para distinguir los atributos **ciudad** en la relación final se ha utilizado como prefijo la inicial del nombre de las relaciones de entrada.

Ahora, en cada eje de la Figura 5.12(c) se representa una relación completa estableciendo un orden entre todas sus tuplas, no como los ejemplos anteriores en los que se representaba un atributo. Y también en la Figura 5.12(c) se puede comprobar que el cardinal del producto cartesiano  $|\text{persona} \times \text{ciudad}|$  es igual al producto de los cardinales de las relaciones que lo componen,  $|\text{persona}| * |\text{ciudad}|$ . O sea,  $3 * 2 = 6$ .

**ejemplo 5.6.** *¿En qué provincia vive luber?*

**solución**  $\Pi_{\text{provincia}} (\sigma_{\text{nombres}='luber' \wedge \text{p.ciudad}=\text{c.ciudad}} (\text{persona} \times \text{ciudad}))$

Atención al ejemplo 5.6. Son tres pasos.

- Calcular todas las parejas posibles.
- Seleccionar aquellas que las columnas vinculadas coincidan en valor y al mismo tiempo satisfagan el criterio especificado en el enunciado.
- Projectar los atributos que se pidan.

En definitiva, el producto cartesiano entre dos relaciones vinculadas sirve para disponer de una relación con la unión de todos los atributos en la cual solo caben

aquellas filas en las que las columnas vinculadas coincidan en valor.

Esta historia se resume muy groseramente en cinco viñetas, aunque diciendo mentiras. A ver si las descubres! Es la tira de la Figura 5.13. La historieta empieza en la viñeta 1, con una relación representada por un conjunto de puntos. La línea horizontal. En la segunda, hay dos relaciones.

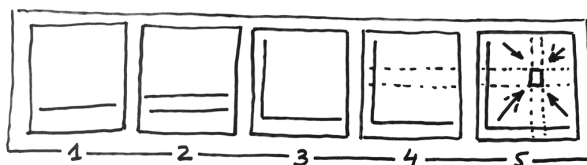


Figura 5.13: *La historieta del álgebra relacional.*

En la tercera, el producto cartesiano. Después, a partir del resultado de la tercera, seleccionamos algunas filas en la cuarta, y las proyectamos en la quinta obteniendo el resultado del cuadradito pequeño.

Quizás no os hayáis dado cuenta, pero la explicación de la historieta es un razonamiento falaz. La trampa está en la interpretación del plano, que en la tercera viñeta representa el plano cartesiano formado por todas las parejas posibles de elementos de las dos relaciones, o sea, que una fila representa un elemento de una de ellas, y una columna representa un elemento de la otra. En cambio, en la cuarta viñeta el plano representa la relación resultante del producto cartesiano. Como si una fila fuese un elemento y una columna un atributo.

En cualquier caso, todo ello resulta una imagen bastante mnemotécnica.

#### 5.4.4 Unión de Relaciones

La unión de relaciones es como la unión de conjuntos, y utiliza el símbolo de la Caja 5.9,

$$r \cup s$$

Caja 5.9. *Expresión relacional de la unión de dos relaciones.*

siendo  $r$  y  $s$  dos relaciones.

La unión resultante de dos relaciones es una relación con las tuplas de las dos.

La operación de unión entre relaciones solo está definida entre relaciones compatibles, tal como se ha definido en la Sección 5.2.1. Lógicamente, el esquema de la unión tomará el dominio mayor, o sea el menos restrictivo, entre los esquemas de las relaciones de entrada para cada uno de los atributos. Sin embargo, casi siempre los esquemas son exactamente iguales, o sea que el esquema de la unión es igual a los esquemas de las relaciones entrantes.

En esencia, sirve para la inserción de nuevas tuplas en relaciones existentes. Podemos añadir elementos a las relaciones utilizando la operación de unión. En los casos más sencillos insertaremos tuplas constantes, como en el ejemplo 5.7.

**ejemplo 5.7.** *Añadir a paola, de 25 años, que vive en ambato a la relación persona.*

**solución**  $\text{persona} \cup \{('paola', 25, 'ambato')\}$

Fíjate que en la expresión relacional que resuelve el ejemplo 5.7 el segundo operando de la expresión es una relación, y por esto las llaves de obertura y cierre, compuesta de una sola tupla, y por eso los paréntesis.

En la Figura 5.14 se ha incorporado este nuevo elemento al conjunto, identificándolo como  $t^*$ .

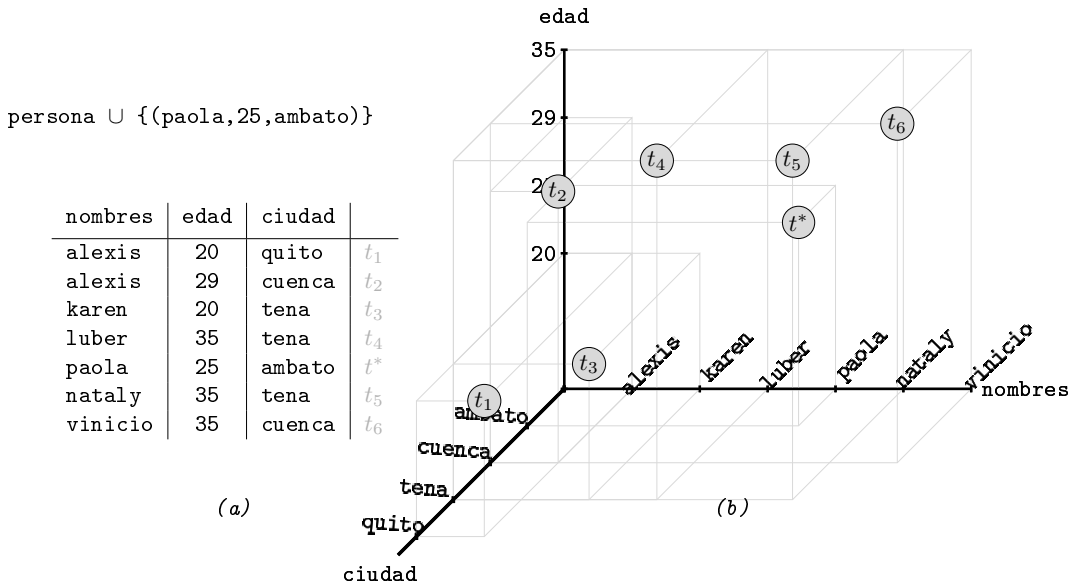


Figura 5.14: *Inserción de la tupla ('paola',25,'ambato'). (a) Descripción tabular. (b) Descripción cartesiana.*

### 5.4.5 Diferencia de Relaciones

Para la diferencia entre dos relaciones se utiliza el mismo símbolo que para la resta entre dos números enteros, como se indica en la Caja 5.10,

$$r - s$$

Caja 5.10. *Expresión relacional de la diferencia de relaciones.*

siendo  $r$  y  $s$  dos relaciones compatibles, como en el caso de la unión.

La relación resultante de la diferencia de la relación  $r$  menos la relación  $s$  es la relación de las tuplas de  $r$  que no están a  $s$ . El esquema de la diferencia es el esquema de la primera de las dos relaciones de entrada, aunque normalmente coinciden.

Fíjate pues que todas aquellas tuplas de la relación  $s$  que no pertenezcan a  $r$  no tienen ningún impacto en la relación resultante.

**ejemplo 5.8.** *Eliminar todas las personas de 35 años de la relación persona.*

**solución**  $\text{persona} - \sigma_{\text{edad}=35}(\text{persona})$

La expresión solución del ejemplo 5.8 quita todas las tuplas del plano horizontal correspondiente a la edad de 35 años, tal como se ilustra en la Figura 5.15.

### 5.4.6 Renombramiento de Relaciones

La operación de renombramiento de relaciones se expresa con la letra griega  $\rho$ , como dice la Caja 5.11,

$$\rho_{x(A_1, A_2, \dots, A_n)}(E)$$

Caja 5.11. *Expresión relacional para el renombramiento de cualquier expresión relacional.*

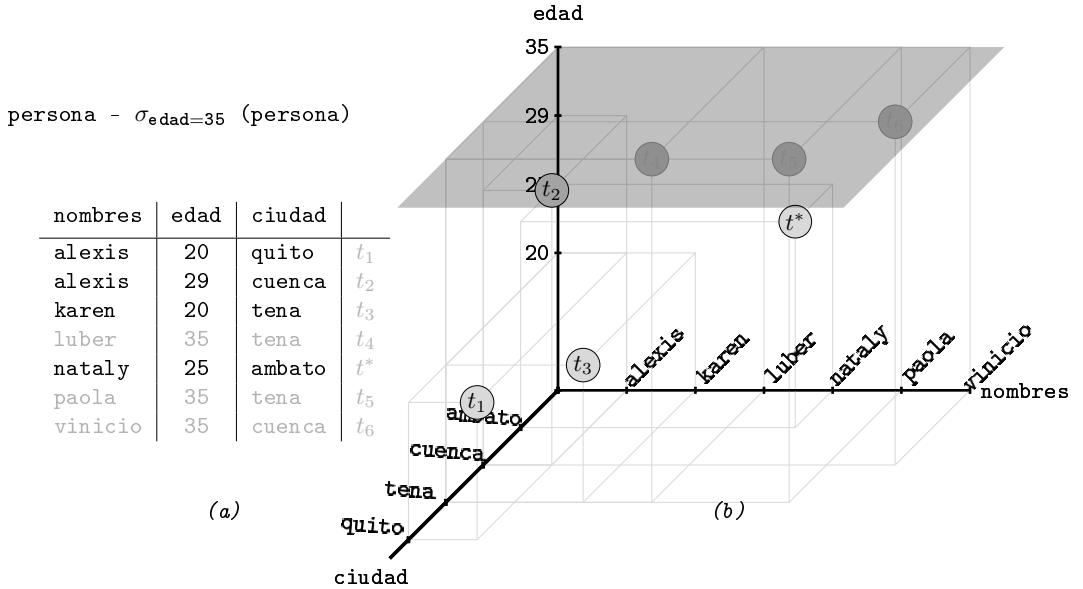


Figura 5.15: Eliminación de las personas de 35 años. (a) Descripción tabular. (b) Descripción cartesiana.

donde  $x$  es el nuevo nombre para la expresión  $E$ , y  $A_1, A_2, \dots, A_n$  es un argumento optativo para los nombres de los atributos de la relación resultante de evaluar  $E$ . O sea el esquema de  $x$ .

Si se proporciona el argumento optativo  $A_1, A_2, \dots, A_n$ , el esquema del renombramiento es el argumento dado. Si no, es el mismo esquema de la expresión relacional que recibe como relación de entrada, entonces los nombres de los atributos son los que tengan las relaciones involucradas en la expresión relacional  $E$ , con el prefijo  $x$ . En esta definición se pone  $E$  en lugar de  $r$  para indicar que es frecuente el uso del renombramiento a partir de una expresión.

Hay dos usos diferenciados de esta operación.

Por una parte podemos usarla tan solo con la finalidad de cambiar los nombres de los atributos de una relación. Este uso es particularmente útil en productos cartesianos de relaciones con atributos homónimos.

**ejemplo 5.9.** Renombrar los atributos de la relación `persona`, o sea `nombres`, `edad` y `ciudad`, a `nombres`, `años`, y `lugar`.

**solución**  $\rho_{\text{persona}(\text{nombres}, \text{años}, \text{lugar})}(\text{persona})$

En la Figura 5.16 se muestra la solución del ejemplo 5.9.



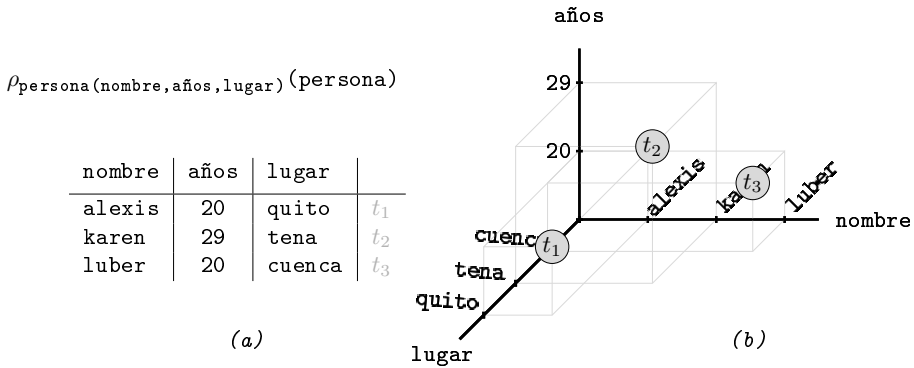


Figura 5.16: Renombramiento de los atributos de la relación `persona`. (a) Descripción tabular. (b) Descripción cartesiana.

Y por otro lado, el renombramiento tiene un uso bastante más críptico. Es el caso en que se pretende referenciar dos veces una misma relación en una composición de operaciones.

Podemos asignar un nombre a una expresión del núcleo de una composición de operaciones, para utilizarlo desde las operaciones más exteriores, o sea finales, de esta misma composición. Cuando se trata de comparar todos los elementos de una relación entre ellos, por ejemplo. En general, cualquier tratamiento que desde el punto de vista de la programación requiriese un bucle dentro de un bucle. Y eso significa, operaciones que involucran todas las parejas posibles de tuplas de una misma relación. Son consultas más complicadas.

Para el ejemplo 5.10 se parte de la relación `persona(nombres, años, lugar)` de la Figura 5.16.

**ejemplo 5.10.** *Obtener los nombres de la persona mayor.*

**solución**  $\Pi_{\text{nombres}}(\text{persona}) - \Pi_{\text{x.nombres}}(\sigma_{\text{x.años} < \text{y.años}}(\rho_{\text{x}}(\text{persona}) \times \rho_{\text{y}}(\text{persona})))$

Desgranemos la solución del ejemplo 5.10. Atacamos de entrada el segundo operando de la diferencia,

$$\Pi_{\text{x.nombres}}(\sigma_{\text{x.años} < \text{y.años}}(\rho_{\text{x}}(\text{persona}) \times \rho_{\text{y}}(\text{persona})))$$

Eso es una relación de un solo atributo llamado `nombres`. Está formada por las personas tales que existe alguien mayor que ellas. Paso a paso. Empezamos por observar cual es el contenido de la relación correspondiente al producto cartesiano

$$\rho_{\text{x}}(\text{persona}) \times \rho_{\text{y}}(\text{persona})$$

que se muestra en la Tabla 5.4.

x.nombres	x.años	x.lugar	y.nombres	y.años	y.lugar
alexis	20	quito	alexis	20	quito
alexis	20	quito	karen	29	tena
alexis	20	quito	luber	20	cuenca
karen	29	tena	alexis	20	quito
karen	29	tena	karen	29	tena
karen	29	tena	luber	20	cuenca
luber	20	cuenca	alexis	20	quito
luber	20	cuenca	karen	29	tena
luber	20	cuenca	luber	20	cuenca

Tabla 5.4: *Contenido de la relación  $\rho_x(\text{persona}) \times \rho_y(\text{persona})$ .*

Otra vez se puede observar que la manera de rellenar la relación de la Tabla 5.4 es como si las tuplas de la relación persona fueran cifras de un sistema de enumeración, y en la relación hubiéramos puesto todos los números de dos cifras de este sistema.

Dicho esto, observamos que hace la selección

$$\sigma_{x.años < y.años}(\rho_x(\text{persona}) \times \rho_y(\text{persona})).$$

Se está pidiendo pues, que de la Tabla 5.4 se eliminen todas las filas en las que la primera columna edad sea inferior a alguna otra de la tabla, ya que en el producto cartesiano cada fila se combina con todos los valores existentes en la otra tabla. Es decir, nos quedamos con las dos tuplas en las que  $x.años < y.años$  de la Tabla 5.4. En la Tabla 5.5 se ha dejado en tono más claro las tuplas eliminadas de la solución.

x.nombres	x.años	x.lugar	y.nombres	y.años	y.lugar
alexis	20	quito	alexis	20	quito
alexis	20	quito	karen	29	tena
alexis	20	quito	luber	20	cuenca
karen	29	tena	alexis	20	quito
karen	29	tena	karen	29	tena
karen	29	tena	luber	20	cuenca
luber	20	cuenca	alexis	20	quito
luber	20	cuenca	karen	29	tena
luber	20	cuenca	luber	20	cuenca

Tabla 5.5: *Selección  $\sigma_{x.años < y.años}(\rho_x(\text{persona}) \times \rho_y(\text{persona}))$ .*

Y a partir de aquí, ya es muy sencillo. Proyectamos los dos nombres, *alexis* y *luber*, como se ve en la Tabla 5.6 que contiene los nombres de las personas tales que hay alguien mayor que ellas. O sea, todas las personas excepto la mayor.

nombres
-----
alexis
luber

Tabla 5.6: *Proyección*  $\Pi_{\text{nombres}}(\sigma_{x.\text{años} < y.\text{años}}(\rho_x(\text{persona}) \times \rho_y(\text{persona})))$

Finalmente, restamos de todos los nombres, los que aparezcan en la relación de la Tabla 5.6. En la Figura 5.17 se muestra la parte final del procedimiento de la consulta del ejemplo 5.10.

nombres		nombres		nombres
-----		-----	=	-----
alexis	-	alexis		karen
karen		luber		
luber				

Figura 5.17: *Última etapa de la resolución del ejemplo 5.10.*

El problema de la persona mayor se ha resuelto utilizando el concepto de complementario. Es decir, se ha calculado primero la relación de todos excepto el mayor, y luego se ha tomado la relación complementaria respecto a la relación inicial. No hay otra manera. Si la selección hubiera cogido las tuplas tales que existiera alguien mayor que todos los demás, hubiera salido una relación vacía. Y si se hubiera pedido por mayor o igual, se habrían seleccionado todas las tuplas.

El ejemplo 5.10 es complicado. Sirve para mostrar la capacidad expresiva de las operaciones básicas que aquí cerramos. En adelante, nos aprovisionaremos de operaciones más sofisticadas que nos permitirán seleccionar directamente un máximo.

## 5.5 Operaciones Adicionales

Las operaciones adicionales son operaciones que no contribuyen al potencial expresivo del álgebra. La demostración matemática de esta afirmación consiste en expresar cada una de estas operaciones en términos de operaciones básicas, como efectivamente se hace a continuación. Estas operaciones se deben entender como herramientas auxiliares que hacen las consultas más concisas, y por tanto más legibles.

En primer lugar hay la *intersección*, que se puede construir a partir de la diferencia. Después, la *reunión natural* que compacta producto cartesiano, selección y proyección. Una generalización de la reunión natural es la reunión interna, que da paso a las otras tres operaciones de reunión, las externas. También se considera la *división* de relaciones, que se define en base al producto cartesiano, y finalmente, para facilitar la descripción de consultas, se incluye una instrucción de asignación. La *asignación* se presenta como última herramienta auxiliar porque los resultados de las

consultas son expresiones, no instrucciones. Las asignaciones permiten hacer pasos intermedios, pero en cualquier caso la consulta final debe resultar ser una expresión como en todos los ejemplos anteriores.

Las relaciones `persona(nombres,edad,ciudad)` y `ciudad(ciudad,provincia)` que se muestran en la Tabla 5.7 servirán para los ejemplos de esta sección.

nombres	edad	ciudad	ciudad	provincia
luber	35	tena	cuenca	azuay
nataly	35	tena	tena	napo
vinicio	35	cuenca	guayaquil	guayas
walter	23	quito		

Tabla 5.7: *Relaciones persona y ciudad.*

### 5.5.1 Intersección de Relaciones

El símbolo de la intersección de relaciones, como el de la unión, es igual que para los conjuntos, como se puede ver en la Caja 5.12,

$$r \cap s$$

Caja 5.12. *Expresión relacional de la intersección de relaciones.*

siendo  $r$  y  $s$  compatibles como en la unión o la diferencia.

El resultado de la intersección de relaciones es la relación de tuplas que estén en  $r$  y en  $s$ , como ya nos podíamos imaginar. Respecto a su esquema, lo mismo que se ha dicho de la unión. O sea, la combinación de dominios menos restrictiva de los atributos. Aún así, la mayor parte de las veces los esquemas de las relaciones de entrada coinciden, y entonces también con el de la intersección.

Una tupla está en  $r$  y en  $s$  si tienen los mismos valores en todos y cada uno de los atributos.

No es una operación básica, como se demuestra en la Caja 5.13,

$$r \cap s = r - (r - s)$$

Caja 5.13. *La intersección no es una operación básica.*

Ya se sabe, la intersección entre dos conjuntos, o también dos mentalidades, es lo que queda luego de eliminar las diferencias.

Es curioso que siendo una operación conmutativa se pueda expresar con un aspecto tan poco conmutativo como el de la Caja 5.13. Pero aún así, recordar los mnemogramas vistos para estas operaciones cuando se definieron entre conjuntos, en la Sección 1.3, facilita la comprensión de la igualdad de la Caja 5.13.

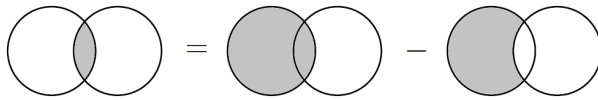


Figura 5.18: *La intersección es uno menos la diferencia.*

En la Figura 5.18 se puede contemplar lo que podríamos llamar una operación entre ideogramas. Es importante tener presente que esta forma de expresarse no tiene ningún rigor, y tan solo sirve para facilitar la comprensión de la idea de limar diferencias.

El enunciado del ejemplo 5.11 ya se vio en la Sección 5.4.1. Aquí se resuelve por medio de la intersección de relaciones.

**ejemplo 5.11.** *Obtener los nombres de las personas de tena, de 35 años.*

**solución**  $\Pi_{\text{nombres}}(\sigma_{\text{edad}=35}(\text{persona})) \cap \Pi_{\text{nombres}}(\sigma_{\text{ciudad}='tena'}(\text{persona}))$

## 5.5.2 Reunión Natural

La operación de reunión natural entre dos relaciones se indica con el signo de la Caja 5.14,

$$r \bowtie s$$

Caja 5.14. *Expresión relacional de la reunión natural entre dos relaciones.*

donde  $r$  y  $s$  son dos relaciones, supuestamente con algunos atributos homónimos. La expresión de la Caja 5.14 se pronuncia "erre reunión natural ese", aunque a menudo se usa el anglicismo *natural join*, "erre natural join ese". Al símbolo  $\bowtie$ , también hay quien le llama corbatín o lacito. Tiene gracia, "erre lacito ese".

La reunión natural retorna la selección de las tuplas del producto cartesiano que tengan valores coincidentes en las columnas homónimas.

Estas columnas homónimas resultan por tanto equivalentes, y la reunión natural tan solo las proyecta una vez.

Eso significa que el esquema de la reunión natural tendrá la unión, en el sentido más riguroso, de los esquemas de las relaciones de entrada. Numéricamente, el esquema tendrá la suma de las cantidades de atributos de las relaciones de entrada menos la cantidad de atributos comunes, es decir homónimos. Para el caso del ejemplo, **persona** tiene tres columnas, **ciudad** dos, y tienen una columna homónima. El esquema de la reunión natural tendrá por tanto cuatro columnas.

Volvamos al ejemplo 5.6 de la página 134 donde la consulta era en qué provincia vivía *luber*. La expresión que solucionaba la consulta era

$$\Pi_{\text{provincia}} (\sigma_{\text{nombres}='luber'} \wedge \text{p.ciudad}=\text{c.ciudad} (\text{persona} \times \text{ciudad})).$$

Está bien claro que el predicado de la selección,

$$\text{nombres} = 'luber' \wedge \text{p.ciudad} = \text{c.ciudad},$$

se compone de dos proposiciones. La primera, **nombres = 'luber'**, es consecuencia directa del enunciado de la consulta, en qué provincia vive *luber*. Pero la segunda no. No depende directamente de la consulta. Hay muchas otras consultas que también requerirían la proposición **p.ciudad = c.ciudad** en su predicado. En definitiva, es una proposición interna. Si en lugar del producto cartesiano utilizamos la reunión natural, nos ahorraremos añadir esta proposición en todas esas consultas.

De esta forma podemos reformular la solución del ejemplo 5.6 como se puede ver en el ejemplo 5.12.

**ejemplo 5.12.** *¿En qué provincia vive luber?*

**solución**  $\Pi_{\text{provincia}} (\sigma_{\text{nombres}='luber'} (\text{persona} \bowtie \text{ciudad})).$

Independientemente de que la mejora sea poco notable a nivel lexicográfico, sí que es interesante a nivel filosófico, ya que en la solución del ejemplo 5.12 todos los argumentos, el de la proyección y el de la selección, forman parte del enunciado del problema, o sea que son específicos de esta consulta.

Observa pues que la segunda proposición del predicado de la selección, **p.ciudad = c.ciudad**, queda asumida por el hecho de que las columnas se llamen igual, y por tanto reflejada en la estructura del modelo.

En la Tabla 5.8 hay el contenido de la reunión natural de las relaciones de la Tabla 5.7.

nombres	edad	ciudad	provincia
luber	35	tena	napo
nataly	35	tena	napo
vinicio	35	cuenca	azuay

Tabla 5.8: *Reunión natural persona  $\bowtie$  ciudad con los valores de la Tabla 5.7.*

La reunión natural elimina las filas que no tenían sentido en la composición selección de producto cartesiano, además de una de las dos columnas equivalentes.

A partir de la relación resultante de esta operación, podemos proyectar cualquier atributo de las dos relaciones y también seleccionar a partir de cualquier otro atributo. Ya se ve que la potencia de esta operación es extraordinaria.

## Reunión interna

La operación de reunión interna, o reunión zeta, entre dos relaciones se denota con el mismo signo que la reunión natural, con un predicado simbolizado por la letra zeta minúscula griega  $\theta$ , que se pronuncia zeta. En la Caja 5.15 se muestra la expresión.

$$r \bowtie_{\theta} s$$

Caja 5.15. *Expresión relacional de la reunión interna entre dos relaciones.*

El resultado de la reunión interna es la selección de las filas del producto cartesiano que satisfagan el predicado  $\theta$ . Es decir,

$$r \bowtie_{\theta} s = \sigma_{\theta}(r \times s).$$

Se trata de una generalización de la reunión natural que permite asociar tuplas de las dos relaciones aunque las columnas no se llamen igual.

El esquema de la reunión interna no agrega columnas homónimas, tiene tantos atributos como la suma de las cantidades de atributos de las relaciones de entrada. Eso significa que si hay columnas homónimas habrá títulos repetidos en la relación resultante, y por tanto para referenciarlos posteriormente se necesitarán renombramientos.

La reunión interna se utiliza en particular para establecer las proposiciones de los predicados que atan atributos estructurales. Es decir, claves foráneas y claves principales, cuando por las razones que sea no puedan llamarse igual. No siempre se puede llamar las claves foráneas con el nombre de la clave primaria a la cual apuntan. Por ejemplo, tres casos.

- Cuando hay dos relaciones 1:N entre la misma pareja de entidades en el modelo ER, como en el ejemplo del Modelo 4.13, donde entre persona y ciudad había dos relaciones 1:N.
- En una autorelación 1:N, como en el caso de la clave foránea *obedece* del Modelo 5.11.
- Cuando hay una autorelación M:N, como en la relación *conoce*, del mismo Modelo 5.11.

En estas situaciones no se puede utilizar la reunión natural. Entonces se acostumbra a hacer con la reunión interna, que casi es una notación alternativa a la selección de un producto cartesiano. Ayuda a justificar su definición, además, que la reunión interna sirve de base para las reuniones externas que se verán más adelante.

El contenido de la reunión interna entre las relaciones de la Tabla 5.7 está en la Tabla 5.9.

nombres	edad	ciudad	ciudad	provincia
luber	35	tena	tena	napo
nataly	35	tena	tena	napo
vinicio	35	cuenca	cuenca	azuay

Tabla 5.9: *Reunión interna* `persona`  $\bowtie_{p.ciudad=c.ciudad}$  `ciudad` con los valores de la Tabla 5.7.

La columna ciudad, pues, aparece repetida. Eso es así porque en la mayoría de los casos que se utiliza la reunión interna es para luego proyectar algunos atributos concretos.

Los dos ejemplos próximos se refieren al Modelo 5.12. Hay trabajadores que pertenecen a departamentos, en los que mandan unos jefes. Y también personas que conocen a personas.

```

persona(pasaporte,nombres,apellidos,ciudad)
conoce(conoce,es_conocida)
trabajador(pasaporte,departamento,obedece)

```



Modelo 5.12. *Fragmento Modelo 5.11 relacional.*

**ejemplo 5.13.** *Obtener los nombres de los jefes de los trabajadores.*

**solución**  $\Pi_{\text{nombres}}(\rho_x(\text{persona}) \bowtie_{x.\text{pasaporte}=\text{obedece}} \text{trabajador})$

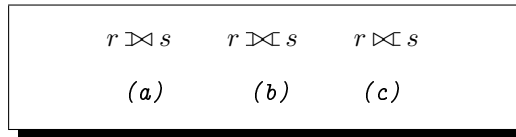
En el ejemplo 5.13 se calcula  $\text{persona} \times \text{trabajador}$  dando siete columnas, y entonces se seleccionan los nombres de todas las personas que aparecen en el valor del atributo *obedece* de alguna tupla de *trabajador*.

**ejemplo 5.14.** *Nombres de las personas conocidas por alguien.*

**solución**  $\Pi_{\text{nombres}}(\rho_x(\text{persona}) \bowtie_{x.\text{pasaporte}=\text{conocida}} \text{conoce})$

### Reuniones externas

Hay tres operaciones de reunión externa. Por la derecha, por la izquierda y completa. El esquema de todas es el mismo que el de la reunión interna. Se diferencian en la cantidad de tuplas del resultado. En la Caja 5.16 se muestran los signos que utilizan. A la izquierda, Caja 5.16(a), la reunión externa por la izquierda, *left outer join*. En la Caja 5.16(b) la reunión externa completa, *full outer join*, y en la Caja 5.16(c) la reunión externa por la derecha, *right outer join*,



Caja 5.16. *Expresión relacional de las reuniones externas. (a) por la izquierda. (b) completa. (c) por la derecha.*

Dada una reunión interna entre dos relaciones  $r$  y  $s$ , o sea, una relación formada por los atributos de  $r$  y los de  $s$ , y por las tuplas que satisfagan el predicado  $\theta$ , la idea de las reuniones externas es útil cuando se desea mantener la información de todas las tuplas de alguna de las relaciones de entrada. Es decir, las que no hayan sido combinadas con ninguna tupla de la otra relación. En otras palabras, si las columnas que se vinculan en el predicado  $\theta$  provocan la desaparición de alguna tupla que interesa, entonces hacemos que aparezca en el resultado. Y claro, eso provoca la introducción de nulos.

La relación resultante de una reunión externa por la izquierda consta de todos los datos de la primera de las dos relaciones de entrada, más los de la segunda que estén vinculados con alguno de la primera. Para los valores de la Tabla 5.7 la relación

resultante se muestra en la Tabla 5.10.

nombres	edad	ciudad	ciudad	provincia
luber	35	tena	tena	napo
nataly	35	tena	tena	napo
vinicio	35	cuena	cuena	azuay
walter	23	quito	quito	null

Tabla 5.10: *Reunión externa por la izquierda, left outer join, entre persona y ciudad de la Tabla 5.7.*

Con una reunión externa completa no se pierde ningún dato de las relaciones de entrada. Eso hace que quedan valores nulos en todas las columnas no vinculadas, como se puede ver en la Tabla 5.11.

nombres	edad	ciudad	ciudad	provincia
luber	35	tena	tena	napo
nataly	35	tena	tena	napo
vinicio	35	cuena	cuena	azuay
walter	23	quito	quito	null
null	null	guayaquil	guayaquil	guayas

Tabla 5.11: *Reunión externa completa, full outer join, entre persona y ciudad de la Tabla 5.7.*

Y la reunión externa por la derecha es la versión simétrica de la de la izquierda. Por eso tiene todas las tuplas de la segunda relación. En la Tabla 5.12 se puede observar el resultado de la reunión externa por la derecha de las relaciones de la Tabla 5.7.

nombres	edad	ciudad	ciudad	provincia
luber	35	tena	tena	napo
nataly	35	tena	tena	napo
vinicio	35	cuena	cuena	azuay
null	null	guayaquil	guayaquil	guayas

Tabla 5.12: *Reunión externa por la derecha, right outer join, entre persona y ciudad de la Tabla 5.7.*

### 5.5.3 División de Relaciones

El signo de la operación de división entre dos relaciones es el de la Caja 5.17

$$r \div s$$

Caja 5.17. *Expresión relacional de la división entre dos relaciones.*

siendo  $r$  y  $s$  dos relaciones tales que los atributos de  $s$  son un subconjunto propio de los atributos de  $r$ . A  $r$  se le llama relación dividenda, a  $s$  se le llama relación divisora, y al resultado se le puede llamar relación cociente.

La relación resultante de una división tiene por esquema los atributos de  $r$  que no están en  $s$ . Su contenido son las partes correspondientes de las tuplas de  $r$  que, en  $r$ , estén combinadas con cada una de las tuplas de  $s$ .

Esta operación sirve para consultas que pidan las tuplas de una relación que esté relacionadas con todas y cada una de las tuplas de la otra.

Para entendernos, miremos el caso más sencillo,  $r$  de dos atributos y  $s$  de uno solo. Sea  $r = r(A, B)$ , y  $s = s(B)$ . Entonces el resultado de  $r$  dividido para  $s$  tendrá solo la columna  $A$ , y tendrá una fila para cada valor de  $A$  que en  $r$  aparezca juntamente con todas y cada una de las tuplas de  $s$ .

Probablemente, la forma más sencilla de recordarlo es que el resultado de la división es aquella relación tal que multiplicada por  $s$  resulte una parte grande de  $r$ , ya que, atención, aquellas tuplas de  $r$  en las cuales el valor del atributo  $A$  no aparezca combinado con cada una de las tuplas de  $s$  no tienen impacto alguno en la división.

No es una operación básica, tal como se demuestra en la Caja 5.18 con unos esquemas de  $r$  y  $s$  genéricos.

$$\begin{aligned} \text{Si } r &= r(A_1, A_2, \dots, A_n, B_1, B_2, \dots, B_m) \\ \text{y } s &= s(B_1, B_2, \dots, B_m) \\ \\ r \div s &= \Pi_{A_1, A_2, \dots, A_n}(r \bowtie s) \end{aligned}$$

Caja 5.18. *La división no es una operación básica.*

El ejemplo 5.15 se plantea con el ejemplo del club deportivo, del Modelo 5.11.

**ejemplo 5.15.** *Nombres de los socios que hacen todos los deportes que hay en el club.*

**solución**  $\Pi_{\text{nombres}}(\text{persona} \times (\Pi_{\text{pasaporte, deporte}}(\text{hace}) \div \Pi_{\text{deporte}}(\text{deporte}))$

Para pensar la solución del ejemplo 5.15 la primera cosa es darse cuenta de que para resolver la consulta hay que obtener en algún lugar la relación de todos los deportes, que hará el papel de divisora es decir, de  $s$ . La relación de todos los deportes es  $\Pi_{\text{deporte}}(\text{deporte})$ .

Una vez disponemos de la relación divisora, nos debemos preocupar de conseguir la relación dividenda, es decir, la que tenga en alguna columna el atributo que sirva para conseguir la salida, y en alguna otra columna los valores de la relación divisora, que en el ejemplo 5.15 es *hace*.

En este mismo ejemplo 5.15 el atributo que obtenemos de la división es *pasaporte*. Por eso acabamos multiplicándolo por *persona* y así resolver los nombres tal como se pide.

### 5.5.4 Asignación de Relaciones

El símbolo que representa una asignación es el de la Caja 5.19

$$r \leftarrow s$$

Caja 5.19. *Expresión para la asignación de relaciones.*

siendo  $s$  una relación, y  $r$  a partir de este momento, también.

El resultado de esta operación es una nueva relación con el mismo contenido que la dada de entrada.

Utilizando esta operación consideraremos que las relaciones son variables que pueden tomar por valor otras relaciones, o más en general, cualquier expresión relacional, es decir, cualquier resultado de una operación del álgebra. Una vez asignado un resultado a una relación se puede usar de nuevo como relación de entrada a otras operaciones, modularizando así la tarea de descripción de una consulta a la base de datos.

En la Caja 5.20 se asigna una relación de una única tupla con los valores (*'slendy macarena', 24, 'baños'*) a la relación *persona(nombres, edad, ciudad)* de los ejemplos

anteriores. Observa que los valores de la tupla pertenecen a los dominios de la relación. Hay una correspondencia posicional. O sea, *slendy macarena* son los nombres, el *24* es la edad, y *baños* la ciudad.

```
persona ← {'slendy macarena', 24, 'baños'}
```

Caja 5.20. *Asignación de una relación con una sola tupla a la relación persona.*

Los paréntesis establecen que los valores van en ese orden. Y la expresión  $\{('slendy macarena', 24, 'baños')\}$  es una relación que tan solo contiene una tupla. O sea que cada una de las tuplas sí que está ordenada interiormente y por eso se describe con paréntesis. En cambio la relación es un conjunto, y como tal, no ordenado, de tuplas, cosa que se expresa con las llaves. En definitiva, la relación dada como operando de entrada en la Caja 5.20 es un conjunto definido por enumeración según aquello visto en la Sección 1.2.2.

A continuación se desgana en etapas el ejemplo 5.16 de la Sección 5.4.6, que se definía sobre la relación  $\text{persona} = \text{persona}(\text{nombres}, \text{años}, \text{lugar})$ . En el ejemplo 5.16 es notorio el hecho de que luego de una serie de asignaciones, se termina dando una expresión como solución.

**ejemplo 5.16.** *Obtener los nombres de la persona mayor.*

**solución**

$$\begin{aligned} r_1 &\leftarrow \Pi_{\text{nombres}}(\text{persona}) \\ x &\leftarrow \text{persona} \\ y &\leftarrow \text{persona} \\ r_2 &\leftarrow \sigma_{x.\text{años} < y.\text{años}}(x \times y) \\ r_1 &- r_2 \end{aligned}$$

## 5.6 Álgebra Relacional Extendida

Consideramos que las operaciones siguientes forman parte del álgebra relacional extendida por el hecho de tener en cuenta los tipos de los atributos.

### 5.6.1 Proyección Generalizada

Podemos generalizar la operación de proyección vista en la Sección 5.4.2 con la notación de la Caja 5.21

$$\Pi_{F_1(A_1), F_2(A_2), \dots, F_n(A_n)}(r)$$

Caja 5.21. *Proyección generalizada de los atributos de una relación.*

La operación de proyección generalizada retorna la relación formada por las transformaciones según las funciones  $F_1, F_2, \dots, F_n$  de los atributos de la relación de entrada.

**ejemplo 5.17.** *A partir de la relación `hace = hace(pasaporte, deporte, cuota)`, obtener la cantidad de impuestos, 21%, que paga cada socio por cada deporte.*

**solución**  $\Pi_{\text{pasaporte, deporte, } 0.21 * \text{cuota}}(\text{hace})$

**ejemplo 5.18.** *A partir a la relación `persona(nombres, edad, ciudad)`, para cada persona de tena, insertar otra con los mismos nombres, pero 5 años mayor.*

**solución**  $\text{persona} \cup \Pi_{\text{nombres, edad+5, 'tena'}}(\sigma_{\text{ciudad='tena'}}(\text{persona}))$

De estos ejemplos se desprende que las funciones de la proyección generalizada pueden involucrar atributos de la relación dada, y también valores constantes, cosa que nos abre la posibilidad de utilizar el álgebra para hacer operaciones aritméticas, como en el ejemplo 5.19

**ejemplo 5.19.** *Calcular la suma de  $2 + 3$ .*

**solución**  $\Pi_{2+3}$

donde como se puede observar ni siquiera le pasamos ninguna relación de entrada.

De la misma manera, una nueva versión de la asignación de la Caja 5.20 se expone en la Caja 5.22.

$$\text{persona} \leftarrow \Pi_{\text{'slendy macarena', 24, 'baños'}}$$

Caja 5.22. *Nueva versión de la asignación de la Caja 5.20.*

Estas proyecciones de atributos constantes en el valor de todas las tuplas del resultado quizá quedan más claras viendo cómo proyectar la constante en más de una tupla.

Pongamos por caso, la expresión

$$\Pi_{\text{nombres}, 200}(\text{persona})$$

retornaría una relación formada de tantas filas como tenga la tabla persona, y de dos columnas. Una con los nombres, y la segunda constante igual a 200 en toda la tabla. Es la relación de la Tabla 5.13.

nombres	?
luber	200
nataly	200
vinicio	200
walter	200

Tabla 5.13: *Resultado de la proyección de un atributo constante.*

Fíjate que en la relación resultante hay un atributo con título desconocido. O sea, para reutilizarla sería necesario darle algún nombre con la operación del renombramiento.

### 5.6.2 Valores Nulos

Un valor *null* es un valor inexistente en una relación. O sea en algún atributo de alguna tupla de una relación. Eso es ambiguo, porque tanto puede ser que el valor sea inexistente porque no se conoce pero podría estar disponible más adelante, como ser nulos estructurales que jamás podrán adquirir un valor.

En cualquier caso, como se ha visto en secciones anteriores, los valores de los atributos se usan en las comparaciones de las selecciones. Y por tanto, hay que dejar bien claro los resultados que esperamos de estas comparaciones con valores nulos.

La filosofía es postponer la interpretación de *null* como falso lógico tanto como se pueda. Es decir, mientras podamos operar con el valor *null*, si tenemos la suerte de que desaparezca en los cálculos intermedios, mejor. Y si no, si al final de la evaluación de un predicado resulta que vale *null*, entonces consideraremos que es falso.

El lenguaje estructurado de consultas, SQL, tiene un mecanismo para discernir si el resultado de una expresión booleana es falso realmente, o es falso debido a que la evaluación ha dado como resultado un valor nulo.

Cualquier comparación de un atributo con *null* dará *null*. Es decir, si en un predicado hay proposiciones que una vez evaluadas acaban calculando si  $A < null$ ,  $A = null$ , o  $A > null$ , entonces el resultado de estas proposiciones es *null*. Y por

tanto si son la única proposición del predicado, falso. Observa que en particular, el predicado  $null = null$  es falso, como también lo es  $null \neq null$ .

Con el SQL podemos preguntar si el valor de un atributo es nulo. Pero cuidado, no con el comparador de igualdad. Hacerlo con el comparador de igualdad es un error frecuente. Y claro, siempre devuelve falso, aunque el valor comparado no sea nulo.

El valor  $null$  en las operaciones aritméticas se comporta de una manera absorbente. Es decir, la expresión  $5 + null$  es  $null$ .

### Lógica de tres valores

Por descontado que siempre que los valores nulos en las proposiciones puedan ser absorbidos por el Álgebra de Boole, mejor. Las tablas de verdad para esta lógica de tres valores son.

$p$	$q$	$p \wedge q$
falso	null	falso
cierto	null	null

(a)

$p$	$q$	$p \vee q$
falso	null	null
cierto	null	cierto

(b)

$p$	$\neg p$
null	null

(c)

Tabla 5.14: *Tablas de verdad con valores nulos. (a) Conjunción. (b) Disyunción. (c) Negación*

Es decir, para la conjunción lógica, el valor nulo se comporta como si fuese falso. Para la disyunción, cierto. Y como cosa nueva, no  $null$ , es  $null$ .

Filosóficamente, desde un punto de vista más esencial, una de las reglas que parece más cuestionable de la deducción natural es que falso sea igual a falso.

### 5.6.3 Funciones de Agregación

Las funciones de agregación son funciones a las cuales se las alimenta con una colección de valores, y nos retornan uno nomás, como la cuenta, la suma, el promedio, o la desviación tipo, que viene a ser el promedio de distancias al promedio.

Es cosa buena interpretar el término *agregar* como una unión irreversible. Como una fusión. Cuando en la operación booleana de suma lógica uno más uno hacen uno, entonces más que una suma, la operación se comporta como una agregación. Y



también en este sentido, cuando los organismos administrativos de estadística desean preservar la confidencialidad de la información, dan los datos agregados.

Esta misma interpretación es la que se hace en el modelo ER cuando incorporamos una clave primaria a una relación M:N convirtiéndola en entidad. En este caso estamos agregando una pareja, o más, de claves foráneas en una sola clave.

Es crucial darse cuenta que agregando un atributo se reduce la cantidad de filas de una relación por el atributo agregado a una sola tupla.

El formato de las funciones de agregación, en su versión más sencilla, se indica en la Caja 5.23

$$\mathcal{G}_{f(A)}(r)$$

Caja 5.23. *Expresión relacional mínima para las funciones de agregación.*

asumiendo que el atributo  $A$  pertenece al esquema de  $r$ . El símbolo  $\mathcal{G}$  se llama  $\mathcal{G}$  caligráfica, y se usa como mnemotécnico para el término *grupo*, o *agrupación*. Se utiliza pasándole como argumento la función de agregación que deseamos calcular, y entre paréntesis el atributo sobre el que deseamos hacer el cálculo.

La relación resultante de operar con funciones de agregación son de una sola tupla y un solo atributo, sin nombre.

Para incorporar estas funciones al álgebra hay que distinguir atributos numéricos, y atributos textuales, por eso estamos en el álgebra extendida.

Las funciones definidas sobre atributos textuales son la cuenta, el mínimo y el máximo. La cuenta retorna la cantidad de valores distintos en el atributo dado a la operación como parámetro de la función. El mínimo y el máximo retornan el primero y el último valor ordenados alfabéticamente según la tabla ASCII, que se muestra en el Apéndice B. Los nombres de estas funciones son  $\text{count}(A)$ ,  $\text{min}(A)$ , y  $\text{max}(A)$ , siendo  $A$  el atributo sobre el que se calculan.

Las funciones definidas sobre atributos numéricos son las de las textuales, y además las aritméticas, o estadísticas. La suma y el promedio. Sus nombres son  $\text{sum}(A)$ ,  $\text{avg}(A)$ .

Veamos ejemplos a partir de las relaciones del Modelo 5.13 para el club deportivo.

```

persona(pasaporte,nombres,apellidos,mail,ciudad)
socio(pasaporte,registro)
deporte(deporte,precio,jugadores)
hace(pasaporte,deporte,cuota)

```

Modelo 5.13. *Fragmento del Modelo 5.11 para los ejemplos de las funciones de agregación.*

**ejemplo 5.20.** *¿Cuántos socios hay en el club?*

**solución**  $\mathcal{G}_{\text{count}(\ast)}(\text{socio})$

La solución del ejemplo 5.20 contaría la cantidad de socios del club. El uso del asterisco en lugar del atributo significa que se desea contar la cantidad de tuplas de la relación dada.

En los ejemplos 5.21, y 5.22 el producto cartesiano sirve para solo tener en cuenta los números de pasaporte que se correspondan a socios.

**ejemplo 5.21.** *¿Cuántos socios hay en el club que se llamen pablo eduardo?*

**solución**  $\mathcal{G}_{\text{count}(\ast)}(\sigma_{\text{nombres}=\text{'pablo eduardo'}}(\text{persona}) \times \text{socio})$

**ejemplo 5.22.** *¿Cómo se llaman el primero y el último socio, alfabéticamente, del club?*

**solución**  $\mathcal{G}_{\text{min}(\text{nombres}),\text{max}(\text{nombres})}(\text{persona} \times \text{socio})$

**ejemplo 5.23.** *¿Cuánto dinero entra cada mes en el club?*

**solución**  $\mathcal{G}_{\text{sum}(\text{cuota})}(\text{hace})$

En los ejemplos anteriores se hace la agregación en base a la relación dada completa. Y en cualquiera de ellos el resultado es una sola tupla.

Hay casos que interesa agregar resultados según los valores concretos de un atributo, o de un conjunto de atributos, que llamamos *criterio de agregación*. Cuidado, esto provocará que por cada valor del criterio de agregación se creará una tupla en la relación de salida. En el fondo, estamos segmentando el resultado de la operación agregada que se haría sobre la relación completa, según los distintos valores del criterio.

Hay que ser conscientes de esto. Si pretendemos añadir atributos adicionales en el resultado de una función de agregación, los cálculos se realizarán para cada uno de los valores, o cada combinación de valores, de esos atributos adicionales. Ergo, estos

atributos no agregados deben ser agrupados. Es una norma tan obligatoria, que como se verá en SQL se convierte en norma sintáctica. No se puede hacer de otra manera.

Para agregar valores según criterios de agregación, hay que poner estos atributos antes de la  $\mathcal{G}$ . En la Caja 5.24 se muestra la versión más sofisticada de una consulta de agregación.

$$A_{k+1}, A_{k+2}, \dots, A_n \mathcal{G}_{f_1(A_1), f_2(A_2), \dots, f_k(A_k)}(r)$$

Caja 5.24. *Expresión relacional para las funciones de agregación.*

Los atributos  $A_{k+1}, A_{k+2}, \dots, A_n$  forman el criterio de agregación. De manera que se puede entender que un valor del criterio de agregación es una combinación de valores de los atributos que lo forman.

La expresión de la Caja 5.24 obtendrá una tupla por cada valor del criterio de agregación, que es cada combinación de valores de los atributos  $A_{k+1}, A_{k+2}, \dots, A_n$ . Por tanto, según los atributos que se utilicen como criterio de agregación variará la cantidad de tuplas de la relación resultante.

En los primeros  $k$  atributos, la tupla debe tener el resultado de agregar los distintos valores de cada  $A_i$  con la correspondiente función  $f_i$ , para  $i = 1, \dots, k$ .

**ejemplo 5.24.** *¿Cuánto ingresa el club por cada deporte?*

**solución** deporte  $\mathcal{G}_{\text{sum}(\text{cuota})}(\text{hace})$

En el ejemplo 5.24 el criterio de agregación es **deporte**. La relación resultante tendrá tantas tuplas como deportes hechos por algún socio, que haya en el club. Si hay algún deporte que no lo hace nadie se supone que el club no ingresa ni un centavo por ese deporte.

**ejemplo 5.25.** *¿Cuánto paga cada socio?*

**solución** pasaporte  $\mathcal{G}_{\text{sum}(\text{cuota})}(\text{hace} \bowtie \text{socio})$

En el ejemplo 5.25 se agrega según el pasaporte sumando la cuota de todos los deportes que haga.

En el SQL se ha tomado la decisión de ignorar en todos los sentidos los valores nulos a la hora de interpretar las funciones de agregación. Así como la expresión  $5 + \text{null}$  es  $\text{null}$ , si hacemos la suma agregada de un atributo numérico,  $A$ , en una relación  $r$  de dos tuplas con un 5 en la primera y nulo en la segunda, entonces la  $\mathcal{G}_{\text{sum}(A)}(r)$

retornará una tupla con el valor del atributo igual a 5.

*En este capítulo ha habido dos partes. Primero, todo aquello que hace referencia a la creación de las bases de datos. Eso ha hecho que nos desviáramos un poco del álgebra relacional para asentar el concepto de modelo relacional como punto de partida en la implementación de las bases de datos. Después, se ha recuperado el hilo del álgebra relacional, pero ya no con la intención de ver la forma de definir las relaciones sino focalizándose en la manera de trabajar con las relaciones definidas según la primera parte. De élla, el concepto más importante es sin duda el de esquema de relación. De la segunda, las operaciones, el producto cartesiano, la reunión interna, y la proyección.*



## Capítulo 6

# Lenguaje Estructurado de Consultas

El lenguaje estructurado de consultas, o SQL de *Structured Query Language*, es sin duda uno de los éxitos de estandarización más notables en la historia de la informática. Seguramente el más antiguo de los lenguajes de programación que se siguen utilizando profusamente en la actualidad. Es de los años setenta, muy anterior a la programación orientada a objetos. Eso hace que haya cosas que no se entenderían si no se ponen en contexto. Es un lenguaje que apareció cuando la programación en lenguajes de alto nivel era mayoritariamente imperativa. Cuando empezó, impresionó a la comunidad científica. Y esta intención pretenciosa se refleja en muchas de sus características. Así mismo, todo ello ha provocado la aparición de distintos dialectos, de manera que en muchas cuestiones es difícil averiguar si el lenguaje que se está utilizando es estándar completamente, o se está haciendo uso de características específicas del SGBD.

En todo este libro no se considera ninguna aplicación de interfaz cliente para desarrollar la base de datos. No conviene. Hacen cosas que hay que aprender a hacer nosotros como responsables de la aplicación. Toman muchas decisiones de formas genéricas, que ensucian el código, y en ningún caso lo optimizan. Como se decía al final del preámbulo, desconfiaremos de las tendencias, y procuraremos mirarlas desde un punto de vista crítico.

Básicamente el SQL se divide en dos lenguajes. El lenguaje que se dedica a la definición de los espacios y al establecimiento de las restricciones, y el que se focaliza en la explotación, o sea inserción, modificación, borrado, y consulta de los datos.

El capítulo abre con una introducción al entorno de trabajo PostgreSQL, que servirá para poder poner en práctica los dos lenguajes que luego se presentan.

## 6.1 Entorno de Trabajo

En adelante nos esperan un montón de pruebas. Es importante, aún que no imprescindible, tener instalado el SGBD PostgreSQL, [7]. El código que se presenta a lo largo de las secciones restantes se puede escribir con cualquier editor de texto, dentro la carpeta del proyecto.

Recuerda del Capítulo 2, que se trabaja sobre una arquitectura cliente servidor tal como se muestra en la Figura 6.1. De todas formas, para poder hacer los experimentos en una sola computadora, podemos conseguirlo montando una máquina virtual, o también conectándonos directamente al *localhost*.

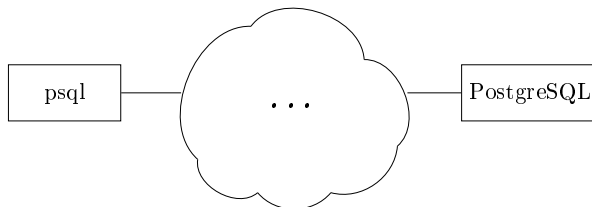


Figura 6.1: *Arquitectura del entorno de trabajo*

Es decir, la gestión de bases de datos utiliza como mínimo dos programas.

Uno es el propio SGBD que reside en el servidor, y como servicio que es, está siempre infatigablemente activo en memoria principal esperando con paciencia solicitudes por los puertos de comunicación, que con el PostgreSQL por defecto es el 5432. El ordenador donde reside el servidor no se para nunca, o casi nunca. Cuando se mide la disponibilidad de las bases de datos se hace con nueves. Un SGBD con una disponibilidad 99 significa que está activo el 99% del tiempo. Y los mejores llegan a tener cuatro nueves, es decir que están un 99.99% del tiempo activos. Un SGBD con una disponibilidad 9999 se para cinco minutos al mes como máximo, para tareas de mantenimiento. Observa que la medida de disponibilidad es logarítmica, o sea que si mejora un 9 significa que el tiempo de no disponibilidad se divide por diez.

El otro es el programa cliente, que desde cualquier computadora puede conectarse dando la ip del servidor, como si llamara por teléfono. El programa que se utilizará para conectarse al servicio se llama `psql`, que es el cliente nativo de postgres. Es un programa de línea de comandos, o sea sin interfaz gráfica. Así nos concentraremos en el SQL, y tomaremos consciencia de qué se puede hacer, y qué no es necesario.

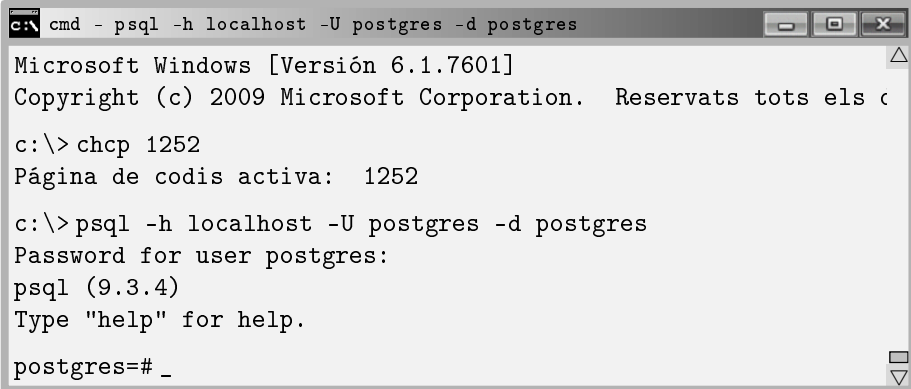
Una vez descargado de [7] y instalado, podemos comprobar que todo está bien antes de empezar invocando directamente el cliente con el comando

```
psql -h localhost -U postgres -d postgres.
```



Es necesario que desde cualquier directorio del cliente se pueda usar el `psql`, cosa que en windows significa modificar el path. Si en las sesiones interactivas interesa poder ver los caracteres de ocho bits, cosa que no tiene más transcendencia, se pueden ajustar los códigos de página haciendo `chcp 1252`.

Todo ello se muestra en la Pantalla 6.1. Las pantallas de ejemplo han sido capturadas sobre el sistema operativo Windows de Microsoft. No obstante, los ejemplos se pueden reproducir sobre terminales con sistemas linux, ya sea Ubuntu, Debian, Red Hat... etcétera.



```

c:\> cmd - psql -h localhost -U postgres -d postgres
Microsoft Windows [Versión 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. Reservats tots els drets.

c:\> chcp 1252
Página de codis activa: 1252

c:\> psql -h localhost -U postgres -d postgres
Password for user postgres:
psql (9.3.4)
Type "help" for help.

postgres=# _
  
```

Pantalla 6.1. *Inicio de sesión.*

Este comando arranca el cliente `psql`, que se conecta al servidor, también llamado *host*, según el parámetro `-h`, con el usuario que diga el parámetro `-U`, en este caso el usuario se llama *postgres*, a la base de datos que indique el parámetro `-d`, que en este caso también es *postgres*. Este usuario se crea durante la instalación, y la base de datos `postgres` también. En PostgreSQL, siempre que un usuario está conectado, lo está en alguna base de datos del clúster. En cambio en MySQL, no.

Todos los parámetros tienen su valor por defecto. Si no se da el `host`, `psql` toma *localhost* como valor por defecto. Si no se da el usuario, `psql` toma el usuario del sistema operativo. Y si no se da el tercer parámetro, entonces el `psql` toma por defecto como nombre de la base de datos el nombre del usuario.

Si en la instalación hemos dado contraseña para el usuario `postgres`, hay que ingresarla en este momento.

Llamamos *prompt* al símbolo que los programas en línea de comandos como el `psql` ponen a principio de línea indicando que esperan alguna entrada de datos. Para el caso que nos ocupa es `postgres=#`, Pantalla 6.1. Se compone de tres partes.

- `postgres` es el nombre de la base de datos actualmente conectada (análoga en

cierta manera a lo que sería un directorio del sistema operativo).

- =, el signo de igual significa que la entrada actual no ha empezado. Prueba a entrar un paréntesis abierto. El prompt cambia a `postgres(#` indicando que tenemos un paréntesis abierto en este comando, y mientras no lo cerremos, se estarán produciendo errores. Lo mismo pasaría si en lugar del paréntesis pulsamos un apóstrofe por ejemplo. Ahora bien, si tecleamos cualquier cosa que no empiece con `\`, y lo entramos, entonces el prompt se transforma en `postgres-#`, o sea un guión en lugar del igual. Eso significa que está esperando un punto y coma para acabar lo que se supone que es un comando SQL. Entra `;` y luego de un mensaje de error recuperarás el estado normal.
- #, el numeral indica que el usuario conectado tiene permisos de superusuario, como en los sistemas `linux`. Para los otros tipos de usuario el prompt tiene un signo de mayor en lugar del numeral, `postgres=>`.

Una vez conectados al servidor como se muestra en la Pantalla 6.1, tenemos dos lenguajes posibles para utilizar. Una de dos,

- O bien empezamos el comando con una contrabarra, `\`, o sea emitimos una orden dirigida al cliente, `psql`.
- O bien acabamos el comando con un punto y coma, `;`, que entonces el `psql` la enviará al PostgreSQL entendiendo que se trata de un comando SQL.

O sea una o la otra, pero no las dos. O empieza con la barra, o acaba con el punto y coma. Prueba a consultar la ayuda del `psql`, con el comando `\?` y la ayuda del SQL haciendo `\h`. Si quieres irte a dormir, pulsa `\q`.

### 6.1.1 Familiarización con el SGBD

Estamos en situación de tocar las primeras teclas. Hagámoslo.

#### Familiarización con el `psql`

Podéis consultar las bases de datos que hay en el clúster haciendo `\l`, que recién instalado el PostgreSQL debería dar la base de datos actual llamada `postgres`, y las dos plantillas que postgres utiliza. La `template0` es una base de datos vacía, a partir de la cual se crea la plantilla `template1` que se usa de origen cuando se crean nuevas bases de datos en este clúster. También se pueden consultar las relaciones existentes en la base de datos, con `\d`, o los usuarios pulsando `\du`. La `d` de estas consultas significa *display*. Todo eso lo dice la ayuda del `psql`.

## Familiarización con el SQL

Una vez conectados y con el cursor en la línea de comandos del PostgreSQL, resulta conveniente familiarizarse probando algunas consultas.

**ejemplo 6.1.** *¿Cuánto suman 2 + 3 y cuánto multiplican?*

**solución** `SELECT 2 + 3, 2 * 3;`

Observa que el PostgreSQL siempre retorna cualquier información en forma de tabla, poniendo `?column?` por defecto a los nombres de los atributos o títulos de las columnas, que es lo mismo. Prueba a poner títulos renombrando las columnas con el operador correspondiente, `AS`, como se indica a continuación.

**solución** `SELECT 2 + 3 AS suma, 2 * 3 AS producto;`

**ejemplo 6.2.** *¿Qué hora es?*

**solución** `SELECT now();`

**ejemplo 6.3.** *¿Cuántos días he vivido?*

**solución** `SELECT now() - '01/01/2001';`

Claro que la solución del ejemplo 6.3 es correcta para una persona que haya nacido el 1 de enero del año 2001.

```

c:\> cmd - psql -h localhost -U postgres -d postgres
postgres=# SELECT 2 + 3 AS suma, 2 * 3 AS producto;
 suma | producto
-----+-----
      5 |         6
(1 row)

postgres=# SELECT now();
      now
-----
2017-02-11 16:12:36.799+02
(1 row)

postgres=# SELECT now() - '01/01/2001';
 ?column?
-----
5885 days 15:12:57.992
(1 row)

postgres=# _

```

Pantalla 6.2. *Familiarización con el SQL de PostgreSQL.*

## 6.2 Lenguaje de Definición de Datos

El Lenguaje de Definición de Datos, o DDL de *Data Definition Language*, es la parte del SQL que se utiliza para construir una base de datos. Y eso significa proporcionar los datos necesarios para poder hacer la explotación.

### 6.2.1 Multiconjuntos

El SQL trata las relaciones como multiconjuntos. O sea, admite que las relaciones tengan elementos repetidos. Esta es la diferencia más importante entre el álgebra relacional y el lenguaje de consultas.

Y, ¿Por qué?

Bien, igual que cualquier magnitud numérica da más información que la simple existencia de la magnitud, así como un valor entero siempre guardará una información más precisa que un booleano, convertimos las relaciones en contadores. En lugar de guardarnos que un cosa ocurre, nos guardamos cuantas veces ocurre la cosa. No es que lo hayamos de hacer siempre, pero sí que nos reservamos la posibilidad de hacerlo. El álgebra relacional no sabe de magnitudes numéricas, y por eso no entra en estas distinciones.

### 6.2.2 Tablas

En adelante, utilizaremos el término *tabla* para indicar aquellas relaciones físicas donde guardamos los datos reales de una base de datos, concepto análogo a los ficheros de un sistema operativo. Sin embargo, cada tabla no solo tiene un nombre sino que, a diferencia de un fichero en un sistema operativo, tiene también un conjunto de atributos, de los que a parte del sus nombres también hay constancia de sus dominios, que significa los tipos o el conjunto de valores que pueden almacenar.

Parece pues que se trate del mismo concepto de relación que se ha estado utilizando en el Capítulo 5. Pero no es el caso.

Cualquier expresión relacional se puede evaluar resultando una relación, pero no una tabla. Es decir, el hecho de que la relación se guarde físicamente en algún dispositivo hace que le llamemos tabla. En definitiva estamos hablando de la persistencia de una relación. Almacenarse en el disco parece que no tenga más trascendencia. Pero sí la tiene, igual que los fundamentos de los edificios o las raíces de los árboles, la parte que toca al suelo sirve para soportar la lógica que ven nuestros ojos.

Y por todo ello, entendemos que la importancia de una relación va directamente atada a su persistencia, cosa que establece una clasificación en las categorías de las relaciones, en función de su volatilidad.

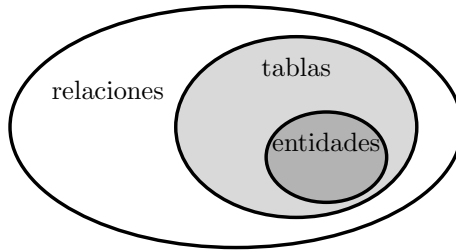


Figura 6.2: Clasificación de las relaciones de una base de datos.

A partir del dibujo de la Figura 6.2 se puede deducir que

- todas las entidades de un modelo ER se implementan en tablas, que son relaciones que en particular tienen soporte físico,
- todas las tablas, entre las que también hay las restantes del modelo relacional, son relaciones que se guardan en soporte físico.
- las consultas que se pueden guardar en una base de datos son relaciones, pero no tienen un soporte físico para los datos, se guarda la expresión relacional que obtiene el resultado de la consulta, no la colección de datos.

Las tablas que constituyen un base de datos son las que aparecen en su modelo relacional.

### 6.2.3 Metadatos

*Metadatos* se podría interpretar como alguna cosa que va más allá de los datos. De hecho, el prefijo *meta*, a parte de "más allá", a menudo conviene entenderlo como autoreferencia. Metamorfosis es algo que va más allá de la forma, metalenguaje es aquella parte del lenguaje que habla del mismo lenguaje. De alguna forma, es como si ir más allá tuviera que ver con la autoreferencia. Y metadatos significa aquellos datos que contienen información sobre los datos. Los metadatos en sí, son una base de datos. La base de datos de la base de datos.

Forman parte de los metadatos de una base de datos, los nombres de las relaciones, los nombres y los dominios de los atributos, las restricciones de clave primaria y clave foránea, restricciones de existencia y de unicidad, y otras cosas.

Los SGBDs guardan toda esta información de la misma manera que lo guardan todo. Es decir, en tablas. A veces a los metadatos se les llama catálogo de la base de datos.

Quien desarrolla una base de datos no tiene ninguna interacción directa con los metadatos. Ahí podría consultar, por ejemplo el tipo de un atributo de una tabla, pero como esa información también la tiene en el código de creación de la tabla, cosa que ha hecho la misma persona, acostumbra a consultar antes el código fuente.

Todo aquello que forma parte de los metadatos pasa automáticamente a considerarse objetos de la base de datos, y por tanto, en PostgreSQL tienen un identificador de objeto, OID. Eso también significa que puede tener un nombre. Es decir, cada restricción, por ejemplo, puede tener un nombre.

## 6.2.4 Dominios de los atributos

Para la creación de las tablas correspondientes al modelo relacional hay que establecer un tipo para cada atributo. Seguidamente se ven las posibilidades existentes.

Cualquier persona que haya desarrollado aplicaciones en lenguajes tipificados tiene consciencia de qué son los tipos de datos de un lenguaje de programación. Y conviene tener presente que, en el fondo, son los tamaños de los datos, en bytes. Esta es una de las abstracciones más notables en la pila de conceptos que constituyen una máquina computadora. A una cosa que estrictamente se corresponde a un tamaño, se le llama tipo. Es muy interesante. Es decir, cuando en un programa se declara que una variable es entera, al compilador tan solo le interesa que ocupa cuatro bytes.

### Tipos primitivos

Se llaman tipos primitivos los que no tienen ningún tratamiento interno asociado, de manera que se corresponden sencillamente con el espacio que se les asigna, y cualquier uso que se haga será por medio de procedimientos implementados en algún lenguaje de programación.

- **BOOLEAN**

Los atributos de tipo booleano pueden valer *cierto*, o *falso*. Ocupan un byte. Y de hecho, no acostumbran a ser demasiado propensos a ser almacenados en bases de datos, ya que normalmente por el mismo precio podemos guardar alguna información más precisa. Tener cualquier valor significa existir, y por tanto siempre es mejor guardar el valor, que la simple existencia de él.

- **CHARACTER**

Este tipo también se puede llamar `CHAR`. Raramente se utilizan atributos de este tipo. Normalmente el tipo carácter es usado como elemento de las secuencias que son las cadenas de caracteres. Pueden valer cualquier carácter de la tabla ASCII. Por tanto ocupan un byte, pero hay que tener en cuenta que de los 256 valores que pueden tomar, solo los 128 primeros son estándar. Y de estos, los 32 primeros no son imprimibles, como el retorno de carro, el salto de línea o la tabulación. La tabla de códigos imprimibles estándar se muestra en el Apéndice B.

- **INTEGER**

Números enteros en el intervalo  $[0..2^{32} - 1]$ , ya que normalmente ocupan cuatro bytes, e igual que con tres cifras decimales podemos representar mil números,  $10^3$ , con 32 cifras binarias podemos representar  $2^{32}$ . Eso significa que si la aplicación que los trata los interpreta sin signo, el valor puede ser como mínimo 0 y como máximo  $2^{32} - 1$ . En cambio, si la aplicación los interpreta con signo, tenemos la mitad de números positivos que teníamos. Entonces se podrá representar números enteros en el intervalo  $[-2^{31}..2^{31} - 1]$ .

- **DECIMAL**

Cuando interesa almacenar variables continuas, como pesos, o áreas en metros cuadrados, entonces utilizamos este tipo de datos. Podemos establecer la cantidad de cifras que deseamos en total, y cuantas tras la coma. Es decir, es un tipo de datos parametrizado con dos cantidades. Por ejemplo, el máximo valor de tipo `DECIMAL(5,2)` es el 999.99, ya que utiliza cinco cifras de las cuales dos son decimales. El hecho de que un tipo de datos se preocupe de su representación impresa no es propio de los SGBDs, que en principio no tienen ninguna consideración relativa a la entrada y salida. Pero bien, es una cuestión atávica del SQL, y así es. En lugar de este tipo, también se puede utilizar el `DOUBLE PRECISION`.

## Tipos alfanuméricos

Los tipos alfanuméricos son secuencias de caracteres. Un valor constante de este tipo va entre comillas simples o apóstrofes. Como `'comercial'`. La concatenación de cadenas de caracteres utiliza dos barras verticales, *AltGr+1*, el símbolo `||`. O sea, `'hola,'||' qué tal'` es igual a `'hola, qué tal'`.

El SQL tiene el operador `LIKE`, que usa patrones con dos caracteres comodín. El guión bajo y el tanto por ciento. El guión bajo sirve para sustituir exactamente una letra. El tanto por ciento para sustituir varias, que pueden ser muchas, una, o ninguna. Este operador retorna un booleano, cierto si el atributo encaja en el patrón. Por ejemplo, `A LIKE 'B_'`, retornará cierto siempre que el valor del atributo `A` sea de dos letras y empiece con B mayúscula. `A LIKE '%pepito%'` retornaría cierto si el valor de la cadena `A` contuviera la subcadena *pepito*.

- **CHARACTER(*n*)**  
También se puede llamar **CHAR(*n*)**. Representa una secuencia de caracteres de longitud fija que si ocupa menos de *n* caracteres se rellena con blancos por la derecha. Si ocupase más, se produciría un error.
- **CHARACTER VARYING(*n*)**  
También se puede llamar **VARCHAR(*n*)**. Secuencia de caracteres de longitud variable, siendo *n* un parámetro optativo que si se da indica la máxima. Y por defecto, si se declara un atributo de una tabla como **VARCHAR** sin el parámetro, entonces la longitud máxima es 255, o sea, que esa longitud se guarda en un byte.
- **TEXT**  
Secuencias largas de caracteres. En principio, la declaración de este tipo hace que el valor se guarde en un espacio diferente a la tabla donde pertenece. De manera que en la tabla tan solo hay una referencia a algún lugar donde se guardan los valores de este tipo. No obstante, en PostgreSQL, no es el caso, de forma que es sinónimo de **VARCHAR**.

## Tipos temporales

Como dice la introducción del capítulo, desde el principio de su tiempo, el SQL marcó diferencias respecto los otros lenguajes de programación por su proximidad extraordinaria al lenguaje humano. En ese sentido incorporó tipos de datos para representar valores temporales juntamente con una serie de operaciones. Para trabajar con constantes de este tipo hay que poner los valores entre comillas simples, como para los textos.

Los tipos de datos relacionados con unidades de tiempo son

- **DATE**  
Para representar fechas. Si un valor de este tipo vale '30/12/2017' y le sumemos 15, entonces valdrá '14/01/2018'.
- **TIME**  
Para representar horas, minutos, segundos, y milésimas de segundo. Un valor, '13:15:12.123'.
- **TIMESTAMP**  
Unión, o concatenación de los dos tipos anteriores. Es decir, un valor podría ser '2017-12-30 15:15:12.123'. Además, existe la posibilidad de declararlo como **TIMESTAMP WITH TIME ZONE**, que significa guardar adicionalmente, en la misma estructura de datos, un entero con signo de -12 hasta a +12, que significa el cambio de huso horario respecto al meridiano de greenwich. Un valor con esta variante es '2017-12-30 15:15:12.123-05'. Este tipo de datos es excepcional, porque depende del lugar donde se represente puede variar el valor numérico.



- **INTERVAL**

Representa la diferencia entre dos de los valores de los tipos anteriores. Utiliza unidades textuales, como por ejemplo '3 days'. Es interesante por las operaciones que proporciona.

Es notorio pues que podemos representar los días en formato '21/01/2017', o bien '2017-01-21'. El primero de los formados porque es al que estamos acostumbrados. El segundo porque es un formato con la ventaja que si quitamos los guiones la cifra resultante es creciente con el tiempo. Es decir, cuando más hacia el futuro, mayor resulta el número que en el ejemplo sería 20170121. Todo ello, pero, trae confusión a menudo. Así que cuidado.

Para cualquiera de estos tipos, los SGBDs proporcionan funciones de extracción de las partes. Podemos obtener fácilmente el año, el mes o el día de una fecha, y también saber a qué día de la semana corresponden.

### **Tipos autoincrementales**

De tipos autoincrementales solo hay uno. En cada SGBD tiene su propio nombre. El Access le llama *autonumérico*, el MySQL *autoincrement*, el PostgreSQL *serial*,...

El funcionamiento de los atributos declarados como autoincrementales es simple. El valor propiamente dicho es un entero positivo. Y para cada nueva inserción se incrementa en una unidad. Entonces, la diferencia entre tuplas de una relación que contenga un atributo autoincremental viene garantizada por el propio SGBD, haciéndolo inútil. La primera virtud de un SGBD es el control de existencia y unicidad de algunos de sus valores. Si utilizamos atributos autoincrementales, estamos desperdiçando esta ventaja.

Si un atributo es de tipo autoincremental, entonces necesariamente debe ser clave primaria, y por tanto, solo puede haber un atributo de este tipo en cada relación. Si no es clave primaria, ser autoincremental no tiene ningún sentido. Las claves foráneas que apuntan claves primarias autoincrementales deben ser números enteros.

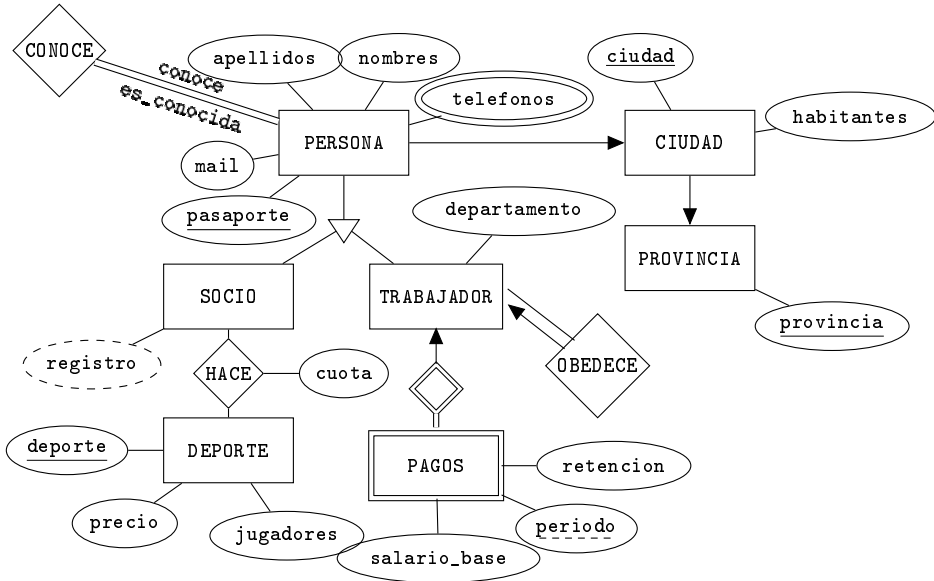
Debido a que desde que surgieron las bases de datos cualquier aplicación comercial ha hecho gala de su uso, ha habido quien sin introducirse en el conocimiento de las bases de datos relacionales las ha usado como si fueran simples sistemas de archivos, haciendo aplicaciones mal hechas. Y para eso sirve esencialmente este tipo de datos. Si identificamos a las personas por número de pasaporte, por ejemplo, no tiene ningún sentido que el sistema permita dar dos personas distintas con el mismo número. Utilizando autoincrementos como claves primarias permitimos acciones que no tienen sentido. Además, en estas aplicaciones mal hechas las tareas de verificación de existencia o de unicidad quedan en manos del programa cliente, recodificando

el trabajo que el SGBD haría con toda seguridad de manera más eficiente, y no es poca. En nuestro planeta mundo, corren barbaridades ingentes de aplicaciones mal programadas en ese sentido, repitiendo malamente muchas tareas, y diluyendo las responsabilidades de manera desordenada. Ese es un extremo en que el intrusismo profesional es palmario.

Su uso tan solo puede justificarse en agregaciones del modelo ER, siempre que el diseñador no tenga bastante imaginación para identificar la relación entre dos o más entidades con alguna palabra más semántica, ya que la característica más importante de los atributos autoincrementales es que de semántica, cero.

### 6.3 Construcción de la Base de Datos

En esta sección se desmenuza minuciosamente la implementación de una base de datos desde la nada. Por eso, recuperamos el diseño de la base para el club deportivo del Capítulo 5 en el Modelo 6.1.



Modelo 6.1. Modelo ER de la base de datos.

En el mismo Capítulo 5 este modelo ER ha sido transformado en el modelo relacional 6.2.

```

provincia(provincia)
ciudad(ciudad,habitantes,provincia)
persona(pasaporte,nombres,apellidos,mail,ciudad)
telefonos(pasaporte,telefono)
conoce(conoce,es_conocida)
socio(pasaporte,registro)
trabajador(pasaporte,departamento,obedece)
deporte(deporte,precio,jugadores)
hace(socio,deporte,cuota)
pagos(pasaporte,periodo,salario_base,retencion)
    
```

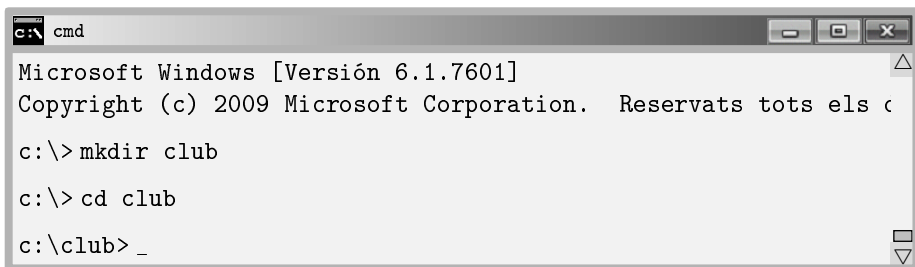
Modelo 6.2. Modelo Relacional del diseño del Modelo Entidad Relación 6.1.

Bien, empezamos. La primera cosa que se hace es dar una palabra al proyecto. Le llamamos `club`, que ya sabemos que significa club deportivo.

### 6.3.1 Espacio de Trabajo

Creamos un directorio nuevo con nombre `club`, donde pondremos todos los archivos relacionados con el proyecto. Un directorio es una carpeta. Para crearla, podemos hacerlo con cualquier interfaz gráfica del sistema operativo. Aún así, como seguidamente debemos abrir una terminal y situarnos dentro, también podemos crear la carpeta desde la misma consola. Utilizamos los términos *consola* y *terminal* como sinónimos. En este caso teclearemos el comando *make directory*, `mkdir`. Y una vez creado, estableceremos ese directorio por defecto, o sea nos meteremos dentro, con el comando de *change directory*, `cd`.

Hasta aquí, pues, todo lo que hemos hecho se puede ver en la Pantalla 6.3.



```
cmd
Microsoft Windows [Versión 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. Reservats tots els drets.

c:\> mkdir club

c:\> cd club

c:\club> _
```

Pantalla 6.3. *Carpeta del proyecto club.*

### 6.3.2 Escrip Principal

Llamaremos *escripts* a los archivos de texto plano, o sea de caracteres de siete bits y con extensión `sql` que escribiremos. Las letras acentuadas, diéresis y eñes no son caracteres de siete bits, son de ocho, y por tanto, no utilizaremos esos caracteres ni en el nombre ni en el código de los escripts. Podrán aparecer, eso sí, en cadenas constantes de caracteres, ya sean datos, o mensajes que se enviarán finalmente a la aplicación cliente para que los pueda mostrar.

Dentro del ciclo de vida de la aplicación, en la fase de desarrollo remontaremos de la nada la base de datos en cada cambio que pueda suponer un impacto en la estructura... cientos de veces. Por eso, la primera cosa que hace el escript principal es eliminar la base de datos del sistema y volverla a crear. En la Caja 6.1 se muestra el contenido del archivo `club.sql`.

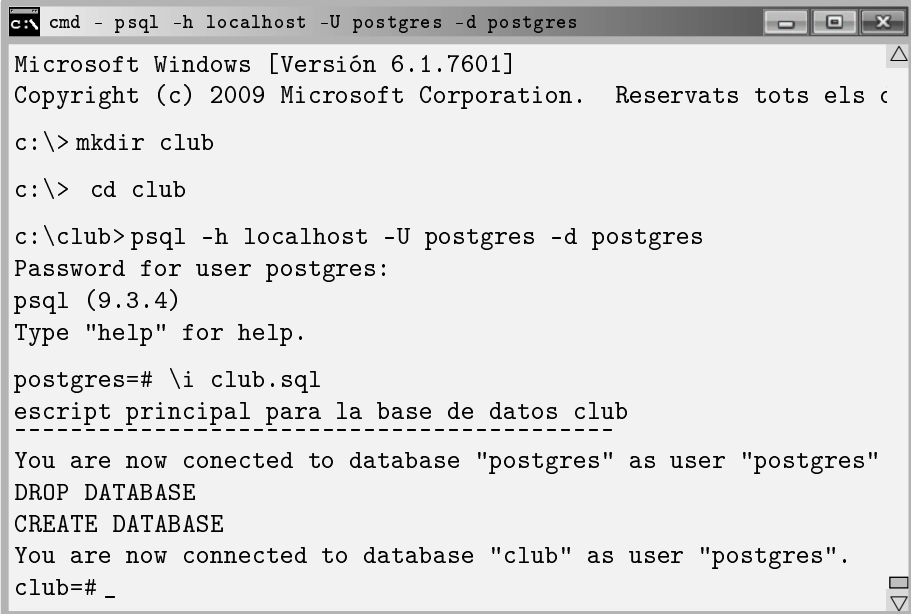
```
\echo escript principal para la base de datos club
\echo -----
\c postgres
DROP DATABASE club;
CREATE DATABASE club;
\c club
```

Caja 6.1. *Contenido inicial del archivo club.sql, escript principal.*

El contenido de la Caja 6.1 se justifica a continuación línea a línea. Para ponerse en situación, hay que entender que ejecutamos este escript estando conectados a la base `club`, que a parte de la primera vez, ocurrirá siempre.

- `\echo mensaje` sirve para que el `psql` imprima por pantalla el texto que sigue al nombre del comando hasta el final de línea. De paso, también es útil como comentario en el mismo escript. Ésa es la función de las dos primeras líneas.
- `\c postgres` sirve para conectarse a la base de datos `postgres`. La intención es dejar de estar conectados a la base actual, `club`, para poderla borrar, ya que no se puede borrar la base activa. Observa pues el uso del comando `\c` del `psql` para cambiar la base actual. Este comando cierra la sesión y abre otra con el mismo usuario en la base a la que se conecta. Si se emite sin ningún parámetro sirve para que el PostgreSQL nos informe de qué usuario somos, y cuál es la base activa, que es la que hay en el prompt.
- `DROP DATABASE club;` borra la base de datos `club`, y por tanto todo lo que hay dentro, aún que de momento no hay nada. Este comando provocará un error la primera vez que se ejecute, porque la base `club` no existirá. Ningún problema. Además, se puede evitar poniendo `DROP DATABASE IF EXISTS club;`. Es decir, borra la base de datos si existe, en tercera persona del singular del presente simple. Pero por una sola vez que se producirá el error, no vale la pena añadir código.
- `CREATE DATABASE club;` crea la base de datos en el clúster. A partir de este momento, cuando hacemos `\l` aparecerá la base `club` como una base más.
- `\c club` vuelve a establecer la base de datos actual en la base `club`. Así las tablas que creemos en adelante se crearán en esta base.

En la Pantalla 6.4 se puede observar la importación del escript desde la línea de comandos del `psql`. Eso es luego de haber hecho la importación por segunda vez. Es decir, no se ha mostrado el error que se comenta más arriba.



```

c:\> cmd - psql -h localhost -U postgres -d postgres
Microsoft Windows [Versión 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. Reservats tots els drets.

c:\> mkdir club

c:\> cd club

c:\club> psql -h localhost -U postgres -d postgres
Password for user postgres:
psql (9.3.4)
Type "help" for help.

postgres=# \i club.sql
escript principal para la base de datos club
-----
You are now conected to database "postgres" as user "postgres"
DROP DATABASE
CREATE DATABASE
You are now connected to database "club" as user "postgres".
club=# _

```

Pantalla 6.4. *Importación del club.sql desde el psql.*

Usamos el comando `\i` para incluir o importar el escript, que debe estar en el mismo directorio desde el que hemos invocado el `psql`.

Cada vez que se emite un comando `\c` de conexión a una base, PostgreSQL nos dice quién somos y en qué base estamos, por eso al incluir el escript, que contiene dos comandos `\c`, aparece en pantalla dos veces los mensajes correspondientes. En medio de ellas, vemos que cuando un comando SQL resulta exitoso, el PostgreSQL nos informa repitiendo el nombre del comando.

Llegados este punto, es el momento de ponerse a crear los objetos de la base de datos `club`.

### 6.3.3 Creación de los Dominios

Tal como se ha visto en la Sección 5.2.1, siempre que se desee definir algún dominio conviene hacerlo antes que nada. La definición de dominios tiene por finalidad dejar la responsabilidad de verificar los valores de un atributo en manos del SGBD. Si un atributo es de un dominio concreto, ya nos podemos despreocupar, que seguro que los valores que contiene responden a las restricciones que imponga el dominio.

La instrucción `CREATE DOMAIN`, inexistente en MySQL, sirve para crear dominios. Después del nombre del dominio hay el tipo de datos que pretendemos restringir. La

restricción viene marcada por la palabra clave `CHECK` seguido de una expresión booleana entre paréntesis con la palabra clave `VALUE` que hace el papel de los valores del dominio a la hora de definir las restricciones. En el caso de la Caja 6.2 se utiliza el operador lógico `IN` que se verá en la Sección 6.4.11.

En el ejemplo tenemos que el atributo `departamento` de la relación `trabajador` puede ser *administración*, *comercial*, o *entrenador*.

```
CREATE DOMAIN dominio_departamento TEXT
    CHECK (VALUE IN ('administración','comercial','entrenador'));
```

Caja 6.2. *Dominio para el atributo departamento de la tabla trabajador.*

Si el dominio se utiliza tan solo en una tabla, podemos crearlo en el mismo script donde creamos la tabla. Si se utilizase en más de una, entonces haríamos un script especial de creación de dominios. Para el caso, tan solo se utiliza en la tabla `trabajador`, así que para esta tabla en concreto, el script empezará como se indica en la Caja 6.2.

No hay demasiada diferencia entre la creación de un dominio definido por enumeración, como el de la Caja 6.2 y una entidad de soporte. O sea, si a la base de datos para el club le añadiésemos una tabla llamada `valores_departamento` con una sola columna, clave primaria, que contuviera tres registros con los valores *administración*, *comercial*, y *entrenador*, entonces se podría indicar en la tabla `trabajador` que el campo `departamento` es clave foránea de la tabla `valores_departamento`. Ventaja. Si algún día aparecen nuevos departamentos, no será necesario remontar la base de datos, tan solo hacer una nueva inserción en la tabla `valores_departamento`. Inconveniente. Hace que las consultas que involucran el atributo `departamento` sean menos eficientes, cosa poca importante. Por ello, MySQL no usa dominios.

Por otra parte, un dominio no se podrá sustituir por una entidad de soporte cuando se defina con otros criterios que no sean por enumeración, como en el ejemplo de la Caja 6.3.

```
CREATE DOMAIN positivo INTEGER
    CHECK (VALUE > 0);
```

Caja 6.3. *Creación de un dominio para los números positivos.*

De la base de datos del club deportivo queda por definir un dominio un poco más complicado. Es el dominio que definimos para el campo `mail` de la relación `persona`.

En la Sección 6.2.4 se ha introducido el operador `LIKE`, que utiliza patrones hechos con dos caracteres comodín que son el guión bajo y el tanto por ciento.

- El guión bajo significa exactamente una letra, cualquiera.
- El tanto por ciento substituye  $n$ , que puede ser ninguna.

Este operador es especialmente útil para restringir formatos de atributos textuales. Por ejemplo, en la Caja 6.4 se presenta un dominio que serviría para verificar las direcciones de correo electrónico.

```
CREATE DOMAIN dominio_mail TEXT
CHECK (VALUE LIKE '%_@%_.%_');
```

Caja 6.4. *Dominio para el atributo mail de la tabla persona.*

En la Caja 6.4 definimos el tipo de datos `dominio_mail`. Los valores de los atributos de este dominio deben ser de tipo texto, deben contener una arroba, y como mínimo una letra antes de la arroba y una luego. Observa que `%_` significa como mínimo una letra. Además, también debe contener un punto y dos letras o más, al final.

Profundizar en el tema de la descripción de formatos pasa por hacer un incursión en el mundo de las expresiones regulares. Es una norma muy extendida, cosa que impide introducirlas aquí, pero queda dicho que existe una nomenclatura estándar de definición de formatos textuales que se llama expresiones regulares.

### 6.3.4 Creación de las Tablas

Pondremos cada tabla en un escript con su nombre. O sea, que desde el escript principal deberemos incluir tantos escripts como tablas haya. Y todo en el mismo orden del modelo relacional.

Actuaremos metódicamente. Primero haremos el escript, luego probaremos a leerlo desde el `psql`, y cuando estemos seguros que no tiene errores de sintaxis, entonces crearemos una carpeta que cuelgue de la carpeta principal y que se llame igual que



el escript, o sea, igual que la tabla que implementa. Moveremos el escript a la nueva carpeta, y entonces añadiremos la línea correspondiente al escript principal.

Empezamos pues haciendo un nuevo archivo de texto llamado `provincia.sql`, con el contenido que se muestra en la Caja 6.5. El comando de SQL `CREATE TABLE` sirve para crear tablas.

```
\echo ----- tabla provincia
CREATE TABLE provincia (
    provincia TEXT,
    CONSTRAINT provincia_repetida PRIMARY KEY(provincia)
);
```

Caja 6.5. Archivo `provincia.sql`.

Para poder probar que la tabla `provincia` ha sido bien creada hay que hacer la importación desde la línea de comandos del `psql` con `\i provincia.sql`.

En la Caja 6.5, la sintaxis para establecer la restricción de clave primaria se presenta en su forma más completa, osea dándole nombre a la restricción, `provincia_repetida`. Veamos su utilidad con un experimento de cuatro pasos, a pesar de que eso signifique anticipar el comando de inserción, `INSERT INTO`.

1. Importamos el archivo con `\i provincia.sql`.
2. Insertamos la provincia *Napo*, con `INSERT INTO provincia VALUES('Napo');`.
3. Volvemos a insertar la provincia *Napo* pulsando flecha arriba, o volviéndolo a teclear.
4. Observamos el mensaje de error.

Todo ello se puede ver en la Pantalla 6.5.

```

c:\ cmd - psql -h localhost -U postgres -d club
club=#
club=# \i provincia.sql
----- tabla provincia
CREATE TABLE
club=# INSERT INTO provincia VALUES('Napo');
INSERT 0 1
club=# INSERT INTO provincia VALUES('Napo');
ERROR: duplicate key value violates unique constraint "provincia_
DETAIL: Key (provincia) = (Napo) already exists.
club=# _

```

Pantalla 6.5. Prueba de clave primaria con nombre repetido.

Es decir, dar nombre a una restricción sirve para que cuando se produzca una violación el nombre dado aparezca en el mensaje de error. Eso puede servir a veces si entre el programa cliente y la base de datos hay algún protocolo relativo a los códigos de error. Si no se da ningún nombre a la restricción, en lugar de `provincia_repetida` se llamaría `provincia_pkey`, de primary key. En adelante aceptaremos estos nombres por defecto.

Una vez importado correctamente el escript, creamos una carpeta `provincia`, y dejamos el archivo `provincia.sql` en ella. La estructura del espacio de trabajo debería ser la que se muestra en la Figura 6.3.

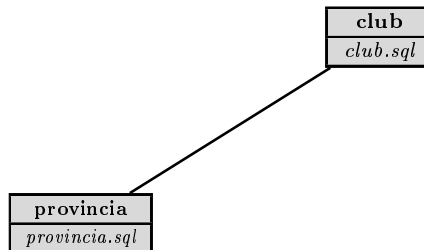


Figura 6.3: Estructura actual del espacio de trabajo.

La caja `provincia` representa la carpeta que cuelga de la carpeta principal. Y contiene el archivo con el mismo nombre que la carpeta y que la tabla que implementa. Cada vez que acabemos el escript de una tabla, añadiremos una carpeta igual que hemos hecho ahora. Y finalmente, al escript principal añadimos la nueva línea

```
\i 'provincia\\provincia.sql'.
```

Como el escript principal se ejecuta estando en la carpeta del proyecto, hay que dar la ruta relativa a ese directorio cuando se hace el import. En `psql` podemos usar

la doble contrabarra para indicar el separador de directorios.

A continuación, hacemos un nuevo escript para la tabla ciudad.

```
\echo ----- tabla ciudad
CREATE TABLE ciudad (
    ciudad TEXT PRIMARY KEY,
    habitantes INTEGER,
    provincia TEXT,
    CONSTRAINT provincia_inexistente
        FOREIGN KEY(provincia) REFERENCES provincia(provincia)
);
```

Caja 6.6. *Primera propuesta para el archivo ciudad.sql.*

En la Caja 6.6 la declaración de clave primaria se ha hecho de la forma más abreviada posible. También se introduce una clave foránea en su forma más completa, dándole nombre. Si no le hubiésemos dado nombre se llamaría `ciudad_provincia_fkey`.

## Integridad referencial

Introduciendo una clave foránea profundizamos en el concepto de integridad referencial. Se establece que los valores posibles para el atributo `provincia` de la tabla `ciudad` son aquellos que haya en la columna `provincia` de la tabla `provincia`. Eso abre distintas opciones por lo que respecta a la actualización. O sea, si una ciudad apunta a una provincia, hay que decidir qué hacemos si la provincia se borra o se cambia de nombre.

- `ON DELETE RESTRICT`, o `ON UPDATE RESTRICT`

Si en la Caja 6.6 hubiésemos añadido una última línea como se muestra en la Caja 6.7, entonces estaríamos imponiendo que no se pueda borrar una provincia mientras exista alguna ciudad que la apunte. Es decir, que el SGBD retorne un mensaje de error si se intenta borrar la provincia. Si en lugar de `DELETE` dijera `UPDATE`, la cosa sería igual cuando se intentase modificar el nombre de una provincia. Es decir, el SGBD lo prohibiría retornando un error.

```

\echo ----- tabla ciudad
CREATE TABLE ciudad (
  ciudad TEXT PRIMARY KEY,
  habitantes INTEGER,
  provincia TEXT,
  CONSTRAINT provincia_inexistente
    FOREIGN KEY(provincia) REFERENCES provincia(provincia)
    ON DELETE RESTRICT
);

```

Caja 6.7. *Restricción de eliminación para la clave apuntada.*

Esta opción es la que hay implementada por defecto, es decir, la Cajas 6.6 y 6.7 ejercen la misma función.

- ON DELETE CASCADE, o ON UPDATE CASCADE

En la Caja 6.8 se detalla otra opción, estipular la eliminación, o actualización, en cascada. Eso significa, en el caso de la eliminación, que si se borra una provincia se borren también todas las ciudades que la apunten. Y en el caso de la actualización, que si se cambia el nombre de una provincia se cambie también en todas las ciudades que la apunten, opción más frecuente.

```

\echo ----- tabla ciudad
CREATE TABLE ciudad (
  ciudad TEXT PRIMARY KEY,
  habitantes INTEGER,
  provincia TEXT,
  CONSTRAINT provincia_inexistente
    FOREIGN KEY(provincia) REFERENCES provincia(provincia)
    ON DELETE CASCADE
);

```

Caja 6.8. *Restricción de participación total de CIUDAD en PROVINCIA*

Se utiliza el término CASCADE en el sentido de que, si por ejemplo se borra una provincia, y como consecuencia se suprimen todas las ciudades que la apuntan, entonces podría ocurrir que si la clave foránea de persona a ciudad también se declarase ON DELETE CASCADE, entonces al eliminar la provincia, además de borrar todas las ciudades de aquella provincia, también se suprimirían todas las personas de aquellas ciudades.

- ON DELETE SET NULL, o ON UPDATE SET NULL

Esta es la opción que nos interesaría si no fuese imprescindible saber de qué provincia son las ciudades. Significa que en caso que se elimine una provincia, automáticamente se ponga un valor nulo en la columna provincia de las ciudades que la apuntaban.

- `ON DELETE SET DEFAULT`, o `ON UPDATE SET DEFAULT`

A cualquier atributo se le puede dar un valor por defecto en el momento de la declaración poniendo la palabra clave `DEFAULT` justo luego del tipo. Si eso se hace con una clave foránea, entonces también se puede establecer que en caso de que el registro apuntado se modifique o se elimine se ponga automáticamente el valor por defecto en la clave foránea.

En cualquier caso, fijate que aunque parezca que hay una analogía entre borrar y actualizar, no es exactamente así. Hay que tener en cuenta que una eliminación es una operación que afecta un registro entero, en cambio una actualización puede afectar tan solo un campo. Por tanto, las normas de comportamiento en los casos de actualización solo harán efecto cuando se actualice el valor de las claves.

Por otra parte, siempre se puede establecer una restricción de existencia para cualquier atributo poniendo `NOT NULL` justo luego del tipo del atributo. Por tanto, una buena implementación debería hacer coherentes las dos restricciones.

Es decir, eliminaciones en cascada deberían ser asociadas a atributos no nulos, cosa que en capítulos anteriores se había llamado participación total de la entidad  $N$ , en la relación  $1:N$  que implementa esta clave foránea. O a la inversa, si permitimos que una clave foránea pueda valer nulo en algún registro, entonces lógicamente deberíamos sentenciar `ON DELETE SET NULL` para esta clave foránea. En este segundo caso estaríamos implementando una participación parcial. Todo ello se sintetiza en la Tabla 6.1.

participación	<code>ON DELETE</code>	clave foránea
total	<code>CASCADE</code>	<code>NOT NULL</code>
parcial	<code>SET NULL</code>	

Tabla 6.1: *Coherencia en las participaciones.*

Es decir, de la Tabla 6.1 hay que decidirse entre un fila, o la otra. Pero no opciones cruzadas, que significarían incoherencia. Y como siempre que se habla de participaciones, vale la pena recordar que la decisión depende de si interesa o no mantener los registros del lado  $n$  cuando se elimine un registro del lado  $1$ , que en el caso del ejemplo es si deseamos mantener las ciudades aunque se suprima su provincia. O dicho de una otra forma, si podemos registrar ciudades sin saber la provincia donde pertenecen.

La opción por defecto, `RESTRICT`, no define ningún tipo de participación. Precisamente es la opción útil cuando quien desarrolla la aplicación de la base de datos no

tiene las cosas claras. No se debe entender como un problema, simplemente se trata de postponer una decisión. Quien no tiene las cosas claras es el usuario final, y por tanto nadie se atreve a tomar ninguna decisión. La opción `RESTRICT` es útil cuando se desea dejar en manos del programa cliente la forma de proceder. Claro, el programa cliente puede dejarlo en manos del usuario final en cada caso. De manera que no hay ninguna regla válida para actuar cuando se borre o se modifique el registro de la tabla apuntada.

Lógicamente, si podemos registrar ciudades ignorando la provincia donde pertenecen, las deberíamos mantener existentes si la provincia desaparece. Eso es lo que dice la Tabla 6.1.

Para continuar, decidimos que la participación será parcial. Queremos mantener las ciudades aunque no sepamos, o no existiera, la provincia donde se encuentran, ya que pueden ser ciudades de otros países, y entonces no existirá ninguna provincia.

Por otra parte, si una provincia cambia de nombre, por la razón que sea, entonces deseamos que todas las ciudades se actualicen con el nuevo nombre. Eso ocurre al corregir un error ortográfico, que alguien cometió tiempo atrás, al insertar un dato.

El `escript` de la tabla `ciudad` queda finalmente como se muestra en la Caja 6.9.

```
\echo ---- tabla ciudad
CREATE TABLE ciudad (
  ciudad TEXT PRIMARY KEY,
  habitantes INTEGER,
  provincia TEXT REFERENCES provincia
    ON DELETE SET NULL
    ON UPDATE CASCADE
);
```

Caja 6.9. *Archivo ciudad.sql.*

Observa que en la versión final del archivo `ciudad.sql` se ha hecho uso de las versiones más breves, tanto para la clave primaria como para la clave foránea. En concreto, se ha omitido el nombre del atributo en la tabla `provincia`, ya que si estamos diciendo que la apuntamos, el SGBD supone que será a su clave primaria. O sea que cuidado. Hay que entender bien sentencias como `provincia TEXT REFERENCES provincia`. La primera aparición de `provincia` es el atributo que se está declarando en esta tabla, y la segunda es la tabla `provincia` declarada en la Caja 6.5.

Estamos inmersos en un proceso iterativo para cada tabla del modelo. La manera de proceder que se está utilizando se repite tantas veces como tablas haya.

- Codificar la creación de la tabla en un escript homónimo.
- Importar el escript principal `club.sql` desde la línea de comandos del `psql`, reconstruyendo así la base desde cero.
- Importar seguidamente el nuevo escript. Corregir los errores, hasta obtener la respuesta `CREATE TABLE` del PostgreSQL.
- Crear una nueva carpeta dentro de la principal del proyecto, con el mismo nombre que la tabla y el escript, y guardar ahí el nuevo escript.
- Añadir la importación con la nueva ruta del escript al archivo `club.sql`.

Así pues, importamos de nuevo el `club.sql`, y como que ya contiene la importación de la tabla `provincia`, inmediatamente luego hacemos la importación de `ciudad.sql` que hasta ahora se encuentra en la carpeta principal del proyecto. Si el PostgreSQL responde `CREATE TABLE`, entonces se repite el procedimiento de antes, dejando la estructura como se ilustra en la Figura 6.4.

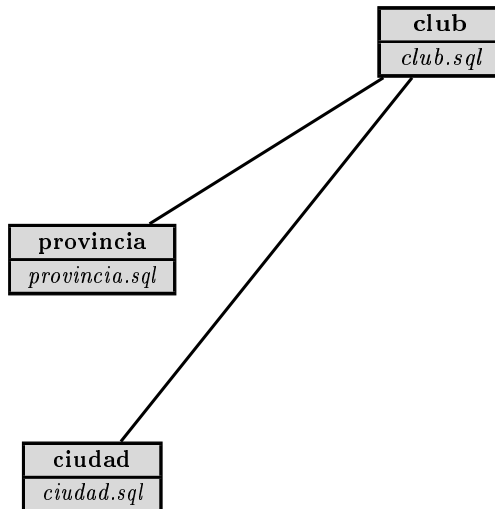


Figura 6.4: Estructura del espacio con la tabla `ciudad`.

Muy bien. Seguimos adelante con la implementación. Empezamos la tercera iteración haciendo un nuevo escript que debe ser el que haya en el modelo relacional del proyecto, ya que es en este modelo donde se establece el orden de creación de las tablas. Como se puede consultar en el Modelo 6.2, ahora es el turno de la tabla `persona`, que se crea la Caja 6.10.

```

\echo ----- tabla persona
CREATE DOMAIN dominio_mail TEXT
    CHECK (VALUE LIKE '%_@%._%__');

CREATE TABLE persona (
    pasaporte TEXT PRIMARY KEY,
    nombres TEXT NOT NULL,
    apellidos TEXT NOT NULL,
    mail dominio_mail NOT NULL,
    ciudad TEXT REFERENCES ciudad
        ON DELETE SET NULL
        ON UPDATE CASCADE,
    CONSTRAINT mail_ya_asignado UNIQUE(mail)
);

```

Caja 6.10. *Archivo persona.sql.*

Haciéndolo de esta manera estamos asumiendo varias decisiones de diseño, de acuerdo con la definición de requerimientos, que tienen un gran impacto en la base de datos. Estas condiciones se enumeran seguidamente.

- Identificamos cada registro con el número de pasaporte. O sea que prohibimos registrar personas si no se sabe su número de pasaporte.
- Tampoco aceptamos personas si no sabemos sus nombre y sus apellidos, aún que para esos atributos se pueden repetir los valores.
- Las direcciones de correo electrónico de las personas también son necesarias. Y además deben ser únicas, no se admiten repeticiones.
- Si además podemos saber la ciudad, mejor. Pero si la persona no la significa, igualmente podrá inscribirse en el club. También como antes, si se elimina una ciudad, mantendremos las personas sin saber de dónde son. Y si cambia de nombre, a las personas de la ciudad en cuestión se les actualizará la ciudad con el nuevo nombre.

En la Caja 6.10, la restricción `UNIQUE` sirve para establecer unicidad en los valores de los mails. No se permitirá que la misma dirección de correo pertenezca a más de una persona. Aquí, la restricción se implementa en su versión más completa, es decir, dándole nombre. En adelante, sin embargo, como con las claves primarias y foráneas, también se usará el nombre por defecto para las restricciones de unicidad, que en este caso sería `u_mail`.



### Claves candidatas

La restricción descrita en la última sentencia de la Caja 6.10 implementa lo que normalmente es conocido como clave alternativa. Es decir, un atributo o conjunto de atributos forman una clave *alternativa*, o *candidata*, cuando se les declara una restricción de unicidad. Eso no impide que esos atributos puedan contener valores nulos, como ya se había dicho. La unicidad no implica la existencia porque no se puede considerar que dos valores nulos sean equivalentes.

El siguiente paso vuelve a ser la importación del escript principal desde la línea de comandos del `psql`. Seguidamente se importa el archivo `persona.sql` que hasta este momento ha estado en la carpeta principal del proyecto. Y una vez rectificamos los errores sintácticos que hubiere, hay que crear una nueva carpeta donde meter ese escript.

Hecho esto, actualizamos la línea en el escript principal. A estas alturas, el `club.sql` debería tener el aspecto que se muestra en la Caja 6.11.

```
\echo escript principal para la base de datos club
\echo -----
\c postgres
DROP DATABASE club;
CREATE DATABASE club;
\c club
\i 'provincia\provincia.sql'
\i 'ciudad\ciudad.sql'
\i 'persona\persona.sql'
```

Caja 6.11. *Contenido actual del escript principal, archivo club.sql.*

Añadida la línea de la nueva tabla en el escript principal nos encontramos en un estado libre de errores. Lo podemos garantizar sin ni siquiera probarlo. Eso hace que sea un buen momento para hacer una copia de seguridad del proyecto completo, es decir de la carpeta principal del proyecto con todas sus carpetas y archivos.

Volvamos a empezar. Creada la tabla `persona`, podemos seguir con las tablas que la apuntan. Como siempre siguiendo los dictados del modelo relacional, abordamos la tabla `telefonos`.

La tabla que se muestra en la Caja 6.12 viene a decir que la persona indicada con

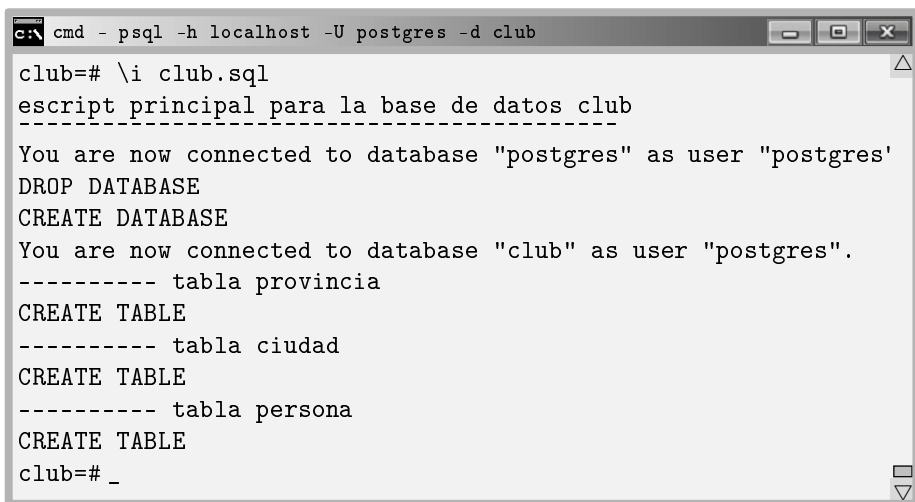
el número de pasaporte tal, tiene el teléfono tal. Esa información no parece que tenga sentido guardarla más de una vez para una persona y un teléfono concreto. Y por eso muchos desarrolladores pondrían la pareja de atributos como clave primaria para evitar repeticiones. Aquí, se implementa de una manera más correcta.

```
\echo ----- tabla telefonos

CREATE TABLE telefonos (
    pasaporte TEXT NOT NULL REFERENCES persona
        ON DELETE CASCADE
        ON UPDATE CASCADE,
    telefono TEXT NOT NULL,
    UNIQUE(pasaporte,telefono)
);
```

Caja 6.12. Archivo telefonos.sql.

Toca volver a ejecutar el escript principal desde la línea de comandos antes de ensamblarle un nuevo import para esta nueva tabla. En la Pantalla 6.6 se puede observar el diálogo que en este momento remonta la base de datos.



```
c:\> cmd - psql -h localhost -U postgres -d club

club=# \i club.sql
escript principal para la base de datos club
-----
You are now connected to database "postgres" as user "postgres"
DROP DATABASE
CREATE DATABASE
You are now connected to database "club" as user "postgres".
----- tabla provincia
CREATE TABLE
----- tabla ciudad
CREATE TABLE
----- tabla persona
CREATE TABLE
club=# _
```

Pantalla 6.6. Importación del escript principal al SGBD.

Es habitual que las tablas que no provienen de entidades del modelo ER, y que por tanto en lugar de tener una clave primaria tienen una o varias claves foráneas, tan solo admitan repetidos cuando además de las claves foráneas haya algún atributo adicional. Es decir, que más que guardar que dos conceptos están relacionados,

guardamos que entre dos conceptos hay una colección de interacciones, cada una de las cuales tiene alguna característica en particular. Si las relaciones son verbos, esos atributos son los adverbios, incluso etimológicamente. Y si como se decía en el preámbulo de una manera filosófica, el tiempo establece diferencias entre una cosa y ella misma, la característica en particular más común que diferencia que dos objetos estén relacionados más de una vez, es el tiempo. Seguiremos hablando del tema.

Seguidamente, hay que hacer el import del archivo `telefonos.sql` corrigiendo los errores que haya hasta que resulte exitoso. Se crea la nueva carpeta `telefonos`, colocándole ese escript nuevo, se añade el nuevo import al escript principal, y a por la siguiente.

En la Caja 6.13 se expone la creación de la tabla `conoce`.

```
\echo ---- tabla conoce
CREATE TABLE conoce (
  conoce TEXT NOT NULL REFERENCES persona
    ON DELETE CASCADE
    ON UPDATE CASCADE,
  es_conocida TEXT NOT NULL REFERENCES persona
    ON DELETE CASCADE
    ON UPDATE CASCADE,
  UNIQUE(conoce,es_conocida)
);
```

Caja 6.13. *Archivo conoce.sql.*

Las autorelaciones M:N tienen doble participación total en la entidad donde son definidas. También en este caso es claro que no tiene sentido guardarse registros con alguno de los dos valores inexistente. Por tanto, otra vez se declaran no nulos cada uno de ellos. Ni duplicados, y por eso la pareja como conjunto, única.

Atención al tiempo. Imagina que en lugar de guardarnos simplemente que dos personas se conocen quisiéramos guardar cuándo han hablado o se han visto por última vez, es decir, cuándo han contactado.

Entonces, en lugar de `conoce` deberíamos llamar `contacta` a la tabla, significando que una persona contacta con la otra. Y sería necesario también añadir un atributo a la autorelación que se llamase `cuando`, y fuese de tipo `TIMESTAMP`. Hasta aquí, con la restricción de unicidad seguiríamos teniendo bastante para registrar cuando fue el último contacto entre la pareja de personas. Pero aún hay más, si una vez añadido el atributo `cuando` a la relación `contacta` que substituyera `conoce`, quitáramos la

restricción de unicidad, entonces nos quedaría un registro histórico de cuando se han visto o han hablado dos personas a lo largo del tiempo. Al poderse repetir la pareja de personas habría registros de *contacta* que tan solo se diferenciarían en *cuando*.

Eso es lo que se pretendía ilustrar cuando se decía que el tiempo establece diferencias entre una cosa y ella misma. Y como consecuencia, tratados importantes de bases de datos distinguen entre bases de datos temporales, y no temporales, [10], [1]. Las extensiones temporales del SQL se aglutinan en un lenguaje llamado *Temporary Structured Query Language 2*, TSQL2.

Fíjate en fin, que habiendo entendido que en un modelo ER las entidades representen sustantivos y las relaciones verbos, aún se entiende más bien que una relación M:N mantenga un registro de tiempo.

Abordamos a continuación las entidades de la especialización del modelo ER.

Para el tema de las herencias en el PostgreSQL hay implementada una cláusula *INHERITS*, que se puede consultar en la documentación oficial, [8]. No obstante, presenta problemas al distinguir un registro heredado, de uno que no lo es. O sea, utilizando esa herramienta el PostgreSQL permite crear una persona con un número de pasaporte AAA0000, y adicionalmente un socio diferente de la persona, con el mismo número AAA0000. Eso hace que si se desea usar la cláusula se requiera de disparadores que controlen esas irregularidades. En cualquier caso, aquí no se utiliza, y en cambio se implementa la herencia como siempre se ha hecho en SQL, es decir, con la clave primaria igual a la clave foránea implementando así relaciones 1:1.

```
\echo ----- tabla socio
CREATE TABLE socio (
    pasaporte TEXT PRIMARY KEY REFERENCES persona,
    registro TIMESTAMP WITH TIME ZONE NOT NULL
);
```

Caja 6.14. *Archivo socio.sql.*

En la Caja 6.14 no aparece ninguna indicación para la clave foránea respecto a la actualización o la eliminación de la clave primaria. Es decir, no se dice qué se debe hacer con un socio cuando se elimine la persona correspondiente. Eso es así porque, otra vez como decisión de diseño, no se permitirá que se borre una persona que es un socio. Por tanto ya va bien la opción *ON DELETE RESTRICT ON UPDATE RESTRICT* que es la que hay por defecto. Lo que se deberá hacer es eliminar el socio. Entonces hay que tomar una nueva decisión. O aceptamos que no hay problema en mantener la información de la persona correspondiente cuando se borra el socio, o bien será

necesario hacer un procedimiento específico para esa situación. Un disparador que cuando se elimine un socio borre también la persona correspondiente. Eso se verá en la Sección 7.7. De momento, aquí se prohíbe la eliminación de un socio si se intenta hacer borrando la persona. Y lo mismo con la actualización. Y todo ello, será igual para el caso de los trabajadores.

Observa también que el atributo calculado `registro` es del tipo más completo de todos los temporales. Si luego interesa tan solo la fecha ya se podrá hacer un truncamiento, esa es la palabra, trucar un instante para obtener un día, sin pérdida de eficiencia. Como que en el mismo instante de la creación de un socio le daremos valor, se declara como atributo requerido.

En la Caja 6.15 hay la otra entidad específica de la herencia. En este caso, además, se usa el dominio definido por el usuario visto en la Caja 6.2.

```
\echo ---- tabla trabajador
CREATE DOMAIN dominio_departamento TEXT
    CHECK (VALUE IN ('administración','comercial','entrenador'));

CREATE TABLE trabajador (
    pasaporte TEXT PRIMARY KEY REFERENCES persona,
    departamento dominio_departamento NOT NULL,
    obedece TEXT REFERENCES trabajador
        ON DELETE SET NULL
        ON UPDATE CASCADE
);
```

Caja 6.15. *Archivo* trabajador.sql.

Como se considera que todos los trabajadores deben pertenecer algún departamento, el atributo correspondiente es requerido. Esta restricción de existencia se hubiera podido incluir en la definición del dominio, añadiendo a la condición `AND VALUE IS NOT NULL`, pero no es aconsejable porque acostumbra a traer problemas si después deseamos añadir disparadores en la inserción de la tabla.

De la Caja 6.15 se puede extraer también que puede haber trabajadores de los cuales no se sepa quién es su jefe. Esta decisión, un pelo arbitraria, se ha tomado para evitar problemas con los registros de los trabajadores el día que su jefe cese. Si se hubiera exigido participación total, entonces en el momento de borrar un jefe, se requeriría asignar un nuevo jefe a cada uno de sus trabajadores, y esto podría representar un trabajo antipático para el usuario final.

Si realmente así se deseara, lo deberíamos explicar bien al usuario. Ahora de momento, lo que pasará es que no se permitirá borrar un trabajador que sea jefe de otro. Una opción razonable sería que al borrar un jefe, sus trabajadores pasasen a depender del siguiente en el escalafón al que se borra, que se podría implementar en un disparador.

En la Caja 6.16 se muestra la tabla deporte, y su comprensión no debería suponer ningún problema.

```
\echo ----- tabla deporte
CREATE TABLE deporte (
    nombre TEXT PRIMARY KEY,
    precio DECIMAL(5,2) DEFAULT 10.0,
    jugadores INTEGER
);
```

Caja 6.16. Archivo deporte.sql.

La definición de la tabla *hace*, en la Caja 6.17, se trata de la implementación clásica de una relación M:N. Igual que en el caso de la autorelación M:N, no parece tener sentido guardarse la información de que un socio hace un deporte más de una vez. Otra vez conviene hacer la reflexión de introducir tiempo. Añadiendo dos atributos a la relación podríamos mantener un registro de cuando se ha inscrito un socio en un deporte y cuando se ha borrado. Pero bien, como ya se dice en el Capítulo 2 de análisis del proyecto, debe quedar muy clara en la definición de requerimientos la distinción entre informaciones que deseamos tener en el presente, y registros históricos de una misma información.

```
\echo ----- tabla hace
CREATE TABLE hace (
    pasaporte TEXT REFERENCES socio
        ON DELETE CASCADE
        ON UPDATE CASCADE,
    deporte TEXT REFERENCES deporte
        ON DELETE CASCADE
        ON UPDATE CASCADE,
    cuota DECIMAL(5,2),
    UNIQUE(pasaporte,deporte)
);
```

Caja 6.17. *Archivo hace.sql.*

Finalmente, para acabar con la construcción de la base de datos club, tan solo queda la creación de la tabla pagos.

```
\echo ----- tabla pagos
CREATE TABLE pagos (
    pasaporte TEXT REFERENCES trabajador
        ON DELETE CASCADE
        ON UPDATE CASCADE,
    periodo DATE,
    salario_base DECIMAL(6,2) NOT NULL,
    retencion DECIMAL (4,2) DEFAULT 2.00,
    PRIMARY KEY(pasaporte,periodo)
);
```

Caja 6.18. *Archivo pagos.sql.*

La definición que se presenta en la Caja 6.18 tiene gran interés. La entidad débil se caracteriza por tener una clave primaria binomial, cosa que impone declarar la restricción de clave primaria en una línea adicional.

Eso es, cuando para una entidad la clave primaria está compuesta de más de un atributo, entonces la forma de declarar el conjunto de campos como clave primaria es en una línea adicional. Al ser clave primaria, establecemos que ninguno de los dos valores puede ser nulo, y que se puede repetir el primero entre distintas tuplas, o bien el segundo, pero no los dos en una misma tupla.

Además, parte de esta clave primaria es clave foránea de la entidad identificadora, **trabajador**. Eso refleja el hecho natural de identificar un pago a partir de la identificación del trabajador a quien pertenece.

En el atributo discriminante, **periodo**, guardaremos la fecha del pago. El salario es requerido. Observa también que como retención se establece un valor por defecto. Este valor debería provenir directamente de la definición de requerimientos. Es decir, debería ser establecido por el usuario final.

Y por fin, ya estamos. El modelo relacional completo se imprime en la Caja 6.19.

```
\set ON_ERROR_STOP on
\echo escript principal para la base de datos club
\echo -----
\c postgres
DROP DATABASE club;
CREATE DATABASE club;
\c club
\i 'provincia\provincia.sql'
\i 'ciudad\ciudad.sql'
\i 'persona\persona.sql'
\i 'telefonos\telefonos.sql'
\i 'conoce\conoce.sql'
\i 'socio\socio.sql'
\i 'trabajador\trabajador.sql'
\i 'deporte\deporte.sql'
\i 'hace\hace.sql'
\i 'pagos\pagos.sql'
```

Caja 6.19. *Contenido final del escript principal, archivo club.sql.*

Sorprende la aparición de la primera línea `\set ON_ERROR_STOP on`. Por la contrabarra inicial, es un comando del programa cliente `psql`. En la ayuda del `psql`, haciendo `\?`, se explica el comando `\set`, para dar valores a sus variables de entorno. La variable del `psql` `ON_ERROR_STOP`, es una variable booleana que como su nombre indica significa que si se produce algún error al ejecutar este escript, se pare. Con esta línea al principio no seguirá intentando crear las tablas siguientes luego de un error.

Ahora, que la cantidad de escripts ya es considerable, si hubiera algún error desecadenaría una hemorragia de mensajes en cascada, ya que cualquier cosa que impida crear una tabla impide crear las que dependan de ella. Y todo ello resultaría incómodo.

El diálogo con el SGBD correspondiente a la importación de este escript se muestra en la Pantalla 6.7.





```

c:\> cmd - psql -h localhost -U postgres -d club
club=# \i club.sql
escript principal para la base de datos club
-----
You are now connected to database "postgres" as user "postgres"
DROP DATABASE
CREATE DATABASE
You are now connected to database "club" as user "postgres".
SET
----- tabla provincia
CREATE TABLE
----- tabla ciudad
CREATE TABLE
----- tabla persona
CREATE DOMAIN
CREATE TABLE
----- tabla telefonos
CREATE TABLE
----- tabla conoce
CREATE TABLE
----- tabla socio
CREATE TABLE
----- tabla trabajador
CREATE DOMAIN
CREATE TABLE
----- tabla deporte
CREATE TABLE
----- tabla hace
CREATE TABLE
----- tabla pagos
CREATE TABLE
club=# _

```

Pantalla 6.7. *Diálogo resultante de importar el archivo club.sql final.*

El escript de la Caja 6.19 se ejecuta, como todos, en el lado del servidor. Igualmente la respuesta mostrada en la Pantalla 6.7 también se emite desde el lado del servidor.

Y por otra parte, o sea por lo que respecta al lado de la computadora cliente, si se han ido haciendo las cosas como se decía al principio de la construcción de la base de datos debería haberse creado una estructura de directorios en el disco local. La estructura completa correspondiente al proyecto tal como debería estar en este momento se muestra en la Figura 6.5. Es importante que así sea para poder trabajar de manera ordenada en las próximas secciones.

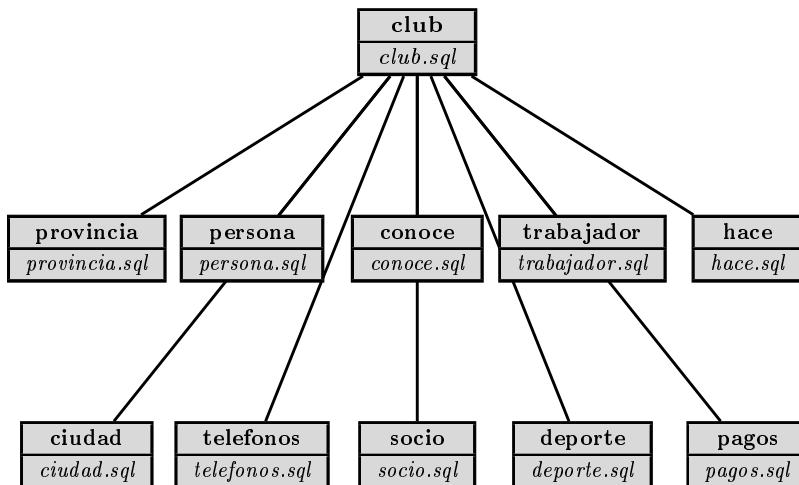


Figura 6.5: Estructura de directorios del proyecto.

## 6.4 Lenguaje de Manipulación de Datos

El Lenguaje de Manipulación de Datos, o DML de *Data Manipulation Language*, es la parte del SQL que se utiliza para consultar y manipular los datos de la base. Eso significa proporcionar las operaciones o funciones necesarias para poder hacer la explotación.

Volvemos a hacer la misma transición que en el Capítulo 5. Pasamos de ocupar de aspectos relacionados con el espacio, a acciones relacionadas con el tiempo. El DDL es al concepto de relación en el álgebra lo que el DML es a las operaciones con relaciones. Lo que haremos a partir de ahora lo llamamos consultas. Haremos consultas de lectura, que podremos guardar en vistas, y consultas de actualización.

### Scripts de inserción

Para facilitar las pruebas de los ejemplos que se verán, conviene montar una nueva jerarquía de scripts. Ahora pero, todos se llamarán igual, `inserts.sql`. La estructura quedará como se ilustra en la Figura 6.6.

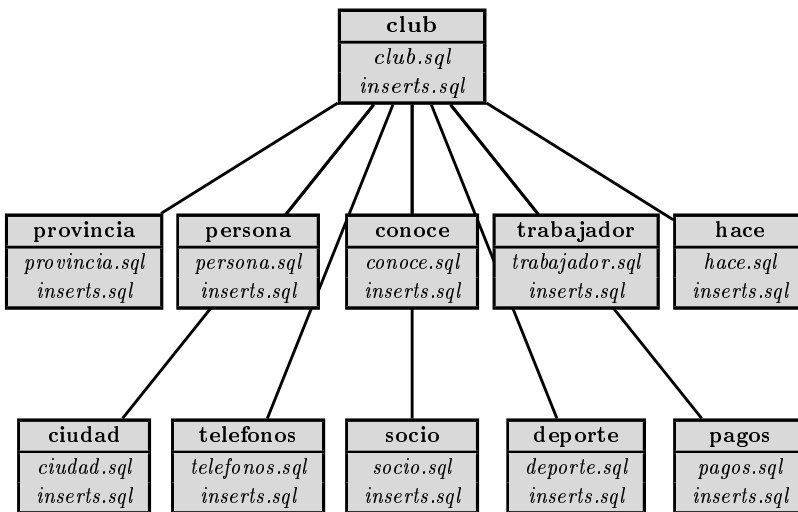


Figura 6.6: Incorporación de scripts de inserción.

El de la raíz, la carpeta del proyecto, será el script principal `inserts.sql` y también hará diez imports, de la forma

```
\i 'provincia\\inserts.sql'.
```

Por eso, será necesario crear un archivo de nombre `inserts.sql` en cada carpeta del proyecto, donde pondremos los datos de ejemplo necesarios para poder hacer las consultas. Y entonces, añadimos una línea al final del escript principal para incluir el escript de inserts. Si en algún momento interesase volver a tener la base vacía, no hace falta nada más que quitar esa última línea del escript principal. Los datos de ejemplo para todas las tablas se muestran en el Apéndice C, y se pueden descargar en la dirección que figura al final del preámbulo.

### 6.4.1 Consultas de Lectura

Llamamos consultas de lectura las que no modifican el contenido de la base de datos. Se caracterizan además porque el tipo de valores que retornan son relaciones. Hay un solo comando para hacer consultas de lectura, el `SELECT`.

En el capítulo anterior nos limitábamos a formular expresiones del álgebra relacional que respondieran a ciertas cuestiones. Eran expresiones teóricas. En adelante no. Ahora emitimos órdenes al SGBD, que es quien realiza la acción. Lo que antes eran expresiones, ahora son instrucciones que llamamos consultas de lectura. Lo que antes hacíamos nosotros pensando, ahora lo hará el SGBD computando.

Seguidamente se presenta el comando emblemático del lenguaje estructurado de consultas SQL. Es una sentencia que como mucho puede tener seis cláusulas, aunque en su forma más famosa tan solo utiliza tres.

#### Estructura fundamental

La estructura fundamental de una consulta de lectura tiene la forma de la Caja 6.20,

```
SELECT  $A_1, A_2, \dots, A_k$ 
FROM  $r$ 
WHERE  $p$ ;
```

Caja 6.20. *Estructura fundamental de una consulta de lectura en SQL.*

siendo  $r = r(A_1, A_2, \dots, A_n)$  una relación con  $k < n$ , y  $p$  un predicado. Este comando proyecta los atributos  $A_1, A_2, \dots, A_k$  de los registros de  $r$  que satisfagan el predicado  $p$ .

Así que cuidado, la cláusula `SELECT` se corresponde con la operación de proyección, y no de selección como podía parecer. Es la cláusula `WHERE` la que permite introducir el predicado de selección del álgebra relacional.

Y estas dos operaciones, selección y proyección, se realizan en la relación formada por el producto cartesiano de las tablas que haya en la cláusula `FROM`, también llamada, la relación de la consulta.

En notación conocida pues, la expresión relacional de la Caja 6.20 es

$$\Pi_{A_1, A_2, \dots, A_k}(\sigma_p(r)).$$

No obstante, la expresión con dos tablas  $r = r(A_1, A_2, \dots, A_n)$  y  $s = s(B_1, B_2, \dots, B_m)$ , siendo  $k \leq n$  y  $\ell \leq m$ , resulta más ilustrativa.

```
SELECT A1, A2, ..., Ak, B1, B2, ..., Bℓ
FROM r, s
WHERE p;
```

Caja 6.21. *Estructura habitual de una consulta de lectura en SQL.*

La consulta de la Caja 6.21 es

$$\Pi_{A_1, A_2, \dots, A_k, B_1, B_2, \dots, B_\ell}(\sigma_p(r \times s)),$$

expresión que debería recordar la tira de la Figura 5.13.

El predicado  $p$ , como en el álgebra relacional, puede implicar valores constantes y atributos de cualquiera de las dos tablas. Si hay columnas homónimas en  $r$  y  $s$  presentes en el predicado se puede poner el nombre de la relación separado por un punto como prefijo del nombre de estas columnas, o utilizar renombramientos, cosa más habitual.

## Datos

Para conseguir que el PostgreSQL dé algún resultado, primero hay que entrar algunos datos. En la Tabla 6.2 se puede observar un posible contenido de la relación `provincia`, que es una parte de los datos de la tabla `provincia` del Apéndice C.

El escript que inserta estos valores en la base está en la Caja 6.22. Utiliza una vez más

provincia
Napo
Los Ríos
El Oro
Chimborazo

Tabla 6.2: *Instancia de la relación provincia.*

el comando `INSERT INTO`, que ya se ha anticipado en la Sección 6.3.4, para construir relaciones enumerando registros de valores constantes.

```
\echo ----- inserts tabla provincia
INSERT INTO provincia VALUES
('Napo'),
('Los Ríos'),
('El Oro'),
('Chimborazo');
```

Caja 6.22. *Archivo club/provincia/inserts.sql.*

Observa que las cadenas de caracteres constantes se escriben entre comillas simples, o apóstrofes. Para escribir el carácter apóstrofe habría que ponerlo por duplicado.

En la Tabla 6.3 hay lo que podría valer la relación ciudad.

nombres	habitantes	provincia
Babahoyo	90191	Los Ríos
Tena	23307	Napo
San Francisco	805235	
Berlín	3499879	
Rio de Janeiro	6320446	
Machala	331260	El Oro

Tabla 6.3: *Instancia de la relación ciudad.*

El script para establecer estos valores se imprime en la Caja 6.23.

```

\echo ----- inserts tabla ciudad

INSERT INTO ciudad VALUES
('Babahoyo',90191,'Los Ríos'),
('Tena',23307,'Napo'),
('San Francisco',805235,null),
('Berlín',3499879,null),
('Rio de Janeiro',6320446,null),
('Machala',331260,'El Oro')

```

Caja 6.23. Archivo club/ciudad/inserts.sql.

Fíjate que números y nulos, sin comillas. Para los booleanos se necesitarían usar las constantes `true` y `false`, también sin comillas.

Y una instancia de la tabla `persona` dados esos valores podría ser la de la Tabla 6.4.

pasaporte	nombres	apellidos	mail	ciudad
0127673812	Carmen Esmeralda	Peralta Gutiérrez	cperalta@ionos.gov	Babahoyo
0123478937	Carlos Sayaro	Sanabria Oña	sayarona1998_2@hotmail.com	Tena
1047548338	Ana Estefanía	Sanabria Oña	anasteona1997@yahoo.ec	Tena
1145493393	Jesús Marcelo	Hortesa Orellana	rexst at 143@rediris.es	Machala
CKU01549	Michael	Bros	mbros1989@aol.com	San Francisco
FFJ904992	Klauss	Stallman	kstallman@dvw.tum.de	Berlín

Tabla 6.4: Contenido de la tabla `persona`.

En la Pantalla 6.8 hay la parte final del diálogo con el PostgreSQL cuando se reconstruye la base de datos con estas inserciones, es decir, cuando hacemos un `import` del `club.sql` actual, que ya incluye el `insert` de inserciones.

```

c:\> cmd - psql -h localhost -U postgres -d club

CREATE TABLE
----- tabla pagos
CREATE TABLE
----- inserts tabla provincia
INSERT 0 4
----- inserts tabla ciudad
INSERT 0 6
----- inserts tabla persona
INSERT 0 6
club=#_

```

Pantalla 6.8. Final del diálogo resultante de importar el archivo `club.sql` con las primeras inserciones.

Se observa en la Pantalla 6.8 que la respuesta exitosa del comando `INSERT` retorna dos cifras. La primera indica el número de objetos de los metadatos afectados, es decir objetos con OID. Que este primer número sea distinto de cero queda fuera del ámbito de este libro. Aquí, siempre será cero. La segunda es el número de filas afectadas, como se puede adivinar.

## Cláusula SELECT

La función de la cláusula `SELECT` es establecer los atributos que forman el esquema del resultado. Coincide con la proyección del álgebra relacional. Es la única que forzosamente debe aparecer en cualquier consulta de lectura. Puede ir sola, es decir sin otras cláusulas, como ya se ha visto en la Sección 6.1.1.

Poniendo un asterisco en lugar de la lista obtendremos todos los atributos de la tabla de la cláusula `FROM` si solo hay una. Para ver el contenido de una tabla completa pues, utilizamos la consulta más habitual de todas las consultas. En la Caja 6.24 se presenta el comando para volcar el contenido de la tabla `persona`.

```
SELECT * FROM persona;
```

Caja 6.24. Consulta de todo el contenido de la tabla `persona`.

La respuesta de PostgreSQL a esta consulta se puede ver a la Pantalla 6.9.



```
cmd - psql -h localhost -U postgres -d club
club=#
club=# SELECT * FROM persona;
 pasaporte |      nombre      |      apellidos      | ciudad
-----+-----+-----+-----
0127673812 | Carmen Esmeralda | Peralta Gutiérrez  | Babahoyo
0123478937 | Carlos Sayaro    | Sanabria Oña       | Tena
1047548338 | Ana Estefanía    | Sanabria Oña       | Tena
1145493393 | Jesús Marcelo    | Hortesa Orellana   | Machala
CKUS01549  | Mick             | Brown               | San Franc
FFJ904992  | Klauss           | Stallman            | Berlín
(6 rows)
club=# _
```

Pantalla 6.9. Contenido de la tabla `persona`.



Más genéricamente, el asterisco sirve para pedir todos los atributos que resulten de la concatenación del producto cartesiano de las relaciones que aparezcan en la cláusula FROM. Fíjate bien. No se trata de la unión de atributos, sino de la concatenación. Este extremo se aclarará en la próxima sección.

### SELECT con multiconjuntos

Como se ha insistido anteriormente, la diferencia más importante entre el álgebra relacional y el SQL es que la primera trabaja con conjuntos, y la segunda con multiconjuntos. En otras palabras, las relaciones en SQL pueden tener elementos repetidos. Eso es fruto que en el álgebra las tuplas de las relaciones son distintas por definición, en cambio en el SQL hay que establecer claves si deseamos garantizar esa condición.

En el ejemplo 6.4 se pretende obtener el nombre de todas las ciudades de las personas de la base. La primera aproximación se limita a obtener el nombre de todas las ciudades de la base de datos.

**ejemplo 6.4.** *Listar las ciudades de las personas de la base de datos.*

**solución** `SELECT ciudad FROM ciudad;`

Y claramente, no es una buena solución, porque nos muestra ciudades de las que no hay nadie. Es el caso de *Rio de Janeiro*. En una segunda aproximación podríamos pensar en

**solución** `SELECT ciudad FROM persona;`

que efectivamente nos muestra las ciudades de las personas de la base de datos. No obstante, cada ciudad aparece tantas veces como personas hay, como se ve en el caso de *Tena*.

Cuando se hace un SELECT de un campo que no es clave primaria, y en particular cuando es clave foránea resulta muy habitual, pueden aparecer elementos repetidos en la relación resultante. Así se comporta el SGBD por defecto. Cuando se quiera evitar ese efecto, se utiliza SELECT DISTINCT. La solución 6.5 es correcta.

**ejemplo 6.5.** *Listar las ciudades de las personas de la base de datos.*

**solución** `SELECT DISTINCT ciudad FROM persona;`

Por defecto, un SELECT es con la opción ALL, que significa que aparezcan los valores repetidos. Eso es interesante si por ejemplo se quisieran contar las personas de cada

ciudad. Se resolvería contando cuántas veces aparece el nombre de cada ciudad en una consulta como la de la segunda solución del ejemplo 6.4. Por eso el SQL mantiene la posibilidad de trabajar con relaciones que sean multiconjuntos.

### Cláusula FROM

La cláusula `FROM` es la que indica el universo a partir del cual hacemos las dos cosas, establecemos criterios para según qué atributos, y solicitamos otros. Por eso, se llama *relación de la consulta* a la relación resultante del producto cartesiano de las tablas presentes en la cláusula `FROM`.

El número de relaciones que aparecen en una cláusula `FROM` es el *orden* de la consulta, y sirve para medir su complejidad.

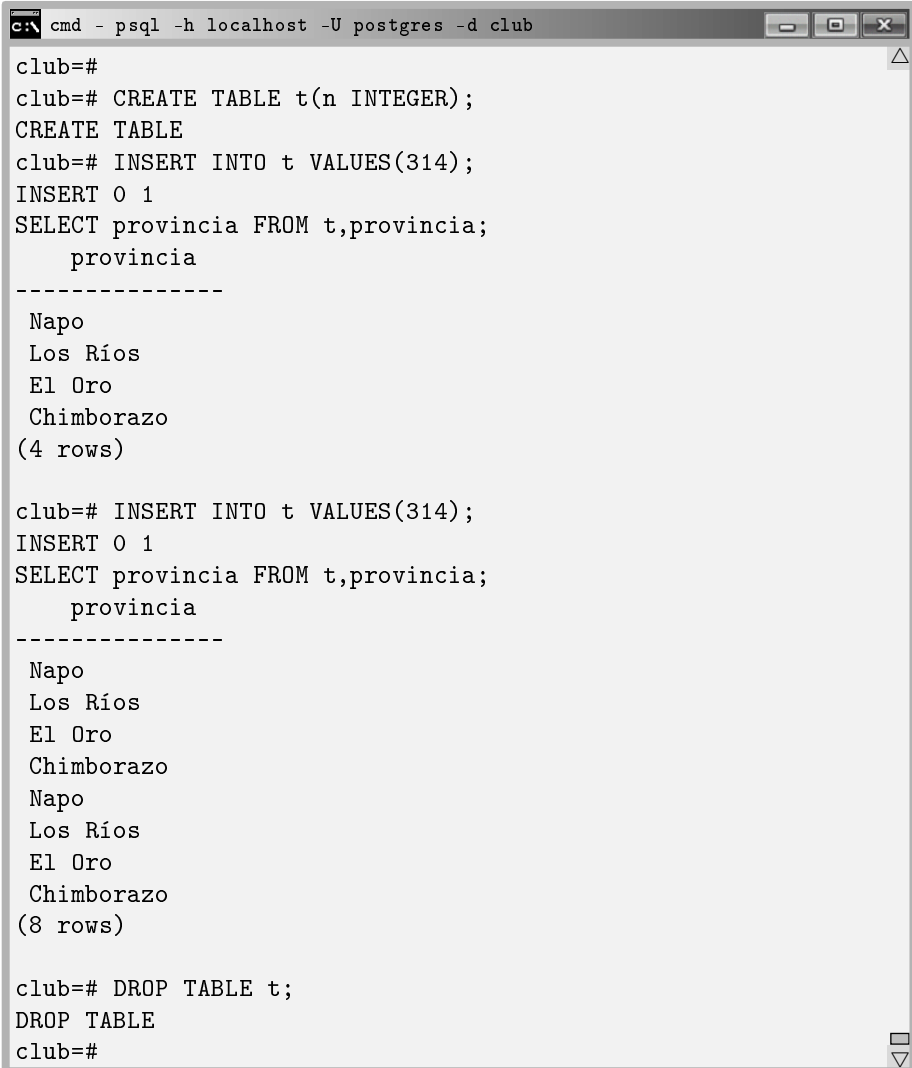
Al computarse, cualquier consulta empieza con la relación de la consulta. Y eso tiene una incidencia más que considerable. Si en una consulta aparece una tabla innecesaria, el tiempo de respuesta se multiplica por el número de elementos que tenga esa tabla. Eso es un error muy grave. Hay que tener en mente que cuando añadimos una tabla a la cláusula `FROM` estamos añadiendo un bucle que la recorre, es decir, que trata un elemento en cada iteración. Visto de otra manera, si hacemos una selección con una sola tabla, nos podemos imaginar el código consistente en un bucle que recorre la tabla. Si ponemos dos tablas en el `FROM` entonces es un bucle dentro de un bucle. Y si ponemos tres, pues un bucle que para cada registro de la primera recorre cada registro de la segunda, y para cada pareja posible de la primera y la segunda recorre el bucle para cada uno de los elementos de la tercera.

Como es importante, hacemos un experimento de ocho pasos con la cantidad de registros resultantes de un producto cartesiano.

1. Crear una tabla, `t`, con un solo atributo que se llame `n` y sea un entero, haciendo `CREATE TABLE t(n INTEGER);`
2. Insertar un número cualquiera en la tabla `t`, haciendo `INSERT INTO t VALUES(314);`
3. Mostrar el nombre de todas las provincias añadiendo innecesariamente la tabla `t` en la cláusula `FROM`, haciendo `SELECT provincia FROM provincia,t;`
4. Observar que hay cuatro resultados, los nombres de las provincias.
5. Insertar un nuevo número cualquiera en `t`, haciendo dos veces flecha arriba, o tecleando otra vez `INSERT INTO t VALUES(314);`
6. Mostrar de nuevo el nombre de todas las provincias añadiendo innecesariamente la tabla `t` en la cláusula `FROM`.
7. Comprender el resultado.

8. Eliminar la tabla `t` haciendo `DROP TABLE t;`

En la Pantalla 6.10 se puede observar el diálogo de este experimento.



```

cmd - psql -h localhost -U postgres -d club
club=#
club=# CREATE TABLE t(n INTEGER);
CREATE TABLE
club=# INSERT INTO t VALUES(314);
INSERT 0 1
SELECT provincia FROM t,provincia;
provincia
-----
Napo
Los Ríos
El Oro
Chimborazo
(4 rows)

club=# INSERT INTO t VALUES(314);
INSERT 0 1
SELECT provincia FROM t,provincia;
provincia
-----
Napo
Los Ríos
El Oro
Chimborazo
Napo
Los Ríos
El Oro
Chimborazo
(8 rows)

club=# DROP TABLE t;
DROP TABLE
club=#

```

Pantalla 6.10. *Resultado del experimento para la cardinalidad del producto cartesiano.*

Un inciso respecto la novedad de este experimento. El comando `DROP TABLE`. Un comando del DDL. Salvo que puede hacerse en `CASCADE`, no hay nada más a decir. Hacer un `DROP TABLE t CASCADE;` serviría para eliminar la tabla `t` y todas las que hagan

referencia con una clave foránea, ya que si se hace un `DROP TABLE` de una tabla que es apuntada desde alguna otra, se produce un error. Ahora pero, como no hay ninguna tabla que referencie `t`, añadir la opción `CASCADE` no tendría ningún efecto.

Lo que se debería comprender de este experimento es que por cada nuevo elemento que añadimos a la tabla `t`, el resultado de la selección posterior mostraría una vez más la lista de todas las provincias, o sea, que la relación resultante se incrementaría en cuatro registros más.

El hecho de solo proyectar un atributo de la tabla `provincia` hace que el experimento resulte más sorprendente. Eso es, si en lugar de insertar dos veces el mismo número en la tabla `t` hubiésemos puesto dos valores distintos, y además, en lugar de un solo atributo los hubiésemos proyectado todos, con un asterisco en el `SELECT`, entonces el resultado se habría comprendido de una manera mucho más clara, ya que de todas las parejas posibles entre las cuatro provincias y los dos elementos de `t`, las cuatro primeras habrían aparecido asociadas al primer número, y las cuatro segundas, al segundo.

Tal como se ha hecho, el experimento expone más claramente el peligro de añadir tablas innecesarias en la cláusula `FROM`.

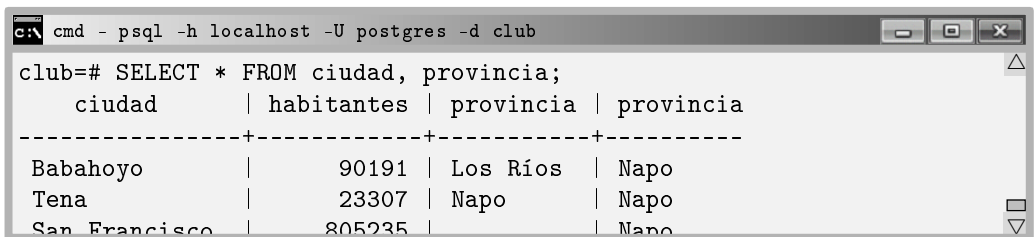
Todo esto por lo que respecta al cardinal del producto cartesiano entre las tablas que aparecen en la cláusula `FROM`. Profundicemos seguidamente en el tema de los atributos, el esquema de la relación de la consulta. Las columnas del producto cartesiano. El PostgreSQL no tiene ningún problema en repetir nombres de columnas en las relaciones resultantes.

Observémos con un resultado que repita la columna `provincia`.

**ejemplo 6.6.** *Obtener todas las combinaciones posibles de ciudades y provincias*

**solución** `SELECT * FROM ciudad,provincia;`

En la Pantalla 6.11 hay la parte inicial del resultado de la solución del ejemplo 6.6.



```

cmd - psql -h localhost -U postgres -d club
club=# SELECT * FROM ciudad, provincia;
 ciudad      | habitantes | provincia | provincia
-----+-----+-----+-----
 Babahoyo    |    90191  | Los Ríos  | Napo
 Tena        |    23307  | Napo      | Napo
 San Francisco |   805235  |           | Napo

```

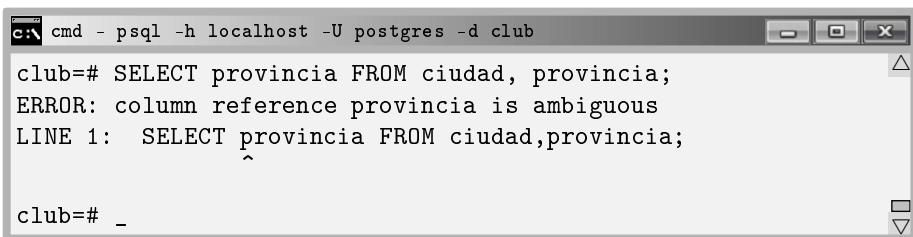
Pantalla 6.11. *Fragmento inicial del resultado del experimento para las columnas del*

*producto cartesiano.*

## Renombramiento

En la Pantalla 6.11, atendiendo a los títulos de las columnas de la respuesta del SGBD se puede comprender hasta qué frontera puede llegar la ambigüedad.

Si se pretendiera ir más allá, pidiendo por la columna `provincia`, en lugar de asterisco, PostgreSQL nos retornaría un error, explicando que no puede saber a qué columna nos estamos refiriendo, si la de la tabla `ciudad`, o `provincia`. La consulta de la Pantalla 6.12 no tiene demasiada lógica desde un punto de vista semántico. El ejemplo es estrictamente técnico. En el fondo se está pidiendo por las provincias que aparezcan en la tabla `ciudad` tantas veces como provincias haya. No tiene sentido, pero ilustra el tema de la ambigüedad. Obtendríamos la respuesta que se muestra.



```

c:\cmd - psql -h localhost -U postgres -d club
club=# SELECT provincia FROM ciudad, provincia;
ERROR: column reference provincia is ambiguous
LINE 1: SELECT provincia FROM ciudad,provincia;
                                     ^
club=# _
  
```

Pantalla 6.12. *Error provocado por ambigüedad de la columna provincia.*

Además, en la Pantalla 6.12 se muestra la forma de imprimir los errores que tiene `psql`. Fíjate en el acento circunflejo que aparece justo luego de la línea y justo en la primera letra de la palabra que ha provocado el error. Eso es muy interesante, y conviene prestarle atención. Cuando puede, PostgreSQL nos indica con una precisión a la letra, qué término es el que ha provocado un error.

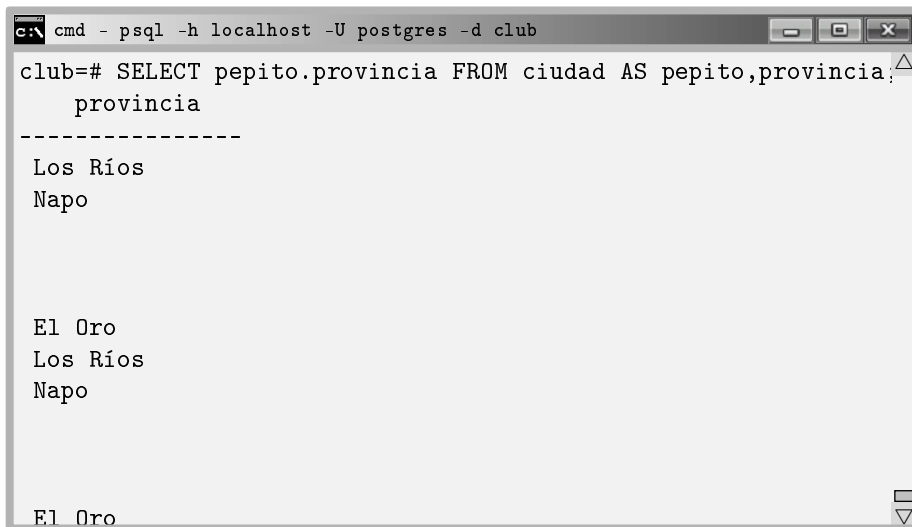
Bien, ese error se podría resolver, como se ha dicho en la Sección 6.4.1, poniendo el nombre de la tabla como prefijo del atributo, en el predicado del `SELECT`. Es decir,

```
SELECT ciudad.provincia FROM ciudad,provincia;.
```

De todas formas, cuando los atributos homónimos aparecen en los predicados del `WHERE` que en seguida se verán, y afectan más de un atributo común a las tablas del `FROM`, o cuando se hace referencia dos veces a la misma tabla en una consulta, entonces conviene usar el operador `AS`.

Es interesante dominar el operador de cambio de nombre de cara las consultas anidadas que se verán en la Sección 6.4.10. De momento tan solo comprender que

la consulta de la Pantalla 6.12 se hubiera podido desambigüizar de la manera que se muestra en la Pantalla 6.13, aunque solo se haya capturado una parte inicial de la respuesta.



```

c:\> cmd - psql -h localhost -U postgres -d club
club=# SELECT pepito.provincia FROM ciudad AS pepito,provincia;
provincia
-----
Los Ríos
Napó

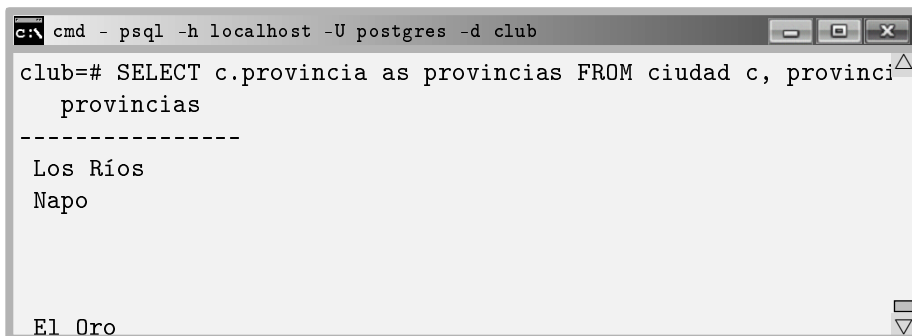
El Oro
Los Ríos
Napó

El Oro
  
```

Pantalla 6.13. *Fragmento inicial de la solución correcta, con cambio de nombre de la tabla ciudad, a pepito.*

Usamos la palabra *pepito* para dejar claro que puede ser cualquier palabra.

Tan frecuente resulta el renombramiento que si no se pone el operador `AS` en el `FROM`, con sólo un espacio, se entiende que la primera palabra es una tabla y la segunda el nombre que recibe en esta consulta. En la Pantalla 6.14 se renombra la tabla a palabra de una letra. Es la manera habitual como se utilizará. Ante la posibilidad que se añadan nuevas columnas a las tablas en el futuro, conviene hacerlo por seguridad.



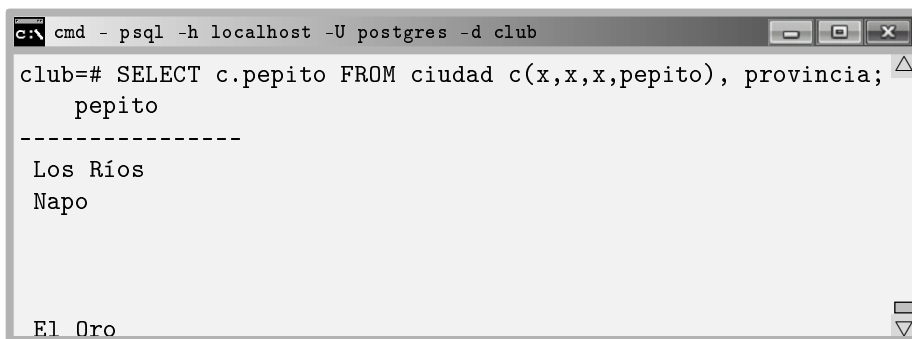
```

c:\> cmd - psql -h localhost -U postgres -d club
club=# SELECT c.provincia as provincias FROM ciudad c, provinci
provincias
-----
Los Ríos
Napó

El Oro
  
```

Pantalla 6.14. *Solución equivalente a la de la Pantalla 6.13 con renombramiento de la tabla ciudad, a c.*

En la Pantalla 6.14, además, se ha renombrado el atributo, y por eso el título *provincias* de la columna. El mismo efecto se consigue renombrando el atributo en el mismo lugar que se da el nombre *c*, o sea en el *FROM*. Eso significa renombrar la tabla y cada uno de sus atributos. La Pantalla 6.15 muestra una solución equivalente a la de la Pantalla 6.14.



```

c:\n cmd - psql -h localhost -U postgres -d club
club=# SELECT c.pepito FROM ciudad c(x,x,x,pepito), provincia;
      pepito
-----
 Los Ríos
  Napo

El Oro

```

Pantalla 6.15. *Cambio de nombre de los atributos de una relación.*

Es notable el hecho de que para renombrar un atributo hayamos de poner un nombre para cada uno de los demás. En la Pantalla 6.15 se ha llamado *x,x,x* a los otros atributos de la tabla *ciudad*. Eso va bien para confirmar que sabemos de qué estamos hablando, cuántos atributos tiene la tabla y qué interesa.

## Cláusula WHERE

La cláusula *WHERE* porta un predicado. Que quede claro, luego de la cláusula debe ir alguna expresión que pueda ser evaluada como cierta o falsa para cada una de las tuplas de la relación de la consulta, que como se ha dicho en la sección anterior, es la relación formada por el producto cartesiano de las relaciones de la cláusula *FROM*.

Recuerda del Capítulo 1.1 que un predicado está formado de proposiciones unidas por medio de operaciones conjuntivas o disyuntivas, o sea de las operaciones lógicas *AND* y *OR*.

El ejemplo más simple de la cláusula *WHERE* se hace en una consulta de una sola tabla.

**ejemplo 6.7.** *Nombres de las personas de Babahoyo.*

**solución** `SELECT nombres FROM persona WHERE ciudad = 'Babahoyo';`

Y otro ejemplo, esta vez implementando un *join*. Decimos *join* aquellas proposiciones que forman parte de los predicados, en los cuales se pide que valores de columnas

vinculadas coincidan. Vinculadas no significa necesariamente homónimas. Cuando se introducía la reunión interna, en la Sección 5.5.2, se ha precisado este punto.

**ejemplo 6.8.** *Nombres de las personas de la provincia del Napo.*

```
solución SELECT p.nombres
           FROM persona p,ciudad c
           WHERE p.ciudad = c.ciudad
           AND c.provincia = 'Napo';
```

La finalidad de haber declarado los mismos nombres para las claves primarias y las foráneas que las apunten, es decir, las columnas con las cuales se harán los joins es precisamente no tener que expresar las proposiciones correspondientes en los predicados. El ejemplo 6.8 se podrá simplificar a partir de la Sección 6.4.7.

Finalmente, un último ejemplo para mostrar la potencia de la estructura fundamental de una consulta. Para eso, será necesario rellenar algunos datos de la tabla `conoce`. Supongamos las siguientes amistades.

```
\echo ----- inserts tabla conoce

INSERT INTO conoce VALUES
('0127673812','0123478937'),
('0123478937','0127673812'),
('0127673812','1047548338'),
('0127673812','FFJ904992'),
('FFJ904992','0127673812'),
('FFJ904992','CKUS01549');
```

Caja 6.25. *Archivo club/conoce/inserts.sql.*

Para introducir estos datos, aunque no sea imprescindible, conviene remontar la base de datos completa importando el script `club.sql`.

<b>conoce</b>	<b>es _ conocida</b>
0127673812 Carmen Esmeralda	0123478937 Carlos Sayaro
0123478937 Carlos Sayaro	0127673812 Carmen Esmeralda
0127673812 Carmen Esmeralda	1047548338 Ana Estefanía
0127673812 Carmen Esmeralda	FFJ904992 Klauss
FFJ904992 Klauss	0127673812 Carmen Esmeralda
FFJ904992 Klauss	CKUS01549 Mick

Tabla 6.5: *Relaciones de amistad, no necesariamente simétricas, entre las personas.*



En la Tabla 6.5 se muestra el mismo contenido de la Caja 6.25 pero con los nombres de las personas al lado del número de pasaporte. En particular, en la Caja 6.25 se dice que Carmen Esmeralda de Babahoyo (0127673812) conoce Carlos Sayaro y Ana Estefanía Sanábria de Tena, i a Klauss de Berlín.

Una vez esos datos hayan sido introducidos, hay la capacidad de responder a cuestiones tan complicadas como la del ejemplo 6.9.

**ejemplo 6.9.** *¿A quien conoce Carmen Esmeralda?*

**solución**

```
SELECT q.nombre
FROM persona p, conoce, persona q
WHERE p.nombres = 'Carmen Esmeralda'
AND p.pasaporte = conoce
AND q.pasaporte = es_conocida;
```

Analicemos la solución del ejemplo 6.9.

Para pensar este tipo de consulta es imprescindible darse cuenta de que lo que se pide hace dos referencias a la tabla **persona**.

Una vez comprendido esto, empezamos poniendo un prefijo, en este caso **q**, al atributo del **SELECT**.

Entonces, en el **WHERE** hay que actuar en coherencia según el prefijo que se ha dado en el **SELECT**, ya que indica la tupla o el registro del cual en última instancia estamos pidiendo el resultado.

Describimos las ataduras que hay que respetar, teniendo en cuenta cual es el nombre que le hemos dado a la relación en el **SELECT**, o sea la **q**, y cuál nombre tiene la tabla que sirve para establecer los criterios que nos han dado, o sea **p**. Es sencillo ver que de las dos referencias a la tabla **persona** una es para establecer criterios y la otra para pedir resultados.

La respuesta del PostgreSQL al ejemplo 6.9 se puede observar a la Pantalla 6.16.

```

c:\cmd - psql -h localhost -U postgres -d club
----- inserts tabla persona
INSERT 0 6
----- inserts tabla conoce
INSERT 0 6
club=# SELECT q.nombre FROM persona p, conoce, persona q
club=# WHERE p.nombres = 'Carmen Esmeralda'
club=# AND p.pasaporte = conoce
club=# AND q.pasaporte = es_conocida;
      nombre
-----
 Carlos Sayaro
 Ana Estefanía
 Klauss
(3 rows)
club=# _

```

Pantalla 6.16. *Personas conocidas por Carmen Esmeralda.*

## 6.4.2 Funciones de Agregación

Las funciones de agregación en el SQL son un reflejo fiel de las definiciones del álgebra relacional, de la Sección 5.6.3. En esencia, son funciones que retornan un solo registro, que normalmente es de un solo atributo, calculado a partir de una colección de registros. Como consecuencia, si en una columna pasamos de muchas filas a una sola, tendremos que agrupar valores iguales cuando se pida por más de un atributo, y esa agrupación regirá las agregaciones calculadas. Hay que concentrarse para entenderlo. La agrupación segmentará las agregaciones calculadas. Eso se verá en la próxima sección.

Para los títulos de las columnas resultantes, PostgreSQL utiliza el nombre de la misma función de agregación. Siempre se puede renombrar con la cláusula `AS`, o inclusive, como en el caso del `FROM`, sin ninguna palabra. Si en la cláusula `SELECT` aparece una expresión y una palabra separadas solo por un espacio en blanco, el SQL entiende que la segunda es un nuevo nombre para la primera, como se ha visto en el ejemplo 6.14.

En adelante se necesitarán más cantidad de datos de los que se han usado hasta ahora. Por eso, quien desee hacer el seguimiento ejecutando cada ejemplo puede descargar de la dirección que se dice al final del preámbulo todas las necesarias. Los resultados que se muestran en las próximas secciones han sido calculados con la totalidad de estos datos. O también hay la posibilidad de hacerlo con papel y lapicero, teniendo en cuenta que los datos insertados en la base son los que se muestran en el Apéndice C.

Las funciones de agregación estándar en SQL se listan seguidamente.

### De todos los tipos de atributos

Las funciones de agregación para atributos alfanuméricos que podemos garantizar que nos retornarán una sola tupla, es decir, un solo registro, son la cuenta, el mínimo, y el máximo.

#### Función COUNT()

Función de conteo de la cantidad de registros que habría en la relación resultante de la consulta. Puesto que la cláusula `SELECT` trabaja por defecto con multiconjuntos, la forma natural de una relación que contiene tuplas repetidas, o equivalentes, es tomándolas como elementos distintos del conjunto relación. Eso significa que el resultado de un `SELECT COUNT(A) FROM r` para una relación  $r$  con un atributo requerido  $A$  que tuviera tres tuplas equivalentes sería tres. Por consiguiente, para contabilizar el número de tuplas de una relación podemos hacerlo contabilizando cualquiera de sus atributos requeridos, aunque por elegancia y legibilidad, es preferente el uso del asterisco. Caso que el atributo no fuera requerido, el resultado de la función sería el número de valores distintos de nulo que hubiere para ese atributo, contabilizando los repetidos tantas veces como estuvieren. En el siguiente ejemplo se resuelve una misma consulta de dos formas.

**ejemplo 6.10.** *¿Cuántas personas hay a la base de datos?*

**solución** `SELECT COUNT(pasaporte) FROM persona;`

Y como se ha dicho, la manera más correcta de hacerlo es poniendo un asterisco.

**solución** `SELECT COUNT(*) FROM persona;`

En particular, al seleccionar un solo atributo que se repite en el resultado, se cuenta tantas veces como aparezca. Para los valores distintos nomás, utilizamos el `DISTINCT` visto a la Sección 6.4.1.

**ejemplo 6.11.** *¿De cuántas provincias hay personas en la base de datos?*

**solución** `SELECT COUNT(DISTINCT c.provincia)  
FROM persona p, ciudad c  
WHERE p.ciudad = c.ciudad;`

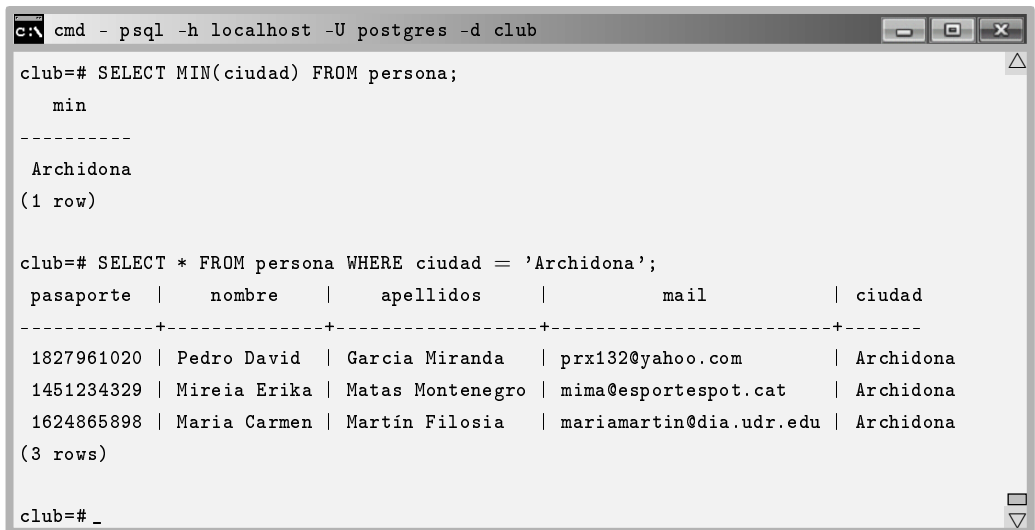
## Funciones MIN() y MAX()

Funciones que retornan el valor mínimo, o el máximo, del atributo dado según el orden establecido por defecto para ese tipo, a no ser que se indique explícitamente otro orden en la misma consulta. Es decir, cada tipo de datos tiene un orden por defecto asociado.

**ejemplo 6.12.** *¿Cuál es la primera ciudad por orden alfabético, de las que vive alguien de la base de datos?*

**solución** `SELECT MIN(ciudad) FROM persona;`

La captura correspondiente al ejemplo 6.12 se muestra en la Pantalla 6.17. Observa que si el mínimo valor se produce en varios registros. Igualmente retorna una sola tupla. Por tanto, no tiene sentido usar la cláusula `DISTINCT`.



```

c:\n cmd - psql -h localhost -U postgres -d club
club=# SELECT MIN(ciudad) FROM persona;
      min
-----
Archidona
(1 row)

club=# SELECT * FROM persona WHERE ciudad = 'Archidona';
 pasaporte | nombre | apellidos | mail | ciudad
-----+-----+-----+-----+-----
 1827961020 | Pedro David | Garcia Miranda | prx132@yahoo.com | Archidona
 1451234329 | Mireia Erika | Matas Montenegro | mima@esportespot.cat | Archidona
 1624865898 | Maria Carmen | Martín Filosia | mariamartin@dia.udr.edu | Archidona
(3 rows)

club=# _

```

Pantalla 6.17. *El mínimo es único, aunque se produzca en varios registros.*

## De atributos numéricos

Las funciones de agregación para atributos numéricos retornan siempre un solo número. De hecho, los atributos que no son numéricos en las consultas que utilizan funciones de agregación hacen el papel de *criterio de agregación*. Ese es un concepto que a menudo supone confusión para los neófitos, y se verá más adelante en esta misma sección. Por el momento observamos los usos más sencillos de estas funciones.

Para los tipos numéricos se dispone adicionalmente de las funciones descritas a continuación. Nomás se presentan dos. Sin embargo, las funciones de agregación disponibles dependen del SGBD. Todos saben calcular desviaciones tipo, y muchísimas otras funciones, pero no coinciden en los nombres que utilizan. Para eso, hay que consultar la ayuda.

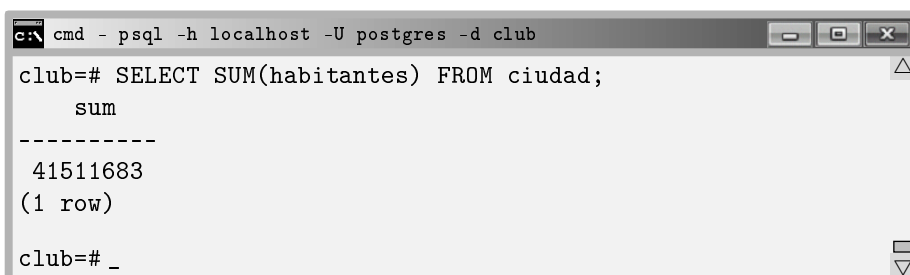
### Función SUM()

Retorna la suma de los valores dados. Si hay nulos se ignoran, es decir, los trata como si fueran ceros.

**ejemplo 6.13.** *¿Cuántos habitantes hay entre todas las ciudades de la base de datos?*

**solución** `SELECT SUM(habitantes) FROM ciudad;`

En la Pantalla 6.19 se imprime la ejecución del ejemplo 6.13.



```
cmd - psql -h localhost -U postgres -d club
club=# SELECT SUM(habitantes) FROM ciudad;
      sum
-----
 41511683
(1 row)
club=# _
```

Pantalla 6.18. Consulta de agregación con un atributo numérico.

### Función AVG()

Retorna el promedio de los valores dados. Si hay nulos los ignora, es decir, no contabilizan al hacer la división para calcular el promedio.

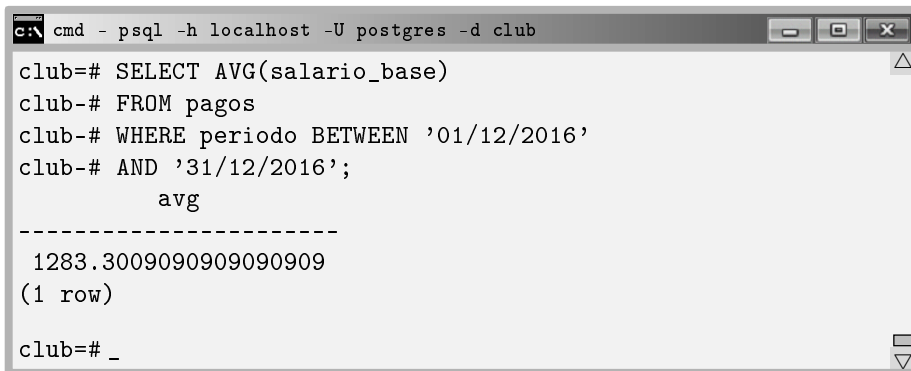
**ejemplo 6.14.** *Calcular el salario medio de los trabajadores del club deportivo el mes de diciembre del 2016.*

**solución** `SELECT AVG(salario_base)  
FROM pagos  
WHERE periodo BETWEEN '01/12/2016' AND '31/12/2016';`

En el ejemplo 6.14 hemos usado un operador booleano que se llama `BETWEEN AND` y es equivalente a poner dos proposiciones una de menor o igual y la otra de mayor o

igual. Son cosas del SQL. Ya se ha comentado que cuando nació pretendía impresionar entre otras cosas por su grado de humanidad en el lenguaje.

En la Pantalla 6.19 se puede ver el resultado del ejemplo 6.14.



```

cmd - psql -h localhost -U postgres -d club
club=# SELECT AVG(salario_base)
club=# FROM pagos
club=# WHERE periodo BETWEEN '01/12/2016'
club=# AND '31/12/2016';
          avg
-----
1283.3009090909090909
(1 row)
club=# _

```

Pantalla 6.19. *Consultas de agregación numéricas.*

### Cláusula GROUP BY

En la Caja 6.26 se muestra el formato de este tipo de consultas, siendo  $r = r(A_1, A_2, \dots, A_n)$  con  $n \geq k + \ell$ .

```

SELECT A1, A2, ..., Ak, f1(Ak+1), f2(Ak+2), ..., fℓ(Ak+ℓ)
FROM r
GROUP BY A1, A2, ..., Ak;

```

Caja 6.26. *Formato de una consulta con criterios de agregación.*

Los atributos  $A_1, A_2, \dots, A_k$  forman el criterio de agregación. El hecho que se hayan de repetir forzosamente en la cláusula **GROUP BY** significa que si deseamos saber alguna información adicional, a parte del resultado de la función de agregación, la información que queramos saber se utilizará para segmentar el resultado global.

Por tanto, la cláusula **GROUP BY** debe aparecer obligatoriamente al final de las consultas donde a parte de la función de agregación aparezcan más atributos en el **SELECT**. Es decir que si no, se produce un error sintáctico.

Visto como en la Caja 6.26 este tipo de consultas parecen más complicadas de lo que son. Normalmente,  $k = 1$ , y  $\ell = 2$ . O sea, que pedimos por la suma o el promedio de un atributo segmentado según otro.

Como se puede suponer, el impacto del criterio de agregación en la consulta es total. Según los atributos que se proyecten acompañando una función de agregación varía radicalmente el nombre de registros que se obtienen. Veámoslo en el caso de la suma por ejemplo, en dos consultas.

**ejemplo 6.15.** *¿Qué insumo mensual se tiene en el club deportivo?*

**solución**

```
SELECT SUM(cuota)
FROM hace;
```

El ejemplo 6.15 calcula una suma agregada total. Al no poner ningún atributo más que aquel sobre el cual se calcula la función de agregación, la consulta se convierte en una agregación para toda la relación `hace`. Es lo que se ha visto en la sección anterior.

Ahora bien, si deseamos saber cualquier otra información relativa a la suma del ejemplo 6.15, esta misma información segmentará la suma total.

**ejemplo 6.16.** *¿Cuál es el insumo mensual que se obtiene en el club deportivo por cada deporte?*

**solución**

```
SELECT deporte, SUM(cuota)
FROM hace
GROUP BY deporte;
```

Eso es así de claro. Mandan los atributos alfabéticos del criterio de agregación.

Para entendernos suponemos que el criterio de agregación es de un solo atributo, como en el caso del ejemplo 6.16. Entonces se supone que los valores de este atributo aparecen repetidamente en la relación de la consulta. También se supone que la relación de la consulta tiene algún atributo numérico. Bien, pues la función de agregación se calcula a partir de los valores de la columna numérica para todas las filas que coincidan en el valor del criterio de agregación. Si los valores del atributo que forma el criterio de agregación no se repiten, entonces la agregación no tendrá demasiado sentido. Todo ello también vale entendiendo que el criterio de agregación puede estar formado de varios atributos, en cuyo caso entendemos que distintos valores puede significar distintas combinaciones de valores.

Ergo, si el resultado de una consulta con funciones de agregación tiene más de una fila, la consulta debe tener una cláusula `GROUP BY`.

A menudo los usuarios finales se confunden en el momento de pedir atributos no agregados juntamente con agregados.

La ley básica es que cualquier relación resultante debe tener el mismo número de filas en todas sus columnas. Por tanto, si solo agregamos algunas columnas, las otras deben agruparse tanto como puedan, es decir, que cada valor distinto provoque una nueva fila. Y como consecuencia, la agregación se segmentará por esos valores.

En síntesis, cuando en un mismo `SELECT` se añaden atributos textuales y funciones de agregación numérica hay que tener en cuenta que si fuera por los numéricos se reduciría el resultado a una sola fila. En ese caso, y el valor de los textuales esa fila no quedaría bien definido. Por lo tanto, se agrupan los textuales tanto como pueden (lo que significa que se agrupan por valores coincidentes) determinando así el número de filas resultantes. Ese fenómeno se puede comprobar claramente con el Access de Microsoft, que en modo diseño de consultas, al establecer una función de agregación numérica en algún campo del resultado de la consulta, los textuales cambian automáticamente a la opción "agregar por".

En la Figura 6.7 se puede observar la captura de la pantalla para los ejemplos 6.15 y 6.16. Sobre esta captura se ha dibujado lo que representa la implicación sintáctica, y de hecho semántica, del atributo `deporte` haciendo de criterio de agregación.

### Cláusula `HAVING`

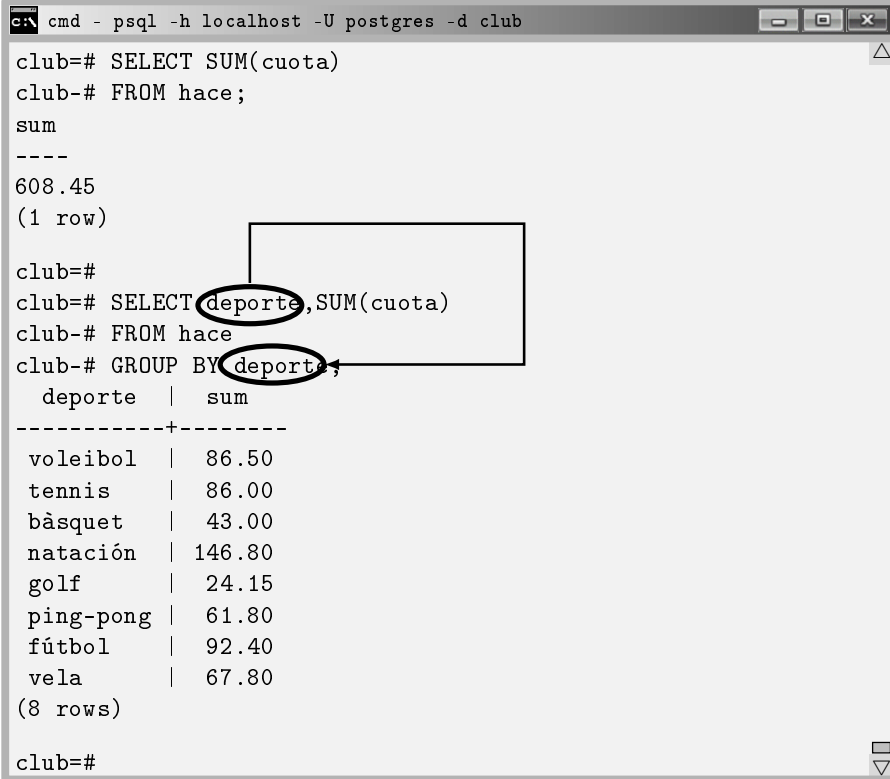
La cláusula `HAVING` sirve para establecer predicados sobre el resultado del valor de la función agregada. O sea, que incorpora un expresión booleana que involucra el resultado de una función de agregación. Por ejemplo, un promedio más alto que un valor.

Es pues, parecido al `WHERE`, pero en lugar de operar a partir de valores de atributos lo hace a partir del resultado de las funciones.

Razonemos. Si se trata de filtrar el resultado de una consulta agregada, es decir, dar como resultado solo las tuplas que satisfagan el predicado de la cláusula `HAVING`, entonces es que estamos hablando de una selección que en principio obtendrá varias tuplas. Y si el resultado de la consulta de agregación antes de seleccionar las que cumplan el predicado tiene varias columnas, entonces es que la consulta agregada tiene un `GROUP BY`, ya que la única manera posible que una consulta con una función de agregación tenga más de un registro resultante es que haya un `GROUP BY`. En consecuencia, en la mayor parte de los casos que se utiliza `HAVING`, en la misma consulta se ha utilizado el `GROUP BY`.

**ejemplo 6.17.** *¿Qué provincias tienen más de cien mil habitantes?*





```

c:\cmd - psql -h localhost -U postgres -d club
club=# SELECT SUM(cuota)
club=# FROM hace;
sum
----
608.45
(1 row)

club=#
club=# SELECT deporte,SUM(cuota)
club=# FROM hace
club=# GROUP BY(deporte);
  deporte  | sum
-----+-----
 voleibol  | 86.50
  tennis   | 86.00
 básquet   | 43.00
 natación  | 146.80
  golf     | 24.15
 ping-pong | 61.80
 fútbol    | 92.40
  vela     | 67.80
(8 rows)

club=#

```

Figura 6.7: Implicación sintáctica de la cláusula GROUP BY.

```

solución SELECT provincia, SUM(habitantes)
            FROM ciudad
            GROUP BY provincia
            HAVING SUM (habitantes) > 100000;

```

Esta consecuencia se puede llevar más lejos aún. Se puede asegurar que si una consulta contiene la cláusula HAVING y no la GROUP BY, entonces es una consulta de existencia. Es decir, que el resultado solo sirve para saber si el valor de la función de agregación para toda la relación de la consulta satisface el predicado. Es el caso del ejemplo 6.18, que puede retornar un registro, o ninguno.

**ejemplo 6.18.** *Entre todas las ciudades de la base, ¿Hay más de un millón de habitantes?*

```

solución SELECT SUM(habitantes)
            FROM ciudad
            HAVING SUM (habitantes) > 1000000;

```

### 6.4.3 Cláusula de Ordenación

La última de las cláusulas que puede traer una consulta de lectura es la que permite ordenar, ascendente o descendientemente, la relación resultante según un atributo, o más de uno si se quieren establecer órdenes secundarios para desempatar en el orden principal.

#### Cláusula ORDER BY

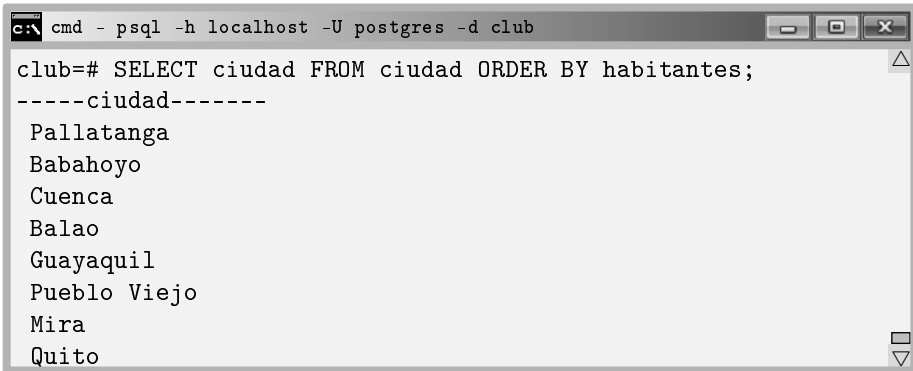
Cuando se quiere pedir la relación ordenada por algún atributo de la relación de la consulta se puede añadir la cláusula `ORDER BY` seguido del nombre del atributo en cuestión. Por defecto, la opción para ordenar es `ASC` que significa ascendente. Podemos invertirlo poniendo `DESC` luego del atributo.

A continuación se detalla qué significa ascendente y descendente según el tipo de los atributos.

- Para los numéricos los sentidos de orden ascendente y descendente son bien claros. Ascendente es creciente.
- Para los alfanuméricos el orden alfabético es el orden ascendente. Bien, más que el alfabético, el orden entre todos los caracteres de la tabla ASCII que se muestra al Apéndice B. O sea, que no es estrictamente alfabético, porque la `a` minúscula va luego de la `Z` mayúscula.
- Para los temporales, el pasado es menor que el futuro.
- Y para los booleanos, falso es menor que cierto.

Cuando no se pide ningún orden en una consulta que solo consta de una tabla en el `FROM`, PostgreSQL la da ordenada según el orden cronológico de las inserciones. Es decir, en la relación resultante aparecen primero los registros más antiguos. Eso es fácilmente comprobable consultando una tabla, borrando el primer elemento, y volviéndolo a insertar.

El criterio de ordenación debe ser un atributo presente en la relación de la consulta, y no necesariamente de la cláusula `SELECT`. Por ejemplo, en la Pantalla 6.20 se lista el nombre de las ciudades ordenadas por el número de habitantes, que no aparece.



```

c:\cmd - psql -h localhost -U postgres -d club
club=# SELECT ciudad FROM ciudad ORDER BY habitantes;
-----ciudad-----
Pallatanga
Babahoyo
Cuenca
Balao
Guayaquil
Pueblo Viejo
Mira
Quito

```

Pantalla 6.20. Ciudades ordenadas por número de habitantes.

Como criterio de ordenación se puede dar más de un atributo para el caso que en el primero haya valores repetidos en la relación resultante. Por ejemplo si se quisiera las personas de la base de datos ordenadas alfabéticamente, y para aquellas que se llamen igual, ordenadas según el orden alfabético de su ciudad, la consulta sería

```
SELECT nombres, ciudad FROM persona ORDER BY nombres, ciudad;
```

Adicionalmente, con la cláusula `ORDER BY` a menudo se da la cláusula `LIMIT` para limitar el número de registros de la relación resultante. Eso es especialmente útil para aplicaciones cliente que trabajan con páginas de tamaño fijo para las consultas.

### Consulta de lectura con todas las cláusulas

Cerramos el tema de las consultas de lectura con un ejemplo que utiliza las seis cláusulas. Para pensarla, hay que tener en cuenta que deseamos un filtro antes y un luego de una función de agregación. En el ejemplo 6.19 se expone la estructura completa que puede tener una consulta de lectura.

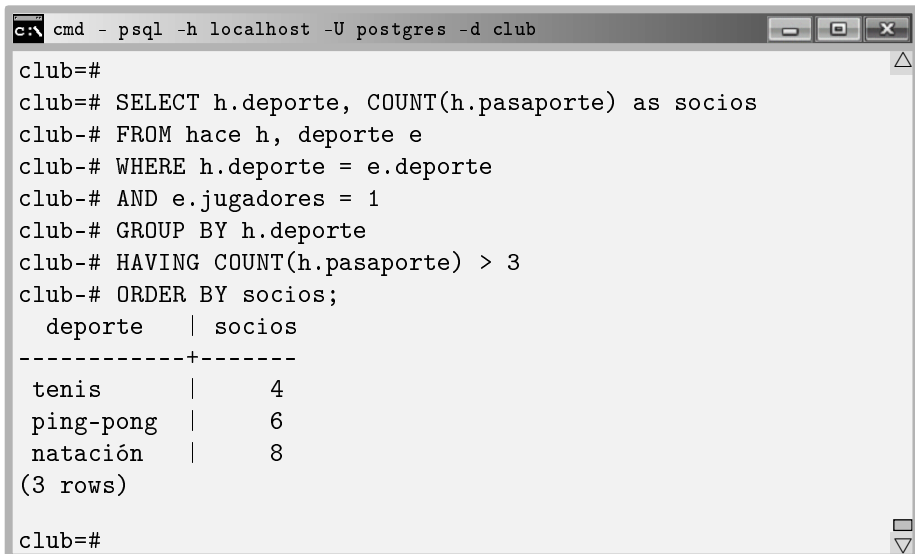
**ejemplo 6.19.** *Dar el nombre de los deportes y la cantidad de socios que los practican, de los deportes que practican más de tres socios y que se jueguen en solitario, ordenados por número de socios.*

```

solución SELECT h.deporte, COUNT(h.pasaporte) as socios
            FROM hace h, deporte e
            WHERE h.deporte = e.nombre
            AND e.jugadores = 1
            GROUP BY h.deporte
            HAVING COUNT(h.pasaporte) > 3
            ORDER BY socios;

```

Y en la Pantalla 6.21 se puede ver la relación resultante.



```

c:\n cmd - psql -h localhost -U postgres -d club
club=#
club=# SELECT h.deporte, COUNT(h.pasaporte) as socios
club=# FROM hace h, deporte e
club=# WHERE h.deporte = e.deporte
club=# AND e.jugadores = 1
club=# GROUP BY h.deporte
club=# HAVING COUNT(h.pasaporte) > 3
club=# ORDER BY socios;
  deporte | socios
-----+-----
   tenis |      4
 ping-pong |      6
 natación |      8
(3 rows)
club=#

```

Pantalla 6.21. Consulta de lectura con todas las cláusulas.

En SQL se puede utilizar el nombre de una columna renombrada en el `SELECT` para ordenar con la cláusula `ORDER BY`, pero no en el `HAVING`. O sea, si en la consulta de la Pantalla 6.21 hubiésemos puesto `ORDER BY COUNT(h.pasaporte)` ningún problema, en cambio si hubiésemos puesto `HAVING socios > 3`, entonces se habría producido un error.

#### 6.4.4 Vistas

Una vista es el soporte físico para una consulta. Es claro que como los datos de la base van variando con el tiempo puede interesar la misma consulta en distintos momentos. Y por tanto, conviene guardarlas de alguna manera para no tener que volverlas a teclear.

Los objetos de tipo vista, como las tablas, se guardan en los metadatos de la base. Pero a diferencia de una tabla que guarda físicamente los datos, de las vistas tan solo se guarda el código SQL de la consulta que implementan. Eso significa que si una vista muestra una columna de una tabla y se inserta, se modifica o se borra algún registro de la tabla, la relación resultante de la vista se verá afectada por estos cambios.

El concepto de vista es de los primeros SQLs que existieron, a principios de los setenta. Después, el SQL del 92 introdujo las funciones que pueden hacer todo lo

que hace una vista, y más cosas. De todas formas, las vistas son un concepto más estándar, ya que las funciones utilizan lenguajes de programación que dependen del SGBD. Y en cambio, las vistas tan solo utilizan SQL puro.

La sintaxis de creación de vistas se presenta en la Caja 6.27.

```
CREATE VIEW nombre_vista AS <consulta de lectura>
```

Caja 6.27. *Sentencia de creación de una vista.*

Hay que entender que una vista es como una tabla por lo que respecta a las operaciones. Por ejemplo, para consultar la relación resultante de una vista se hace con un `SELECT` exactamente igual que si fuese una tabla. Las vistas también se pueden poner en el `FROM` de cualquier consulta, y combinarlas con otras tablas o inclusive con otras vistas provocando dependencias en cascada.

Para borrar una vista, hay el comando `DROP VIEW nombre_vista`. Y también igual que en el caso de las tablas, si añadimos la opción `CASCADE` luego del nombre, antes del punto y coma, entonces se borrarán todas las vistas que usen ésta en su definición.

De la misma manera, si hacemos un `DROP TABLE nombre_tabla CASCADE`; de una tabla utilizada en la definición de alguna vista también se eliminará la vista en cuestión.

En la Caja 6.28 se implementa una vista con la relación de lo que paga cada socio cada mes, que le llamamos factura.

```
\echo ----- vista facturas_socio

CREATE VIEW facturas_socio AS
SELECT p.pasaporte,
       p.nombres,
       p.apellidos,
       SUM(f.cuota) AS factura
FROM persona p, hace f
WHERE p.pasaporte = f.pasaporte
GROUP BY p.pasaporte,p.nombres,p.apellidos
ORDER BY p.apellidos;
```

Caja 6.28. *Archivo club/hace/facturas\_socio.sql.*

Exactamente igual que en la etapa de construcción, creamos el archivo `facturas_socio.sql` en la carpeta raíz del proyecto. Entonces desde la línea de comandos del `psql` lo importamos haciendo `\i facturas_socio.sql`. Y verificamos su funcionamiento, tal como se ve en la Pantalla 6.22.

```

c:\> cmd - psql -h localhost -U postgres -d club
club=#
club=# \i facturas_socio.sql
----- vista facturas_socio
CREATE VIEW
club=# SELECT * FROM facturas_socio;
 pasaporte | nombre | apellidos | factura
-----+-----+-----+-----
1239238229 | Samantha Anabel | Aragall Tumbaco | 33.75
CKUS01549 | Michael | Bros | 31.20
1759119283 | Pedro Fernando | Camprubí Villasana | 32.85
HAT481338 | Rita | Derbeken | 24.00
1827961020 | Pedro David | Garcia Miranda | 32.85
1145493393 | Jesús Marcelo | Hortesa Orellana | 44.10
1238433548 | Anabel Madelyn | Zurita Margalef | 28.65
1624865898 | Maria Carmen | Martín Filoasia | 31.80
2142065765 | Camila Odalys | Noriega Pastuña | 43.00
0127673812 | Carmen Esmeralda | Peralta Gutiérrez | 44.05
1124637238 | Rosario Carmela | Puente Pizarro | 33.75
C01X01TN | Roberto | Rietto | 28.65
0123478937 | Carlos Savaro | Sanabria Oña | 44.05

```

Pantalla 6.22. Verificación del *escript* de la Caja 6.28.

En el espacio del proyecto guardaremos los *escripts* que implementen las vistas en la carpeta correspondiente a la última tabla que utilicen en su definición, según el orden del modelo relacional. Este nuevo *escript*, `facturas_socio.sql`, lo guardaremos en la carpeta `club/hace` del proyecto. Y la importación irá en el *escript* principal, justo luego de la importación del *escript* de creación de la tabla `hace`.

## 6.4.5 Consultas de Actualización

Llamamos consultas de actualización las que actualizan el estado de la base de datos. Es extraño llamar consulta a una inserción o eliminación, pero así es. Sí que se entiende en cambio que una inserción en una tabla es una actualización de la base de datos. Cuidado pues, que esto puede traer confusión. Hay tres comandos para hacer las consultas de actualización, `INSERT INTO`, `UPDATE SET`, y `DELETE FROM`. Eso contempla nuevos registros, modificaciones y eliminaciones. Las consultas de actualización se

caracterizan por el tipo de los datos que retornan. Un número entero. El resultado de una consulta de actualización es la cantidad de registros que se han visto afectados en la base de datos.

### Cláusula INSERT INTO

En la Sección 6.3.4 ya se ha anticipado el uso del comando para insertar registros de valores constantes. En su versión más sencilla se puede ver en la Caja 6.29.

```
INSERT INTO r VALUES (a1, a2, ..., an);
```

Caja 6.29. *Forma básica del comando de inserción.*

siendo  $r = r(A_1, A_2, \dots, A_n)$ , con  $A_i \subseteq D_i$  y  $a_i \in D_i$ , para  $i = 1, \dots, n$ .

En todos los scripts de inserción que hay en la carpeta del proyecto se utiliza profusamente la forma de la Caja 6.29, donde además, ya se ha visto que en una sola inserción se pueden introducir muchos registros constantes, separados por comas.

Si  $D_i$  es un dominio de tipo alfanumérico o temporal, entonces el valor  $a_i$  del registro insertado debe ir entre comillas simples, o apóstrofes, de manera que para introducir un apóstrofe hay que poner dos seguidos. Si  $D_i$  es de tipos numéricos se ponen sin comillas. Para los booleanos se da su valor con las palabras clave *true* o *false*. Y para un valor nulo, se escribe la palabra *null*.

Observa pues que en su forma más simple, el comando `INSERT INTO` tiene muy presente el esquema de la relación  $r$ , ya que los valores dados deben ser compatibles con ese esquema.

Ahora bien, si se desconoce el orden de los atributos en el esquema de  $r$  entonces también se puede hacer una inserción. Si no se sabe o no se puede utilizar el orden de los atributos en el esquema, hay que forzosamente saber sus nombres, así como los dominios para poder dar los valores. Además, también si son requeridos o no, ya que si son requeridos será necesario darlos en cualquier caso.

Con la forma que se muestra en la Caja 6.30 el comando hace explícito el esquema de  $r$ , al menos, la parte que afecta a la inserción.

```
INSERT INTO  $r(A_1, A_2, \dots, A_k)$  VALUES ( $a_1, a_2, \dots, a_k$ );
```

Caja 6.30. *Forma alternativa del comando de inserción.*

siendo  $k \leq n$ .

De esta manera, la asociación entre valores y atributos es posicional. O sea, el primer valor dado se corresponde con el primer atributo del esquema dado, y el segundo con el segundo. Se supone que los atributos de  $r$  no presentes en el comando,  $A_{k+1}, \dots, A_n$ , no son requeridos, y por tanto quedan nulos luego de la inserción, a no ser que en la tabla donde se está insertando haya valores definidos por defecto con la opción `DEFAULT`. Si algún atributo no dado en el comando fuese requerido y no tuviera declarado un valor por defecto, se produciría un error.

Así pues, esta forma alternativa de insertar valores constantes en una tabla consiste en dar el nombre de los atributos que se están suministrando del nuevo registro. Además, con la forma 6.30 tenemos la posibilidad de dar los atributos en cualquier orden. O sea, que esta forma para la instrucción de inserción tiene una doble vertiente. Aunque los demos todos, o sea  $k = n$ , también tiene sentido utilizarla si es porque interesa introducirlos en algún orden específico.

**ejemplo 6.20.** *Insertar un nuevo deporte, atletismo, con un jugador por equipo.*

**solución** `INSERT INTO deporte VALUES('atletismo', null, 1);`

o alternativamente

**solución** `INSERT INTO deporte(jugadores, nombre) VALUES(1, 'atletismo');`

En las dos soluciones del ejemplo 6.20 hacemos uso del valor por defecto del atributo `precio`, que es 10.0.

La forma más compleja de la cláusula es dando una expresión relacional, es decir, sin la palabra clave `VALUES`.

```
INSERT INTO  $r$  ( $E$ );
```

Caja 6.31. *Forma más compleja del comando de inserción.*

En la Caja 6.31 se presenta la forma de insertar el resultado de una consulta a una tabla. Fíjate que hay que tener muy claro el esquema resultante de la expresión




$E$  a fin que sea compatible con el de la relación  $r$ .

**ejemplo 6.21.** *Regalar una inscripción para hacer atletismo a los diez socios que más pagan.*

```
solución INSERT INTO hace (
        SELECT pasaporte,'atletismo',0.0
        FROM facturas_socio
        ORDER BY pagos DESC
        LIMIT 10
    );
```

En la Pantalla 6.23 se captura la ejecución de estos últimos comandos.



```
cmd - psql -h localhost -U postgres -d club
club=#
club=# INSERT INTO deporte(jugadores,nombre)
club=# VALUES(1,'atletismo');
INSERT 0 1
club=# INSERT INTO hace (
club=# SELECT pasaporte,'atletismo',0.0
club=# FROM facturas_socio
club=# ORDER BY pagos DESC
club=# LIMIT 10
club=# );
INSERT 0 10
club=#
```

Pantalla 6.23. *Inserción compleja en base a una consulta.*

Dos comentarios respecto al contenido de la Pantalla 6.23. Por una parte mencionar el uso de la proyección de un valor constante, *atletismo*, o el 0.0 de la cuota en la cláusula `SELECT` de la inserción. Y también fíjate en el uso del orden descendente, `DESC` y de la cláusula `LIMIT`. Se había hablado de esta estructura en la Sección 6.4.3.

Errores habituales provocados por esta instrucción son el de existencia previa de clave primaria, o el de necesidad de valores requeridos que no se dan. Otro error muy común es, en tablas con claves foráneas, que no existía ningún registro en la tabla apuntada con el valor de clave primaria que se está dando.

### Cláusula `UPDATE SET`

Por la misma naturaleza que al hacer una inserción hay que hacer explícitos todos los valores requeridos de los registros que se insertan, para hacer una modificación de un

conjunto de registros existentes, tan solo hay que identificarlos y conocer el nombre y los tipos de los atributos que se pretenden modificar, a parte de los nuevos valores, claro.

En la Caja 6.32 se presenta la cláusula que sirve para hacer modificaciones en el contenido de la base de datos. El orden en que se den los atributos  $A_i, \dots, A_{y+k-1}$  es irrelevante. Si no se pone el **WHERE** la modificación se hace en todos los registros de la tabla. Los  $a_1, a_2, \dots, a_k$  son los nuevos valores, o expresiones que dan un resultado del dominio del atributo al que se asignen.

La expresión de la Caja 6.32 modifica todos los registros de  $r$  para los que el predicado dé cierto. No obstante, en su versión más simple solo se modifica el valor de un atributo. Y además, el predicado  $p$  consta de una sola proposición que involucra la clave primaria de  $r$ .

```
UPDATE r SET Ai = a1, Ai+1 = a2, ..., Ai+k-1 = ak WHERE p;
```

Caja 6.32. *Sintaxis del comando de actualización.*

siendo  $r = r(A_1, \dots, A_i, \dots, A_{i+k-1}, \dots, A_n)$  y  $p$  un predicado que involucra atributos de  $r$  y constantes.

En el ejemplo 6.22 se hace una modificación de tantos registros como tenistas haya en la base de datos.

**ejemplo 6.22.** *Aumentar un 10% la cuota del tenis a los socios.*

**solución** UPDATE hace  
SET cuota = cuota \* 1.10  
WHERE deporte = 'tenis' ;

El ejemplo 6.23 es más complejo, puesto que para hallar las tuplas que deben actualizarse se requiere una subconsulta sobre la autorelación **OBEDece** del modelo.

**ejemplo 6.23.** *Nombrar nueva jefe de entrenadores a Elena Angustias Fernández Mierdaza.*

**solución** UPDATE trabajador  
SET obedece = (SELECT pasaporte  
FROM persona  
WHERE nombres = 'Elena Angustias'  
AND apellidos = 'Fernández Mierdaza')  
WHERE departamento = 'entrenador';

La consulta del ejemplo 6.23 supone una única *Elena Angustias Fernández Mierdaza*. Y como se puede ver en la Pantalla 6.24 tres registros de la tabla trabajador quedan afectados por esta modificación, es decir, hay tres entrenadores en la base de datos. En la misma pantalla se ilustran los ejemplos 6.22 y 6.23.

```

c:\> cmd - psql -h localhost -U postgres -d club
club=#
club=# UPDATE hace
club=# SET cuota = cuota * 1.10
club=# WHERE deporte = 'tenis';
UPDATE 4
club=# UPDATE trabajador
club=# SET obedece = (SELECT pasaporte
club(# FROM persona
club(# WHERE nombre = 'Elena Angustias'
club(# AND apellidos = 'Fernández Mierdaza')
club=# WHERE departamento = 'entrenador';
UPDATE 3
club=#

```

Pantalla 6.24. Actualizaciones con la cláusula UPDATE SET.

Errores frecuentes de esta instrucción pasan por la definición de las restricciones en el momento de creación de la base de datos. Si no se ha declarado la manera de actualizar claves foráneas cuando se modifica el valor de la clave apuntada, por defecto es `ON UPDATE RESTRICT`, que significa que se prohíba la actualización y por tanto se emita un mensaje de error en caso de intento de modificación del valor de una clave principal.

### Cláusula DELETE FROM

Es natural que para borrar registros de la base de datos tan solo se requiera identificarlos. Eso se hace exactamente igual que con un `SELECT` de una sola tabla. La manera de eliminar registros de la base de datos es la que se muestra en la Caja 6.33.

`DELETE FROM  $r$  WHERE  $p$ ;`

Caja 6.33. Sintaxis del comando de eliminación.

siendo  $r$  una relación y  $p$  un predicado definido sobre sus atributos.

Todos los registros de  $r$  en los que se satisfaga el predicado  $p$  resultarán eliminados de la relación  $r$ . Huelga decir pues que en ese predicado deben constar proposiciones que involucren los atributos de la relación  $r$ .

Efectivamente, igual cuesta encontrar un registro para consultarlo, o sea con un `SELECT`, que para borrarlo con un `DELETE FROM`. De hecho, que el esfuerzo computacional de la consulta y de la eliminación sean equivalentes sucede con todas las estructuras de datos. Por eso tiene su lógica que la sintaxis para la eliminación sea exactamente igual que la de la selección con asterisco, pero sin el asterisco.

En el ejemplo 6.24 se utiliza el operador `LIKE` para el test de si un valor se ajusta al de un patrón. Se ha introducido en la Sección 6.2.4, y también ha sido utilizado en la creación del dominio de los mails en la Sección 6.3.4.

**ejemplo 6.24.** *Borra las provincias que empiecen con la letra A.*

**solución** `DELETE FROM provincia WHERE provincia LIKE 'A%';`

Con la ejecución de esta consulta se elimina la provincia *Azuay*, ya que es la única con esa inicial.

Observa en un segundo ejemplo la directiva `ON DELETE SET NULL` que se establece en la creación de la tabla *persona* para el campo *ciudad*.

**ejemplo 6.25.** *Eliminar la ciudad de Klaus Stallman.*

**solución** `DELETE FROM ciudad  
WHERE ciudad = (SELECT ciudad  
FROM persona  
WHERE nombres = 'Klauss'  
AND apellidos = 'Stallman');`

En la Pantalla 6.25 se describe la ejecución de los dos ejemplos anteriores.

Para el 6.25 se verifica previamente que la ciudad existe, y en lugar de anidar la consulta se hace en dos pasos.

Es conveniente analizar la diferencia espacio temporal entre realizar una misma consulta como anidada en otra, que requiere más tiempo de respuesta, o como producto cartesiano en la cláusula `FROM`, que requiere más espacio.

Tomar esa decisión debe tener en cuenta los recursos disponibles cuando hablamos de tablas grandes, así como de la cantidad de veces que la consulta deberá ser efectuada, pues una consulta que se requiera un millón de veces al día, conviene que sea rápida. Si no tenemos restricciones, entonces que prime la legibilidad.

```

c:\cmd - psql -h localhost -U postgres -d club
club=# DELETE FROM provincia WHERE nombre LIKE '0%';
club=# DELETE 1
club=# SELECT * FROM persona
club=# WHERE nombres = 'Klauss' AND apellidos = 'Stallman';
 pasaporte | nombres | apellidos | ciudad
-----+-----+-----+-----
 FFJ904992 | Klauss  | Stallman  | Berlín
(1 row)
club=# DELETE FROM ciudad WHERE ciudad = 'Berlín';
club=# DELETE 1
club=# SELECT * FROM persona
club=# WHERE nombres = 'Klauss' AND apellidos = 'Stallman';
 pasaporte | nombres | apellidos | ciudad
-----+-----+-----+-----
 FFJ904992 | Klauss  | Stallman  |
(1 row)
club=# _

```

Pantalla 6.25. *Eliminaciones con la cláusula DELETE FROM.*

### 6.4.6 Operaciones con Conjuntos

Aunque solo fuese por respecto a sus ancestros, el álgebra relacional y aún más allá la teoría de conjuntos, el SQL implementa de la manera que puede las tres operaciones básicas de la teoría de conjuntos. De la manera que puede significa que teniendo en cuenta que las relaciones en este capítulo son multiconjuntos, hay que formalizar el comportamiento de las operaciones en este nuevo entorno.

Estas tres operaciones son la unión, la diferencia y la intersección de conjuntos. Las tres operaciones son binarias, o sea que operan a partir de dos relaciones de entrada. Y también todas ellas requieren la compatibilidad de esas relaciones. La compatibilidad entre relaciones se ha definido en la Sección 5.4.4, en la página 135, en base a la compatibilidad entre conjuntos, introducida ya en la Caja 1.1.2.

En definitiva, cada uno de los atributos de los `SELECTS` de las relaciones de entrada deben pertenecer a dominios compatibles dos a dos. O sea, que o son iguales, o uno es subconjunto del otro.

Seguidamente se presentan cada de esas tres cláusulas, con sus pormenores respecto a las consecuencias de trabajar con multiconjuntos.

**Cláusula UNION**

La operación de unión de relaciones implementada en SQL tiene la sintaxis que se muestra en la Caja 6.34,

```
SELECT  $A_1, A_2, \dots, A_k$ 
FROM  $r$ 
WHERE  $p$ 
UNION
SELECT  $B_1, B_2, \dots, B_k$ 
FROM  $s$ 
WHERE  $q$ ;
```

Caja 6.34. *Sintaxis de la operación de unión de relaciones.*

siendo los esquemas  $A_1, A_2, \dots, A_k$  y  $B_1, B_2, \dots, B_k$  compatibles.

El resultado es una relación formada por los registros resultantes de la primera consulta y luego los de la segunda, a no ser que se cambie el orden explícitamente en la misma consulta.

Está claro que muchas relaciones que podrían calcularse con esta operación resuelven el problema añadiendo una disyuntiva en el predicado del **WHERE**. Aun así, en otros casos no es posible.

El ejemplo 6.26 obtiene un listado de las personas del club juntamente con la factura si son socios, o el salario base del rol de pago si son trabajadores.

Además del número de pasaporte, los nombres, apellidos y la factura, que será un valor negativo cuando se trate de pagos a trabajadores, también muestra en una columna adicional que llama **vínculo**. Esa columna tan solo puede contener los valores de tipo texto "socio" o bien "trabajador".

Respecto a la columna factura, se puede observar que se imprimen como números negativos todos aquellos que supongan un gasto para el club. Es decir, los pagos a los trabajadores. Por consiguiente, en la solución del ejemplo 6.26 se produce una redundancia. Si el signo del valor numérico presenta bajo el título **factura** es negativo, entonces se cooresponderá a una tupla referente a un trabajador. Y si por el contrario ese signo es positivo, entonces se trata de un socio del club deportivo.

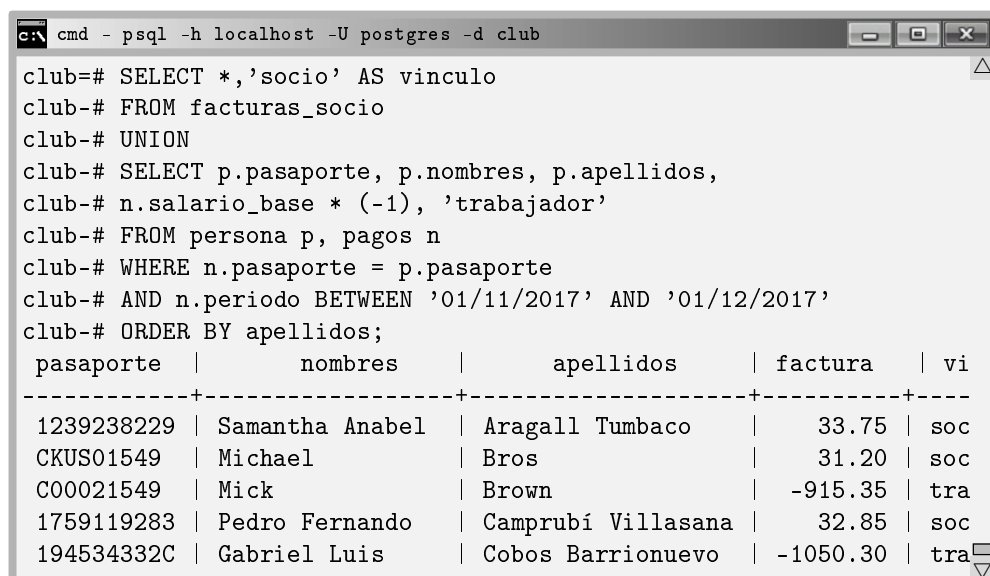
Sin duda, el esfuerzo mental más relevante que se debe efectuar para formular la consulta del ejemplo 6.26 es la de mantener en mente que los esquemas de las

dos subconsultas deben coincidir. Por tanto, antes que nada, hay que consultar cuáles son las columnas que devuelve la vista usada en primer lugar, para ajustar la segunda subconsulta a ese esquema. Eso es, de la Pantalla 6.22 de la página 224, `factura_pagos = factura_pagos(pasaporte, nombres, apellidos, factura)`.

**ejemplo 6.26.** *¿Qué balance de gastos y pagos ha supuesto para el club cada persona el noviembre del 2017?*

```
solución SELECT *, 'socio' AS vinculo
FROM facturas_socio
UNION
SELECT p.pasaporte, p.nombres, p.apellidos,
       n.salario_base * (-1), 'trabajador'
FROM persona p, pagos n
WHERE n.pasaporte = p.pasaporte
AND n.periodo BETWEEN '01/11/2017' AND '01/12/2017'
ORDER BY apellidos;
```

El hecho de proyectar atributos constantes, como 'socio' o 'trabajador', en la última columna ya se había visto en la Sección 6.4.5. Concretamente en el comando de inserción.



```
c:\N cmd - psql -h localhost -U postgres -d club
club=# SELECT *, 'socio' AS vinculo
club=# FROM facturas_socio
club=# UNION
club=# SELECT p.pasaporte, p.nombres, p.apellidos,
club=# n.salario_base * (-1), 'trabajador'
club=# FROM persona p, pagos n
club=# WHERE n.pasaporte = p.pasaporte
club=# AND n.periodo BETWEEN '01/11/2017' AND '01/12/2017'
club=# ORDER BY apellidos;
```

pasaporte	nombres	apellidos	factura	vi
1239238229	Samantha Anabel	Aragall Tumbaco	33.75	soc
CKUS01549	Michael	Bros	31.20	soc
C00021549	Mick	Brown	-915.35	tra
1759119283	Pedro Fernando	Camprubí Villasana	32.85	soc
194534332C	Gabriel Luis	Cobos Barrionuevo	-1050.30	tra

Pantalla 6.26. *Unión de relaciones.*

Además, en la Pantalla 6.26 es puede ver la forma como la unión de dos relaciones asigna títulos de las columnas. Tan solo que una de las dos relaciones tenga título,

lo toma. Por eso no hay que dar un título a la columna con el salario base negativo. En la relación resultante se llamará *factura*, ya que la primera de las relaciones de entrada sí que tiene título para esa columna. Exactamente lo mismo pasa con la columna *vinculo*.

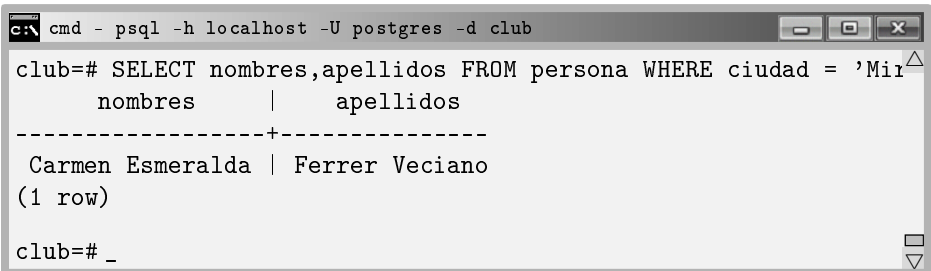
## Multiconjuntos

La unión de relaciones agrega por defecto los registros equivalentes. Eso se puede evitar con la opción `ALL`. O sea, exactamente al contrario que el `SELECT`, que por defecto repite repeticiones, y requiere la opción `DISTINCT` para agregarlas.

A fin de interiorizar la diferencia entre trabajar con conjuntos y trabajar con multiconjuntos, analizamos meticulosamente las dos opciones con un experimento de cinco pasos. Hay que entender que se trata de un experimento técnico con este objetivo.

Está claro que los resultados de las consultas que a continuación se detallan se podrían obtener más sencillamente tan solo con la estructura fundamental de las consultas de lectura.

1. Seleccionamos nombres y apellidos de las personas que viven en *Mira*.



```

c:\> cmd - psql -h localhost -U postgres -d club
club=# SELECT nombres,apellidos FROM persona WHERE ciudad = 'Mira'
      nombres      |      apellidos
-----+-----
 Carmen Esmeralda | Ferrer Veciano
(1 row)
club=# _
  
```

Pantalla 6.27. Paso 1. Personas de Mira, 1 resultado.

2. Seleccionamos nombres y apellidos de las personas que viven en *Babahoyo*.



```

cmd - psql -h localhost -U postgres -d club
club=# SELECT nombres,apellidos FROM persona WHERE ciudad = 'Bab
      nombres      |      apellidos
-----+-----
 Carmen Esmeralda | Peralta Gutiérrez
(1 row)
club=# _

```

Pantalla 6.28. Paso 2. Personas de Babahoyo, 1 resultado.

3. Seleccionamos nombres y apellidos de las personas que viven en *Mira* y de las personas que viven en *Babahoyo*.

```

cmd - psql -h localhost -U postgres -d club
club=# SELECT nombres,apellidos FROM persona WHERE ciudad = 'Mir
club=# UNION
club=# SELECT nombres,apellidos FROM persona WHERE ciudad = 'Bat
      nombres      |      apellidos
-----+-----
 Carmen Esmeralda | Ferrer Veciano
 Carmen Esmeralda | Peralta Gutiérrez
(2 rows)
club=# _

```

Pantalla 6.29. Paso 3. Personas de Mira o de Babahoyo, 2 resultados.

4. Seleccionamos solo los nombres de las personas que viven en *Mira* y de las personas que viven en *Babahoyo*.

```

cmd - psql -h localhost -U postgres -d club
club=# SELECT nombres FROM persona WHERE ciudad = 'Mira'
club=# UNION
club=# SELECT nombres FROM persona WHERE ciudad = 'Baba
      nombres      |
-----+-----
 Carmen Esmeralda
(1 row)
club=# _

```

Pantalla 6.30. Paso 4. Los nombres nomás, de las personas de Mira y de Babahoyo, 1 resultado.

Hasta aquí, ya se ve que la operación se comporta de la manera más fiel a la teoría de conjuntos. Hay que entender que en la Pantalla 6.29 aparecen dos resultados gracias a que los apellidos distinguen los dos registros. En el momento que dejamos de proyectar los apellidos, pasan a ser iguales, y la operación de unión los agrega por defecto.

Si se quiere trabajar con la versión multiconjunto de la operación hay que añadir la opción `ALL` luego de la cláusula `UNION`. La operación `UNION ALL` es la versión multiconjunto de la operación `UNION`. Eso significa que queremos que las tuplas equivalentes en la relación resultante no sean agregadas, sino que aparezcan tantas veces como efectivamente resulte tras efectuar los cálculos de la relación.

Como en el caso del `SELECT` y el `SELECT DISTINCT`, las versiones multiconjunto sirven para cuando se desean contar ocurrencias de los datos, cosa que se realiza mediante el uso de funciones de agregación.

5. Seleccionamos solo el nombre de todas las personas de *Mira* y *Babahoyo*.

```

cmd - psql -h localhost -U postgres -d club
club=# SELECT nombre FROM persona WHERE ciudad = 'Mira'
club=# UNION ALL
club=# SELECT nombre FROM persona WHERE ciudad = 'Babahoyo';
      nombres      |      apellidos
-----+-----
 Carmen Esmeralda
 Carmen Esmeralda
(2 rows)

club=# _

```

Pantalla 6.31. Paso 5. Nomás los nombres de todas las personas de *Mira* y de *Babahoyo*, 2 resultados.

En definitiva, por defecto las operaciones `SELECT` y `UNION` actúan de maneras opuestas, cosa que podría hacer reflexionar.

### Cláusula EXCEPT

La operación de diferencia de relaciones se implementa con el `EXCEPT`. Su sintaxis es como la de la operación `UNION` de la sección anterior. Se presenta en la Caja 6.35,

```

SELECT A1, A2, ..., Ak
FROM r
WHERE p
EXCEPT
SELECT B1, B2, ..., Bk
FROM s
WHERE q;

```

Caja 6.35. *Sintaxis de la operación de diferencia de relaciones.*

con esquemas  $A_1, A_2, \dots, A_k$  y  $B_1, B_2, \dots, B_k$  compatibles.

El resultado es la relación formada por los registros de la primera relación que no están en la segunda. Fíjate pues que los que haya en la segunda y no en la primera no tienen ninguna incidencia en la operación.

La operación de diferencia de relaciones es un reflejo de la definida en la Sección 5.4.5 del álgebra relacional. Observa que el término *except* es un término nuevo para este concepto.

Filosóficamente, la misma sinapsis que hay entre la disyunción *o* y la operación de unión, o la conjunción *y* y la operación de intersección, es la que hay entre la diferencia y la intersección *y no*, también llamado *pero*. La operación **EXCEPT** retorna los elementos del primer operando pero no del segundo.

**ejemplo 6.27.** *Obtener los nombres y apellidos de los trabajadores que no son jefes de nadie.*

**solución**

```

SELECT p.nombres,p.apellidos
FROM persona p, trabajador t
WHERE p.pasaporte = t.pasaporte
EXCEPT
SELECT p.nombres,p.apellidos
FROM persona p, trabajador t
WHERE p.pasaporte = t.obedece
ORDER BY apellidos;

```

La resolución del ejemplo 6.27 se ha implementado a partir del razonamiento de que los números de pasaporte de los trabajadores que figuran en la columna **obedece** de la tabla **trabajador** corresponden a jefes de alguien. Por tanto lo que se pide son los nombres de todos los trabajadores excepto esos.

Por otra parte, esta solución se toma la libertad de ordenar los registros alfabéticamente. La ejecución de esta consulta se puede ver en la Pantalla 6.32.

```

c:\N cmd - psql -h localhost -U postgres -d club
club=# SELECT p.nombres,p.apellidos
club=# FROM persona p, trabajador t
club=# WHERE p.pasaporte = t.pasaporte
club=# EXCEPT
club=# SELECT p.nombres,p.apellidos
club=# FROM persona p, trabajador t
club=# WHERE p.pasaporte = t.obedece
club=# ORDER BY apellidos;
nombres      | apellidos
-----+-----
Mick          | Brown
Gabriel Luis  | Cobos Barrionuevo
Sonia Erika   | Colmena Robles
Carmen Esmeralda | Ferrer Veciano
José Antonio  | González Huertas

```

Pantalla 6.32. *Diferencia de relaciones.*

### Cláusula INTERSECT

Para la implementación de la operación de intersección de relaciones se utiliza la cláusula `INTERSECT`, y se utiliza igual que las dos operaciones anteriores. También es la operación de intersección que se ha visto en la Sección 5.5.1 del álgebra relacional. Su uso se introduce en la Caja 6.36,

```

SELECT A1, A2, ..., Ak
FROM r
WHERE p
INTERSECT
SELECT B1, B2, ..., Bk
FROM s
WHERE q;

```

Caja 6.36. *Sintaxis de la operación de intersección de relaciones.*

siendo los esquemas  $A_1, A_2, \dots, A_k$  y  $B_1, B_2, \dots, B_k$  compatibles, como siempre.

El resultado es una relación formada por los registros de la primera consulta que también están en la segunda. Y como en el caso de la unión, en muchas ocasiones

se puede prescindir de esta operación con el predicado adecuado en la estructura fundamental. Es decir, con proposiciones conjuntivas en el predicado del **WHERE**.

Igualmente como antes, hay casos en que esto resulta complicado, como en el ejemplo 6.28.

**ejemplo 6.28.** *¿Qué socios practican fútbol y básquet?*

```
solución SELECT p.pasaporte,p.nombres,p.apellidos
FROM persona p, hace h
WHERE p.pasaporte = h.pasaporte
AND h.deporte = 'fútbol'
INTERSECT
SELECT p.pasaporte,p.nombres,p.apellidos
FROM persona p, hace h
WHERE p.pasaporte = h.pasaporte
AND h.deporte = 'básquet'
ORDER BY apellidos;
```

Resulta complicado prescindir de la intersección cuando el predicado del **WHERE** debería tener proposiciones incidentes en distintos registros de una misma tabla. Fíjate pero, que la complejidad de este tipo de consultas estriba en la doble referencia a una misma tabla en la consulta, y por tanto se pueden resolver como se muestra en la Caja 6.37. O sea que esta expresión se basa en un tratamiento de todas las parejas posibles de tuplas de una relación. Se trata de una solución alternativa.

$$r_1 \leftarrow \sigma_{x.deporte='básquet' \wedge y.deporte='fútbol'}(\rho_x(\text{hace}) \times \rho_y(\text{hace}))$$

$$r_2 \leftarrow \Pi_{\text{pasaporte}}(r_1)$$

$$\Pi_{\text{pasaporte,nombres,apellidos}}(r_2 \bowtie \text{persona})$$

Caja 6.37. *Expresión alternativa a la consulta de intersección del ejemplo 6.37.*

La forma de la solución de la Caja 6.37 es muy común. La estrategia pasa por conseguir la lista de claves primarias de la relación donde se hallan los atributos que se pretende finalmente conseguir. En este caso en particular, **persona**, puesto que lo que se pide en la solución son pasaporte, nombres y apellidos. Así pues, los primeros pasos de la consulta tienen por objetivo conseguir los valores de pasaporte de las tuplas resultantes. Y una vez con ellos, tan solo con un natural join ya conseguimos la solución.

Y como se puede ver en la Pantalla 6.33, por casualidad, son los dos de Berlín.

```

c:\cmd - psql -h localhost -U postgres -d club
club=# SELECT p.pasaporte,p.nombres,p.apellidos
club-# FROM persona p, hace h
club-# WHERE p.pasaporte = h.pasaporte
club-# AND h.deporte = 'fútbol'
club-# INTERSECT
club-# SELECT p.pasaporte,p.nombres,p.apellidos
club-# FROM persona p, hace h
club-# WHERE p.pasaporte = h.pasaporte
club-# AND h.deporte = 'básquet'
club-# ORDER BY apellidos;
 pasaporte | nombres | apellidos
-----+-----+-----
 HAT481338 | Rita    | Derbeken
 FFJ904992 | Klauss  | Stallman

(2 rows)
club=#

```

Pantalla 6.33. *Intersección de relaciones.*

### 6.4.7 Operaciones de Reunión de la Cláusula FROM

Las operaciones de reunión del álgebra relacional tienen su versión en SQL. A lo largo de estas secciones supondremos que  $r$  es una relación con un esquema de  $n$  atributos de los cuales hay  $k$  que coinciden, o sea se llaman igual que otros  $k$  atributos de otra relación  $s$ , que tiene  $m$ . Sin pérdida de generalidad podemos suponer que los  $k$  atributos comunes a las dos relaciones son los  $k$  primeros. Es decir,  $r = r(C_1, \dots, C_k, A_{k+1}, \dots, A_n)$ , y  $s = s(C_1, \dots, C_k, B_{k+1}, \dots, B_m)$ .

Todas las operaciones de reunión se definen según una condición. Se le llama *condición de reunión*, y puede ser o bien **NATURAL**, o bien **USING**, o bien **ON**. Forzosamente debe aparecer alguna de las tres y solo una. Cada condición relaja más y más el predicado interno de la reunión. Es decir, la reunión interna con la condición **NATURAL** es la más restrictiva. La condición **USING** es una relajación de la reunión natural, y la condición **ON** es una relajación de la condición **USING**.

Si la reunión es **NATURAL**, entonces no hay que decir nada más. El esquema resultante tendrá  $m + n - k$  columnas, para cualquiera de los cuatro tipos de reunión. Si se usa la condición **USING** se debe dar una lista ordenada de atributos que sea un subconjunto de los comunes, que normalmente será un subconjunto propio. Entonces las relaciones resultantes tendrán, de entre los atributos comunes, una sola vez los que se haya dado en la lista, y repetidos los demás. Con la condición **ON** hay que dar algún predicado. En este caso los esquemas resultantes tendrán todos los  $m + n$  atributos de los cuales

$k$  serán repetidos y por tanto los nombres de las columnas de la relación resultante, iguales.

En concreto, las reunión interna natural selecciona filas con valores iguales en todas las columnas homónimas. Todas.

Con la condición `USING` se seleccionan filas con valores iguales en algunas, solo algunas, de las columnas homónimas.

Con la condición `ON`, se puede establecer cualquier predicado como si fuese una cláusula `WHERE`.

En las próximas secciones, el tipo de reunión se puede omitir. Es decir, las palabras clave `INNER` y `OUTER` no se precisan. Para los ejemplos, se van a utilizar las relaciones

- `persona(pasaporte,nombres,apellidos,mail,ciudad)` de 31 registros.
- `ciudad(ciudad,habitantes,provincia)` que tiene 21.

A continuación se ve la sintaxis de cada una de las operaciones de reunión, y se ilustran con algunos ejemplos.

#### Cláusula `NATURAL INNER JOIN`

La reunión natural interna en versión SQL tiene el aspecto de la Caja 6.38,

```
SELECT  $X_1, X_2, \dots, X_\ell$ 
FROM  $r$  NATURAL JOIN  $s$ 
WHERE  $p$ ;
```

Caja 6.38. *Sintaxis de la reunión natural.*

siendo  $X_1, \dots, X_\ell$  cualquiera de los atributos de  $r$  o de  $s$ . Y  $p$ , un predicado que involucra atributos de  $r$ , atributos de  $s$ , y constantes.

El resultado de esta consulta tendrá por esquema los atributos de  $r$  y a continuación los que falten de  $s$ . PostgreSQL pondrá primero las columnas comunes, es decir,  $(C_1, \dots, C_k, A_{k+1}, \dots, A_n, B_{k+1}, \dots, B_m)$ .

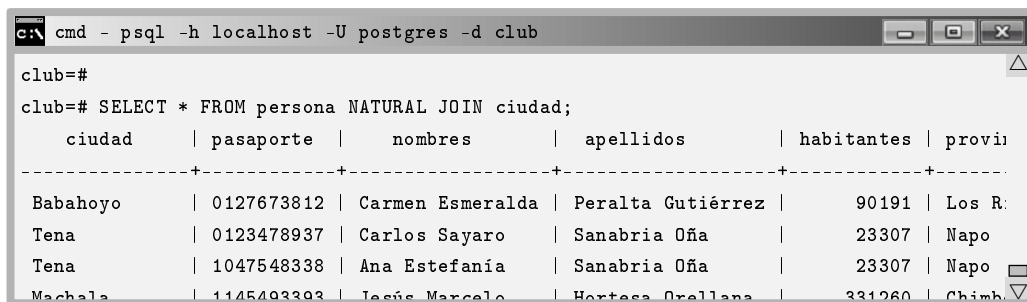
Las tuplas de la relación resultante son aquellas en las que los valores de las columnas homónimas coincidan. Además, claro, si hay la cláusula `WHERE` deben satisfacer el predicado.

Así pues, recogemos por fin el fruto de haber llamado igual, siempre que se ha podido, las claves primarias que las foráneas que las apuntan.

**ejemplo 6.29.** *Dar todos los datos de las personas con las ciudades donde viven.*

**solución** `SELECT * FROM persona NATURAL JOIN ciudad;`

Teniendo en cuenta que `persona` tiene cuatro columnas, `ciudad` tres, y hay una única columna homónima, el esquema de la reunión natural tendrá seis columnas. En la Pantalla 6.34 se lista la parte inicial de la relación resultante.



```

c:\> cmd - psql -h localhost -U postgres -d club
club=#
club=# SELECT * FROM persona NATURAL JOIN ciudad;
 ciudad | pasaporte | nombres | apellidos | habitantes | provi
-----+-----+-----+-----+-----+-----
 Babahoyo | 0127673812 | Carmen Esmeralda | Peralta Gutiérrez | 90191 | Los R.
 Tena | 0123478937 | Carlos Sayaro | Sanabria Oña | 23307 | Napo
 Tena | 1047548338 | Ana Estefanía | Sanabria Oña | 23307 | Napo
 Machala | 1145493393 | Jesús Marcelo | Hortosa Orallana | 331260 | Chimb

```

Pantalla 6.34. *Reunión natural de ciudad y persona.*

Observa el orden de los títulos. Tenemos como columnas homónimas `ciudad`. Por tanto, sólo las filas del producto cartesiano en las cuales ese valor coincida. Si en lugar de ser una sola columna hubiesen varias columnas comunes, todas ellas aparecerían una sola vez, y el número de filas sería inferior o igual al actual. Por otra parte, en la Pantalla 6.34 no aparecen las personas de las que no se conozca su ciudad, ni las ciudades de las que no hay ninguna persona, como se verá seguidamente.

### Cláusula `NATURAL LEFT OUTER JOIN`

La reunión natural externa por la izquierda, además de los registros resultantes de la reunión natural interna, añadirá cualquier fila de la relación de la izquierda a la relación resultante, con nulos a las columnas correspondientes a la relación de la derecha.

**ejemplo 6.30.** *Dar los datos de todas las personas y si se sabe, de las ciudades donde*



viven.

**solución** `SELECT * FROM persona NATURAL LEFT JOIN ciudad;`

La parte inicial de la ejecución del ejemplo 6.30 se muestra en la Pantalla 6.35.

```

c:\> cmd - psql -h localhost -U postgres -d club
club=#
club=# SELECT * FROM persona NATURAL LEFT JOIN ciudad;
 ciudad      | pasaporte | nombres      | apellidos      | habitantes | pro
-----+-----+-----+-----+-----+-----
 Babahoyo    | 0127673812 | Carmen Esmeralda | Peralta Gutiérrez | 90191 | Los I
 Tena        | 0123478937 | Carlos Sayaro    | Sanabria Oña     | 23307 | Napo
 Tena        | 1047548338 | Ana Estefanía    | Sanabria Oña     | 23307 | Napo
 Machala     | 1145493393 | Jesús Marcelo    | Hortesa Orellana | 331260 | Chiml
 San Francisco | CKUS01549 | Michael         | Bros             | 805235 | 
 Berlín     | FFJ904992  | Klauss          | Stallman        | 3499879 | 
            | 1239238229 | Samantha Anabel | Aragall Tumbaco  |         | 
 Berlín     | HAT481338  | Rita           | Derbeken        | 3499879 | 
 Balao      | 1124637238 | Rosario Carmela | Puente Pizarro   | 8221  | Guay:
 Roma       | C01X01TN   | Roberto        | Rietto          | 2796102 | 
 Balao      | 1238433548 | Anabel Madelyn  | Zurita Margalef | 8221  | Guay:
 Machala    | 1232234958 | José Marcelo    | Sanlúcar Flavia  | 331260 | Chiml
 Pallatanga | 2038474483 | Miquel Pablo    | Vila Valverde    | 3160  | Chiml
 Machala    | 2142065765 | Camila Odalys   | Noriega Pastuña  | 331260 | Chiml
            | 1759119283 | Pedro Fernando  | Camprubí Villasana |         | 
 Archidona  | 1827961020 | Pedro David     | Garcia Miranda   | 5478  | Napo

```

Pantalla 6.35. *Reunión natural externa por la izquierda de ciudad y persona.*

No sabemos las ciudades de Samantha Aragall y Pedro Camprubí. Se nota que se ha remontado la base de datos, ya que en la Pantalla 6.35, *Berlín* vuelve a existir. Esta relación tiene 31 registros, tantos como *persona*.

### Cláusula NATURAL RIGHT OUTER JOIN

La reunión natural externa por la derecha, además de los registros resultantes de la reunión natural interna, añadirá cualquier fila de la relación de la derecha con nulos en las columnas de la relación de la izquierda.

En este caso las filas se añadirán al final.

**ejemplo 6.31.** *Dar los datos de todas las ciudades y las personas que viven en ellas si se sabe.*

**solución** `SELECT * FROM persona NATURAL RIGHT JOIN ciudad;`

```

club=# SELECT * FROM persona NATURAL RIGHT JOIN ciudad;
 ciudad | pasaporte | nombres | apellidos | habitantes | pr
-----+-----+-----+-----+-----+--
 Babahoyo | 0127673812 | Carmen Esmeralda | Peralta Gutiérrez | 90191 | Los
 Tena | 0123478937 | Carlos Sayaro | Sanabria Oña | 23307 | Napo
 Tena | 1047548338 | Ana Estefanía | Sanabria Oña | 23307 | Napo
 Machala | 1145493393 | Jesús Marcelo | Hortesa Orellana | 331260 | Chiriquí
 San Francisco | CKUS01549 | Michael | Bros | 805235 | 
 Berlin | FFJ904992 | Klauss | Stallman | 3499879 | 
 Berlin | HAT481338 | Rita | Derbeken | 3499879 | 
 Balao | 1124637238 | Rosario Carmela | Puente Pizarro | 8221 | Guayas
 Roma | C01X01TN | Roberto | Rietto | 2796102 | 
 Balao | 1238433548 | Anabel Madelyn | Zurita Margalef | 8221 | Guayas
 Machala | 1232234958 | José Marcelo | Sanlúcar Flavia | 331260 | Chiriquí
 Pallatanga | 2038474483 | Miquel Pablo | Vila Valverde | 154 | Chiriquí
 Machala | 2142065765 | Camila Odalys | Noriega Pastuña | 331260 | Chiriquí
 Archidona | 1827961020 | Pedro David | Garcia Miranda | 5478 | Napo
 ...
 Chilla | 193439185 | Boris Mauro | Santos Auyacu | 65444 | Chiriquí
 Rio de Janeiro | | | | 6320446 | 
 Zaruma | | | | 22222 | Chiriquí
 Atenas | | | | 664046 | 
(32 rows)
club=# _

```

Pantalla 6.36. *Reunión natural externa por la derecha de persona y ciudad.*

En la Pantalla 6.36 se han suprimido quince filas centrales para poder ver que Samantha y Pedro del ejemplo anterior ya no están. Y al final, han aparecido tres ciudades en las que no vive nadie.

### Cláusula NATURAL FULL OUTER JOIN

En la relación resultante aparecen todos los valores de todos los registros de las dos tablas, y por tanto muchos valores nulos.

**ejemplo 6.32.** *Obtener todos los datos de las personas y las ciudades en una sola relación*

**solución** `SELECT * FROM persona NATURAL FULL JOIN ciudad;`

La ejecución del ejemplo 6.32 se muestra en la Pantalla 6.37.

```

c:\n cmd - psql -h localhost -U postgres -d club
club=# SELECT * FROM persona NATURAL FULL JOIN ciudad;
 ciudad | pasaporte | nombres | apellidos | habitantes | pro
-----+-----+-----+-----+-----+-----
 Babahoyo | 0127673812 | Carmen Esmeralda | Peralta Gutiérrez | 90191 | Los
 Tena | 0123478937 | Carlos Sayaro | Sanabria Oña | 23307 | Nap
 Tena | 1047548338 | Ana Estefanía | Sanabria Oña | 23307 | Nap
 Machala | 1145493393 | Jesús Marcelo | Hortesa Orellana | 331260 | Chi
 San Francisco | CKUS01549 | Michael | Bros | 805235 |
 Berlín | FFJ904992 | Klauss | Stallman | 3499879 |
 | 1239238229 | Samantha Anabel | Aragall Tumbaco | |
 Berlín | HAT481338 | Rita | Derbeken | 3499879 |
 Balao | 1124637238 | Rosario Carmela | Puente Pizarro | 8221 | Gua
 Roma | C01X01TN | Roberto | Rietto | 2796102 |
 Balao | 1238433548 | Anabel Madelyn | Zurita Margalef | 8221 | Gua
 Machala | 1232234958 | José Marcelo | Sanlúcar Flavia | 331260 | Chi
 Pallatanga | 2038474483 | Miquel Pablo | Vila Valverde | 154 | Chi
 Machala | 2142065765 | Camila Odalys | Noriega Pastuña | 331260 | Chi
 | 1759119283 | Pedro Fernando | Camprubi Villasana | |
 Archidona | 1827961020 | Pedro David | Garcia Miranda | 5478 | Nap
 ...
 Chilla | 193439185 | Boris Mauro | Santos Auyacu | 65444 | Chi
 Rio de Janeiro | | | | 6320446 |
 Zaruma | | | | 22222 | Chi
 Atenas | | | | 664046 |
(34 rows)

club=# _

```

Pantalla 6.37. *Reunión natural externa completa de ciudad y persona.*

### Cláusula INNER JOIN USING

La condición `USING` sirve cuando  $k > 1$ . Eso es, cuando haya varias columnas con los mismos nombres. Si deseamos evitar lo que nos resultaría de la reunión natural, y

solo atar algunos de los atributos homónimos pero no todos, entonces utilizamos esta condición.

```
SELECT X1, X2, ..., Xℓ
FROM r JOIN s USING (C1, ..., Cj);
```

Caja 6.39. *Sintaxis de la reunión interna con la condición USING.*

siendo  $j < k$ .

Es decir, tras la condición `USING` se añade una lista entre paréntesis de atributos comunes a las dos relaciones. Se trata de un subconjunto propio de la intersección de atributos ordenado de manera que refleja el orden en la relación resultante.

O sea, primero se inventó la reunión natural, hasta que llegó alguien y dijo que le molestaba que atara todas las columnas comunes. Entonces se inventó la reunión interna con la condición `USING` para relajar la reunión natural. Y luego llegó otro y dijo que ya puestos, podrían relajar aún más el predicado y dejarlo abierto a cualquier otra expresión sin necesidad de hacer referencia a las columnas homónimas, y apareció el `ON`.

El ejemplo 6.29 con la condición `USING` se muestra en la Pantalla 6.38.



```
c:\N cmd - psql -h localhost -U postgres -d club
club=#
club=# SELECT * FROM persona JOIN ciudad USING (ciudad);
 ciudad | pasaporte | nombres | apellidos | habitantes | provi
-----+-----+-----+-----+-----+-----
 Babahoyo | 0127673812 | Carmen Esmeralda | Peralta Gutiérrez | 90191 | Los R...
 Tena | 0123478937 | Carlos Sayaro | Sanabria Oña | 23307 | Napo
 Tena | 1047548338 | Ana Estefanía | Sanabria Oña | 23307 | Napo
 Machala | 1145493393 | Jesús Marcelo | Hortesa Orellana | 331260 | Chimb...
 San Francisco | CKUS01549 | Michael | Bros | 805235 |
 Berlín | FFJ904992 | Klaus | Stallman | 3499879 |
 Berlín | UAT481228 | Rita | Darbaker | 2400870 |
```

Pantalla 6.38. *Reunión interna con la condición USING.*

Como entre las dos relaciones solo hay una columna homónima, el resultado de la Pantalla 6.38 coincide totalmente con el de la reunión natural de la Pantalla 6.34. Pero

bien, sirve para ilustrar la ejecución. Observa en el orden de las columnas que aquellas que siendo comunes se han agregado en el resultado se ponen primero, ciudad. Si hubieran columnas comunes no presentes en la cláusula USING se habrían mostrado en el lugar correspondiente de la tabla correspondiente.

Las cláusulas LEFT OUTER JOIN USING, RIGHT OUTER JOIN USING, y FULL OUTER JOIN USING funcionan de manera análoga.

### Cláusula INNER JOIN ON

La condición ON sirve para introducir cualquier tipo de predicado. Esta operación toma la forma indicada en la Caja 6.40. La relación resultante de la operación viene condicionada por el predicado obligatorio  $p$  que puede involucrar cualquier atributo de las dos relaciones de entrada, y constantes.

```
SELECT  $X_1, X_2, \dots, X_\ell$ 
FROM  $r$  JOIN  $s$  ON  $p$ ;
```

Caja 6.40. *Sintaxis de la reunión interna con la condición ON.*

Cuando  $p$  es una conjunción de proposiciones que pidan la coincidencia de columnas homónimas, el resultado es como la reunión natural con columnas repetidas, Pantalla 6.39.

```
cmd - psql -h localhost -U postgres -d club
club=#
club=# SELECT * FROM persona JOIN ciudad ON persona.ciudad = ciudad.ciudad;
 pasaporte |      nombres      |  apellidos   | ciudad | ciudad | habitantes | provin
-----+-----+-----+-----+-----+-----+-----
 0127673812 | Carmen Esmeralda | Peralta Gutiérrez | Babahoyo | Babahoyo | 90191 | Los Ríos
 0123478937 | Carlos Sayaro    | Sanabria Oña    | Tena    | Tena    | 23307 | Napo
 1047548338 | Ana Estefanía    | Sanabria Oña    | Tena    | Tena    | 23307 | Napo
 1145493393 | Jesús Marcelo    | Hortesa Orellana | Machala | Machala | 331260 | Chimborra
 CKUS01549 | Michael          | Bros            | San Francisco | San Francisco | 805235 |
 FF1004992 | Klaus           | Stallman        | Berlín   | Berlín   | 349879 |
```

Pantalla 6.39. *Reunión interna con la condición ON.*

Observa como siempre que se usa la condición `ON` el esquema resultante es, rigurosamente, la concatenación de los dos esquemas de las relaciones entrantes.

Usar predicados que no afecten a las claves primarias o foráneas en la cláusula `ON` funciona, pero no es habitual.

Así pues, las razones que justifican el uso de la reunión interna son primero, que las claves foráneas no se llamen como las claves primarias. Esta está bien clara. Pero aún hay otro motivo por el cual esta operación está definida. Se trata de cuando no deseamos todos los atributos de una reunión natural. Observa que si de las relaciones resultantes de las Pantallas 6.34 y 6.39 se pidiera por tan solo algún atributo concreto darían el mismo resultado.

En síntesis, cuando se proyectan solo algunos atributos de la reunión natural, el resultado es el mismo que cuando se proyectan los mismos atributos en la reunión interna con otras condiciones de reunión, siempre y cuando el predicado involucre columnas homónimas.

Las cláusulas `LEFT OUTER JOIN ON`, `RIGHT OUTER JOIN ON`, y `FULL OUTER JOIN ON` funcionan de manera análoga.

### 6.4.8 Valores Nulos

Esencialmente hay dos cláusulas directamente relacionadas con el valor nulo.

#### **Predicado IS NULL**

Para establecer los predicados en las selecciones de las cláusulas `WHERE` o `JOIN ON` podemos requerir que el valor de un atributo de la relación de la consulta no exista, es decir, que se trate de un valor nulo. Cuidado, la proposición `A = NULL` es falsa, independientemente de si `A` tiene valor a no. En cualquier caso, `NULL` no es un valor. Para preguntar si `A` no tiene valor la sintaxis es `A IS NULL`.

Más en general, `E IS NULL` es un predicado donde `E` puede ser cualquier expresión formada de atributos y constantes. Eso se presta a confusión, y por eso lo experimentamos en dos pasos.

1. Pedir incorrectamente las personas que no se sabe dónde viven haciendo

```
SELECT * FROM persona WHERE ciudad = NULL;
```

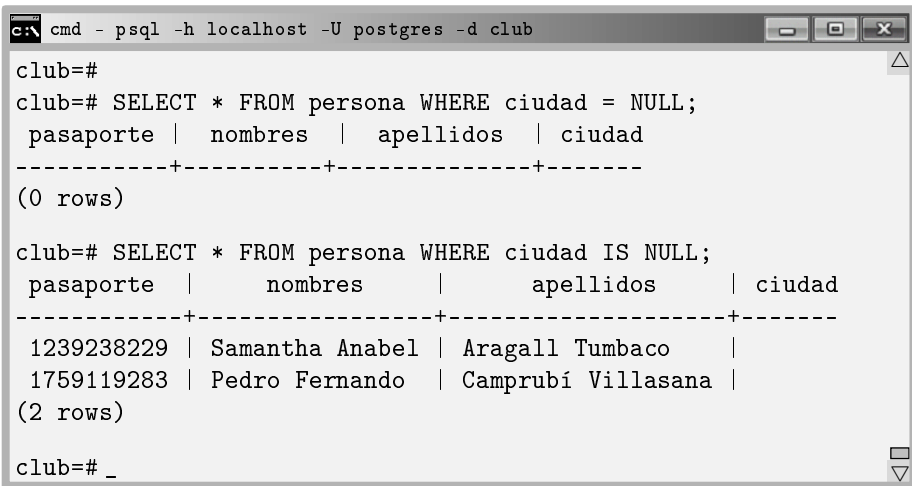
No hay resultados. Las comparaciones con nulos siempre retornan falso.

2. Pedir correctamente las personas que no se sabe dónde viven, haciendo

```
SELECT * FROM persona WHERE ciudad IS NULL;
```

Observa como el resultado tiene las dos personas que ya conocíamos de la Sección 6.4.7

El experimento se muestra en la Pantalla 6.40.



```

c:\n cmd - psql -h localhost -U postgres -d club
club=#
club=# SELECT * FROM persona WHERE ciudad = NULL;
 pasaporte | nombres | apellidos | ciudad
-----+-----+-----+-----
(0 rows)

club=# SELECT * FROM persona WHERE ciudad IS NULL;
 pasaporte | nombres | apellidos | ciudad
-----+-----+-----+-----
 1239238229 | Samantha Anabel | Aragall Tumbaco |
 1759119283 | Pedro Fernando | Camprubí Villasana |
(2 rows)

club=# _

```

Pantalla 6.40. *Experimento con el predicado IS NULL.*

Huelga decir que para el caso que interese la pregunta contraria, o sea requerir la existencia de valor en algún atributo en un predicado, entonces pondremos

```
WHERE A IS NOT NULL.
```

### **Predicados IS UNKNOWN, IS DISTINCT OF**

Para el caso específico de los atributos booleanos se debe usar `IS UNKNOWN` en lugar de `IS NULL`. Pero no deja de ser una cuestión de legibilidad.

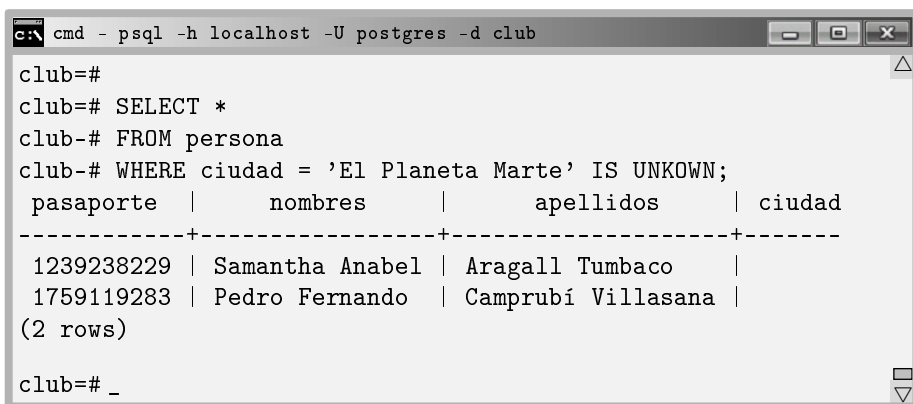
Para atributos de cualquier tipo de datos, existen algunos otros operadores, como el `IS DISTINCT OF` garantizando que no por operar con valores nulos nos devuelva falso. Es decir, que si comparamos dos valores nulos, nos devuelve cierto.

Hay algunos otros operadores interesantes. Aquí se han citado los más habituales. Para todos ellos existe la versión negada, `IS NOT` . . .

Como ejemplo del uso de `IS UNKNOWN`, en la Pantalla 6.41 se puede ver una manera alternativa de resolver el experimento anterior, personas que no se sabe donde viven.

En este caso, se resuelve preguntando las personas que no se sabe si viven en *El Planeta Marte*. Es decir, cualquier valor concreto. Como que la evaluación resulta conocida para todas aquellas personas de las que se sepa la ciudad, se obtienen otra vez los mismos registros que en la segunda parte del experimento de la sección anterior.

Sin lugar a dudas, hay mucha lógica en este comportamiento. Si no sé dónde vives, no sé si vives en el planeta Marte.



```

c:\> cmd - psql -h localhost -U postgres -d club
club=#
club=# SELECT *
club=# FROM persona
club=# WHERE ciudad = 'El Planeta Marte' IS UNKOWN;
 pasaporte |      nombres      |      apellidos      | ciudad
-----+-----+-----+-----
 1239238229 | Samantha Anabel  | Aragall Tumbaco     |
 1759119283 | Pedro Fernando  | Camprubí Villasana |
(2 rows)
club=# _

```

Pantalla 6.41. *Uso del predicado IS UNKNOWN.*

### 6.4.9 Tablas temporales

Si la cuestión se complica demasiado, siempre queda la opción de crear vistas, o tablas temporales con la `CREATE TEMP TABLE`, que crea tablas que se borran automáticamente al finalizar una sesión. En la Pantalla 6.42 se muestra el hecho.



```

c:\cmd - psql -h localhost -U postgres -d club
club=# CREATE TEMP TABLE a(n integer);
CREATE TABLE
club=# \c postgres
You are now connected to database "postgres" as user "postgres"
postgres=# \c club
You are now connected to database "club" as user "postgres".
club=# SELECT * FROM a;
ERROR: relation "a" does not exist
LINE 1: SELECT * FROM a;
                        ^
club=# _

```

Pantalla 6.42. *Las tablas temporales desaparecen al salir de la sesión.*

### 6.4.10 Consultas Anidadas

Llegados a este estadio, ya podemos resolver consultas más complicadas. Consultas dentro de consultas, cosa que ya se ha ido anticipando en las últimas secciones.

En la estructura fundamental `SELECT FROM WHERE` cualquiera de las tres cláusulas acepta consultas anidadas en lugar de lo que se ha visto hasta ahora. Sin embargo, poner un `SELECT` dentro un `SELECT` o en un `FROM` es feo, y de difícil legibilidad, como se explica seguidamente.

#### En el argumento de la cláusula `SELECT`

Funciona poner una consulta anidada en un `SELECT`. Eso es un hecho. Sin embargo, es mala práctica. Tan solo es admisible hacerlo para pruebas intermedias durante el desarrollo. Se trata de legibilidad.

Ese tipo de anidamientos permite obtener valores de atributos descriptivos a partir de claves primarias utilizando renombramientos.

Atención, las anidadas han de ser consultas que solo retornen una columna. Es decir, no se puede poner un `SELECT` que retorne dos atributos anidado en otro `SELECT`, pero sí uno con dos anidados retornando dos columnas. Quizás se comprenda mejor así:

```

SELECT (SELECT A,B FROM ...) está prohibido,
SELECT (SELECT A FROM ...),(SELECT B FROM ...) es correcto.

```

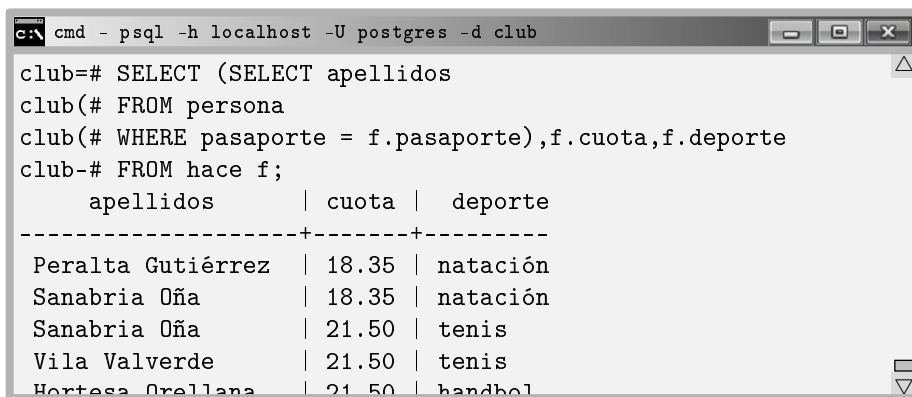
**ejemplo 6.33.** *Obtener los apellidos de los socios, con lo que pagan por cada deporte.*

**solución**

```
SELECT (SELECT apellidos
        FROM persona
        WHERE pasaporte = h.pasaporte),h.cuota,h.deporte
FROM hace h;
```

El PostgreSQL nos retornaría un error si en el ejemplo 6.33 se hubiera pedido por los nombres, además de los apellidos.

En la Pantalla 6.43 se visualiza la parte inicial de la ejecución de esta consulta, que lista 38 registros.



```
cmd - psql -h localhost -U postgres -d club
club=# SELECT (SELECT apellidos
club(# FROM persona
club(# WHERE pasaporte = f.pasaporte),f.cuota,f.deporte
club-# FROM hace f;
      apellidos      | cuota | deporte
-----+-----+-----
Peralta Gutiérrez  | 18.35 | natación
Sanabria Oña       | 18.35 | natación
Sanabria Oña       | 21.50 | tenis
Vila Valverde      | 21.50 | tenis
Hortesa Orellana   | 21.50 | handbol
```

Pantalla 6.43. *SELECT dentro de SELECT.*

El interés del ejemplo 6.33 está en la sintaxis de la consulta anidada.

En sí, esto no aporta potencia expresiva adicional, ya que la misma consulta podría haber sido resuelta sin anidamientos.

**solución**

```
SELECT apellidos,cuota,deporte
FROM persona NATURAL JOIN hace;
```

### En el argumento de la cláusula FROM

Igualmente, el SQL también soporta poner consultas en lugar de tablas o vistas en la cláusula FROM. Y también resulta mala práctica por cuestiones de legibilidad y conviene evitarlo. Como se ha dicho antes estos gazapos son útiles mientras se desarrolla, para

probar si el procedimiento que se está desarrollando funcionaría con un dato que en este momento se lo proporcionamos así, pero que en cuanto verifiquemos su ejecución debemos poner en una función o vista auxiliar y quitarlo `SELECT` llamando en su lugar a esa función breve.

Siempre que se ponga una subconsulta en la cláusula `FROM` se debe renombrar.

**ejemplo 6.34.** *Socios que practican fútbol.*

```
solución SELECT pasaporte,nombres,apellidos
          FROM (SELECT pasaporte
                FROM hace
                WHERE deporte='fútbol') AS futbol NATURAL JOIN persona;
```

La solución de esta consulta se puede ver en la Pantalla 6.44.



```
cmd - psql -h localhost -U postgres -d club
club=# SELECT pasaporte,nombres,apellidos
club-# FROM (SELECT pasaporte
club(# FROM hace
club(# WHERE deporte='fútbol') AS futbol NATURAL JOIN persona;
 pasaporte |      nombres      | apellidos
-----+-----+-----
 0127673812 | Carmen Esmeralda  | Peralta Gutiérrez
 0123478937 | Carlos Sayaro     | Sanabria Oña
 FFJ904992  | Klauss           | Stallman
 1239238229 | Samantha Anabel  | Aragall Tumbaco
 HAT481338  | Rita             | Derbeken
 1124637238 | Rosario Carmela  | Puente Pizarro
(6 rows)
club=#
```

Pantalla 6.44. `SELECT dentro de FROM`.

Otra vez, una solución alternativa y más clásica es la siguiente.

```
solución SELECT pasaporte,nombres,apellidos
          FROM persona NATURAL JOIN hace
          WHERE deporte='fútbol';
```

### 6.4.11 Consultas Anidadas en la Cláusula WHERE

Las consultas anidadas en la cláusula WHERE es donde deben ir.

Siempre se puede comparar el valor de un atributo con el resultado de una consulta en lugar de compararlo con una constante o otro atributo. Se hace simplemente substituyendo la constante o el segundo atributo por la consulta entre paréntesis. En la Caja 6.41 se presenta la sintaxis de estas comparaciones.

```

SELECT A1, A2, ..., Ak
FROM r
WHERE Aj = (
    SELECT B
    FROM s
);

```

Caja 6.41. *Consulta anidada.*

siendo  $r = r(A_1, A_2, \dots, A_n)$ ,  $A_i \subseteq D_i$  para  $i = 1, \dots, n$ ,  $k \leq n$ ,  $j \leq n$ , y  $s$  una relación con el atributo  $B \subseteq D_B$  como mínimo. El operador de igualdad puede ser cualquier operador lógico de comparación. Es importante que los dominios de  $A_j$  y  $B$  sean compatibles,  $D_j \sim D_B$ . Además, hay que poder demostrar previamente que la consulta anidada solo retornará un único valor, o ninguno. Veamos el típico error.

**ejemplo 6.35.** *Nombres de los socios con alguna cuota más alta que algún salario de algún trabajador.*

**solución**

```

SELECT nombres
FROM persona NATURAL JOIN hace
WHERE cuota > (
    SELECT salario_base
    FROM pagos
);

```

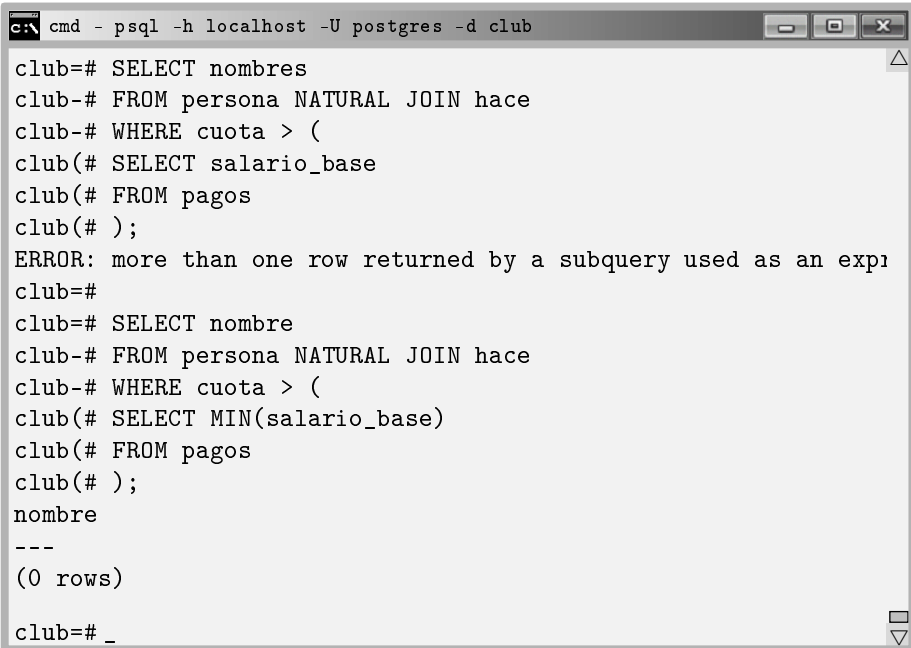
Esta solución al ejemplo 6.35 es incorrecta y causará un error por respuesta informando que la subconsulta usada como expresión, o sea como valor, retorna más de un registro.

En este caso particular, no hay que esforzarse demasiado para entender que si se pide que la cuota sea mayor algún salario, se puede solucionar fácilmente pidiendo por el mínimo salario en lugar de para todos los salarios. Eso hará que la subconsulta

retorne un solo valor, y por lo tanto la consulta sea correcta, así.

```
solución SELECT nombres
          FROM persona NATURAL JOIN hace
          WHERE cuota > (
                SELECT MIN(salario_base)
                FROM pagos
            );
```

En la Pantalla 6.45 se imprime la introducción del ejemplo 6.35, la primera aproximación que provoca un error, y la correcta que nos informa que no hay ningún socio que cumpla esta condición.



```
cmd - psql -h localhost -U postgres -d club
club=# SELECT nombres
club=# FROM persona NATURAL JOIN hace
club=# WHERE cuota > (
club(# SELECT salario_base
club(# FROM pagos
club(# );
ERROR: more than one row returned by a subquery used as an expression
club=#
club=# SELECT nombre
club=# FROM persona NATURAL JOIN hace
club=# WHERE cuota > (
club(# SELECT MIN(salario_base)
club(# FROM pagos
club(# );
nombre
---
(0 rows)
club=# _
```

Pantalla 6.45. *Uso de subconsultas como expresiones.*

Más allá de utilizar subconsultas como expresiones, el hecho de poderlas anidar en más niveles convierte este mecanismo en el que más capacidad expresiva tiene del SQL. Esta potencia viene reforzada por unos nuevos operadores lógicos, que retornan cierto o falso, a los cuales se les da un valor y una relación. Son operadores que nos responden cuestiones de pertenencia de elementos en conjuntos, o de existencia de valores en las relaciones. Se presentan a continuación.

**Cláusula IN**

La cláusula `IN` actúa como un operador lógico en notación infija, es decir, entre sus dos operandos. Se le pasa un valor y una relación con el esquema compatible con este valor. Retorna cierto si el valor pertenece a la relación.

```

SELECT A1, A2, ..., Ak
FROM r
WHERE Ai IN (
    SELECT Bj
    FROM s
);

```

Caja 6.42. *Cláusula lógica IN.*

Una intersección de consultas es fácilmente implementable con esta cláusula, como se ve en el ejemplo 6.36.

**ejemplo 6.36.** *Socios que practican fútbol y básquet.*

**solución**

```

SELECT pasaporte, nombres, apellidos
FROM hace NATURAL JOIN persona
WHERE deporte = 'fútbol'
AND pasaporte IN (SELECT pasaporte
    FROM hace
    WHERE deporte = 'básquet');

```

Anidando unas consultas dentro de otras se pueden resolver cuestiones que afecten a multitud de registros de tablas distintas. Esta forma de trabajar se basa en la operación de cambio de nombre.

El procedimiento mental para formular estas consultas más complicadas pasa por considerar que cada renombramiento que hacemos en alguna tabla es como crear una variable de tipo tupla con los campos correspondientes a los atributos de la tabla. Visto así, podemos disponer de tuplas para cualquier tabla de la base de datos que nos sirven para comparar y restringir lo que finalmente será la tupla resultante, que es la que aparece en la cláusula `SELECT` principal.

Como se puede comprobar en el ejemplo 6.37, con este método se pueden resolver consultas que entren por un extremo del modelo ER para acabar seleccionando registros de la otra punta. Se trata sin duda de las expresiones más complicadas.

**ejemplo 6.37.** *Personas conocidas por los trabajadores de Napo con salarios superiores a 1000.00.*

```
solución SELECT p.pasaporte,p.nombres,p.apellidos
FROM conoce c JOIN persona p
ON c.es_conocida = p.pasaporte
AND c.conoce IN (SELECT n.pasaporte
FROM pagos p
WHERE p.salario_base > 1000.00
AND p.pasaporte IN (SELECT r.pasaporte
FROM persona r NATURAL JOIN ciudad cc
WHERE cc.provincia = 'Napo'
)
);
```

La solución 6.37 contiene dos tuplas.

Cuidado con el concepto de contrario cuando se trabaja con conjuntos. La cláusula `IN` significa "pertenece". Y por tanto con la negación, `NOT IN` significa no pertenece.

### Cláusulas `SOME` y `ALL`

El término *some* se puede traducir por *algún*.

Un operador de comparación seguido de la cláusula `SOME` actúa como un operador lógico entre sus dos operandos, que como con la cláusula `IN` son: un valor y una relación (con el esquema compatible con este valor). Según el comparador, el valor dado, y los de la relación, retorna cierto o falso.

En lenguaje formal se puede definir la operación `SOME` de forma sencilla, como se presenta en la Caja 6.43, que utiliza el símbolo  $\star$  para indicar cualquier operador de comparación.

$$F \star \text{SOME}(r) \Leftrightarrow \exists t \in r \mid F \star t$$

Caja 6.43. *Definición de la cláusula `SOME` con el comparador genérico.*

siendo  $F$  una expresión,  $r$  una relación, y el símbolo estrella  $\star$  cualquier comparador del conjunto  $\star \in \{=, \neq, <, >, \leq, \geq\}$ .

En la Caja 6.44 se indica la manera de utilizar esta cláusula para el caso de la

igualdad, en SQL.

```

SELECT A1, A2, ..., Ak
FROM r
WHERE Ai = SOME (
    SELECT Bj
    FROM s
);

```

Caja 6.44. Cláusula lógica `SOME`.

El predicado de la cláusula `WHERE` de la Caja 6.44 retornará cierto si algún valor de la columna  $B_j$  de la relación  $s$  es igual al valor del atributo  $A_i$  de  $r$ . Es importante la presencia del operador de comparación. Igual podría ser menor, mayor, diferente, menor o igual, y mayor o igual.

En la Figura 6.8 se exponen cuatro ejemplos de evaluaciones con el correspondiente resultado.

$$\begin{array}{l}
 \left( 5 < \text{SOME} \begin{array}{|c|} \hline 6 \\ \hline 5 \\ \hline 0 \\ \hline \end{array} \right) \text{ cierto} \quad \left( 6 > \text{SOME} \begin{array}{|c|} \hline 6 \\ \hline 5 \\ \hline 0 \\ \hline \end{array} \right) \text{ falso} \\
 \left( 5 = \text{SOME} \begin{array}{|c|} \hline 6 \\ \hline 5 \\ \hline 0 \\ \hline \end{array} \right) \text{ cierto} \quad \left( 5 \neq \text{SOME} \begin{array}{|c|} \hline 6 \\ \hline 5 \\ \hline 0 \\ \hline \end{array} \right) \text{ cierto}
 \end{array}$$

Figura 6.8: Ejemplos de la cláusula `SOME`.

Así pues, `= SOME` es equivalente a `IN`. Pero en cambio, `≠ SOME` no es equivalente a `NOT IN`. Estamos en un terreno resbaladizo. Hay que diferenciar entre el contrario de la pertenencia y la pertenencia del contrario.

Filosóficamente, la cláusula `SOME` es en cierta forma media generalización de la cláusula `IN`. Media generalización en el sentido que la cláusula `IN` siempre pregunta si el valor es igual a alguno de la relación. En cambio con la cláusula `SOME`, se relaja el operador, y en lugar del igual podemos preguntar por si es distinto, o mayor, o menor, etcétera. En este sentido, es una generalización. No obstante, la versión negada de la cláusula `IN`, es decir, la cláusula `NOT IN` es imposible de construir a partir de la cláusula `SOME`. Por eso media generalización, ya que en positivo es más genérico, pero no negativo, no es reproducible.

Análogamente a la cláusula `SOME`, existe la cláusula `ALL`. Como es bien sabido, el



término *all* se puede traducir por *todos*. En la Caja 6.45 se muestra la definición formal de esta cláusula.

$$F \star \text{ALL}(r) \Leftrightarrow \forall t \in r \mid F \star t$$

Caja 6.45. *Definición de la cláusula ALL con el comparador genérico.*

La diferencia entre las dos operaciones, pues, estriba en el cuantificador existencial  $\exists$  de la cláusula *SOME* versus el cuantificador universal  $\forall$  de la cláusula *ALL*.

Se utiliza como en la Caja 6.46.

```
SELECT A1, A2, ..., Ak
FROM r
WHERE Ai <> ALL (
    SELECT Bj
    FROM s
);
```

Caja 6.46. *Cláusula lógica ALL.*

En SQL el comparador de desigualdad  $\neq$  se escribe como menor mayor,  $<>$ . El predicado de la cláusula *WHERE* de la Caja 6.46 retornará cierto si todos los valores de la columna  $B_j$  de la relación  $s$  son distintos al valor del atributo  $A_i$  de  $r$ .

El operador de desigualdad también puede ser cualquiera de comparación.

Otra vez se muestran cuatro evaluaciones en la Figura 6.9

$$\begin{aligned} \left( 5 < \text{ALL} \begin{array}{|c|} \hline 6 \\ \hline 5 \\ \hline 0 \\ \hline \end{array} \right) & \text{ falso} & \left( 6 > \text{ALL} \begin{array}{|c|} \hline 6 \\ \hline 5 \\ \hline 0 \\ \hline \end{array} \right) & \text{ falso} \\ \left( 4 = \text{ALL} \begin{array}{|c|} \hline 6 \\ \hline 5 \\ \hline 0 \\ \hline \end{array} \right) & \text{ falso} & \left( 5 \neq \text{ALL} \begin{array}{|c|} \hline 6 \\ \hline 4 \\ \hline 0 \\ \hline \end{array} \right) & \text{ cierto} \end{aligned}$$

Figura 6.9: *Ejemplos de la cláusula ALL.*

Se ve que  $\neq$  ALL es equivalente a NOT IN. Pero = ALL no es equivalente a IN.

Así como en la Caja 6.44 se ha usado la igualdad, aquí en la Caja 6.46 se utiliza la desigualdad por ser los operadores más habituales en las operaciones correspondientes. Los signos que se utilizan para los operadores de la estrella de la definición formal, en SQL son  $\{=, <>, <, >, <=, >=\}$ .

Observa que para el caso de la cláusula ALL se puede producir una situación absorbente, es decir, que el resultado no dependa del valor por el cual se está preguntando. Para el caso en que se utilice el operador de igualdad en una relación donde haya valores distintos la evaluación resultará negativa para cualquier valor que se le pase.

Y también desde un prisma más espiritual, en el fondo, la cláusula ALL es la otra mitad de la generalización de la cláusula IN.

## Cláusula EXISTS

El término *exists* se puede traducir por *existe*, en tercera persona del singular del presente simple.

Es lo mismo utilizar la cláusula EXISTS con una relación que preguntar si la relación no es vacía. La definición de esta cláusula en lenguaje formal se muestra en la Caja 6.47.

$$\text{EXISTS}(r) \Leftrightarrow r \neq \emptyset$$

Caja 6.47. Definición de la cláusula EXISTS.

Es decir, que esta es la cláusula que nos permite preguntar si el resultado de una expresión relacional es la relación vacía. Para usarla en un predicado de la cláusula WHERE hay que hacerlo como en la Caja 6.48.

```

SELECT A1, A2, ..., Ak
FROM r
WHERE EXISTS (
    SELECT Bj
    FROM s
);

```

Caja 6.48. *Cláusula lógica EXISTS.*

Es una cláusula especialmente útil para verificar la existencia de referencias a una clave primaria.

**ejemplo 6.38.** *Nombre de las ciudades de las que no haya ninguna persona.*

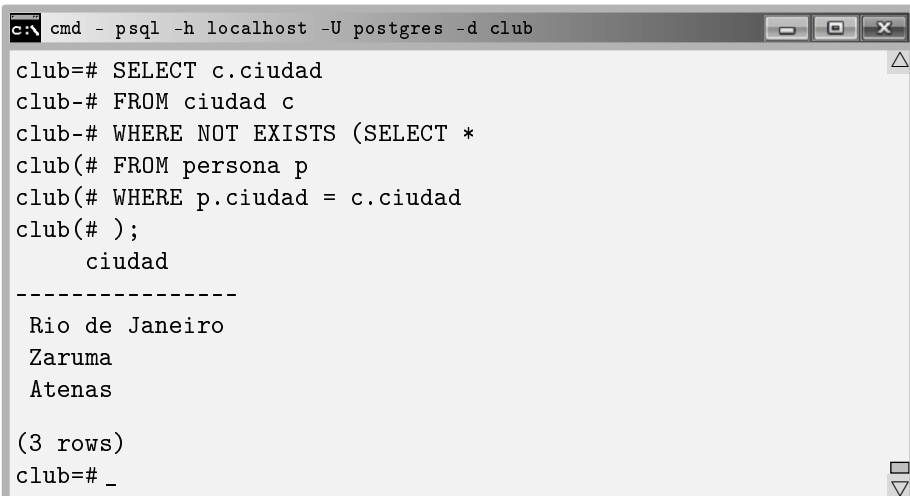
**solución**

```

SELECT c.ciudad
FROM ciudad c
WHERE NOT EXISTS (SELECT *
                  FROM persona p
                  WHERE p.ciudad = c.ciudad
                  );

```

La respuesta del ejemplo 6.38 se imprime en la Pantalla 6.46.



```

c:\cmd - psql -h localhost -U postgres -d club
club=# SELECT c.ciudad
club=# FROM ciudad c
club=# WHERE NOT EXISTS (SELECT *
club(# FROM persona p
club(# WHERE p.ciudad = c.ciudad
club(# );
      ciudad
-----
Rio de Janeiro
Zaruma
Atenas
(3 rows)
club=# _

```

Pantalla 6.46. *Ejemplo de uso de la cláusula EXISTS.*

Usándola con intersecciones o diferencias de relaciones se pueden responder cuestiones como la del ejemplo 6.39.

**ejemplo 6.39.** *Dar el nombre de los deportes que practican socios de todas las ciudades de la provincia del Napo.*

**solución**

```

SELECT DISTINCT h.deporte
FROM hace h
WHERE NOT EXISTS (
    SELECT ciudad
    FROM ciudad
    WHERE provincia = 'Napo'
EXCEPT
    SELECT p.ciudad
    FROM hace hh, persona p
    WHERE hh.pasaporte = p.pasaporte
    AND hh.deporte = h.deporte
);

```

Analícemos este último ejemplo. Con las dos primeras líneas pedimos los deportes que practica algún socio. Y además los nombramos con el prefijo *h*. Hay que retenerlo en mente. El prefijo *h* hace referencia a la tupla que formará parte de la relación resultante si satisface el predicado.

Entonces, con el **WHERE** establecemos que además para aparecer en la solución de la consulta cada uno de esos deportes debe cumplir una condición: que las ciudades de Napo sea un subconjunto de las ciudades de los socios que lo practican.

Hay que tener en cuenta que

$$X - Y = \emptyset \Rightarrow X \subseteq Y.$$

Y por tanto, se puede comprender fácilmente la solución 6.39.

Paso a paso. Los dos **SELECTS** internos computan los conjuntos

- *X* : Conjunto de ciudades de la provincia del Napo.
- *Y* : Conjunto de ciudades de socios que practican el deporte **h.deporte**.

El **NOT EXISTS** es el  $= \emptyset$ .

En palabras, cuando eliminando todos los elementos de un conjunto que estén en otro nos quedamos sin ningún elemento, es que el primer conjunto es un subconjunto del segundo. El hecho de que si de un conjunto de gatos quitamos todos los animales nos quedamos sin nada, significa que los gatos son un subconjunto de los animales. Por la misma razón, si quitando de la lista de ciudades de Napo las de los socios que practican el deporte tal se obtiene la lista vacía, entonces es que tal deporte se practica por socios de todas las ciudades de Napo.

---

*En este capítulo se ha visto primeramente como construir una base de datos a partir de un modelo relacional utilizando la creación de tablas con las correspondientes restricciones de integridad referencial. Después se ha hecho un recorrido a las estructuras básicas de las consultas relacionales, muchas de las cuales reflejan operaciones del álgebra relacional vistas en el Capítulo 5, y otras más sofisticadas propias de estar operando con multiconjuntos y no con conjuntos como en aquel capítulo. Todo ello, provee el material necesario para formular respuestas a las consultas que se puedan plantear.*



## Capítulo 7

# El Sistema Gestor de Bases de Datos PostgreSQL

El Sistema Gestor de Bases de Datos PostgreSQL [7], es el software de código abierto que más se ajusta a los estándares SQL. A diferencia del capítulo anterior donde todo el contenido era estándar del lenguaje estructurado de consultas, en este capítulo hacemos una inmersión en este SGBD en particular. Esto no significa que todo lo que se muestra aquí sea exclusivo de este sistema. Tan solo quiere decir que no se debe asumir que valga para cualquiera. Y a pesar de que las funcionalidades que se exponen se muestran mediante diferentes facetas en los otros sistemas, las maneras de hacerlo no dejan de ser singularidades de cada uno de ellos.

Este capítulo empieza introduciéndose en la sintaxis de las sinopsis del manual. Aprender a interpretar la gramática que en ellas se utiliza. Gran parte de este capítulo no es más que una selección de las instrucciones más representativas con las opciones más habituales que hay en la documentación de PostgreSQL, en [8]. Es muy recomendable familiarizarse con este tipo de sinopsis.

Luego, se hace una breve reestructuración del espacio para el proyecto del club deportivo, introducido en el capítulo anterior. Se deja todo listo para dejar la aplicación acabada.

Seguidamente se aborda la administración de usuarios y grupos de usuarios describiendo meticulosamente la gestión de permisos y privilegios. Esto introducirá un problema tan interesante como es el de la concurrencia en el acceso y modificación de los datos. Como resulta previsible, la concurrencia genera conflictos. En consecuencia, se abordará el concepto de transacción como unidad indivisible de acción modificadora de la base de datos.

Después se presenta una clasificación de los componentes lógicos de las bases de datos para profundizar entonces en los componentes de control, o funcionales.

Más allá del almacenamiento de datos un SGBD dispone de herramientas para gestionarlas que en el mejor de los casos contienen incluso un lenguaje de programación nativo, como en el caso del PostgreSQL. Con este lenguaje se debería ser capaz de dejar la aplicación para el club deportivo acabada, por lo que respecta al lado del servidor. Para esto se hace una breve incursión en el lenguaje procedimental del SGBD PostgreSQL, que se llama PL/pgSQL. Este nombre raro proviene de *procedural language PostgreSQL*. Se utilizará este lenguaje de programación sencillo para implementar funciones. Y luego, las funciones servirán para añadir disparadores.

Llegados a este punto, se da por concluida la adquisición de conocimiento relativo al diseño y la implementación de bases de datos y se abordan otros temas implicados. Esto incluye algunas herramientas básicas para al desarrollo de programas de interfaz que hagan de clientes, así como los detalles de la instalación, iniciación y parada del sistema, copias de seguridad, y algunos parámetros básicos de la configuración del servidor.

## 7.1 Documentación de PostgreSQL

La Figura 7.1 se trata de una captura de la parte inicial de la página web correspondiente a la instrucción `SELECT` de PostgreSQL.

### Name

`SELECT` -- retrieve rows from a table or view

### Synopsis

```
SELECT [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
* | expression [ AS output_name ] [, ...]
[ FROM from_item [, ...] ]
[ WHERE condition ]
[ GROUP BY expression [, ...] ]
[ HAVING condition [, ...] ]
[ { UNION | INTERSECT | EXCEPT } [ ALL ] select ]
[ ORDER BY expression [ ASC | DESC | USING operator ] [, ...] ]
[ LIMIT { count | ALL } ]
[ OFFSET start ]
[ FOR { UPDATE | SHARE } [ OF table_name [, ...] ] [ NOWAIT ] [...]
```

Figura 7.1: *Página web del manual.*

Hay que concentrarse. En la Figura 7.1, todo aquello que está fuera de corchetes, es obligatorio que aparezca en un `SELECT`. Es decir, lo que está entre corchetes es



optativo. La barra vertical significa disyunción exclusiva. O sea, para cada barra vertical se puede poner lo que tenga a la izquierda o lo que tenga a la derecha, pero no las dos cosas. Cuando hay más de dos opciones, entonces se utilizan llaves para definir el conjunto de posibilidades, y entre cada una de ellas hay una barra vertical. La secuencia `[ , ... ]` luego de una palabra en minúsculas, y por tanto en negrita y en cursiva, significa que aquella misma última cosa se puede repetir. Todos los símbolos están equilibrados en cada línea. Tantos corchetes o llaves se abren como se cierran. Para agilizar la comprensión de este lenguaje, podemos interpretar que donde dice *expresión* significa atributo. Aunque en rigor, también significa, por ejemplo, `2 * sueldo_base`.

Empecemos.

Para hacer un `SELECT` forzosamente hay que escribir la misma palabra `SELECT`, y un asterisco o una expresión. Nada más. Fíjate bien. Porque no se hace explícito de ningún manera, o sea no se dice en ningún sitio, que en el caso de optar por el asterisco, entonces es obligatoria alguna cosa más. Y en cambio, todo lo que sigue es optativo. Es decir, somos nosotros que tenemos que saberlo. Y por tanto, las sinopsis no lo dicen todo, pero sí que son muy útiles.

Interpretando lo que se dice en la Figura 7.1 pues, se ve que luego de la palabra `SELECT` se puede poner `ALL`, o bien `DISTINCT`. En este segundo caso, o sea si ponemos `DISTINCT`, entonces podemos añadir la cláusula `ON` seguido de un atributo respecto al cual se darán los diferentes valores en la solución. Por ejemplo, con la base de datos del club deportivo, la instrucción

```
SELECT DISTINCT ON (ciudad) nombres,ciudad FROM persona;
```

retorna una persona de cada ciudad. ¿Cuál? La primera que encuentra según el orden que se dé en la cláusula `ORDER BY`, que en este caso, como no se da, sería el orden por defecto, es decir, el orden de inserción que en su día se dio en la tabla.

Pero no salgamos del tema. El propósito de esta sección es comprender este tipo de sinopsis. Resulta materialmente imposible explicar todo lo que se puede hacer con un `SELECT` en una sola sección. Por tanto, agilizando, después se sigue con o bien un asterisco, o bien un atributo, y si optamos por este segundo caso, entonces podemos renombrar la columna con el operador `AS`, dando un *output\_name* que podemos intuir que significa un nombre cualquiera. Además, también en este segundo caso, podemos repetir la estructura añadiendo más atributos, cada uno de los cuales podemos renombrar, o no.

Después podemos poner `FROM`. En ninguna parte dice, tampoco que si no ponemos `FROM` está prohibido poner `WHERE`. El significado de *from\_item* se describe más abajo en la misma página web.

Seguidamente podemos poner la cláusula `WHERE` seguida de una condición que también podemos intuir que representa una expresión booleana.

A continuación hay la posibilidad de agrupar según algún atributo. No se indica nada respecto al tema de las funciones de agregación. Aún así, la persona que consulta el manual sabe que si se agrupa para algún atributo, es para agregar otros. En la Figura 7.1 también se explica que luego, a la agrupación se le pueden establecer condiciones con la cláusula `HAVING`, que acostumbra a ser utilizada si se ha utilizado el `GROUP BY`, aunque eso tampoco queda plasmado en la sinopsis.

En cambio, lo que sí que se muestra es la posibilidad de realizar operaciones lógicas entre relaciones con las cláusulas correspondientes. Después del `HAVING` podemos poner una de tres, `UNION`, `INTERSECT`, o bien `EXCEPT`, y continuar con otro `SELECT`, cosa que convierte la sinopsis en recursiva. Además, a cualquier opción del trilema se le puede añadir el operador `ALL` antes del `SELECT` anidado.

La siguiente cláusula que se ilustra es la de ordenación, con la disyuntiva de si es ascendente, descendente, o bien de otra forma que no trataremos.

Finalmente, la opción de limitar el número de filas de la relación resultante, o no, obteniendo todas las filas que resulten del cómputo. En esta cláusula, la opción `ALL`, es la que se da por defecto. Esto sirve para programas cliente que utilicen paginación de medida fija para las transferencias. Igualmente, la cláusula siguiente, `OFFSET` indica el primer registro que se desea obtener. Y de la última opción, tampoco haremos ningún comentario.

En fin, más o menos, sería interesante comprender bien estas sinopsis ya que son de gran ayuda a la hora de consultar los formatos de las instrucciones.

## 7.2 Estructura del Proyecto

El proyecto para el club deportivo ha sido introducido en la Sección 2.6. En el Capítulo 5 se ha mostrado la forma de transformar el diseño en un modelo relacional. Y cuando se ha presentado el DDL se ha implementado el modelo, y algunos archivos para disponer de ejemplos de inserciones.

Ha llegado la hora de engrandecer la jerarquía de escripts que estructuran el proyecto. Hasta ahora el escript principal era el mismo que el de la creación de las tablas. Sin embargo el archivo de gestión de usuarios debe estar a la misma altura en la estructura que el archivo de creación de tablas, lógicamente, ya que son conceptos independientes.

Por tanto, el objetivo de esta transformación es que estos dos archivos ocupen un mismo nivel en la jerarquía. Empezaremos dividiendo en cuatro partes el escript principal de ahora. Como eso va a provocar cierta proliferación de archivos, en adelante los enumeraremos por orden secuencial que indique cómo se han de importar desde el `psql` para crear la aplicación.

Empecemos pues fragmentando el contenido del actual `club.sql` en cuatro archivos. El principal, uno de inicialización, el de creación de tablas, y el de inserts.

En la Caja 7.1 se describe el fichero `1-init.sql` que tan solo contiene la creación.

```
\echo ----- 1-init.sql
\c postgres
DROP DATABASE club;
CREATE DATABASE club;
\c club
```

Caja 7.1. Archivo `1-init.sql`.

El archivo `2-tablas.sql` queda exclusivamente con la creación de las tablas, sin los inserts. Su contenido se detalla en la Caja 7.2.

```
\echo ----- 2-tablas.sql
\i 'provincia\\provincia.sql'
\i 'ciudad\\ciudad.sql'
\i 'persona\\persona.sql'
\i 'telefonos\\telefonos.sql'
\i 'conoce\\conoce.sql'
\i 'socio\\socio.sql'
\i 'trabajador\\trabajador.sql'
\i 'deporte\\deporte.sql'
\i 'hace\\hace.sql'
\i 'roles_de_pago\\roles_de_pago.sql'
```

Caja 7.2. Archivo `2-tablas.sql`.

Entonces, renombramos el archivo `inserts.sql` a `3-inserts.sql`, creamos un archivo de nombre `4-usuarios.sql`, y dejamos el principal como la Caja 7.3.

```

\set ON_ERROR_STOP on
\echo escript principal para la base de datos club
\echo -----

\i 1-init.sql
\i 2-tablas.sql
\i 3-inserts.sql
\i 4-usuarios.sql

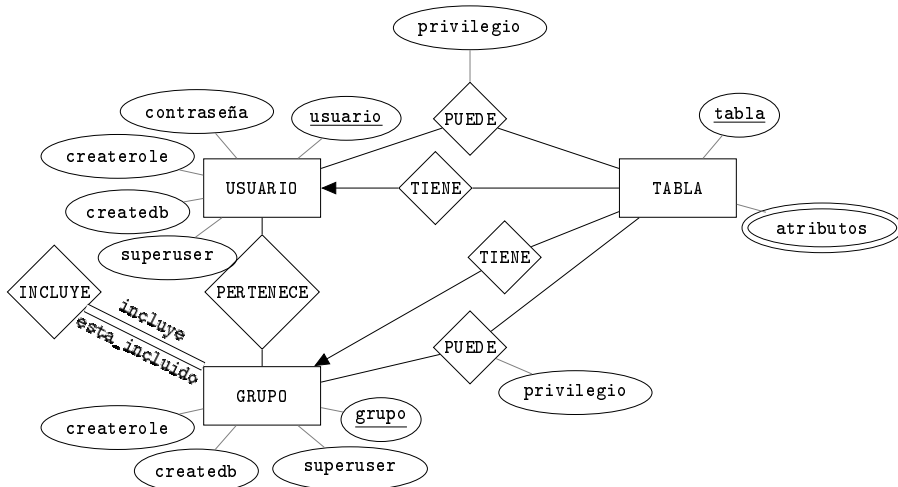
```

Caja 7.3. Archivo principal, club.sql.

## 7.3 Usuarios y Seguridad

Los usuarios no son exclusivos de una base de datos en concreto, sino de todas las bases de datos que se gestionen en un servidor.

Para una sola base, el Modelo 7.1 explica como se referencian los diferentes objetos implicados en la seguridad de la base de datos.



Modelo 7.1. Permisos y privilegios.

El Modelo 7.1 es un modelo grosero. Un croquis para hacernos una idea.

A lo largo de estas explicaciones, el término `TABLA` se utiliza para simplificar. Realmente significa objeto, que podemos entender como tabla, vista, función,... y más cosas.

En el Modelo 7.1, los tres permisos que tienen los usuarios y los grupos se pueden entender así como booleanos, aunque en un buen diseño se debería declarar un dominio. Tal como se muestra sirve para hacerse una idea de como va el tema de permisos y privilegios, no se trata de ninguna implementación. Es exclusivamente ilustrativo, utilizamos los modelos ER para expresar ideas.

La relación `TIENE` realmente en PostgreSQL se llama `owner`, que significa propietario. Cualquier tabla tiene un propietario que puede ser un usuario o un grupo.

La relación `PUEDE` realmente no tiene un nombre conocido.

Los valores del atributo `privilegio` son cadenas de caracteres, tal como se verá más adelante.

Notar que los permisos son atributos del usuario, y los privilegios, atributos de la relación M:N entre usuarios y tablas.

### 7.3.1 Permisos

En la Sección 7.9.3 se verá que la declaración de qué usuarios se pueden conectar a qué bases de datos se da en uno de los archivos de configuración.

Cualquier usuario tiene permisos para crear tablas, y otros objetos, en las bases de datos donde se pueda conectar. Esto lo convierte en el propietario de los objetos que crea, y le permite dar privilegios a otros usuarios para poder leer, modificar, y borrar.

Cada permiso se identifica con una palabra. Hay tres. El permiso de crear bases de datos, `CREATEDB`, el de crear usuarios y grupos, `CREATEROLE`, y el que puede hacerlo todo, `SUPERUSER`. Estan pseudo ordenados. Cada uno, en cierta manera, está incluido en el siguiente.

- `CREATEDB`. Este permiso permite a los usuarios la creación de nuevas bases de datos en el clúster.
- `CREATEROLE`. Con este permiso se pueden crear usuarios y grupos, pero no superusuarios. Tampoco se pueden crear bases de datos directamente. O sea, quien tiene permiso de `CREATEROLE` puede crear usuarios que tengan permiso de

`CREATEDB`, y conectarse posteriormente con ese otro usuario para crear bases de datos, pero no pueden crear bases de datos con la instrucción correspondiente.

- `SUPERUSER`, o `CREATEUSER`. Estos dos son el mismo, son sinónimos, y quién lo tiene es superusuario. La forma `CREATEUSER` está obsoleta, aunque sigue siendo aceptada. Quien tiene permiso de `SUPERUSER` también tiene el permiso `CREATEROLE`, y el `CREATEDB`. Normalmente tan solo hay un superusuario para cada base de datos. Los superusuarios son los únicos que tienen todos los privilegios para leer, modificar y borrar, todos los objetos de todo el mundo.

Seguidamente, echamos una ojeada a las instrucciones que sirven para crear bases de datos, usuarios y grupos. Es decir, las instrucciones que tan solo se pueden ejecutar según qué permisos se tengan.

Quién tiene el permiso `CREATEDB` puede crear bases de datos usando la orden de la Caja 7.4, conocida ya desde la Sección 6.3.2.

```
CREATE DATABASE club;
```

Caja 7.4. *Creación de bases de datos.*

Si se tiene el permiso `CREATEROLE`, una forma habitual de crear un usuario es con la instrucción que se muestra en la Caja 7.5.

```
CREATE USER pepito;
```

Caja 7.5. *Creación de usuarios.*

Con la instrucción de la Caja 7.5 se crea un usuario con código *pepito*. Los nombres de los usuarios en PostgreSQL empiezan con una letra, y no hay diferencia entre mayúsculas y minúsculas. Siempre se guardan en minúsculas.

Para dar permisos a los usuarios que se crean se puede hacer de dos formas.

- En el momento de la creación, añadiendo el nombre del permiso luego del nombre del usuario. Por ejemplo la orden `CREATE USER pepito CREATEDB` crea un usuario de nombre *pepito* con permiso de creación de bases de datos.

- Posteriormente a la creación con la instrucción `ALTER USER pepito CREATEDB`.

De estas dos mismas formas también se puede asignar la contraseña. Añadiendo `PASSWORD '123'` en la creación, por ejemplo, o también posteriormente con la instrucción `ALTER USER pepito PASSWORD '123'`. Cualquier usuario puede cambiarse su propio password de esta segunda manera, poniendo su propio código de usuario. O sea, el usuario pepito puede hacer `ALTER USER pepito PASSWORD '456'`. En principio, un usuario sin contraseña no puede entrar, y por tanto no tiene mucho sentido crearlo. En la Sección 7.9.3 veremos como podemos saltarnos esta restricción, y entrar sin dar contraseñas.


Los permisos se pueden quitar igual que se dan, prefijando el nombre del permiso con `NO`. Por ejemplo, `ALTER USER pepito NOCREATEDB` quita el permiso para crear bases de datos. Y también con los otros dos, `NOCREATEROLE`, y `NOSUPERUSER`.

Un usuario pues, no es más que un conjunto de permisos y de privilegios. Es previsible que varios usuarios tendrán los mismos permisos. Por ello aparece el concepto de grupo de usuarios. Con el permiso `CREATEROLE`, también se pueden crear grupos, tal como se indica en el ejemplo de la Caja 7.6, donde se crea un grupo de usuarios llamado `socios`.

```
CREATE GROUP socios;
```

Caja 7.6. *Creación de grupos de usuarios.*

Todos los usuarios que pertenezcan a un grupo tienen los permisos del grupo como mínimo, ya que para eso sirven los grupos. Si un usuario pertenece a varios grupos tiene los máximos permisos que se puedan derivar. Y luego, es posible que el mismo usuario, en particular, tenga más. En la Pantalla 7.1 se muestra la creación del usuario y el grupo de estas dos últimas cajas.



```
c:\N cmd - psql -h localhost -U postgres -d club
club=# CREATE USER pepito PASSWORD '123' CREATEDB;
CREATE ROLE
club=# CREATE GROUP socios;
CREATE ROLE
club=# _
```

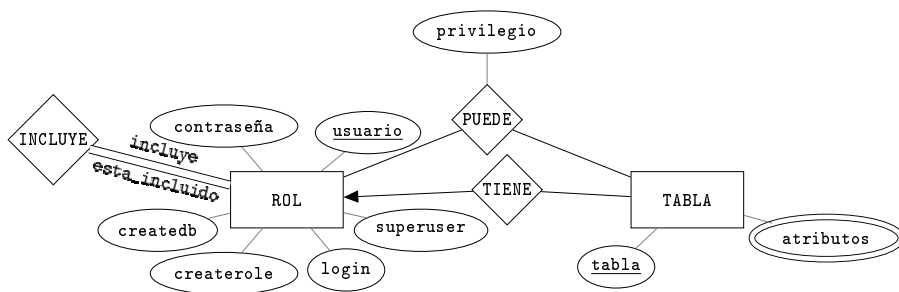
Pantalla 7.1. *Creación de usuarios y de grupos de usuarios.*

Lógicamente, a los grupos más que permisos se les acostumbra asignar privilegios sobre objetos.

Sorprende que PostgreSQL responda `CREATE ROLE` tanto para los usuarios como para los grupos cuando la instrucción ha sido exitosa. Veamos por qué.

O sea, reflexionemos. Un usuario se corresponde con unos permisos, y todos los usuarios de un grupo tienen los permisos que tenga asignados el grupo. Ergo, un usuario y un grupo es casi lo mismo. Con todo, irrumpe el concepto de rol. Cambiamos de filosofía.

En adelante, tanto los usuarios como los grupos son roles. Y entonces, nos inventamos un permiso que sea `LOGIN`. De forma que lo que hasta ahora considerábamos usuarios, a partir de ahora serán roles con permiso de `LOGIN`, y en cambio lo que llamábamos grupos, serán roles sin ese permiso. Todo esto impacta fuertemente en el Modelo 7.1, que queda transformado en el Modelo 7.2.



Modelo 7.2. *Permisos y Privilegios.*

Y de hecho, los comandos para crear usuarios y grupos de las Cajas 7.5 y 7.6 están obsoletos aunque aún aceptados. Por ese motivo, son preferibles las versiones de la Caja 7.7.

```
CREATE ROLE pepito LOGIN;
CREATE ROLE socios;
```

Caja 7.7. *Versiones más actuales para la creación de usuarios y grupos.*

Como se ha dicho más arriba, los roles pueden ser propietarios de objetos. Si el rol es un usuario, en el momento en que crea una tabla se convierte en propietario de esa tabla. Sin embargo, también puede ocurrir que alguien con permiso de `CREATEROLE`



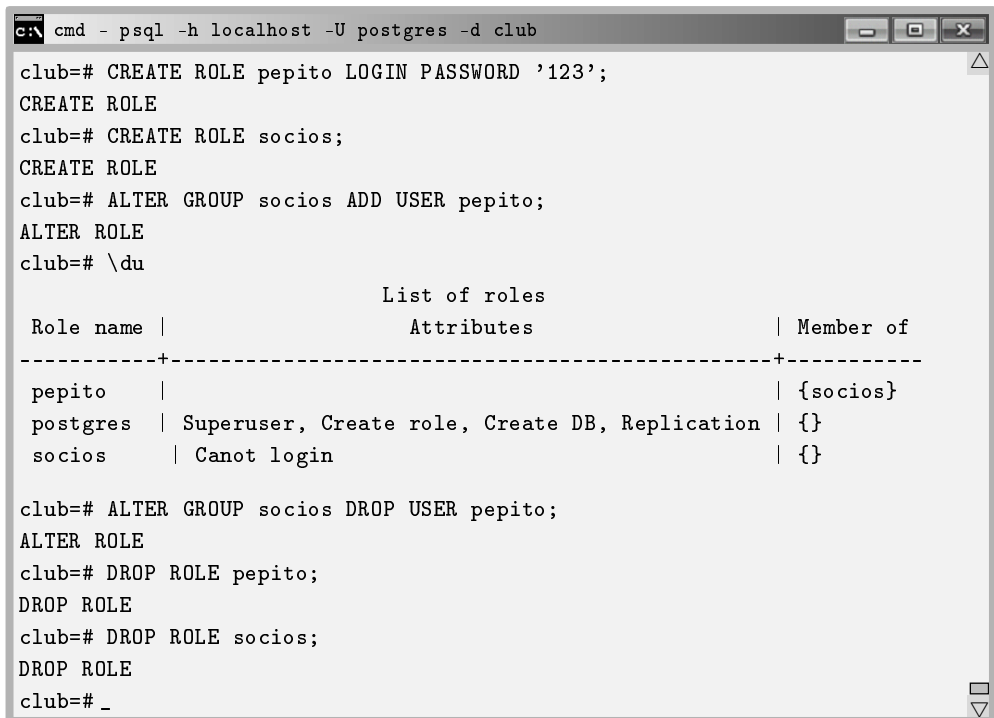
asigne un propietario a una tabla, haciendo `ALTER TABLE tabla OWNER TO pepito`. Esta segunda manera es la única posible si el rol es un grupo de usuarios.

Cuando algún usuario es el propietario de una tabla, o de cualquier otro objeto, puede dar privilegios de lectura, modificación o eliminación a otros usuarios. Cuando un grupo es propietario de algún objeto, entonces todos los miembros del grupo se pueden comportar como propietarios del objeto, es decir, pueden dar privilegios a otros usuarios.

Bien, los conceptos de usuario y grupo se confunden en el de rol. Aún así, para añadir y eliminar usuarios de grupos hay el comando `ALTER GROUP`, al cual se le da el nombre del grupo, `ADD` o `DROP`, y el nombre del usuario, o sea

```
ALTER GROUP socios {ADD | DROP} USER pepito;
```

En la Pantalla 7.2 se muestra un diálogo de ejemplo.



```
c:\> cmd - psql -h localhost -U postgres -d club
club=# CREATE ROLE pepito LOGIN PASSWORD '123';
CREATE ROLE
club=# CREATE ROLE socios;
CREATE ROLE
club=# ALTER GROUP socios ADD USER pepito;
ALTER ROLE
club=# \du
                                List of roles
-----+-----+-----
Role name |          Attributes          | Member of
-----+-----+-----
pepito    |                               | {socios}
postgres  | Superuser, Create role, Create DB, Replication | {}
socios    | Canot login                  | {}

club=# ALTER GROUP socios DROP USER pepito;
ALTER ROLE
club=# DROP ROLE pepito;
DROP ROLE
club=# DROP ROLE socios;
DROP ROLE
club=# _
```

Pantalla 7.2. *Inserción, modificación, y consulta de los usuarios y grupos del clúster.*

Esto no se puede hacer con roles. No se puede añadir un rol a otro, ni un usuario

a un grupo haciendo `ALTER ROLE socios ADD USER pepito`. No funciona. Y eso es así porque los roles como usuarios son indiferentes de los roles como grupos por lo que respecta a los permisos, no a la jerarquía. Un usuario pertenece a un grupo, y un grupo puede pertenecer a otro grupo, pero no puede ser que un grupo pertenezca a un usuario. En definitiva, los usuarios son las hojas del árbol, como los ficheros y las carpetas en el sistema de archivos del sistema operativo. Para borrar un usuario o un grupo se utiliza el comando `DROP ROLE` seguido de su nombre.

Primero se crea un usuario llamado `pepito`, y un grupo llamado `socios`. Después se añade el usuario `pepito` al grupo `socios` y se consulta la información de los usuarios con un comando `psql`. Hecho esto, se quita el usuario del grupo, y se eliminan usuario y grupo.

Algunas consideraciones respecto a la Pantalla 7.2.

- El comando `\du` de `psql` sirve para obtener la información de los roles existentes.
- PostgreSQL llama atributos de usuario a los permisos. Observa que el rol `socios` no tiene permiso de `LOGIN`, cosa que hace que como rol sea un grupo.
- La tercera columna lista la colección de grupos a los que pertenece cada usuario.
- Hay dependencias de existencia entre los elementos. No se puede borrar un grupo si contiene algún usuario.
- El permiso de `REPLICATION` tiene que ver con copias de seguridad. Se hablará con más detalle en la Sección 7.9.2.

## Aportaciones al proyecto del club deportivo

Para la base de datos del club deportivo, creamos en este momento un superusuario que llamamos `club`. Esto no significa que dejemos de conectarnos con el usuario `postgres`. En todo momento, el usuario que ejecutará el script será `postgres`. Esto servirá a la larga, cuando se dé por válida la aplicación. En la Sección 7.9.3 limitaremos el acceso del superusuario `club` a la base de datos `club` del servidor exclusivamente.

Empecemos de una el archivo `4-usuarios.sql` con el que se muestra la Caja 7.8. Además de la creación del superusuario, añadimos la creación de los grupos que finalmente deben estar presentes a la aplicación.

```
DROP ROLE IF EXISTS club;  
CREATE ROLE club LOGIN CREATEROLE PASSWORD '1';  
  
DROP ROLE IF EXISTS socios;  
CREATE ROLE socios;  
DROP ROLE IF EXISTS comerciales;  
CREATE ROLE comerciales;  
DROP ROLE IF EXISTS entrenadores;  
CREATE ROLE entrenadores;  
DROP ROLE IF EXISTS administrativos;  
CREATE ROLE administrativos;
```

Caja 7.8. *Archivo usuarios.sql.*

### 7.3.2 Privilegios

Los usuarios de una base de datos tienen privilegios para actuar en las tablas. Pueden tener privilegio de lectura, de inserción, de modificación y de borrado. Hay más. Cualquier propietario puede dar privilegios a otros usuarios, o a un grupo de usuarios, para poder leer sus tablas. Eso con la instrucción de la Caja 7.9.

```
GRANT SELECT ON tabla TO pepito;
```

Caja 7.9. *Garantizar privilegios.*

Para poder utilizar el comando de la Caja 7.9 hay que o bien ser el propietario de la tabla, o bien ser superusuario. En lugar de `SELECT`, el permiso podría ser con `INSERT`, `UPDATE` o `DELETE`. Para poner varios privilegios hay que separarlos por comas, tipo `GRANT SELECT, DELETE ON tabla TO pepito`. Con `ALL` decimos todos, `GRANT ALL PRIVILEGES ON...`. Y también se pueden asignar privilegios a columnas concretas de tablas. `GRANT SELECT (pasaporte) ON TABLE persona TO pepito`. Si se desea dar el privilegio a todo el mundo, `GRANT SELECT ON tabla TO PUBLIC`.

Para retirar los privilegios se debe hacer tal como se indica en la Caja 7.10.

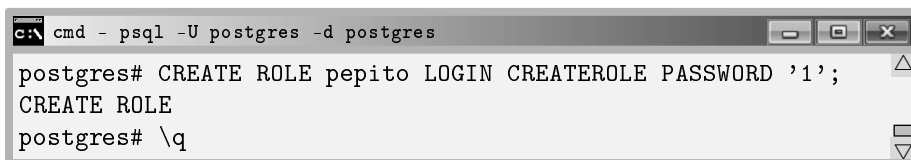
```
REVOKE SELECT ON tabla FROM pepito;
```

Caja 7.10. *Revocar privilegios.*

Y como antes, con las posibilidades INSERT, UPDATE, DELETE... y más que no veremos.

Para poder consultar los privilegios que hay asociados a una tabla en `psql` hay el comando `\dp`. Como siempre, si le damos una tabla nos dirá los privilegios que tenga asociados. Y si no, nos lo dirá de todas las tablas. Pero desgranemos el caso de una tabla con un experimento de cuatro pasos.

1. El usuario `postgres` conectado a la base de datos `postgres` crea el usuario `pepito`, con contraseña `'1'` y permiso `CREATEROLE`. Y se desconecta.

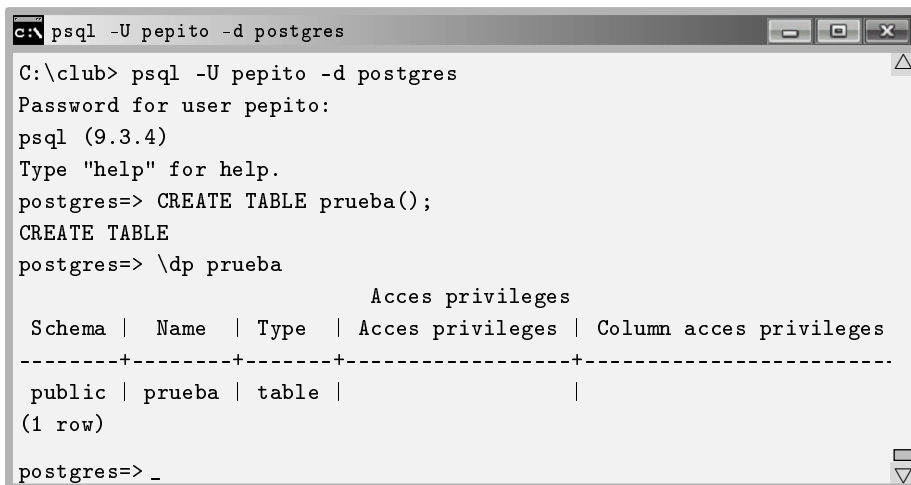


```

c:\> cmd - psql -U postgres -d postgres
postgres# CREATE ROLE pepito LOGIN CREATEROLE PASSWORD '1';
CREATE ROLE
postgres# \q
  
```

Pantalla 7.3. *Creación de un usuario con derecho a crear usuarios*

2. El usuario `pepito` se conecta a la base de datos `postgres`, crea una tabla vacía a la que llama `prueba`, y consulta los privilegios con `\dp prueba`.



```

c:\> psql -U pepito -d postgres
C:\club> psql -U pepito -d postgres
Password for user pepito:
psql (9.3.4)
Type "help" for help.
postgres=> CREATE TABLE prueba();
CREATE TABLE
postgres=> \dp prueba
                Acces privileges
Schema | Name  | Type  | Acces privileges | Column acces privileges
-----+-----+-----+-----+-----
public | prueba | table |                   |
(1 row)
postgres=> _
  
```

Pantalla 7.4. *Consulta de los privilegios de una tabla recién creada.*

3. Crea el usuario `pepita`, y le da permiso para leer la tabla `prueba`. Entonces vuelve a consultar los privilegios de la tabla.

```

c:\> psql -U pepito -d postgres

postgres=> CREATE ROLE pepita LOGIN PASSWORD '1';
CREATE ROLE
postgres=> GRANT SELECT ON prueba TO pepita;
GRANT
postgres=> \dp prueba

          Acces privileges
Schema | Name | Type | Acces privileges | Column acces privi
-----+-----+-----+-----+-----
public | prueba | table | pepito=arwdDxt/pepito+ |
      |      |      | pepita=r/pepito      |
(1 row)

postgres=> _

```

Pantalla 7.5. *Asignación y consulta de privilegios.*

4. Finalmente, revoca el permiso dado volviendo al estado 2, y por tercera vez observa los privilegios.

```

c:\> psql -U pepito -d postgres

postgres=> REVOKE SELECT ON prueba FROM pepita;
REVOKE
postgres=> \dp prueba

          Acces privileges
Schema | Name | Type | Acces privileges | Column acces privi
-----+-----+-----+-----+-----
public | prueba | table | pepito=arwdDxt/pepito |
(1 row)

postgres=> _

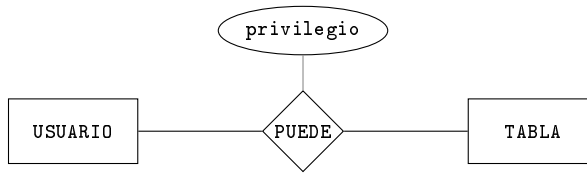
```

Pantalla 7.6. *Gestión de privilegios en una tabla.*

Para el análisis de este experimento, primero conviene conocer los valores que puede tomar el atributo privilegios que aparecía en el Modelo 7.2. En el Modelo 7.3 retomamos el fragmento implicado en el experimento.

Modelo 7.3. *Fragmento superior del Modelo 7.2.*

Este atributo es de tipo TEXT, y contiene una secuencia de letras cada una de las cuales tiene sentido por sí misma. Para el propietario de una tabla, el valor por



defecto del atributo privilegio es

```
privilegio = 'arwdDxt'
```

La traducción de lo que eso significa no es nada mnemotécnica debido a que en lugar de basarse en las palabras del SQL para tomar las iniciales, se basa en los permisos que se utilizan en los sistemas de archivos de los sistemas operativos linux. Para los profanos en la materia, seguidamente se indica la semántica de estas cadenas de caracteres.

- a: Significa append, que es `INSERT`.
- r: Significa read, que es `SELECT`.
- w: Significa write, que es `UPDATE`.
- d: Significa delete, que es `DELETE`.
- D: Significa truncate, que es `TRUNCATE`. Este permiso permite borrar todos los registros de una tabla en una sola transacción.
- x: Significa que el usuario puede referenciar la tabla. O sea, ejecutar consultas que hagan joins con ella.
- t: Permite ejecutar los disparadores asociados a la tabla.

Este valor es el que se asigna cuando se hace un `GRANT ALL PRIVILEGES...`

Lo que puede resultar desconcertante del experimento es sin duda que los resultados de los pasos segundo y cuarto sean distintos. Es decir cuando se acaba de crear, la columna `Acces privileges` está vacía. Y luego de dar el permiso a un tercer usuario, y quitárselo, ya no. O sea, que no se puede recuperar la estado inicial.

Esto es fruto de un comportamiento extraño. La convención es la siguiente. Debido a que estadísticamente la mayor parte de las tablas de las bases de datos mantienen los privilegios por defecto de la creación, convenimos en que un valor nulo en la columna de privilegios de acceso, de la vista de privilegios, significa que el propietario tiene los privilegios `'arwdDxt'` concedidos por él mismo. Así, mientras no haya ninguna modificación de los privilegios, nos habremos ahorrado la tarea de la asignación. Ahora

bien, una vez se ha hecho alguna concesión de privilegios sobre la tabla, entonces sí que se establece el valor real para esa columna.

Por otra parte, observa que es deducible que lo que nos muestra el comando `\dp` debe ser una vista, ya que si fuera una tabla no respetaría ni siquiera la primera forma normal que dice que el número de columnas de una tabla debe ser un número fijo. Y la columna `Access privileges` de las pantallas anteriores no solamente muestra un valor, sino un conjunto de valores separados con el signo `'+'`.

Analicemos meticulosamente el contenido de la columna `Access privileges` de la Pantalla 7.5, que se muestra nuevamente en la Figura 7.2.

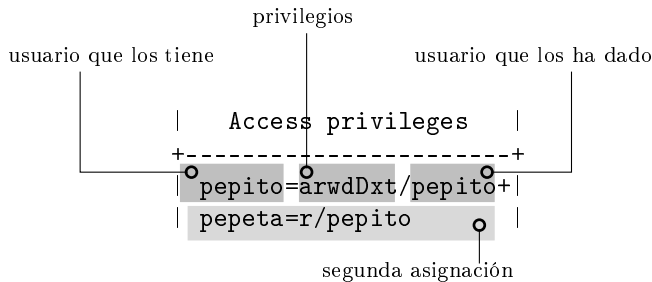


Figura 7.2: Columna de privilegios de acceso a una tabla

El comando `\dp prueba` de la Pantalla 7.5 ha retornado una sola fila correspondiente lógicamente a la tabla `prueba`. Aún así, esta fila tiene una columna, la de privilegios de acceso, que tiene más de un valor, y por tanto, se representa en una celda múltilínea. Pues bien, cada una de estas líneas interiores a la celda representa una asignación de privilegios sobre la tabla `prueba` a algún usuario o rol. Estas asignaciones tienen un formato trivial

```
usuario_que_los_tiene = privilegios_dados / usuario_que_se_los_ha_dado
```

y por eso en la primera línea de la columna de la Figura 7.2 el donador del privilegio acostumbra a corresponder con el receptor, que es el propietario. También podría ser que luego de la barra apareciera algún superusuario. Guardar quién ha dado los permisos a quién sirve para poder revocarlos en cascada.

En la misma Figura 7.2 podemos observar que en su segunda línea se indica que la usuaria `pepita` tiene el permiso de lectura, o sea `SELECT` sobre esta tabla, `prueba`. Y además, que quien le dio este permiso en su momento fue el usuario `pepito`. El signo `'+'` sirve para separar asignaciones.

Como se ha mencionado al inicio de esta sección, recuérdese que toda la explicación anterior ha sido simplificada para facilitar la comprensión de la forma de asignar permisos y privilegios. De todas, la simplificación más notoria es la de haber utilizado el concepto de tabla como objeto de la base de datos. Hay que entender pues, que

en lugar de sobre una tabla, los privilegios se pueden establecer sobre vistas, funciones, bases de datos, esquemas, y más cosas. Como consecuencia, los valores del atributo privilegio mostrados en esta simplificación son exclusivos de los privilegios definidos sobre las tablas. Es decir, si se estuviera hablando de asignación de privilegios sobre una función, por ejemplo, entonces la 'x' de los privilegios de la Figura 7.2 significarían privilegio de ejecución, y no de referencia como es el caso.

### 7.3.3 Concurrencia y Transacciones

Llamamos *transacción* a una secuencia de instrucciones que forman una acción indivisible en la base de datos.

Este concepto entra en escena en este momento porque previsiblemente, con el aumento del número de usuarios actuando en la base de datos surgirán problemas de concurrencia. De todas formas, el concepto de transacción afecta a aspectos que no tienen nada que ver con trabajar en un entorno multiusuario. Como se explica a continuación.

Cuando hacemos un `INSERT`, estamos ejecutando una transacción. La diferencia entre hacer dos inserts de una sola tupla constante, o hacer un solo insert con dos tuplas, es que en el primer caso, si falla el segundo insert (por ejemplo por clave primaria repetida), entonces el primero quedará hecho igualmente. En cambio si se trata de un solo insert con dos tuplas, es decir una sola transacción, entonces si fallase el segundo, el primero tampoco se produciría. De la misma manera que con las inserciones, con las modificaciones y las eliminaciones.

En esta sección observaremos problemas de concurrencia y veremos como implementar transacciones que hagan lo que puedan para resolverlos.

Como ejemplo de transacción, una transferencia de dinero entre dos cuentas corrientes. Si no se puede hacer completamente, que no se haga nada. O sea, que se tire marcha atrás lo que se ha empezado a hacer. Esto es una transacción. Una de dos. O se realiza del todo, o la base queda exactamente como estaba antes de comenzarla. Para ello, supongamos que tenemos una relación `cuenta = cuenta(titular, saldo)` con dos tuplas. Pepita tiene \$2000 y pepito \$1000. Se trata de hacer una transferencia de \$50 que pepito manda a pepita.

```
UPDATE cuenta SET saldo = saldo - 50 WHERE titular = 'pepito';
UPDATE cuenta SET saldo = saldo + 50 WHERE titular = 'pepita';
```

Caja 7.11. *Dos transacciones en dos comandos.*



En PostgreSQL, y más en general en SQL, se puede desgranar la transacción con la secuencia de instrucciones de la Caja 7.11. Pero... cuidado! Si hubiera un terremoto, o cualquier otra cosa que pueda pasar, en mitad de la ejecución de estas dos instrucciones los cincuenta dólares habrían desaparecido, ya que pepito los perdería, pero pepita, debido al terremoto, no los habría cobrado. Y eso no puede ser. Para resolver este tipo de problemas podemos transformar las dos transacciones en una sola, como en la Caja 7.12. Entonces, seguiríamos sin haber podido hacer la transferencia, pero al menos pepito recuperaría los \$50.

```
BEGIN;
UPDATE cuenta SET saldo = saldo - 50 WHERE titular = 'pepito';
UPDATE cuenta SET saldo = saldo + 50 WHERE titular = 'pepita';
COMMIT WORK;
```

Caja 7.12. *Transacción de dos comandos.*

La instrucción `BEGIN` sirve para comenzar una transacción. Internamente, conviene imaginarlo como que crea una copia de la base de datos que es la que verá el usuario que haya ejecutado el `BEGIN`. Entonces entra en un estado de prueba, como si estuviera jugando. Mientras dure este estado desde su perspectiva la base debe mantener la coherencia que tendría si no hubiera hecho el `BEGIN`. Puede hacer inserciones, modificaciones, y eliminaciones en las tablas que desee, que todo ello no dejará de ser un espejismo mientras no concluya la transacción.

La transacción se puede acabar bien, con el comando `COMMIT WORK`, que también se puede decir `COMMIT` a secas. Si acaba bien, los cambios realizados por el usuario serán visibles desde cualquier otra conexión a la base de datos.

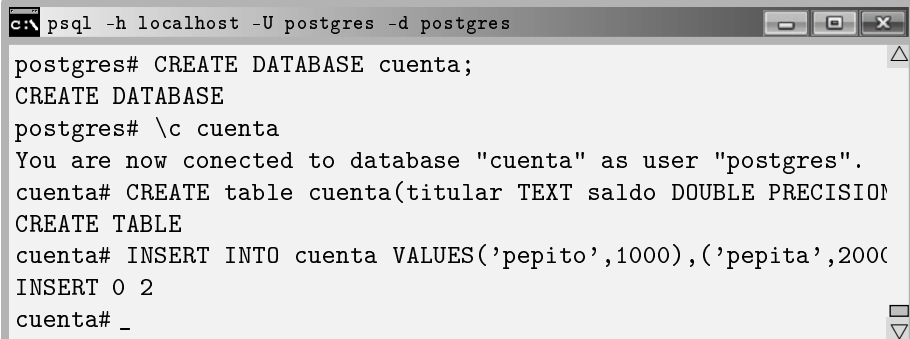
O puede acabar mal. Arrepintiéndose. Eso se hace con el comando `ROLLBACK WORK`, y entonces, tanto si ha hecho inserciones como borrados o lo que sea, el contenido de la base de datos se restaurará al momento inmediatamente anterior a haber empezado la transacción. O sea, antes del `BEGIN`.

En los dos casos se dice que la transacción ha acabado. Si es con `COMMIT`, se dice que ha acabado exitosamente. Si es con `ROLLBACK` se dice que la transacción ha sido abortada.

Hacemos un experimento de ocho pasos.

1. Como usuario `postgres`, o sea como superusuario, creamos la base de datos `cuenta`, en la cual creamos una tabla `cuenta(titular,saldo)`. Para este ex-

perimento no son necesarias claves primarias. En la tabla añadimos las dos tuplas. Esto se hace con las instrucciones que se muestran a la Pantalla 7.7.



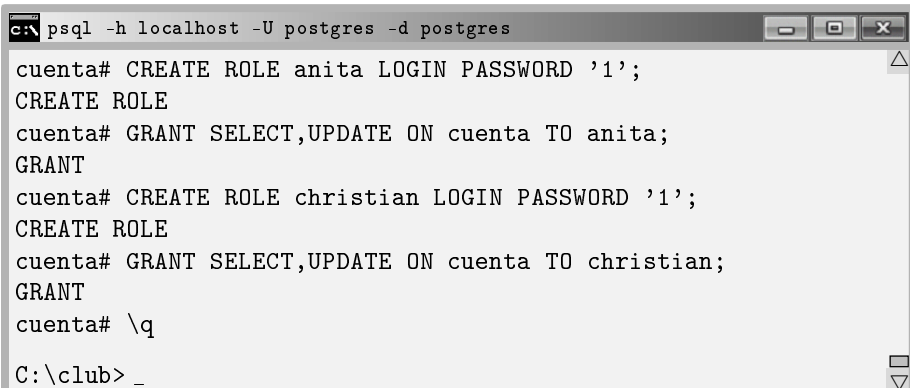
```

c:\n psql -h localhost -U postgres -d postgres
postgres# CREATE DATABASE cuenta;
CREATE DATABASE
postgres# \c cuenta
You are now conected to database "cuenta" as user "postgres".
cuenta# CREATE table cuenta(titular TEXT saldo DOUBLE PRECISION
CREATE TABLE
cuenta# INSERT INTO cuenta VALUES('pepito',1000),('pepita',2000)
INSERT 0 2
cuenta# _

```

Pantalla 7.7. *Creación de la tabla para el experimento de concurrencia*

2. Creamos dos usuarios que se llamen **anita** y **christian** con permisos para leer y modificar la tabla **cuenta**. Y nos desconectamos.



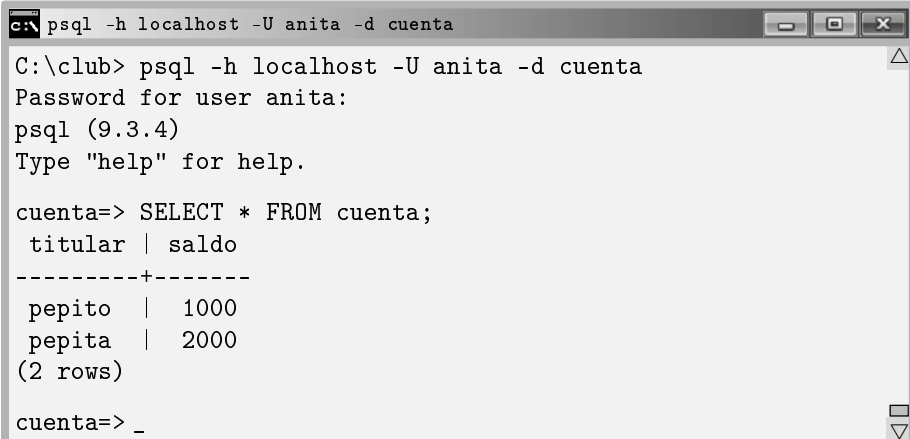
```

c:\n psql -h localhost -U postgres -d postgres
cuenta# CREATE ROLE anita LOGIN PASSWORD '1';
CREATE ROLE
cuenta# GRANT SELECT,UPDATE ON cuenta TO anita;
GRANT
cuenta# CREATE ROLE christian LOGIN PASSWORD '1';
CREATE ROLE
cuenta# GRANT SELECT,UPDATE ON cuenta TO christian;
GRANT
cuenta# \q
C:\club> _

```

Pantalla 7.8. *Creación de dos usuarios, y asignación de privilegios*

3. Nos conectamos a la base de datos **cuenta** con el usuario **anita**, y consultamos el contenido de la tabla **cuenta**.



```

c:\club> psql -h localhost -U anita -d cuenta
C:\club> psql -h localhost -U anita -d cuenta
Password for user anita:
psql (9.3.4)
Type "help" for help.

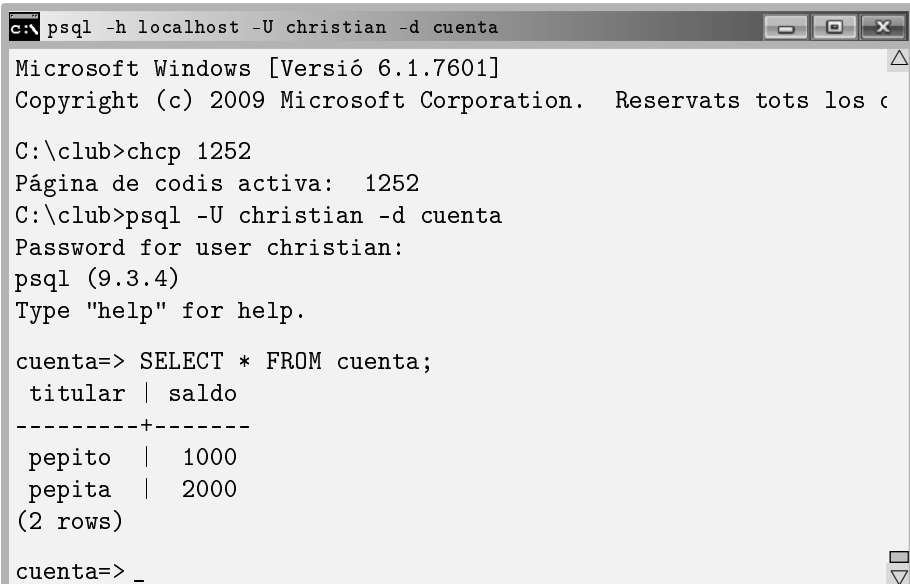
cuenta=> SELECT * FROM cuenta;
 titular | saldo
-----+-----
 pepito  | 1000
 pepita  | 2000
(2 rows)

cuenta=> _

```

Pantalla 7.9. *La usuaria anita consulta la tabla cuenta.*

4. Abrimos otra terminal, nos conectamos con el usuario `christian`, y también consultamos la tabla `cuenta`. En este momento el contenido es igual en las dos terminales.



```

c:\club> psql -h localhost -U christian -d cuenta
Microsoft Windows [Versió 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. Reservats tots los c

C:\club>chcp 1252
Página de codis activa: 1252
C:\club>psql -U christian -d cuenta
Password for user christian:
psql (9.3.4)
Type "help" for help.

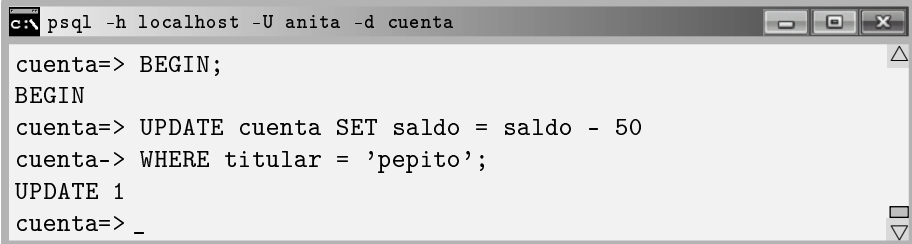
cuenta=> SELECT * FROM cuenta;
 titular | saldo
-----+-----
 pepito  | 1000
 pepita  | 2000
(2 rows)

cuenta=> _

```

Pantalla 7.10. *El usuario christian consulta la tabla cuenta.*

5. Anita inicia la transferencia quitando \$50 de la cuenta de pepito.



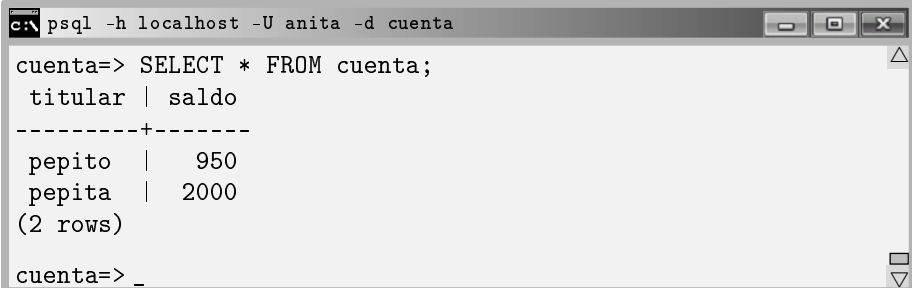
```

c:\N psql -h localhost -U anita -d cuenta
cuenta=> BEGIN;
BEGIN
cuenta=> UPDATE cuenta SET saldo = saldo - 50
cuenta-> WHERE titular = 'pepito';
UPDATE 1
cuenta=> _

```

Pantalla 7.11. *La usuaria anita inicia la transferencia de \$50.*

- En este momento la visión de la base de datos según **christian** es diferente de la visión que tiene **anita**. De hecho, si **christian** consultase la tabla **cuenta** obtendría el mismo resultado que en la Pantalla 7.10. Es decir, según **christian**, pepito tiene \$1000. Observamos como según **anita**, tiene \$950.



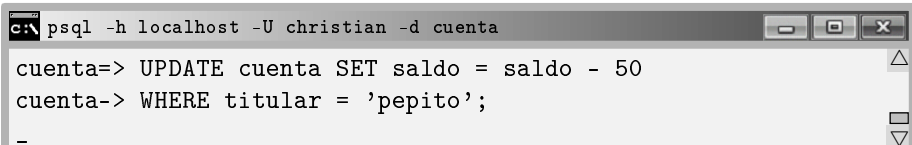
```

c:\N psql -h localhost -U anita -d cuenta
cuenta=> SELECT * FROM cuenta;
titular | saldo
-----+-----
pepito  |    950
pepita  |   2000
(2 rows)
cuenta=> _

```

Pantalla 7.12. *La usuaria anita consulta la tabla cuenta.*

- La incoherencia va más allá hasta llegar al absurdo. O sea, que PostgreSQL se queda colgado. Eso pasaría si en este momento **christian** pretendiera modificar el saldo de pepito.



```

c:\N psql -h localhost -U christian -d cuenta
cuenta=> UPDATE cuenta SET saldo = saldo - 50
cuenta-> WHERE titular = 'pepito';
_

```

Pantalla 7.13. *El usuario christian intenta modificar el saldo de pepito.*

- Observa que en la Pantalla 7.13 el **psql** no muestra el prompt. La terminal está colgada. De hecho, está esperando que **anita** se decida, y seguirá así hasta que haga una de dos.

- O bien acaba la transacción exitosamente con el comando `COMMIT`, dejando un saldo de \$900.
- O bien aborta la transacción, con el comando `ROLLBACK`, dejando un saldo de \$950.

De manera que la actualización hecha en la Pantalla 7.13 se hace en cualquier caso, ya que `christian` no ha iniciado transacción y por tanto, su comando es ejecutado siempre que sea posible. Imagina por ejemplo que `anita` luego del `BEGIN` hubiera eliminado el usuario `pepito`. Entonces al hacer `COMMIT` en la terminal de `anita`, en la terminal de `christian` hubiera aparecido la respuesta `UPDATE 0`.

La conclusión de este experimento es que la cosa no es sencilla. Aquí solamente ha habido un usuario iniciando una transacción, si el otro también hubiera iniciado otra, e hicieran cosas incompatibles, y acabarían abortando o comprometiendo..., todo ello se podría complicar bastante. Y de hecho, este problema aparece en la vida real cuando se hace un overbooking de entradas para una obra de teatro, o para los pasajes de un vuelo.

Que un simple `INSERT`, `UPDATE`, o `DELETE` sea por sí mismo una transacción es debido a la variable global `AUTOCOMMIT` de PostgreSQL. Es booleana, y por defecto vale cierto.

## 7.4 Componentes Lógicos de una Base de Datos

En rigor, entendemos por componentes lógicos todos aquellos objetos registrados con su identificador de objeto, `OID`, en el catálogo de cada una de las bases de datos que aloja un SGBD. Los componentes lógicos de una base de datos son todo aquello con lo que puede interactuar quien la implementa. Se agrupan en dos vertientes. Como siempre, el espacio y el tiempo. Distinguimos entre componentes de datos y componentes de control.

### 7.4.1 Componentes de Datos

Los componentes de datos son los dominios, las tablas, las vistas, y los índices. Los comandos para tratar con este tipo de componentes son los más estándar en todos los dialectos de SQL, y por eso ya han sido descritos en el Capítulo 6. De hecho, se han visto los componentes de datos involucrados directamente con la creación de la base, que son todos excepto los objetos de tipo índice, de los que no se ha hablado, y por eso aquí se hace una referencia.

## Atributos indexados

Un índice es un concepto asociado a un atributo o conjunto de atributos de una tabla de la base de datos. Es un objeto que requiere información interna, y por eso tiene una repercusión en los metadatos. Son objetos con OID.

Buscar un valor en una secuencia ordenada de  $n$  valores es un procedimiento logarítmico, como se explica en detalle en [3]. Esto significa proporcional a la cantidad de cifras que tenga  $n$ . En cambio, en una secuencia desordenada es lineal, o sea proporcional al número de datos,  $n$ , que es muchísimo más. Pongamos por caso, para buscar un valor en una secuencia de mil valores. Si la secuencia está ordenada se tarda diez unidades de tiempo, y si no, mil. Esto en el caso peor, o sea teniendo mucha mala suerte, que es el caso que usamos de referencia.

Crear un índice para un atributo es como decir que queremos la tabla ordenada según ese atributo. Podemos ordenar la misma relación según diferentes atributos declarándolos índice cada uno de ellos.

La creación de un índice provoca guardarse una estructura de datos arborescente, es decir, jerárquica.

En la Figura 7.3(a) hay parte del contenido de una tabla. La clave primaria es un número entero. Por eso conviene imaginar que la tabla está físicamente ordenada según este campo, tal como se muestra.

Declarar un índice para el atributo **nombre** de esta tabla provoca la creación y mantenimiento de la estructura de datos de la Figura 7.3(b).

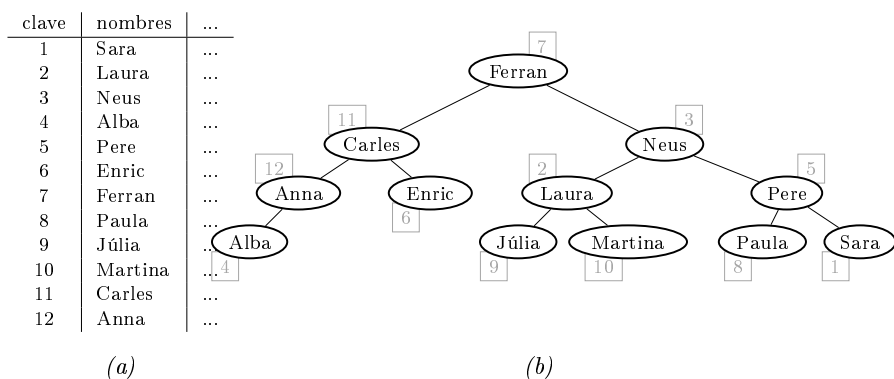


Figura 7.3: (a) Parte de una relación; (b) Estructura para un atributo índice.

Observa que a partir de la raíz del árbol no tardaremos en ningún caso más de tres pasos en encontrar un elemento dentro de un conjunto de doce. Esta estructura se llama árbol binario de búsqueda, o *binary search tree* *BST*, y se caracteriza por el hecho

que siendo un árbol binario, cualquier hijo izquierdo es ordenadamente anterior a su padre, y el padre anterior al hijo derecho, como se puede comprobar concentrándose en la Figura 7.3(b). Si se desea recorrer la tabla completa según el orden marcado por el índice, la información relevante no es más que la permutación que representa el inorden del árbol de la Figura 7.3, o sea 4,12,11,6,7,9,2,10,3,8,5,1. Eso significa la posición de las tuplas en la tabla ordenadas según el atributo **nombres**. Si tan solo interesa acceder algún elemento concreto, entonces con la árbol de la Figura 7.3(b) tardaremos tres unidades de tiempo como máximo para saber dónde está, frente a las doce que tardaríamos, también como máximo, si no existiera el índice.

El árbol de la Figura 7.3(b) es uno de los métodos como puede trabajar un índice. Hay otros.

Aún así, hay que recordar que el orden físico de los registros es el cronológico de inserción, no el de la clave primaria. Claro, si no fuera así se deberían mover registros en cada inserción. Y eso no puede ser porque tardaría un tiempo proporcional a la cantidad de registros que haya en la tabla, que pueden ser muchos centenares de millares. Físicamente, insertar es añadir al final, o más simple aún, al principio.

Se pueden declarar índices a funciones de los atributos. Es decir, en lugar de ordenar según los valores del atributo, ordenar según el resultado de someter esos valores a alguna función de transformación.

Automáticamente se crea un índice para cada restricción de unicidad que definamos en la base. Es indispensable que cada clave primaria tenga su índice, ya que el acceso por clave primaria debe ser eficiente necesariamente.

Tan solo por curiosidad, para tener constancia del momento en que se crean los índices, se puede poner el parámetro de configuración `client_min_messages` a nivel de `'DEBUG'`, con el comando

```
SET client_min_messages TO 'DEBUG';
```

justo antes de la creación de la primera tabla, en la primera línea del archivo `2-tablas.sql`, y volver a importar el `club.sql`.

Ten en cuenta que el comando `\c` borra los valores dados a los parámetros de configuración en la sesión. La última línea del archivo `1-init.sql`, que pone `\c club`, abre una sesión nueva, y por tanto anula los comandos `SET` previos. O sea que si pusieramos el comando `SET` antes, no serviría para nada.

En la Pantalla 7.14 se listan los índices creados en la base de datos `club`.

```

cmd - psql -h localhost -U postgres -d club
club=# \di
                                List of relations
 Schema |          Name          | Type | Owner  | Table
-----+-----+-----+-----+-----
 public | ciudad_pkey           | index | postgres | ciudad
 public | provincia_repetida    | index | postgres | provincia
 public | conoce_conoce_es_conocida_key | index | postgres | conoce
 public | deporte_pkey         | index | postgres | deporte
 public | hace_pasaporte_deporte_key | index | postgres | hace
 public | telefono_ya_asignado  | index | postgres | telefono
 public | pagos_pkey           | index | postgres | pagos
 public | persona_pkey         | index | postgres | persona
 public | socio_pkey           | index | postgres | socio
 public | trabajador_pkey      | index | postgres | trabajador
(10 rows)
club=# _

```

Pantalla 7.14. *Índices creados en la base de datos club.*

La gestión de los índices no tiene un impacto directo en el desarrollo de una base de datos. La intención de esta sección solamente ha sido dar a conocer la existencia de estos objetos y su función, a saber, agilizar los tratamientos con las tablas cuando los accesos se produzcan mediante atributos indexados.

Es decir, los índices sirven para los atributos que no siendo clave primaria se pretende a menudo una de dos, o bien acceder a la totalidad de la tabla ordenadamente según el atributo indexado, o bien acceder a tuplas que contengan valores concretos del atributo. O sea, ordenaciones o búsquedas.

Fíjate pues que si se trata de funciones de agregación, que recorren la tabla completa para tratar los valores de un atributo, entonces el orden es indiferente. Y por tanto no justifica la creación de índices.

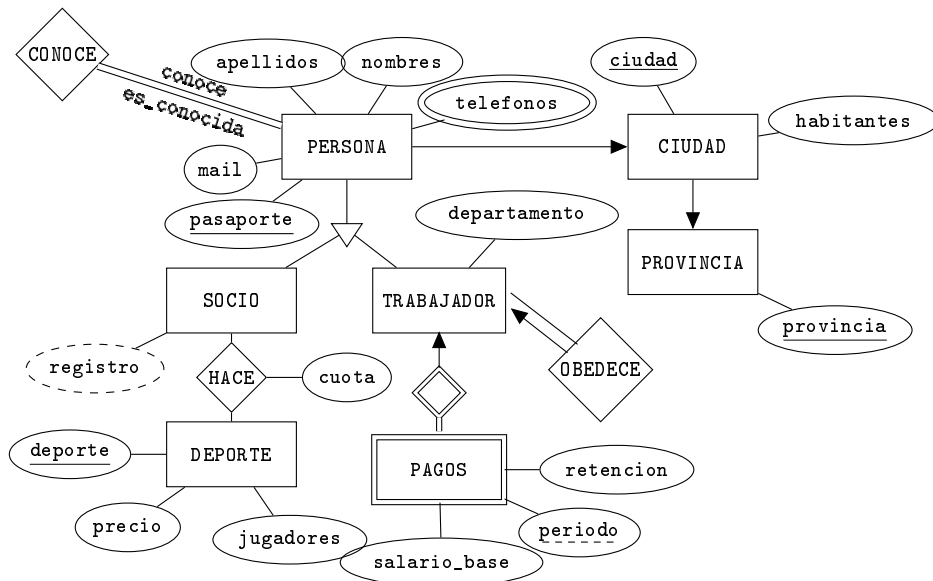
Una base de datos puede crear sus índices adicionales con el comando `CREATE INDEX` con el atributo a indexar, aunque que normalmente se hace de forma automática estableciendo las restricciones de unicidad en la creación de las tablas. Crearlos no es frecuente.



## 7.5 Componentes de Control

Entendemos por componentes de control aquellos que sirven para procesar los datos en instantes concretos. Son programas breves, interpretados, que se pueden ejecutar por solicitud explícita del usuario cliente, o bien asociados a acciones que ocurren en la base datos. En todas sus formas, son funciones. Si se pretende poderlas ejecutar de manera asíncrona, es decir con solicitudes explícitas del usuario, deben retornar algún tipo de datos conocido. En cambio, cuando se asocian a acciones concretas como inserciones o modificaciones de valores en tablas, entonces retornan un tipo de datos reservado, se llaman funciones de tipo disparador, y solamente pueden ser utilizadas para ese efecto.

### 7.5.1 Implementación de la Aplicación del Club Deportivo



Modelo 7.4. Modelo ER para la aplicación del club deportivo.

En los ejemplos de las próximas secciones se implementarán primeramente las funciones necesarias para las inserciones y borrados que requieran procedimientos específicos de los registros de las entidades del modelo ER del proyecto, que se repite en el Modelo 7.4. Eso significa las funciones relativas al registro y borrado de socios y trabajadores.

La parte procedural se implementará por roles. El primer objetivo es satisfacer el usuario administrador, luego socios, y finalmente trabajadores.

De entre las funciones a implementar distinguimos las genéricas que siempre son necesarias para todas las bases de datos. Son las que insertan, modifican, o borran los objetos del modelo de forma específica. Es el caso de las herencias. Y luego, habrá una colección de funciones concretas para cada aplicación, que irán ligadas a la definición de requerimientos para cada rol.

## Implementación de funciones genéricas

Esta tarea se encuentra automatizada en muchas tecnologías de programación. Se acostumbra a utilizar el término *binding* para reflejar en alguna interfaz gráfica cada uno de los objetos de la base de datos.

El *hibernate*, [5], es un entorno de objetos persistentes en el que el programador puede trabajar como si la base de datos estuviera en memoria central pudiendo hacer inserciones de la misma manera que un programa hace asignaciones.

Utilizar *binding* desde java para desarrollar una aplicación de interfaz para una base de datos significa asociar los objetos de la base a estructuras de datos del programa cliente. Por ejemplo, para mapear el resultado de una consulta concreta se proporciona una estructura de tipo tabla además de la sentencia SQL con la consulta en cuestión. Y para registrar un nuevo objeto se le asocia un formulario. Las técnicas de vinculación entre la base de datos y las variables de una aplicación fueron introducidas por Access de Microsoft con los formularios y luego han sido reproducidas profusamente en multitud de entornos, con más o menos requerimientos técnicos por parte del programador.

Aquí se harán las funciones genéricas partiendo de cero para conocer lo que en muchas utilidades va empaquetado en interfaces gráficas.

Está claro que si es una tarea muy automatizada es que debe ser una tarea rutinaria. En muchos casos estas funciones serían simples *wrappers*. Un wrapper (funda), es una función que recibe unos parámetros y la única cosa que hace es llamar a otra pasándole los mismos parámetros que ha recibido. Sirven para ajustar formatos entre entornos y poca cosa más. En este texto, no se mostrarán implementaciones de ningún wrapper, porque son bastante sencillos como para ahorrárnoslas. En concreto, para dar de alta provincias, ciudades o deportes, no se hará ninguna función específica. La aplicación cliente podrá hacer directamente los comandos SQL, `INSERT INTO`, `UPDATE SET`, o `DELETE FROM`.

En ningún caso solaparemos trabajo que haga el SGBD. Eso significa que si se produce un error de integridad, será PostgreSQL quien informe al usuario lanzando una excepción.

## Implementación de funciones específicas

Esta tarea debe reproducir las decisiones de diseño y la definición de requerimientos de la aplicación concreta. Una forma válida de hacerlo es en base al menú de opciones correspondiente a cada rol.

Para cada tipo de usuario se puede empezar haciendo una función que simplemente retorne el nombre de las funciones disponibles, de manera que las aplicaciones cliente puedan automatizar las opciones en sus menús.

## 7.6 Funciones

Las funciones se crean con el comando `CREATE FUNCTION` y se borran con `DROP FUNCTION`. Para implementar una función consideraremos que su estructura se compone de tres partes. La cabecera, el cuerpo, y una última línea constante en la que hay el nombre del lenguaje de programación que se está utilizando. En los dominios de este libro, siempre utilizaremos el mismo lenguaje que en el código lo llamaremos 'plpgsql' de procedural language postgres sql.

En la Figura 7.4 se presenta una función "hola mundo", con la terminología usada para sus partes. Crearemos un archivo como este en la carpeta del proyecto, que es como el banco de pruebas.

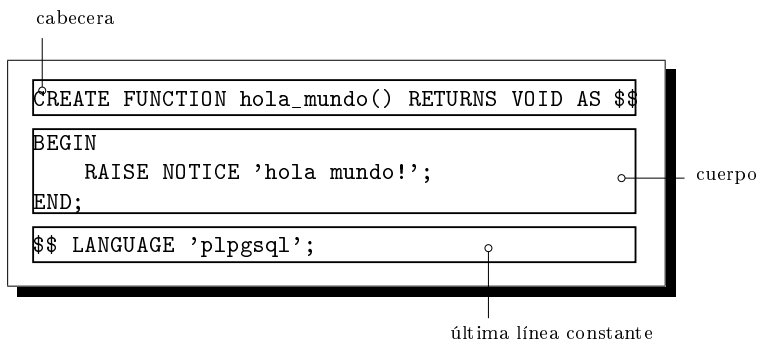


Figura 7.4: Archivo `hola_mundo.sql`

Ya con el archivo, podemos reproducir el diálogo de la Pantalla 7.15. Fíjate que una vez creada la función, se puede invocar de dos maneras. Con `SELECT funcion()`, o bien con `SELECT * FROM funcion()`.

En la Pantalla 7.15 se ven ambas formas de invocar la función, aunque que cuando solamente retorna una columna como es el caso, resulten equivalentes.

```

c:\cmd - psql -h localhost -U postgres -d club
club=# \i hola_mundo.sql
CREATE FUNCTION
club=# SELECT hola_mundo();
NOTICE: hola mundo!
 hola_mundo
-----
(1 row)

club=# SELECT * FROM hola_mundo();
NOTICE: hola mundo!
 hola_mundo
-----
(1 row)
club=# _

```

Pantalla 7.15. *Importación y invocación de una función.*

También se notable de la Pantalla 7.15, que el mensaje aparece luego de la etiqueta `NOTICE`. En PL/pgSQL todos los mensajes tienen un grado de severidad. Por eso podemos poner el parámetro `client_min_messages` al nivel `'DEBUG'`, el más laxo, como hemos hecho en la Sección 7.4.1 para ver los índices. Cada nivel muestra todos los mensajes de los niveles anteriores, más los suyos. Si pusiéramos el valor de este parámetro a `ERROR`, entonces no aparecerían los mensajes de severidad `NOTICE`. Respecto a este punto, podéis consultar el manual, [8].

Que `SELECT funcion()` sea equivalente a `SELECT * FROM funcion()` se debe a que solamente retorna una columna. Estas dos instrucciones se diferencian en los resultados cuando las funciones llamadas retornan varias columnas.

Inmediatamente luego del mensaje que la función escribe con la instrucción `RAISE NOTICE` por la consola, termina con la misma sintaxis que si fuera una relación. El caso de que la función solamente retorne un dato es igual que el de cuando retorna una relación con una columna de una fila. Como se puede contemplar en la Pantalla 7.15, la columna se titula `hola_mundo`, y su contenido consta de una única fila con un valor nulo, la línea en blanco. Este valor es el que ha retornado la función.

## 7.6.1 Cabecera

Mirando el escript de la Figura.7.4 ya se ve que la sentencia para crear una función se llama `CREATE FUNCTION`. También se admite poner `CREATE OR REPLACE FUNCTION`. Cuando desarrollamos funciones complicadas que difícilmente funcionarán a la primera, es

altamente recomendable esta segunda opción, ya que si no, entonces tendremos que borrar la función antes de volverla a importar después de un error, de compilación por ejemplo. Y eso no es tan fácil como podría parecer, ya que en PL/pgSQL se ornamentan los nombres de las funciones. Lo hace el intérprete de PL/pgSQL. Eso significa que automáticamente se añade la lista de tipos de los parámetros en el mismo nombre de la función, al final, y PostgreSQL las identifica mediante ese identificador resultante. Por esa razón, cuando se desea borrar una función mediante `DROP FUNCTION`, se debe dar el nombre de la función seguido de la lista de los tipos entre paréntesis.

Siguiendo con la creación de funciones, luego del nombre de la sentencia va el nombre de la función, `hola_mundo`, y la lista de parámetros de entrada en formato `nombre tipo`, que en la Figura 7.4 es vacía. Luego la palabra clave `RETURNS` en tercera persona de singular del presente simple, ya que es ella, la función, quién retorna alguna cosa. Sigue el tipo de la expresión que se retorna. Si no se desea retornar nada se puede poner `VOID`, que significa retornar un valor nulo. Finalmente, la palabra clave `AS` concluye la cabecera. En SQL estándar el cuerpo de la función va entre apóstrofes. En PL/pgSQL podemos poner doble dólar. Así evitamos colisiones con las cadenas de caracteres constantes que pueda haber en el cuerpo de la función, como es el caso de la Figura 7.4. En SQL estándar los apóstrofes que limitan las cadenas de caracteres constantes se ponen dos veces, haciendo el código más retorcido.

Una función puede retornar una tabla, `RETURNS TABLE(...)`. Para eso, se debe hacer toda la declaración de la tabla, con nombres de columnas y tipos, en la cabecera. En esos casos, son habituales las dos opciones siguientes.

- Que la última instrucción del cuerpo de la función consista en la instrucción `RETURN QUERY(SELECT...)`;
- O si no, en el caso más complicado, una función puede retornar un conjunto de registros retornando uno en cada iteración de un bucle. Esto es un número indeterminado de elementos formados por varios atributos. Cada elemento se retorna con la instrucción `RETURN NEXT A1, A2, ..., An`, dentro del bucle que una vez finalizado hace un `RETURN` para cerrar la secuencia de datos resultantes.

En cualquiera de esos dos casos, los títulos de las columnas retornadas serán los que se declaren en la cabecera de la función.

## 7.6.2 Cuerpo

El cuerpo de la función empieza con una sección que contiene la declaración de variables locales, a pesar de que en la Figura 7.4 no hay. Esta área va precedida del palabra clave `DECLARE`, y se pueden declarar variables de todos los tipos que se ha visto en la Sección 6.2.4, y alguno más que se verá más adelante en esta misma sección. La

declaración de variables locales acaba con la palabra clave `BEGIN`. Y entre `BEGIN` y `END` va el código imperativo que describe la acción que se realiza. En esta parte se puede poner cualquier sentencia de SQL igual que si fuera una instrucción más del lenguaje procedural. En este sentido, el PL/pgSQL es una generalización del SQL.

Además en el cuerpo de las funciones se pueden poner:

- Asignaciones a variables locales, en forma de composición secuencial.
- Sentencias alternativas entre las palabras clave `IF ... THEN ... END IF`;
- Bucles indexados, `FOR ... LOOP ... END LOOP`;
- Bucles de condición final desconocida `WHILE ... LOOP ... END LOOP`;
- Llamadas a otras funciones.

Hay una gran cantidad de funciones definidas para los diversos tipos de datos. Todas ellas disponibles desde el cuerpo de cualquier función. Por ejemplo, la función `length(str)` a la que se le pasa una cadena de caracteres y retorna un entero con su longitud. O bien `substr(str,p0,len)` que retorna la subcadena de caracteres de la cadena `str` desde la posición `p0` que es un número entero, con una longitud `len` que también es un entero. Por ejemplo, `substr('hola',2,1)` es la cadena formada por la letra 'o'.

En la Caja 7.13 se muestra una función sencilla. Se llama `siete_bits`. Sirve para quitar los acentos o diéresis que pueda tener el carácter que se le pasa como parámetro, y lo retorne a siete bits. Si el carácter que se le da ya es de siete bits, lo retorna igual que lo recibe.

```

\echo ----- funcion siete_bits()

CREATE OR REPLACE FUNCTION siete_bits(c CHAR) RETURNS CHAR AS $$
DECLARE
    i INTEGER;
    n INTEGER;
    origen TEXT DEFAULT 'ÁÉÏÍÓÚÛÑçæéííóúúçñ';
    destino TEXT DEFAULT 'AEIIOUUNCaеiiouucn';
BEGIN
    n = length(origen);
    FOR i IN 1..n LOOP
        IF c = substr(origen,i,1) THEN
            c = substr(destino,i,1);
            RETURN c;
        END IF;
    END LOOP;
    RETURN c;
END;
$$ LANGUAGE 'plpgsql';

```

Caja 7.13. Archivo `siete_bits.sql`. Ejemplo de bucle sencillo.

La forma de transformar un carácter a siete bits es simple. Se declaran dos cadenas de caracteres, `origen` y `destino`. Se inicializa `origen` con los caracteres que se transformarán, y `destino` con los caracteres correspondientes transformados. Estas inicializaciones se hacen en la misma declaración utilizando el operador `DEFAULT`. Así pues, hay una correspondencia posicional entre los caracteres de las dos cadenas.

La función empieza guardando en la variable entera `n` la longitud de la cadena `origen`, igual que podría ser la de la cadena `destino`. De hecho, podría ser una constante igual a 24, pero tal como está resulta más sencillo poder ampliar los caracteres que se pueda desear transformar en el futuro. Y luego, entra en un bucle en el que compara la letra que ha recibido como parámetro con cada una de las que está en la cadena `origen`. Si la encuentra, retorna la letra correspondiente a la misma posición de la cadena `destino`.

Esta función se utilizará más adelante. Al no depender de ninguna tabla conviene guardarla en una nueva carpeta `club/_librería`, y añadir la importación al final del escript que inicializa la base de datos `1-init.sql`, diciendo

```
\i '_libreria\_libreria.sql'
```

prefijando los nombres del escript y de la carpeta con un guión bajo para que queden los primeros de la carpeta alfabéticamente. Y claro, dentro de la carpeta `_libreria`

tenemos que crear un escript que se llame `_libreria.sql`.

Es en el escript `_libreria\libreria.sql` donde debemos poner la importación del archivo con el código fuente de la función.

```
\i '_libreria\libreria.sql'
```

Mucho trabajo para una función, pero bien, ya queda preparado para poder añadir funciones adicionales siempre que sean independientes de los objetos de la base. Podemos ir implementando funciones a medida que desarrollamos la aplicación. Estas funciones que no dependen del contenido de la base de datos se llaman *inmutables*.

### 7.6.3 Entrada y salida de datos con PL/pgSQL

Atención a la entrada y salida de datos en las funciones hechas en PL/pgSQL. No hay ninguna manera de introducir un dato por teclado. El PL/pgSQL no incluye ninguna instrucción para poder leer datos del teclado porque no son necesarias.

Para poder imprimir datos por pantalla hay tan solo una instrucción, `RAISE`, que se utiliza como se ha visto a la Figura 7.4 para escribir una cadena de caracteres constante. El nombre hace reflexionar. No se dice ni *print*, ni *write*, ni nada que suene a acción frecuente. `RAISE` significa lanzar. Y este verbo desea significar que no se tiene claro el destino donde va a ir a parar el mensaje. Es más, utilizar este verbo hace que sea más normal lanzar excepciones que mensajes, como efectivamente será.

El comando `RAISE` sirve a menudo para poner trazas. En cualquier lenguaje de programación, poner una traza en el código fuente significa escribir por el canal de salida (ya sea pantalla, fichero de log, o cualquier otro medio) el valor de alguna variable. Esto se hace exclusivamente en la fase de desarrollo, y es un trabajo que los entornos de desarrollo integrados, como el *eclipse* o la *netbeans* han evitado a los programadores. Años atrás no había forma alternativa de depurar un código que no fuera con el uso de trazas. Y así se sigue trabajando cuando se desarrolla una base de datos desde la línea de comandos como lo estamos haciendo aquí. Para averiguar qué valor tiene una variable en un punto concreto de la ejecución utilizamos la instrucción

```
RAISE NOTICE 'la variable pepito = %',pepito;
```

para una variable que se llamase `pepito`. O sea, el símbolo de tanto por ciento será sustituido por el valor real en el momento de la ejecución. Igualmente pueden aparecer varios tanto por cientos en la cadena de caracteres, y varias variables separadas por comas en lugar de `pepito`. Entonces, la asociación entre los tantos por ciento y las variables es posicional. Es de agradecer que la manera de hacerlo, con el carácter de tanto por ciento, no dependa del tipo de la variable, como habitualmente sucede en la mayoría de lenguajes de programación de alto nivel. Claro, por otra parte, tampoco



podemos, ni hay que especificar formatos.

Una forma alternativa con la que podemos hacer uso de la instrucción `RAISE` es con el calificador `EXCEPTION` en lugar de `NOTICE`. Si hacemos eso abortaremos la ejecución de la función en este punto, y retornaremos el mensaje dado como texto de la excepción. Los programas cliente podrán así atender excepciones de tipo `SQLException` de la manera que tengan definida, como puede ser el mecanismo `try-catch`.

Para clarificar el uso de la instrucción `RAISE` del PL/pgSQL observemos la función de la Caja 7.14. Con espíritu divulgativo emitimos un mensaje calculando el número  $\pi$  como cuatro veces el arcotangente de 1.0.

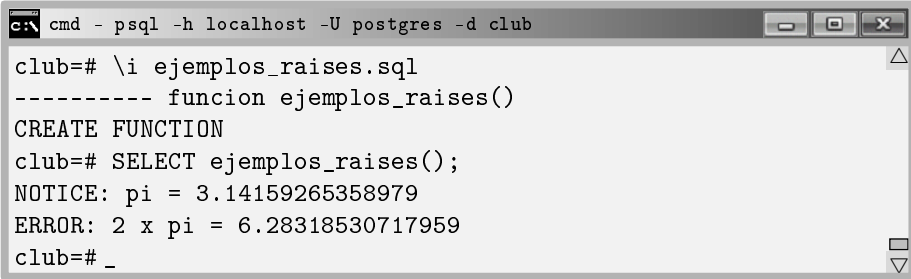
```
\echo ----- funcion ejemplos_raises()

CREATE FUNCTION ejemplos_raises() RETURNS INTEGER AS $$
DECLARE
    pi DOUBLE PRECISION;
BEGIN
    pi = 4 * atan(1.0);
    RAISE NOTICE 'pi = %',pi;
    RAISE EXCEPTION '2 x pi = %',2*pi;
    RAISE NOTICE 'Esto ya no se escribirá';
END;
$$ LANGUAGE 'plpgsql';
```

Caja 7.14. Archivo `ejemplos_raises.sql`. Uso de la instrucción `RAISE`.

Se demuestra que estamos tratando con un intérprete más que con un compilador porque a pesar de decir en la cabecera que se retorna un entero, la carencia de una instrucción `RETURN` en el cuerpo no impide que pueda ser ejecutada.

En la Pantalla 7.16 se detalla el diálogo resultante de la importación y ejecución de la función `ejemplos_raises()`.



```

c:\cmd - psql -h localhost -U postgres -d club
club=# \i ejemplos_raises.sql
----- funcion ejemplos_raises()
CREATE FUNCTION
club=# SELECT ejemplos_raises();
NOTICE: pi = 3.14159265358979
ERROR: 2 x pi = 6.28318530717959
club=# _

```

Pantalla 7.16. *Ejecución del escript de la Caja 7.14.*

Está claro que la instrucción `RAISE EXCEPTION` aborta la ejecución. Eso provoca que no salga ni el mensaje siguiente, ni el error (que diría que ha llegado al final de la función sin encontrar una instrucción `RETURN`). Tampoco sale la impresión con aspecto de relación, ese con el título de la columna que aparecía en la Figura 7.4. Observa pues que el nivel de severidad del último mensaje, `ERROR`, provoca el final de la ejecución.

#### 7.6.4 Consultas de actualización en PL/pgSQL

Las consultas de actualización se hacen exactamente igual que cuando se trabaja interactivamente con el `psql`, pudiendo usar variables donde hasta ahora se usaban valores constantes. Dicho eso ya podemos implementar las dos funciones necesarias para introducir nuevos registros. Los de socios y los de trabajadores.

En la Caja 7.15 se describe la función `registrar_socio()`. Recibe los datos de los socios y los distribuye entre las tablas `persona` y `socio`. Además, fíjate que establecemos el valor del atributo derivado `registro` de la tabla `socio`. La variable `registro` de tipo `TIMESTAMP WITH TIME ZONE` tiene un carácter meramente ilustrativo, y es perfectamente prescindible.

```
\echo ----- funcion registrar_socio()

CREATE FUNCTION registrar_socio(p TEXT, n TEXT, c TEXT,
                               m dominio_mail, ci TEXT) RETURNS INTEGER AS $$
DECLARE
    registro TIMESTAMP WITH TIME ZONE;
BEGIN
    INSERT INTO persona VALUES(p,n,c,m,ci);
    registro = now();
    INSERT INTO socio VALUES(p,registro);
    RETURN 1;
END;
$$ LANGUAGE 'plpgsql';
```

Caja 7.15. Archivo registrar\_socio.sql. Consultas de actualización.

El valor de retorno consiste en el número de socios que se han insertado, que puede ser uno o ninguno. Si se llega a ejecutar el `RETURN` es que se trata de un caso exitoso, que puede no ser siempre, ya que las dos transacciones previas podrían resultar en una excepción abortando la ejecución.

En la Caja 7.15 registramos la nueva persona antes que el socio para no producir un error de integridad referencial. El error posible, o sea que el pasaporte que pretendamos insertar como socio no exista en la tabla persona, queda resuelto haciéndolo en ese orden.

A pesar de todo, hay aún dos otros errores de integridad referencial que efectivamente se pueden producir. Uno por duplicidad de clave primaria, y el otro por inexistencia de valor apuntado por la clave foránea ciudad. Conviene conocer los mensajes de error asociados a cada uno de ellos. Por ello se describen a continuación, con ejemplos.

Presta atención a los mensajes que nos retornan esas excepciones, puesto que, como se puede ver, son mensajes dirigidos a un usuario que, en el fondo, no se sabe si es un programa informático o un ser humano. Se debe asociar la emisión de esos mensajes a las interrupciones de los procedimientos que los han emitido, puesto que ambas cosas son consecuencias de una misma excepción.

- Que el número de pasaporte ya exista en la base de datos. La cédula 0127673812 es el de Carmen Peralta de Babahoyo. Observa el mensaje de la Pantalla 7.17, que tiene tres partes indicando un error, el detalle y el contexto, donde también se puede ver la ornamentación que practica PostgreSQL con los nombres de las funciones.

```

cmd - psql -h localhost -U postgres -d club
club=# \i registrar_socio.sql
----- funcion registrar_socio()
club=# CREATE FUNCTION
club=# SELECT registrar_socio('0127673812','Matthew Pablo','Robledo Ze
club-# 'mrob@gmail.com','Ibarra');
ERROR: duplicate key value violates unique constraint "persona_pkey"
DETAIL: Key (pasaporte)=(0127673812) already exists.
CONTEXT: SQL statement "INSERT INTO persona VALUES(p,n,c,ci)"
PL/pgSQL function registrar_socio(texto,texto,texto,dominio_mail,texto
club=# _

```

Pantalla 7.17. *Error de inserción clave primaria duplicada.*

- Que la ciudad, siendo un valor válido, no exista en la tabla de ciudades. Entonces se producirá el error de la Pantalla 7.18.

```

cmd - psql -h localhost -U postgres -d club
club=# SELECT registrar_socio('111111111','Matthew Pablo','Robledo Ze
club-# 'mrob@gmail.com','Santo Domingo');
ERROR: insert or update on table "persona" violates foreign key cons
DETAIL: Key (ciudad)=(Santo Domingo) is not present in table "ciudad".
CONTEXT: SQL statement "INSERT INTO persona VALUES(p,n,c,ci)"
PL/pgSQL function registrar_socio(texto,texto,texto,dominio_mail,texto
club=# _

```

Pantalla 7.18. *Error de inserción clave foránea inexistente.*

A continuación se implementa la función para registrar nuevos trabajadores. No tiene más complicación que la anterior. Es decir, se limita a registrar en primer lugar la persona para que el pasaporte sea existente, y seguidamente registra los datos adicionales en la tabla TRABAJADOR.

```

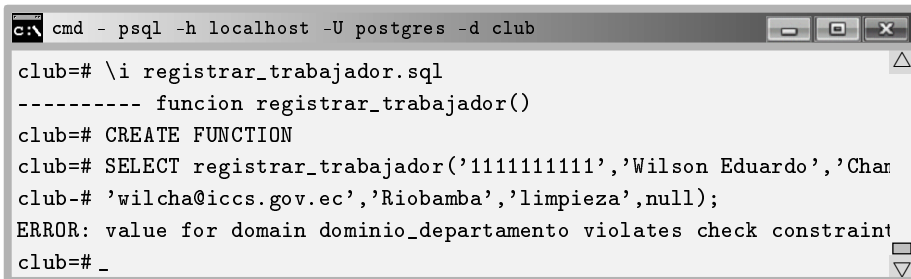
\echo ----- funcion registrar_trabajador()

CREATE FUNCTION registrar_trabajador(p TEXT, n TEXT, c TEXT,
    m dominio_mail, ci TEXT,
    d dominio_departamento, o TEXT) RETURNS INTEGER AS $$
BEGIN
    INSERT INTO persona VALUES(p,n,c,m,ci);
    INSERT INTO trabajador VALUES(p,d,o);
    RETURN 1;
END;
$$ LANGUAGE 'plpgsql';

```

Caja 7.16. Archivo registrar\_trabajador.sql. Consultas de actualización.

Un nuevo error se puede ocasionar con el código de la Caja 7.16. En la Pantalla 7.19 se muestra el caso. Se da un nombre para el departamento, *limpieza*, que no pertenece al dominio\_departamento.



```

c:\> cmd - psql -h localhost -U postgres -d club
club=# \i registrar_trabajador.sql
----- funcion registrar_trabajador()
club=# CREATE FUNCTION
club=# SELECT registrar_trabajador('1111111111','Wilson Eduardo','Char
club-# 'wilcha@iccs.gov.ec','Riobamba','limpieza',null);
ERROR: value for domain dominio_departamento violates check constraint
club=# _

```

Pantalla 7.19. Error valor incorrecto del dominio.

Este par de funciones, registrar\_socio() y registrar\_trabajador() solamente deberían de ser ejecutadas por trabajadores administrativos. Para eso hay que añadir un par de líneas al final del archivo 4-usuarios.sql que establezcan estos privilegios, como en la Caja 7.17.

```

...
CREATE ROLE entrenadores;
DROP ROLE IF EXISTS administrativos;
CREATE ROLE administrativos;

GRANT EXECUTE ON FUNCTION registrar_socio(texto,texto,texto,
    dominio_mail,texto) TO administrativos;
GRANT EXECUTE ON registrar_trabajador(texto,texto,texto,
    dominio_mail,texto,texto,texto) TO administrativos;

```

Caja 7.17. Adición de privilegios sobre las funciones en el archivo 4-usuarios.sql.

Para agilizar, no vamos a mostrar los errores que se producen cuando un usuario sin los permisos necesarios intenta ejecutar alguna de esas funciones, pero de todas formas, te animo a probarlo por tu cuenta.

### 7.6.5 Consultas de lectura en PL/pgSQL

Las consultas de lectura se caracterizan por el hecho de que la información transita de la base de datos hacia el usuario. En cambio, en las consultas de actualización pasa lo contrario. Esto no tiene más trascendencia cuando se trabaja interactivamente, ya que hay la pantalla que soporta esa transmisión. Es decir, si hacemos la selección de todo el contenido de una tabla, simplemente la obtenemos a través del monitor. Hacer una consulta de lectura desde un programa debe ser diferente, ya que no tendría ningún sentido que el programa volcara la información al monitor cuando nadie lo estuviera mirando. Por tanto, son necesarias variables para alojar los datos que obtenemos vía consulta de lectura. Y eso significa discernir entre si la consulta retorna un solo valor, o una colección. Por esa razón se ha dividido esta sección en dos subsecciones.

En cualquiera de los dos casos, en PL/pgSQL hay una variable de entorno que se llama `FOUND`. Es una variable booleana que se actualiza luego de cualquier `SELECT` y vale cierto si ese `SELECT` ha dado alguna tupla de resultado.

Las consultas de lectura introducen una novedad interesante. Nuevos tipos para las variables. El sufijo `%ROWTYPE` sirve para expresar que una variable es del tipo que sea una fila de una tabla. Por ejemplo, `persona%ROWTYPE` significa un tipo de variable que se corresponde con una fila de la tabla `persona`. Y eso significa una estructura con cinco campos, pasaporte, nombres, apellidos, mail, y ciudad. Básicamente, con variables de estos tipos podemos hacer dos cosas. O bien un `SELECT * INTO` la variable (que se explica en breve), o bien un bucle `FOR` la variable `IN (SELECT * FROM persona) LOOP`, que se explica más adelante.

Es importante comprender lo que significan los tipos %ROWTYPE porque se utilizarán intensamente en los disparadores.

### Consultas que retornan un solo valor

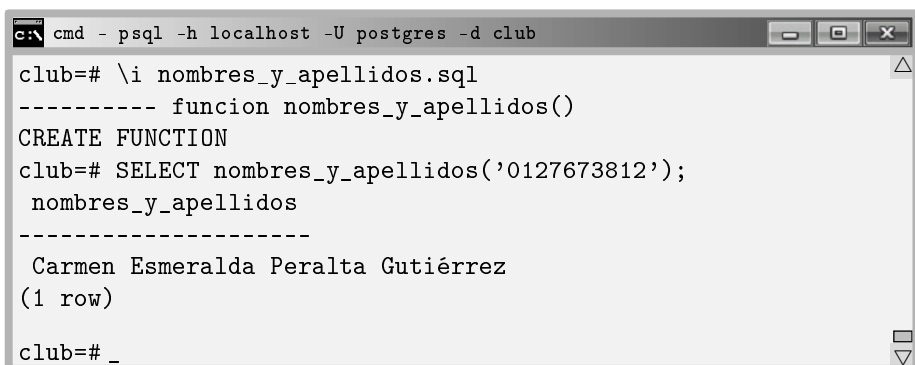
En este caso podemos guardar el resultado del SELECT en una variable local de un tipo compatible con el resultado de la consulta poniendo la palabra clave INTO seguido del nombre de la variable luego de la palabra SELECT.

En la Caja 7.18 se muestra una función que dado un número de pasaporte retorna los nombres y los apellidos de la persona correspondiente, separados por un blanco.

```
\echo ----- funcion nombres_y_apellidos()
CREATE FUNCTION nombres_y_apellidos(p TEXT) RETURNS TEXT AS $$
DECLARE
    rp persona%ROWTYPE; -- registro con la persona.
BEGIN
    SELECT INTO rp * FROM persona WHERE pasaporte = p;
    RETURN rp.nombres || ' ' || rp.apellidos;
END;
$$ LANGUAGE 'plpgsql';
```

Caja 7.18. *Primera versión archivo nombres\_y\_apellidos.sql. Selección de un solo valor.*

Después de haber hecho una importación desde el psql podemos comprobar su funcionamiento como se hace en la Pantalla 7.20.



```
cmd - psql -h localhost -U postgres -d club
club=# \i nombres_y_apellidos.sql
----- funcion nombres_y_apellidos()
CREATE FUNCTION
club=# SELECT nombres_y_apellidos('0127673812');
nombres_y_apellidos
-----
Carmen Esmeralda Peralta Gutiérrez
(1 row)
club=# _
```

Pantalla 7.20. *Importación y uso de la función nombres\_y\_apellidos().*

Parece funcionar. No obstante, si se le pasa un número de pasaporte no presente en la tabla `persona`, la función de la Caja 7.18 peta por intento de uso de puntero nulo en la última línea, `NullPointerException`, retornando una excepción. Y probablemente este no debería ser el comportamiento esperado. Una versión alternativa, pues, se muestra en la Caja 7.19.

```
\echo ----- funcion nombres_y_apellidos()

CREATE FUNCTION nombres_y_apellidos(p TEXT) RETURNS TEXT AS $$
DECLARE
    rp persona%ROWTYPE; -- registro con la persona.
BEGIN
    SELECT INTO rp * FROM persona WHERE pasaporte = p;
    IF FOUND THEN
        RETURN rp.nombres || ' ' || rp.apellidos;
    END IF;
    RETURN 'el número de pasaporte % es inexistente.',p;
END;
$$ LANGUAGE 'plpgsql';
```

Caja 7.19. Archivo `nombres_y_apellidos.sql`. Selección y verificación de un solo valor.

Una vez verificado su correcto funcionamiento, probando la función con números de pasaporte existentes e inventados, procedemos guardando el script en la carpeta `club/persona`, añadimos en el archivo `persona.sql` la línea para importar esa función, `\i 'persona\nombres_y_apellidos.sql'`, después de la creación de la tabla, y volvemos a remontar la base desde la nada importando el archivo principal, `club.sql`.

Observa que mientras se desarrolla la aplicación como lo estamos haciendo quizás nos interesa más la versión de la Caja 7.19, pero en cambio a partir del momento en que esta base de datos haya de ser utilizada para aplicaciones GUI puede resultar más conveniente lanzar una excepción con la versión de la Caja 7.18.

La única fuente de errores a los que la aplicación se expone consiste en los datos introducidos para el usuario. Por ello, es normal que no ahorremos esfuerzos en controlar tanto como se pueda esa entrada de datos, y posible fuente de inconsistencias. Utilizaremos varias funciones adicionales para verificar que no se den valores con errores de acentuación a la hora de dar ciudades, provincias, y deportes. Es decir, en aquellas tablas en las que la clave primaria sea textual y el usuario pueda introducir errores en los acentos.

En ese sentido, nos aprovisionamos primero de una función inmutable adicional que



nos retorne la forma canónica de una cadena de caracteres. Diremos forma canónica a la misma cadena pero sin acentos ni blancos, y en mayúsculas. Es una definición que establecemos en este momento. Hay quien le llamaría forma normalizada, o forma estándar. En cualquier caso, representa la forma de referencia de cada término.

En la Caja 7.20 se muestra la función para canonizar cualquier palabra. Y a continuación se hace un análisis minucioso.

```
\echo ----- funcion canoniza()

CREATE FUNCTION canoniza(palabra TEXT) RETURNS TEXT AS $$
DECLARE
    i INTEGER;
    n INTEGER;
    c CHAR;
    str TEXT DEFAULT '';
BEGIN
    n = length(palabra);
    palabra = upper(palabra);
    FOR i IN 1..n LOOP -- para cada letra del mot
        c = substr(mot,i,1);
        IF c <> ' ' THEN
            str = str || siete_bits(c);
        END IF;
    END LOOP;
    RETURN str;
END;
$$ LANGUAGE 'plpgsql';
```

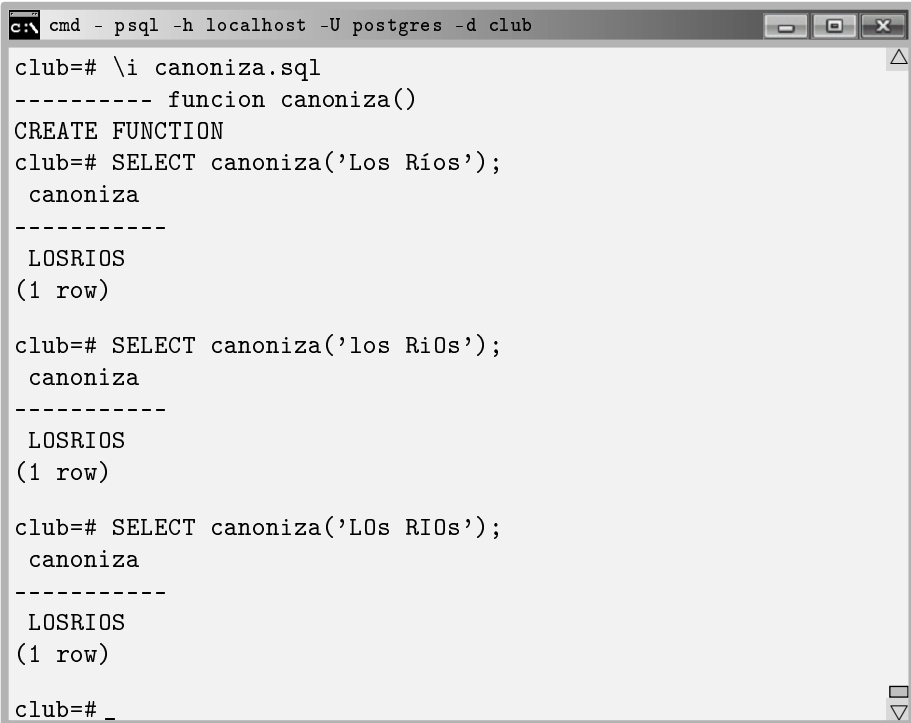
Caja 7.20. Archivo `canoniza.sql`. Función auxiliar.

Esta función empieza guardándose la longitud de la cadena de caracteres que recibe como parámetro de entrada, y seguidamente la pasa a mayúsculas con la función `upper()`, predefinida en el PL/pgSQL. Después hace un recorrido eliminando blancos y transformando cada letra con la función `siete_bits()` de la Caja 7.13. El valor de la variable `str` es el que finalmente se retorna.

Para el correcto funcionamiento de la función `canoniza()` es indispensable que la variable `str` se inicialice con una cadena de caracteres vacía. En la Caja 7.20 se ha hecho con la cláusula `DEFAULT`. También se hubiera podido hacer en cualquier lugar previo al bucle con una asignación, `str = ''`. Este valor inicial resulta necesario en la forma de funcionar del procedimiento.

En cada iteración del bucle concatena una letra más a la variable `str`. Ergo, si no se le hubiera dado un valor inicial habría empezado la ejecución con un valor nulo, y entonces, concatenar cualquier cosa a `str` no tendría efecto alguno, ya que en todo momento el valor sería nulo. En cierta forma pues, esta variable juega el papel de acumulador, y como tal, hay que inicializarla al string vacío, como lo haría a cero si fuera numérico.

En la Pantalla 7.21 se muestra el resultado de esta función para una cadena de caracteres de ejemplo, *Los Ríos*.



```

c:\> cmd - psql -h localhost -U postgres -d club
club=# \i canoniza.sql
----- funcion canoniza()
CREATE FUNCTION
club=# SELECT canoniza('Los Ríos');
 canoniza
-----
 LOSRIOS
(1 row)

club=# SELECT canoniza('los RiOs');
 canoniza
-----
 LOSRIOS
(1 row)

club=# SELECT canoniza('LOs RlOs');
 canoniza
-----
 LOSRIOS
(1 row)

club=# _

```

Pantalla 7.21. *Importación y usos de la función canoniza().*

A partir de la Pantalla 7.21 se debe comprender que el valor canónico de la cadena "LOs RlOs" es el mismo que el de "los RiOs", y también que "Los Ríos". O en otras palabras, se trata de cadenas de caracteres canónicamente equivalentes.

Por lo dicho, queremos ir más allá en la protección contra los errores frente a los datos de entrada que el usuario pueda introducir. Y la función `canoniza()` de la Caja 7.20 nos va a resultar útil para hacer una función de contraste entre los nuevos datos y los existentes. Es decir, insistiendo en nuestro empeño, nos aprovisionamos además de una función que llamamos `provincia_parecida()`, que ya no es inmutable. Las funciones *volátiles* son las que dependen del contenido de la base de datos. Que

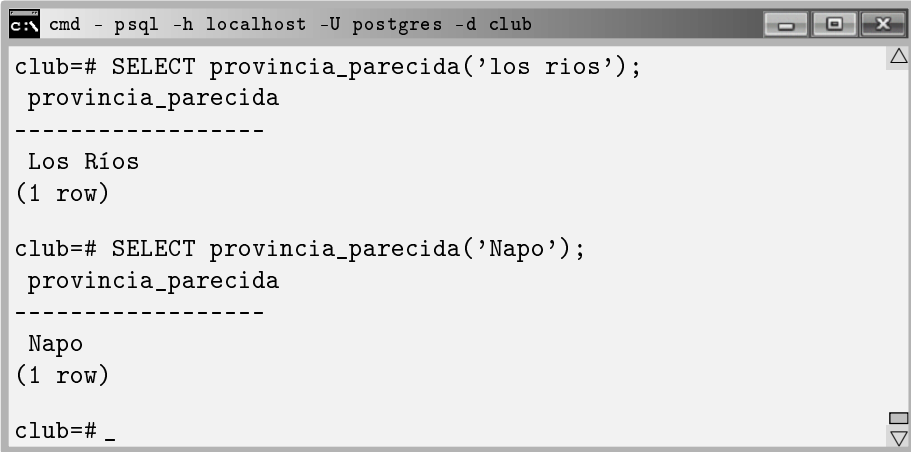
sea inmutable o volátil impacta sobre dónde guardarla. Las inmutables en la librería, en cambio las volátiles en la carpeta correspondiente a la última tabla que referencien. La función recibe una palabra, supuestamente un nombre de provincia, y en caso que en la base de datos haya alguna otra provincia que sea canónicamente equivalente (o sea, parecida en realidad), retorna la existente. Si no es así, retorna el mismo parámetro de entrada.

```
\echo ----- funcion provincia_parecida()

CREATE FUNCTION provincia_parecida(c TEXT) RETURNS TEXT AS $$
DECLARE
    vc; -- nombre correcto de la provincia preexistente.
    cc; -- nombre canonico de la provincia introduciéndose.
BEGIN
    cc = canoniza(c);
    SELECT provincia INTO vc
    FROM provincia
    WHERE cc = canoniza(provincia);
    IF FOUND THEN
        RETURN vc;
    END IF;
    return c;
END;
$$ LANGUAGE 'plpgsql';
```

Caja 7.21. Archivo `provincia_parecida.sql`. *Función auxiliar.*

La función de la Caja 7.21 es útil para corregir algunos errores básicos de deletreo que puede cometer el usuario en la introducción del nombre de las provincias. En la Pantalla 7.22 vemos un par de ejecuciones de esta función.



```

c:\> cmd - psql -h localhost -U postgres -d club
club=# SELECT provincia_parecida('los rios');
provincia_parecida
-----
Los Ríos
(1 row)

club=# SELECT provincia_parecida('Napo');
provincia_parecida
-----
Napo
(1 row)

club=# _

```

Pantalla 7.22. Ejemplos de uso de la función `provincia_parecida()`.

Esta función se usará en el disparador de inserción que asociaremos a la tabla `provincia`. Se dejan al lector las funciones correspondientes para las ciudades, en el escript `ciudad_parecida.sql`, y para los deportes en la `deporte_parecido.sql` se pueden crear análogamente, y a pesar de no mostrarse aquí, sí que se encuentran colgadas en el repositorio que se indica al final del preámbulo.

## Cláusula `PERFORM`

El uso de la variable `FOUND` es muy útil. Tanto, que a veces más que estar interesados en lo que retorna un `SELECT` estamos interesados únicamente en si retorna alguna cosa. En esos casos, tanto la variable que se debería declarar como la cláusula `INTO` del `SELECT` pueden evitarse utilizando la instrucción `PERFORM`.

La cláusula `PERFORM` funciona exactamente igual que un `SELECT`, con la ventaja de no tener que dar ninguna variable para guardar el resultado de la consulta. Ahora bien, la variable `FOUND` sí que quedará actualizada con el valor correcto igual que si el `SELECT` correspondiente al `PERFORM` obtuviera algún resultado.

Por ejemplo, se podría usar la cláusula `PERFORM` en las inserciones de los roles de pago. Consideraremos que tanto el salario base como la retención de los trabajadores, cuando no se pase por parámetro, será el mismo que el del rol de pago anterior. Para el caso del `salario_base` que es un atributo requerido, si no se da y es el primer rol de pago del trabajador se producirá un error.

Entonces, para saber si cuando se registra una nuevo rol de pago es el primero del trabajador se debería usar la cláusula `PERFORM`.

Esta cláusula también se puede utilizar para invocar una función que sea un procedimiento, es decir, que haga cosas pero no retorne ninguna información. Por ejemplo, en el cuerpo de una función podría haber la instrucción `PERFORM hola_mundo()`.

Filosóficamente, esos aparentemente dos usos de la cláusula no son en realidad más que uno. No debe sorprendernos, ya que para cualquiera que sea el tipo de una variable digamos `var`, por ejemplo, ésta podría aparecer en la instrucción `SELECT INTO var hola_nombre()`, que teniendo en cuenta que retorna un valor nulo, la instrucción sería válida. Pura abstracción.

Se puede concluir así en que luego de llamar a una función desde el cuerpo de otra con la cláusula `perform`, cuando esa función es un procedimiento es decir devuelve `VOID`, entonces la variable `FOUND` vale falso.

### Consultas que retornan registros

Sin duda, las funciones más complicadas del PL/pgSQL son las que retornan múltiples registros, puesto que en ningún caso pueden usarse como expresión. Pueden ser fundamentalmente de dos formas. Con `RETURN QUERY` o con `RETURN NEXT`. También, una función puede retornar una variable de tipo `RECORD`, pero en este libro no vamos a profundizar lo suficiente como para ilustrar ese tipo de datos que puedes consultar a partir de la ayuda en línea.

Las funciones con `RETURN QUERY` son parecidas a una vista. En la Caja 7.22 se implementa una función que retorna la cantidad de socios inscritos en cada deporte. Esta manera de resolverlo se definió en el SQL 92. Está claro que lo mismo se podría hacer con una vista, que forma parte del SQL estándar desde la primera versión.

```
\echo ----- funcion deportes_socios()

CREATE FUNCTION deportes_socios()
  RETURNS TABLE(deportes TEXT, socios INTEGER) AS $$
BEGIN
  RETURN QUERY(
    SELECT deporte, count(*)
    FROM hace
    GROUP BY deporte
  );
END;
$$ LANGUAGE 'plpgsql';
```

Caja 7.22. Archivo `deportes_socios.sql`. Uso de la instrucción `RETURN QUERY`.

En este caso, las dos formas de invocar la función resultan diferentes.

En la Pantalla 7.23 se muestra la forma de una única columna. Fíjate que cada fila se cierra entre paréntesis, y toda ella forma una sola cadena de caracteres que es el tipo de la columna resultante. El título de esta columna es el mismo nombre de la función, como antes.



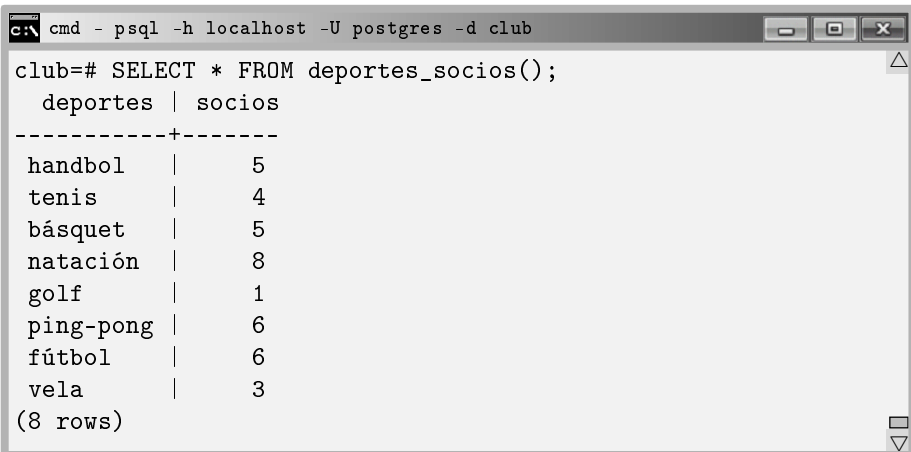
```

c:\> cmd - psql -h localhost -U postgres -d club
club=# SELECT deportes_socios();
deportes_socios
-----
(handbol,5)
(tenis,4)
(básquet,5)
(natación,8)
(golf,1)
(ping-pong,6)
(fútbol,6)
(vela,3)
(8 rows)

```

Pantalla 7.23. *Forma poco aconsejable de llamar la función de la Caja 7.22.*

La confusión entre la llamada ilustrada en la Pantalla 7.23 y la segunda manera de hacerlo, la de la Pantalla 7.24 puede traer graves consecuencias para los programas clientes.



```

c:\> cmd - psql -h localhost -U postgres -d club
club=# SELECT * FROM deportes_socios();
deportes | socios
-----+-----
handbol  |      5
tenis    |      4
básquet  |      5
natación |      8
golf     |      1
ping-pong |      6
fútbol   |      6
vela     |      3
(8 rows)

```

Pantalla 7.24. *Forma correcta de llamar la función de la Caja 7.22.*

Es notorio que en esta segunda forma de invocar la función los títulos de las columnas son los que se han expresado en la cabecera de la función.

Guardaremos el archivo `deportes_socios.sql` en la carpeta `club/hace`, lógicamente.

El error que supone confundir la llamada de una sola columna a la de tantas como la consulta interna dé es frecuente mientras se desarrolla la aplicación. Y difícil de detectar, porque cuando el programa cliente pretende acceder a la segunda columna se produce una excepción de puntero nulo, ya que la consulta tan solo ha retornado una sola columna.

Respecto a las consultas de lectura que retornan varios registros pero para cada uno de ellos hay un tratamiento específico, se usan intensivamente los tipos fila. Como se ha mostrado en la función `nombres_y_apellidos()` de las Cajas 7.18 y 7.19, esos tipos de variables van asociados a las tuplas de las tablas de la base de datos. Sus nombres se forman con el nombre de la tabla en cuestión seguido de la cadena `%ROWTYPE`. Por ejemplo, el tipo `ciudad%ROWTYPE` significa una estructura de datos de cuatro campos con los atributos de la tabla `ciudad`.

Además, las variables declaradas de tipo fila pueden utilizarse como índices de bucles que recorran una consulta. Nos encontramos con ello, ante una de las capacidades expresivas más potentes del lenguaje PL/pgSQL.

La función `ciudades_deportes_socios()` se muestra en la Caja 7.23. Obtiene la cantidad de socios de cada ciudad que practican cada deporte.

```

\echo ----- funcion ciudades_deportes_socios()

CREATE FUNCTION ciudades_deportes_socios()
  RETURNS TABLE(ciudad TEXT, deporte TEXT, socios BIGINT) AS $$
DECLARE
  rc ciudad%ROWTYPE; -- registro de la tabla ciudad
  rd deporte%ROWTYPE;
  s BIGINT;
BEGIN
  FOR rc IN (SELECT * FROM ciudad c) LOOP
    FOR rd IN (SELECT * FROM deporte d) LOOP
      SELECT INTO s COUNT(*)
        FROM hace h
        WHERE h.deporte = rd.deporte
        AND h.pasaporte IN (SELECT pasaporte
          FROM persona
          WHERE ciudad = rc.ciudad);
      IF s > 0 THEN
        RETURN NEXT c.ciudad,d.deporte,s;
      END IF;
    END LOOP;
  END LOOP;
  RETURN NULL;
END;
$$ LANGUAGE 'plpgsql';

```

Caja 7.23. Archivo `ciudades_deportes_socios.sql`. Recorridos de consultas.

Este archivo se debería guardar en la carpeta de la tabla `deporte`. Si de una ciudad no hay nadie que practique un deporte concreto, entonces no aparecerá la fila correspondiente en la relación resultante. Esto es un tratamiento especial para cada elemento del producto cartesiano entre deportes y ciudades. De hecho, observa que los dos bucles de la función de la Caja 7.23 no hacen más que recorrer este producto cartesiano, y tratar el caso de cada pareja posible una por una con un `IF`.

Atención al tipo `BIGINT` que tiene la columna `socios` de la tabla resultante. Esto es para alojar el resultado de la función de agregación `COUNT(*)`. En inglés, *big* significa grande pero no de magnitud, sino de cosa. Cosa grande. Para los números, para decir que un número es grande se usa el adjetivo *large*. Este matiz en este par de adjetivos es un punto a favor de la lengua inglesa, ya que sin duda es más correcta que las latinas por lo que respecta a ese extremo. Un número es más bien largo que grande, ya que reside en una sola dimensión, la que cuantifica. Pero bueno, tampoco vamos a ir más allá. El uso del tipo `BIGINT`, contracción de *big integer*, en la función de la Caja 7.23 no significa pues que los números de socios tengan que ser números grandes,



sinó que el espacio de memoria que facilitamos para alojarlos sí que efectivamente será mayor que para un número entero cualquiera. Con todo, no tiene más trascendencia.

## 7.7 Disparadores

Los disparadores, *triggers* en inglés, son funciones que no se ejecutan cuando el usuario o programa cliente lo pide, sinó que van asociados a inserciones, borrados, o modificaciones de las tablas de la base de datos. Los parámetros que hay que establecer para crear un disparador son cinco.

1. La función. Lógicamente, hay que dar la función que se ejecutará. Debe ser una función de tipo `TRIGGER`. Una función es del tipo del valor que retorna. Si una función retorna un entero, es una función entera.
2. Aún así, el primer parámetro que caracteriza un disparador es la tabla en la que está declarado. Hay una dependencia de existencia del disparador con la tabla. Esto significa que como consecuencia de una orden `DROP TABLE tabla CASCADE` también se eliminan los disparadores que haya declarados, que no significa las funciones. La función de un trigger y el trigger mismo son objetos diferentes en los metadatos, con nombres diferentes.
3. El evento. Hay que establecer si el disparador se ejecutará cuando haya inserciones, modificaciones, o eliminaciones. O cualquier combinación de estos comandos.
4. Antes o luego que se produzca el evento.
5. El tipo de disparador, que puede ser `STATEMENT`, o bien `ROW`. Esto es, el disparador se ejecutará un vez para cada transacción, o un vez para cada evento. Mirando los scripts de inserción del capítulo anterior se ve que con una sola orden de inserción se pueden añadir múltiples registros. Y por tanto hay que decidir si se desea ejecutar el disparador un vez para cada transacción, o un vez para cada registro insertado.

### 7.7.1 Funciones de Tipo TRIGGER

Las funciones de tipo `TRIGGER` se ejecutan asíncronamente. Por tanto, nadie sabe cuando se ejecutarán. Nadie las llama. Y por tanto, no pueden tener parámetros de entrada. Por la misma razón, tampoco pueden retornar ningún valor, aunque ese extremo se matizará en breve.

Tenemos dos conjuntos. Uno formado por los eventos `{INSERT,UPDATE,DELETE}` y el otro formado por los modificadores antes y luego, `{BEFORE,AFTER}`. Pues bien, para

cada posible pareja de elementos formada por un elemento de cada conjunto hay la posibilidad de asociar un disparador.

Con la experiencia, quién desarrolla bases de datos aprende que los disparadores asociados a antes de los eventos acostumbran a practicar tareas de verificación de los valores que se van a insertar o a modificar. Los asociados a luego de los eventos, hacen tareas de publicación o notificación del hecho.

### Cabecera de las funciones TRIGGER

La cabecera de las funciones de este tipo tienen una estructura muy rígida. Siempre igual. En la Caja 7.24 se muestra la única estructura posible para todas las funciones de disparadores.

```
CREATE FUNCTION nombre_funcion() RETURNS TRIGGER AS $$
```

Caja 7.24. *Cabecera de una función de tipo TRIGGER.*

La única palabra que puede variar en la cabecera de la Caja 7.24 es el nombre de la función, que aquí se ha llamado `nombre_funcion`. Estas funciones se pueden codificar después de la creación de la tabla a la que van asociadas, en el mismo archivo. Y luego de su implementación, entonces se podrá declarar el disparador.

### Cuerpo de las funciones TRIGGER

Las funciones del PL/pgSQL, tan solo por el hecho de declarar que retornan un tipo `TRIGGER`, tienen disponibles dos variables. Bien, depende del evento al que el disparador se asocie finalmente. Comprender el sentido de estas variables es fundamental, ya que en última instancia son las únicas candidatas como posibles valores de retorno para este tipo de funciones.

- Disparadores asociados a inserciones.  
Siempre que una función se declara como disparador para inserciones en una tabla `t` dispone dentro de su ámbito de visibilidad de una variable con el nombre clave `NEW`. Esta variable es de tipo fila de la tabla, `T%ROWTYPE`. Su contenido son los valores del nuevo registro que se va a insertar. La función puede modificar esos valores antes de insertar el nuevo registro, o incluso anular la inserción.

- Disparadores asociados a modificaciones.  
Para el caso de los disparadores asociados a eventos de modificación se dispone de dos variables del mismo tipo fila de la tabla donde pertenecen esos disparadores. Ambos, con palabras clave `NEW` y `OLD`. La variable `NEW` funciona igual que para el caso de la inserción. La variable `OLD` contiene los valores que van a ser sobrescritos en la modificación.
- Disparadores asociados a eliminaciones.  
Siempre que asociamos un disparador a la eliminación en una tabla tendremos de la variable `OLD` en el código de la función adjunta a ese disparador. Contiene los valores del registro que está a punto de ser eliminado.

Estas variables solamente están disponibles para los row triggers, es decir, para los que se ejecutan un vez por cada registro afectado en la operación.

### Valores de retorno de los disparadores

Para los disparadores asociados antes de los eventos hay dos opciones. Si retornan `NULL` anulan el evento. Por ejemplo, para el caso de una inserción. Un disparador asociado a antes de una inserción en una tabla puede evitar que las inserciones se produzcan retornando un valor nulo, o puede indicar que la inserción proceda retornando la variable de tipo fila con el nombre clave `NEW`. Eso mismo sucede para el caso de las modificaciones. En cambio, si el disparador se asocia a antes de una eliminación, para evitar que se borre el registro también puede retornar un valor nulo, pero para proceder con la supresión debería retornar la variable de tipo fila `OLD`.

Todos los disparadores asociados a después de los eventos han de retornar `NULL` forzosamente.

#### 7.7.2 Sintaxis de Declaración de un Disparador

En esta sección se propone el desarrollo de un disparador que se ejecutará antes de hacer una inserción en la tabla `provincia`. Su función es asegurarse de que no se hayan producido errores de deletreo en la introducción del nombre. La forma de proceder es sencilla. Antes de hacer la inserción se mira si el nombre que se pretende insertar es equivalente canónicamente a alguno que ya esté en la tabla. Esto se hace tomando el valor resultante de la función `provincia_parecida()` de la Caja 7.22. Si lo es, anula la inserción retornando un valor nulo. Y si no, se indica que se prosiga retornando el valor `NEW`.

```

\echo ----- funcion verifica_provincia()

CREATE FUNCTION verifica_provincia() RETURNS TRIGGER AS $$
DECLARE
    nc TEXT; -- nombre correcto de la provincia
BEGIN
    nc = provincia_parecida(NEW.provincia);
    PERFORM * FROM provincia WHERE provincia = nc;
    IF FOUND THEN
        RETURN NULL;
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE 'plpgsql';

```

Caja 7.25. *Disparador antes de la inserción, para cada fila, en la tabla provincia.*

En la Caja 7.26 se muestra la declaración del disparador que servirá para corregir errores de deletreo en la introducción de nuevas provincias.

```

CREATE TRIGGER BEFORE INSERT ON provincia
FOR EACH ROW EXECUTE PROCEDURE verifica_provincia();

```

Caja 7.26. *Declaración de un disparador para la tabla provincia.*

El contenido de las Cajas 7.25 y 7.26 se puede poner a continuación de la misma creación de la tabla `provincia` ya que son conceptos totalmente vinculados.

Observa que a pesar todo, en ningún momento hemos establecido un formato canónico de interfaz, o lo que llamamos formatos normalizados. Es decir, no estaría de más implementar una función llamada `provincia_normalizada()` en la que se establecieran las reglas que se deseen para mostrar los nombres de las provincias a las aplicaciones cliente. Por ejemplo, se podría regular que los nombres de las provincias empiecen siempre con la primera letra mayúscula, y también en mayúscula cualquier palabra de más de tres letras que haya en el nombre de la provincia, es decir cualquier letra que prosiga un blanco y venga sucedida de más de dos letras. Esto, para filtrar los relativos, "de" o los artículos "la" o "el". Ese sería un formato normalizado, o canónico, de cánon.

## 7.8 Aplicaciones Cliente

Es importante darse cuenta que estrictamente, el contenido de esta sección no es propio de un libro de bases de datos. Si se desea estudiar un robot hay que hacerlo comprendiendo como realiza las acciones que se le piden. Esto acaba resultando en una colección de comandos que activen el robot para hacer alguna cosa. Si luego, además, queremos manipular el robot desde un programa informático, requerimos de quién ha construido el robot nos proporcione una interfaz de comandos para poderlas ejecutar desde el programa en java, por ejemplo. Pues eso es lo mismo para las bases de datos. El tema de las aplicaciones cliente para las bases de datos es exactamente igual que el tema de las aplicaciones cliente para cualquier otro tipo de sistema que se pueda manipular mediante un programa en lenguaje estándar. Es decir, todo se reduce a la librería. Cuando los sistemas que se ofrecen para ser manipulados informáticamente contienen mecanismos físicos y dispositivos materiales, entonces estas librerías reciben el nombre de *drivers*. De todas formas, también es frecuente llamar driver de postgres a esa librería.

La librería para poder trabajar desde un programa con una base de datos PostgreSQL la proporciona el mismo PostgreSQL, claro. Como siempre. La librería para poder manipular un robot también la proporciona el fabricante del robot. De manera que la única cosa que debe hacer el programa es incluir esta librería en la compilación. Entonces tiene disponibles las funciones a las que podrá llamar desde el código para ejecutar las acciones que desee.

El nombre de la librería de postgres para java, para el caso de la versión 9.2, tiene la forma `postgresql-9.2-1002-jdbc4.jar`. Este archivo se debe importar al proyecto java para poder utilizar las clases que se ven a continuación.

Cualquier conexión a una base de datos requiere un nombre de usuario y una contraseña. Incluso las aplicaciones cliente `GUI`. Para que un programa cliente pueda conectarse a una base de datos debe dar el nombre del usuario y la contraseña como parámetros de alguna función que conste en la librería.

### 7.8.1 Lenguaje Anfitrión y SQL Incrustado

Se entiende por lenguaje anfitrión aquel lenguaje de programación de alto nivel que permite la incorporación de comandos SQL. Los tres más usados son el java, el python y el php. Para poder ofrecer esta prestación se debe hacer uso de librerías que proporcionan las clases necesarias para ejecutar, esencialmente, tres tipos de interacción con la base de datos.

## Conexión

Para poderse conectar desde un programa cliente la librería del programa debe contener alguna función a la que se le pasen los parámetros de conexión. Normalmente la dirección internet de la computadora servidor, un código de usuario, y una contraseña. Asimismo, también se acostumbra a ofrecer una función de desconexión. En java, la clase que soporta estos métodos se llama `Connection`.

Existe un dilema frecuente. Muchos programadores se preguntan si conviene abrir una conexión cada vez que se hace una consulta, o bien hacerlo una sola vez cuando arranca el programa cliente y mantenerla abierta durante toda la ejecución. Incuestionablemente la respuesta es la primera opción. Hay que tener en cuenta que cualquier proceso informático con intervención humana de manera regular muy difícilmente podrá fluir en cuestiones de eficiencia. En otras palabras, los usuarios finales difícilmente notarán el tiempo adicional que supone abrir y cerrar la conexión en cada consulta. Y además, considerando que como administradores nos interesa poder controlar la base de datos, si utilizáramos la segunda opción difícilmente podríamos hacer el mantenimiento en exclusión mútua respecto los usuarios finales.

## Consultas de lectura (SELECT)

En las librerías de los SGBDs para los lenguajes de programación que implementan aplicaciones cliente se ofrecen clases cuyas instancias son objetos especialmente dedicados a las consultas de lectura. Se llaman objetos de tipo `RecordSet`, y forman parte de las librerías ODBC, de *Object Data Base Connection*, que en java se llama JDBC. Normalmente se acostumbran a utilizar dos métodos de esta clase de objetos. Uno que retorna un booleano, `next()`. Llamar a ese método nos retornará cierto mientras haya resultados por obtener, aumentando el cursor interno privado de la clase para prepararse para la próxima llamada. Y retornará falso cuando se acaben los resultados de la consulta efectuada. Hay también otro para obtener el valor de una columna del registro actual, `getColumn(int c)`. Así pues, resulta previsible que las funciones o procedimientos realizados en el lenguaje anfitrión deberán implementar recorridos con variables de ese tipo para almacenar los resultados provenientes de PostgreSQL en las estructuras de datos internas de esos programas.

## Consultas de actualización (INSERT, UPDATE y DELETE)

Para las consultas de actualización hay el método `executeQuery(String str)` de la clase `Statement`. Este método recibe una cadena de caracteres que contiene la sentencia SQL que se pretende ejecutar. Y retorna un número entero, el mismo que retorna PostgreSQL cuando la conexión es vía `psql`. Es decir, la cantidad de filas afectadas por la consulta de actualización.

## 7.9 Administración de la Base de Datos

En esta sección se pretende conocer las órdenes básicas que hay que utilizar para poder trabajar con PostgreSQL tanto desde el sistema operativo MS-DOS como de los sistemas linux. Eso incluye la consulta de si el servicio está activo, la parada, y el arranque del sistema. Después se ven las órdenes para los backups y de mantenimiento del clúster, y finalmente se habla de algunos parámetros de configuración. Con todo ello tan solo se pretende familiarizar al lector en esas cuestiones, sin profundizar demasiado en ellas.

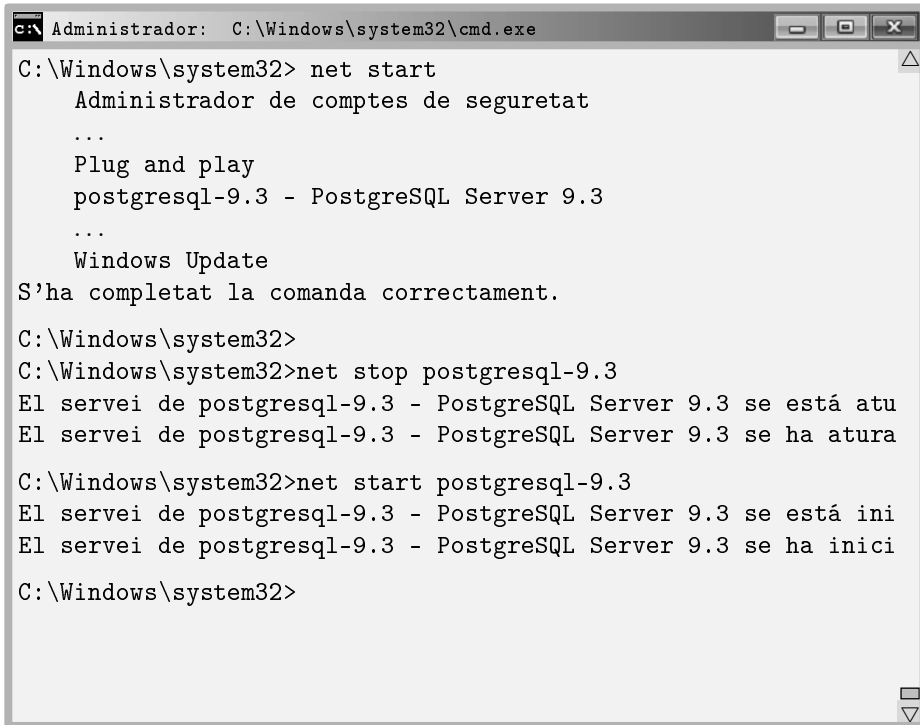
### 7.9.1 Parada e Iniciación del Sistema

Veamos los procedimientos en los diferentes sistemas operativos para llevar a cabo las operaciones más importantes que se pueden hacer con un SGBD, pararlo e iniciarlo. Como estamos hablando de un servicio, lo normal es que esté activo, ya que se inicia automáticamente cuando lo hace la computadora. Por ese motivo, el orden en que se presentan las operaciones parte de esa base.

#### MS-DOS

Lo primero, abrimos una consola con permisos de administración. Una manera de hacer eso es desde el menú *inicio* escribiendo `cmd` en la caja *ejecutar*: y pulsando las tres teclas, control y mayúsculas, cuando se hace intro. Si se tienen permisos de administración, el sistema pide confirmación sombreando la pantalla. Aceptamos, y estamos dentro.

De entrada, podemos consultar si el PostgreSQL está iniciado con el comando `net stat`. Esto provocará una hemorragia de servicios por el monitor. Con la barra deslizante podemos buscar `postgresql-9.3 - PostgreSQL Server 9.3`. Detenemos PostgreSQL con el comando `net stop postgresql-9.3`, y lo volvemos a iniciar con `net start postgresql-9.3`. Todo ello se describe en la Pantalla 7.25, habiéndose abreviado la lista de servicios.



```
Administrador: C:\Windows\system32\cmd.exe
C:\Windows\system32> net start
Administrador de comptes de seguretat
...
Plug and play
postgresql-9.3 - PostgreSQL Server 9.3
...
Windows Update
S'ha completat la comanda correctament.

C:\Windows\system32>
C:\Windows\system32>net stop postgresql-9.3
El servei de postgresql-9.3 - PostgreSQL Server 9.3 se está aturando
El servei de postgresql-9.3 - PostgreSQL Server 9.3 se ha aturado

C:\Windows\system32>net start postgresql-9.3
El servei de postgresql-9.3 - PostgreSQL Server 9.3 se está iniciando
El servei de postgresql-9.3 - PostgreSQL Server 9.3 se ha iniciado

C:\Windows\system32>
```

Pantalla 7.25. Verificación de servidor iniciado, parada e inicio.

## Línx

Desde una terminal línx hay una manera rápida, aunque que poco precisa, de saber si PostgreSQL está iniciado en el servidor. La instrucción `top`. En la Pantalla 7.26 se muestra una ejecución.



```

root@ubuntu:~
root@ubuntu:~# top
top -19:34:41 up 26 min, 2 users, load average: 0.02, 0.04, 0.13
Tasks: 73 total, 1 running, 72 sleeping, 0 stopped, 0 zombie
Cpu(s): 0.3%us, 3.0%sy, 0.0%ni, 95.7%id, 0.0%wa, 0.0%hi, 0.1%si, 0.0%st
Mem: 766936k total, 169352k used, 597584k free, 18108k buffers
Swap: 784380k total, 0k used, 784380 free, 82140 cached

  PID USER      PR  NI  VIRT  RES  SHR  S  %CPU  % MEM   TIME+  COMMAND
 1956 root       20   0   2720 1096  880  R   1.9   0.1   0:24.55 top
     4 root       20   0     0    0    0   S   0.6   0.0   0:06.92 kworker/0
 1516 postgres  20   0 50484 1568  584  S   0.6   0.2   0:10.68 postgres
 1517 postgres  20   0 50484 1324  352  S   0.6   0.2   0:10.79 postgres
 1776 root       20   0  9648  3136 2520  S   0.6   0.4   0:07.06 shd
root@ubuntu:~# _

```

Pantalla 7.26. Verificación de servidor iniciado con la utilidad `top`.

Este comando nos proporciona el mismo tipo de información que el administrador de tareas en windows. Esto es, para cada proceso podemos ver el uso del procesador en tanto por ciento del tiempo, cantidad de memoria que utiliza el proceso, cantidad de memoria de paginación, y otros indicadores.

Si se desea tener una idea más clara de los cuatro procesos que forman el SGBD se puede mirar con el comando `proces status`, `ps auxww`. Y para evitar un alud de información, se puede usar el comando `grep` que busca las cadenas que se le pasen utilizando expresiones regulares. En concreto, para expresar la condición de inicio de palabra, las expresiones regulares utilizan el acento circunflejo.

Entre el `ps` y el `grep` hay que hacer un *pipe*. Pipear la salida de un proceso como entrada de otro significa que el segundo va a trabajar con la salida del primero como archivo de entrada. En linux eso se indica con una barra vertical entre las llamadas a los programas.

Por tanto, con el comando `ps auxww | grep ^ postgres` conectamos la salida del `ps auxww` como entrada al programa `grep` al cual le pedimos que tan solo nos muestre aquellas líneas donde aparezca la palabra `postgres` como principio de palabra, es decir, luego de un separador. Para parar y arrancar el servidor PostgreSQL desde linux tenemos los comandos `service postgresql stop`, y `service postgresql start`. En la Pantalla 7.27 se muestra con todo.

```

root@ubuntu:~
root@ubuntu:~# ps auxww | grep ^ postgres
postgres  2667  1.2  1.0  50504  8048 ?        S   00:36  0:03  /usr/lib/postgr
ql/9.1/bin/postgres -D /var/lib/postgresql/9.1/main/postgresql.conf
postgres  2670  0.8  0.2  50488  1568 ?        Ss  00:36  0:02  postgres: writ
proces
postgres  2672  0.1  0.3  50912  2444 ?        Ss  00:36  0:00  postgres: auto
uum launcher proces
postgres  2673  0.1  0.1  20736  1388 ?        Ss  00:36  0:00  postgres: sdad
ollector launcher proces
root@ubuntu:~# service postgresql stop
  Stopping PostgreSQL 9.1 database server                    [0]
root@ubuntu:~# service postgresql start
  Starting PostgreSQL 9.1 database server                    [0]
root@ubuntu:~# _

```

Pantalla 7.27. Verificación de servidor iniciado con `ps`, parada e inicio.

Con esta segunda opción vemos el proceso principal, *writer proces*, el proceso dedicado a la replicación que escribe los archivos *write ahead log*, el disparador que arranca el regenerador, *autovacuum*, y el recolector de estadísticas.

## El programa `pg_ctl`

Internamente, las formas utilizadas en las Pantallas 7.25 y 7.27 para iniciar o parar el servidor ejecutan el programa `pg_ctl` con los parámetros predefinidos para las llamadas automáticas. Si se desea variar alguna de estas opciones, entonces se debe invocar al programa directamente. Por ejemplo, a la hora de pararlo, existe la posibilidad de hacerlo con la opción *smart*, que significa que se espere a que los programas clientes se desconecten, hacerlo con la opción *fast*, que espera que se acaben las transacciones que puedan estar realizándose, y la opción *immediate*, que para abruptamente el servidor, y requiere hacer una restauración cuando se vuelva a iniciar.

El programa `pg_ctl`, además de *start* y *stop*, también tiene las opciones de *restart*, y *reload*. Esta última tan solo vuelve a cargar el archivo de configuración.

Este programa requiere de un parámetro cuyo contenido debe ser el nombre del directorio donde se encuentran los archivos de configuración, `-D`. O sea que desde el sistema operativo se invoca con el comando

```
pg_ctl -D "C:\Program Files (x86)\PostgreSQL\9.3\data"
```

para el caso de MS-DOS. Si no se pasa este parámetro en la llamada al programa, PostgreSQL también puede obtener esta información a partir de una variable de sistema, como se verá más adelante.

## 7.9.2 Mantenimiento y Copias de Seguridad

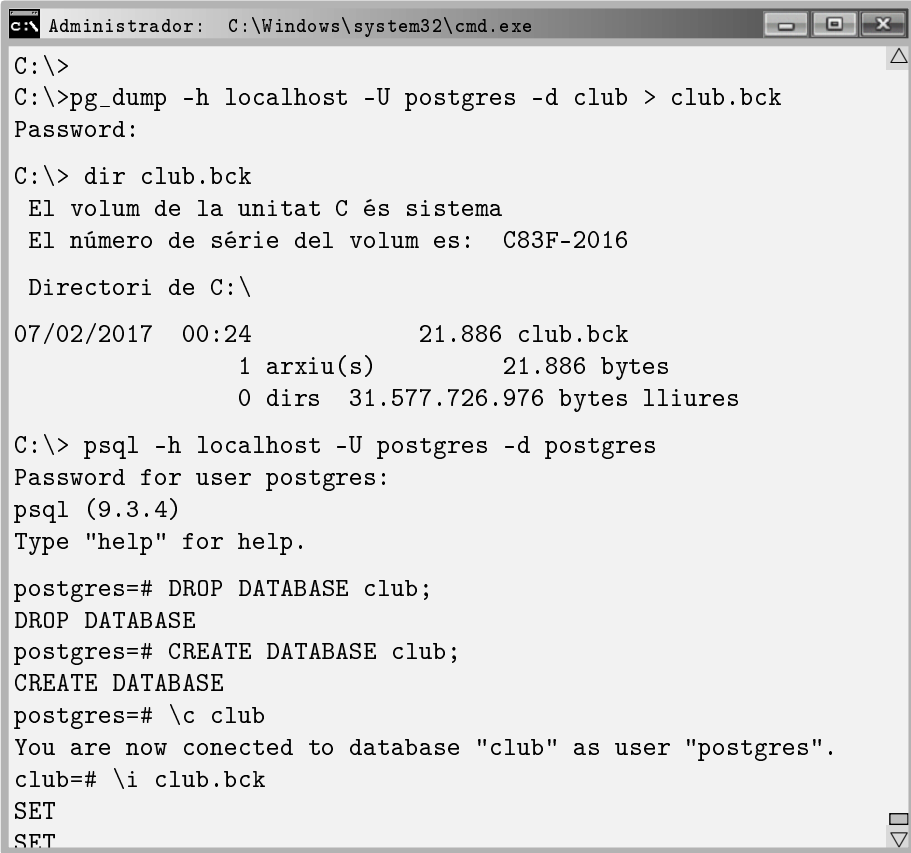
La manera más comprensible de hacer una copia de seguridad es con el programa cliente `pg_dump`. Se le pasan los parámetros de conexión de la misma manera que al `psql`, y se añade un redireccionamiento de la salida a un fichero. Al terminar la ejecución tenemos un archivo de texto con la secuencia de comandos SQL que restauran la base de datos al mismo punto en el que estaba al ejecutar el comando `pg_dump`. Hagámoslo con un experimento de seis pasos.

1. Desde una consola con permisos de administración, situados en la carpeta de donde cuelga la del proyecto club, hacemos un backup de la base de datos con el programa `pg_dump`, redireccionando la salida al fichero `club.bck`. La instrucción introducida al sistema operativo es

```
pg_dump -h localhost -U postgres -d club > club.bck
```

2. Comprobamos que existe el nuevo archivo `club.bck` en el directorio actual con el comando `dir` de MS-DOS. También resulta interesante abrirlo con cualquier editor de texto plano y hacer una ojeada a su contenido, que en gran parte consiste en el mismo escript de creación que habíamos compilado en capítulos anteriores.
3. Nos conectamos a PostgreSQL como siempre, con `psql -h localhost -U postgres -d postgres`.
4. Eliminamos la base de datos club con el comando de siempre, `DROP DATABASE club` sin miedo, confiando en el backup. Claro, si tenemos los escripts de creación y de inserción eso no tiene tanta gracia.
5. Creamos la base de datos club, y nos conectamos de la manera que lo hace el escript principal, `CREATE DATABASE club, y \c club`.
6. Importamos el archivo `club.bck` también igual que con cualquier otro archivo.

Finalmente, nos encontramos en el mismo estado que al principio. La incidencia de cada uno de los pasos se puede comprobar añadiendo comandos al experimento. Pero vaya, ya se puede intuir que a partir de aquí podemos implementar cualquier política de copias de seguridad. En la Pantalla 7.28 se reproduce el diálogo de estos seis pasos.



```

Administrador: C:\Windows\system32\cmd.exe
C:\>
C:\>pg_dump -h localhost -U postgres -d club > club.bck
Password:
C:\> dir club.bck
El volum de la unitat C és sistema
El número de sèrie del volum es: C83F-2016
Directori de C:\
07/02/2017  00:24                21.886 club.bck
                1 arxiu(s)                21.886 bytes
                0 dirs  31.577.726.976 bytes lliures
C:\> psql -h localhost -U postgres -d postgres
Password for user postgres:
psql (9.3.4)
Type "help" for help.

postgres=# DROP DATABASE club;
DROP DATABASE
postgres=# CREATE DATABASE club;
CREATE DATABASE
postgres=# \c club
You are now conected to database "club" as user "postgres".
club=# \i club.bck
SET
SET

```

Pantalla 7.28. *Copia de seguridad al archivo C:\club.bck.*

El ejemplo de la Pantalla 7.28 es la mínima expresión para poder hacer un backup de la base de datos. Con las políticas usadas para las copias de seguridad hay un abanico de posibilidades con diferentes niveles de protección. Se distingue entre backups totales, e incrementales. Es decir los que realizan una copia de la base desde cero, y los que lo hacen a partir del último backup realizado. Para los tipos incrementales de backup, la restauración puede ser complicada, ya que se debe reseguir la secuencia hecha desde el último backup total.

Realizar backups cíclicos robustece la metodología usada. Normalmente, en cada instalación se determina la antigüedad con la que se desea mantener los datos, por ejemplo cinco años, y entonces se establece un formato de nombres de archivos que contiene la fecha del backup, como `club-01-01-2017.bck`. Hacerlo cíclicamente sirve para asegurarse de que la cantidad de memoria secundaria que se desea dedicar a esta finalidad se mantenga más o menos estable. Para el caso del ejemplo, en el momento de escribir el archivo `club-01-01-2020.bck` se borraría automáticamente el `club-01-01-2015.bck`.

Hay otras maneras más sofisticadas para gestionar copias de seguridad de la base de datos. Probablemente la más innovadora consista en la replicación. La filosofía es que un servidor haga el papel de gestor del clúster, y otros permanecen copiando el archivo de log que el servidor principal va emitiendo.

El atributo `REPLICATION` que tiene el usuario `postgres` en la Pantalla 7.2, a principios de este capítulo, en la página 275, está relacionado con la replicación de bases de datos. Se trata de un sistema de copias múltiples que a partir de un servidor principal se copian en una cantidad cualquiera de servidores *estantes*. Esto se hace con ficheros de extensión `WAL`, *de write ahead log*. Y un usuario con permiso de replicación puede acceder a estos archivos, que normalmente se guardan en dispositivos de memoria secundaria de los servidores estantes, o sea, los secundarios. La diferencia entre los dos tipos de servidor es que el único con capacidad para modificar el contenido de los datos es el servidor principal. Los otros pueden hacer consultas de lectura en las copias que han conseguido mediante estos ficheros. Todo eso requiere mecanismos de sincronización.

### 7.9.3 Configuración de un servidor PostgreSQL

El archivo `postgres.conf`, así como los otros dos archivos de configuración de un servidor PostgreSQL, son archivos de texto plano. Así, pueden editarse con los editores más sencillos de cada sistema operativo, `notepad` para windows o `le`, o `vi`, para los sistemas linux. Son tres archivos. El principal, el de autenticación basada en el host, y el de mapeo de usuarios. En todos ellos se utiliza el símbolo numeral, `#`, para introducir comentarios de línea.

#### Archivo de parámetros de configuración

El nombre del archivo principal de configuración del SGBD PostgreSQL es `postgresql.conf`. Y el directorio por defecto, claro, depende del sistema operativo. En cualquier caso, estando conectados con el `psql` con permisos de superusuario, siempre se puede consultar cuál es el archivo de configuración que se está utilizando mediante el comando `show config_file`. Esa acción se muestra en las Pantallas 7.29 y 7.30.



```

c:\> psql -h localhost -U postgres -d postgres
postgres=# show config_file;
              config_file
-----
C:/Program Files (x86)/PostgreSQL/9.3/data/postgresql.conf
(1 row)

```

Pantalla 7.29. Identificación del archivo de configuración desde MS-DOS.



```

root@ubuntu:~
postgres=# show config_file;
              config_file
-----
/etc/postgresql/8.1/main/postgresql.conf
(1 row)

```

Pantalla 7.30. Identificación del archivo de configuración desde sistemas *linux*.

Observa por el prompt que hay que ser superusuario.

Este archivo contiene los parámetros de configuración. Cada línea tiene una estructura **nombre = valor**, siendo el nombre el parámetro en cuestión que puede ser de tipo booleano, entero, real, cadenas o enumeraciones. Inicialmente, los parámetros están comentados y además, contienen los valores por defecto. Esto obliga a pensárselo dos veces antes de cambiar cualquier valor. Cuando se descomenta, y cuando se da el valor. En el archivo `postgresql.conf`, para cada parámetro se comenta si su modificación implica reinicio del sistema para que surja efecto.

Cada vez que nos conectamos a una base de datos, ya sea desde el sistema operativo con el programa `psql`, o desde dentro del mismo haciendo `\c`, se leen todos los parámetros del fichero, y un vez leídos quedan en la tabla `pg_settings` del SGBD. En esa tabla, las dos primeras columnas son el nombre del parámetro, `name`, y el valor actual, `setting`.

También podemos utilizar la función `pg_reload_conf()` a fin de leer el fichero de configuración de forma asíncrona, haciendo `SELECT pg_reload_conf()`.

El archivo `postgresql.conf` está segmentado en secciones. En la primera se dice el lugar para los ficheros, y el primero de todos los parámetros es precisamente el nombre del directorio donde se encuentra el archivo de configuración, `data_directory`. Parece contradictorio que dentro del archivo de configuración haya un parámetro, el primero de todos, que dice una información que hay que saber previamente para poder acceder a ella. Y de contradictorio no tiene nada. Esta información puede ser utilizada por las funciones que haya en el SGBD, o también por las aplicaciones cliente que se puedan conectar a este servidor. La función de este parámetro es pues, de espejo. Y por tanto, para iniciar el PostgreSQL hay que dar valor al parámetro `data_directory` de alguna otra manera.

Una posibilidad es en la línea de comandos con la opción `-D` cuando se inicia con el programa `pg_ctl`, tal como se ha visto en la Sección 7.9.1. Y la otra posible manera de hacerlo es declarando una variable en el sistema operativo con el nombre clave `PGDATA`.

Si se cambia el valor de este parámetro en el archivo de configuración hay que reiniciar el servidor para que tenga efecto. No hay bastante con una simple relectura del archivo.

Además, dentro de esta misma sección del archivo `postgresql.conf`, de lugares para los ficheros, también hay la ruta de los otros dos archivos de configuración.

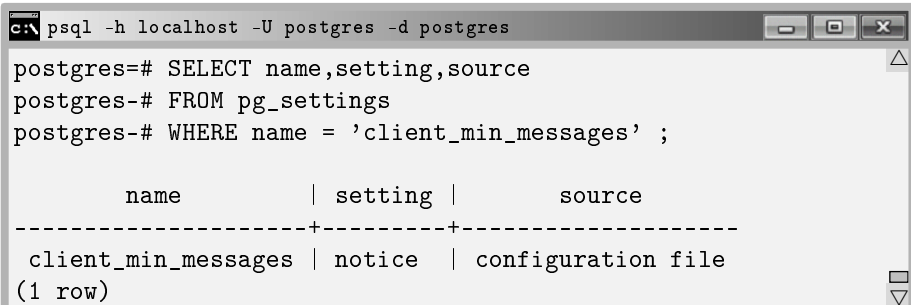
En la siguiente sección del archivo de configuración, conexiones y autenticación, hay el parámetro `listen_addresses` que sirve para restringir las direcciones de internet de las computadoras cliente para las cuales se aceptan conexiones. Es habitual asignarle el valor `*` indicando que se aceptan conexiones desde todas partes. En esta sección también hay los parámetros de conexión que aceptamos por defecto en el momento de la instalación. O sea, el puerto, y el máximo número de conexiones.

No es propósito de esta sección explicar el significado de cada parámetro. Además, muchos de ellos resultarían difíciles de comprender para quién no tenga un conocimiento más o menos profundo de administración de redes. No obstante, sí que resulta interesante comprender qué prioridades hay cuando se modifican los valores de los parámetros. Para esto, tomamos de ejemplo el parámetro `client_min_messages`. En la Caja 7.27 se muestra la consulta que nos permite saber quién ha establecido el valor de un parámetro.

```
SELECT name,setting,source
FROM pg_settings
WHERE name = 'client_min_messages';
```

Caja 7.27. Consulta del valor y el origen de un parámetro.

Con esta consulta podemos averiguar el valor que tiene un parámetro y quién ha puesto este valor. El resultado obtenido se describe en la Pantalla 7.31.



```
c:\> psql -h localhost -U postgres -d postgres
postgres=# SELECT name,setting,source
postgres=# FROM pg_settings
postgres=# WHERE name = 'client_min_messages' ;
```

name	setting	source
client_min_messages	notice	configuration file

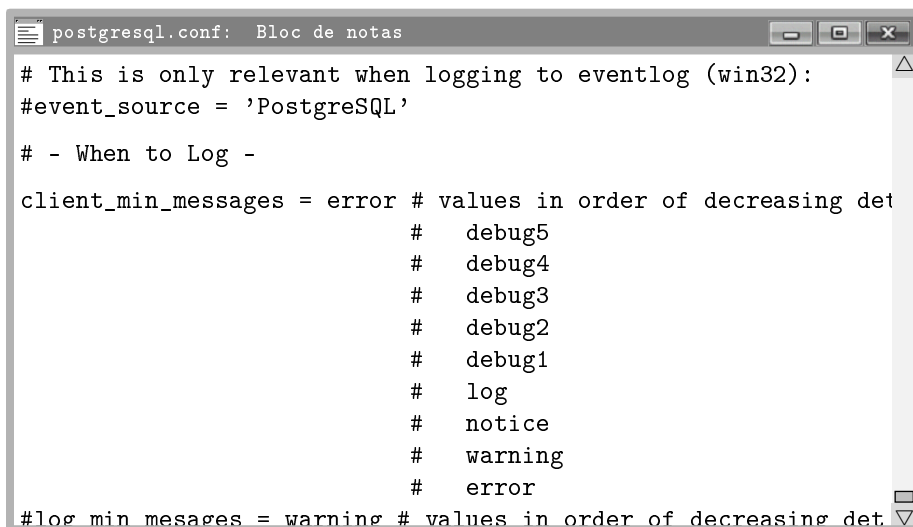
```
(1 row)
```

Pantalla 7.31. Valor y origen del parámetro `client_min_messages`.

El valor que prevalece para los parámetros de configuración viene determinado por el origen desde donde se haya asignado, cosa que se puede ver en la columna `source` de la tabla `pg_settings`.

Con el mismo parámetro `client_min_messages` hacemos un experimento de siete pasos.

1. Modificamos con algún editor de texto el archivo de configuración. Establecemos el valor del parámetro igual a `error`. Para esto, tenemos que descomentar la línea correspondiente y modificar el valor que hay luego del signo igual. La situación después de haber hecho el cambio se muestra en la Pantalla 7.32.



```

postgresql.conf: Bloc de notas
# This is only relevant when logging to eventlog (win32):
#event_source = 'PostgreSQL'

# - When to Log -

client_min_messages = error # values in order of decreasing det
                        #   debug5
                        #   debug4
                        #   debug3
                        #   debug2
                        #   debug1
                        #   log
                        #   notice
                        #   warning
                        #   error

#log_min_messages = warning # values in order of decreasing det

```

Pantalla 7.32. *Modificación de un valor en el archivo postgresql.conf.*

2. Releemos el archivo de configuración con la función `pg_reload_conf()`.
3. Volvemos a mirar el valor que tiene el parámetro con el comando de la Caja 7.27.
4. Establecemos un nuevo valor para el parámetro desde la misma línea de comandos del `psql`, tecleando `SET client_min_messages = debug;`
5. Volvemos a mirar el valor que tiene el parámetro pulsando dos veces la flecha arriba. Observamos el cambio en la columna `source`.
6. Reconectamos con la base a la que estemos conectados, tecleando `\c`.
7. Volvemos a mirar el valor que tiene el parámetro pulsando otra vez dos veces la flecha arriba. Observamos el cambio en la columna `source`.

En la Pantalla 7.33 se muestran los últimos pasos del experimento.



```

c:\> psql -h localhost -U postgres -d postgres

postgres=# SELECT pg_reload_conf();
 pg_reload_conf
-----
 t
(1 row)

postgres=# SELECT name,setting,source FROM pg_settings WHERE name = 'client_min_messages ;
      name      | setting | source
-----+-----+-----
 client_min_messages | error   | configuration file
(1 row)

postgres=# SET client_min_messages = debug;
SET
postgres=# SELECT name,setting,source FROM pg_settings WHERE name = 'client_min_messages ;
      name      | setting | source
-----+-----+-----
 client_min_messages | debug   | sesion
(1 row)

postgres=# \c
You are now connected to database "postgres" as user "postgres".
postgres=# SELECT name,setting,source FROM pg_settings WHERE name = 'client_min_messages ;
      name      | setting | source
-----+-----+-----
 client_min_messages | error   | configuration file
(1 row)

postgres=# _

```

Pantalla 7.33. Orígenes y valores del parámetro `client_min_messages`.

Un último parámetro que puede traer problemas de sincronización entre el servidor y los clientes, y por eso hay que tenerlo en cuenta, es `log_timezone` que aquí en Ecuador acostumbra a valer `UTM-05`. Se utiliza a la hora de traducir entre los tipos temporales del SQL.

Observando más cuidadosamente al archivo de configuración es notable que la mayor parte de los parámetros están comentados. Los otros, es decir los activos, se establecieron en el momento de la instalación del PostgreSQL.

## Archivo de autenticación basada en el host

El archivo `pg_hba.conf` es el de la configuración de la autenticación basada en el host, *host based authentication*. Establece una asociación entre el formato físico de la dirección internet que se conecta y el tipo de contraseña que se le solicitará. Este tipo de archivo tiene un muy bajo índice de volatilidad. Es habitual establecer los valores que sean necesarios en el momento de la instalación de la aplicación en la red, y ya no volverlo a modificar, o hacerlo muy excepcionalmente.

Estando conectados al servidor con el `psql`, podemos consultar su ubicación en el host preguntando por el valor de la variable de sistema que lo guarda, `hba_file`, tal como se muestra en las Pantallas 7.34 y 7.35, según el sistema operativo que se utilice.



```

c:\> psql -h localhost -U postgres -d postgres
postgres=# show hba_file;
                hba_file
-----
C:/Program Files (x86)/PostgreSQL/9.3/data/pg_hba.conf
(1 row)

```

Pantalla 7.34. Identificación del archivo de autenticación basada en el host desde *MS-DOS*.



```

root@ubuntu: ~
postgres=# show hba_file;
                hba_file
-----
/etc/postgresql/8.1/main/pg_hba.conf
(1 row)

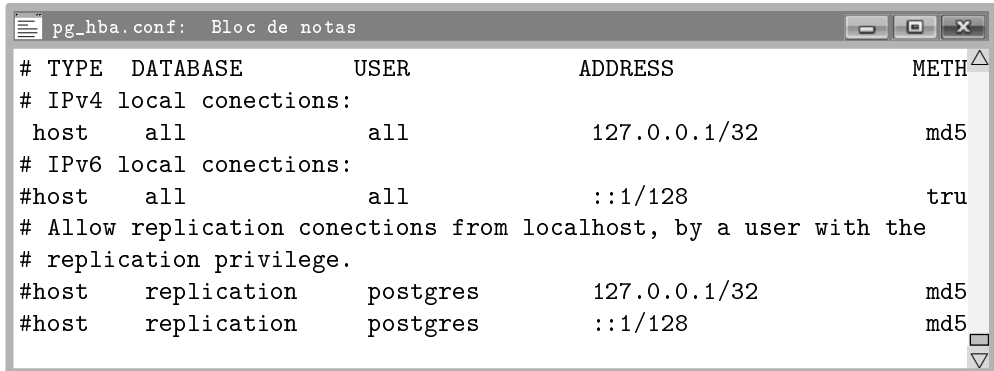
```

Pantalla 7.35. Identificación del archivo de autenticación basada en el host desde *linux*.

Los registros de este archivo ocupan un línea. Cinco columnas. Para comprender en profundidad la funcionalidad de cada una de ellas hay que dominar conceptos propios de sistemas operativos más que de bases de datos. Elementos definidos en capas muy nucleicas de los sistemas operativos como la sincronización entre procesos, el acceso compartido a recursos, el tratamiento de exclusiones mutuas y otros conceptos de la ingeniería.

Por tanto, aquí se expondrá un mínimo de contenido referente aquello que tiene incidencia cuando pretendamos poner la aplicación en producción. En la Pantalla 7.36

se volca el contenido de la parte final de este archivo. Toda la parte inicial son comentarios.



```

pg_hba.conf: Bloc de notas
# TYPE DATABASE USER ADDRESS METH
# IPv4 local connections:
host all all 127.0.0.1/32 md5
# IPv6 local connections:
#host all all ::1/128 tru
# Allow replication connections from localhost, by a user with the
# replication privilege.
#host replication postgres 127.0.0.1/32 md5
#host replication postgres ::1/128 md5

```

Pantalla 7.36. Archivo de configuración de la autenticación basada en el host.

Si le echáis una ojeada, veréis que el archivo `pg_hba` tiene poco menos de cien líneas, de las cuales las que no están comentadas se pueden contar con los dedos de una mano. Es decir, este archivo está comentado con cuidado, explicando su estructura. Lo que se ve seguidamente no es más que la traducción de algunas de las partes iniciales de este archivo.

Desde MS-DOS, todos los registros tienen cinco columnas. La primera, `TYPE`, indica el tipo de conexión al que hace referencia esta regla de autenticación. Los valores que puede contener es conexión vía internet, `IPv4` y `IPv6`, o conexión desde computadoras conectadas directamente a la del servidor, `host`. En linux, esta columna acepta un valor adicional, `local`, que representa las conexiones hechas desde la misma computadora donde reside el servidor, y que por tanto se pueden hacer en capas inferiores de la OSI, ya sea utilizando *sockets* o otros mecanismos de comunicación entre procesos. Cuando se utiliza este tipo de conexión exclusiva de los sistemas linux, entonces no hay la columna `ADDRESS` en la línea correspondiente.

La segunda columna dice a qué bases de datos se pueden conectar los usuarios que utilicen este tipo de conexión, como se ve en la Pantalla 7.36 podemos utilizar la palabra clave `all` para decir a todas las bases de datos del clúster. Observa en las dos últimas líneas del archivo, aunque estén comentadas, que como base de datos también se puede poner la palabra clave `replication`. Esto está relacionado con lo que se ha dicho en la Sección 7.9.2, aunque que se ha visto bien poco.

La tercera columna restringe los usuarios PostgreSQL que pueden conectarse de la manera indicada en esta línea. También se puede poner `all`, y si efectivamente se desea restringir el acceso a un conjunto concreto de usuarios, se puede poner el nombre de un archivo de texto plano. Entonces, en coherencia, en el mismo directorio

que indica el parámetro de configuración `data_directory` hay que figurar el archivo con un nombre de usuario en cada línea, y nada más.

La columna `ADDRESS` restringe las direcciones de internet que pueden conectarse. Esto explica la dependencia funcional con la primera columna. En el ejemplo de la Pantalla 7.36 se ve que la única línea descomentada permite el acceso desde cualquier computadora que se encuentre en la misma red local que la del servidor. Esto significa, en el caso más sencillo, todas las computadoras que compartan el router. Esta es la configuración para trabajar con el localhost.

Y la última columna describe el algoritmo de verificación o encriptación de contraseñas que se utilizará para este tipo de conexiones. El `md5` es uno de los más utilizados. Hay muchos otros. Y si tenemos plena confianza, podemos poner la palabra clave `trust`, y entonces para este tipo de conexión no se pedirá contraseña. Esto puede resultar muy interesante cuando no recordemos alguna.

### Archivo de mapeo de usuarios

Como se ha dicho en la Sección 6.1, si al establecer una conexión a PostgreSQL no se da el nombre de usuario con el parámetro `-U`, entonces se interpretará que el nombre del usuario PostgreSQL es el mismo que el del sistema operativo.

El archivo `pg_ident.conf` sirve para establecer asociaciones entre usuarios del sistema operativo y del SGBD.

De los tres archivos de configuración, sin duda este es el menos importante, ya que en la mayor parte de las instalaciones no se utiliza. Lo puedes consultar igual que los otros dos archivos, y verás que está completamente comentado. Para usarlo hay que dar nombres clave que identifiquen los mapas de transformación, es decir las asociaciones. Y finalmente, en el archivo de configuración `pg_hba.conf` se le debe decir que para el tipo de conexión que se desee se utilice el mapa de transformación de los nombres de los usuarios del sistema operativo, a los nombres de los usuarios PostgreSQL.

---

*Lo deseable luego de haber seguido este último capítulo es que el lector se atreva a investigar por él mismo si no todo, al menos una parte bastante significativa de lo que permite un sistema gestor de bases de datos. A partir de los instrumentos suministrados no debería haber problema para completar la aplicación de ejemplo que se ha ido observando a lo largo de todo el libro. Es importante saber consultar la ayuda, y saber organizarse el espacio de trabajo reservando la carpeta principal como banco de pruebas. Todo el conocimiento adquirido respecto al lenguaje procedural puede servir asimismo para descubrir más cosas. No es un lenguaje demasiado extenso y sin mucho esfuerzo se pueden conocer todas sus prestaciones. Una vez adquirida cierta fluidez, la implementación de disparadores no tiene más secreto. Por otra parte, se ha dado a entender que el desarrollo de aplicaciones cliente que hagan las tareas de interfaz no debe suponer ningún esfuerzo para una persona que desarrolle código en lenguajes de alto nivel. Y se ha cerrado el capítulo presentando la manera de funcionar de los parámetros de configuración del PostgreSQL.*

- Gracias por vuestra atención -



# Apéndice A

## Estilo

Tal como se explica en el preámbulo del libro, el diseño y la implementación de bases de datos es una teoría abierta. Eso significa que además de los procedimientos que hay que efectuar para completar un proyecto nos queda aún la posibilidad de hacerlo de distintas formas. En este apéndice se detallan las normas que se siguen a lo largo del libro respecto a aspectos menores que no tienen impacto funcional, pero de cara a la legibilidad del código resulta conveniente establecerlas.

### A.1 Modelo Entidad Relación

Los identificadores que se plasman para las relaciones en un modelo ER son exclusivamente los que trascienden en la implementación. Eso hace que las relaciones 1:N se representen como simples flechas.

- Todos los identificadores que aparecen en los diagramas ER ya sea de entidades, relaciones o atributos están en ASCII de siete bits. Eso significa que no hay tildes, ni diéresis, ni eñes.
- Los identificadores no tienen espacios en blanco, o sea, en la mayor parte de los casos constan de una sola palabra. En caso de necesidad, se pone un guión bajo para separar palabras.
- Las entidades se identifican con sustantivos en singular, excepto para el caso de entidades débiles. Se representan en mayúsculas en los diagramas.
- Cuando hay que poner nombre a relaciones se utilizan verbos en tercera persona del singular del presente de indicativo, también en mayúsculas.

- Los nombres de los atributos se ponen en minúscula.
- No se pone en ningún caso el tipo de un atributo en su nombre. Un atributo que guarda una fecha inicial se llama `inicio`, y no `fecha_inicio`.
- Los identificadores de atributos que por nombre tienen un sustantivo en singular se acostumbran a implementar en strings.
- Los identificadores de atributos que por nombre tienen un sustantivo en plural representan cantidades que se acostumbran a implementar en enteros.
- También se utilizan sustantivos en plural en nombres de entidades débiles y de atributos multivalorados.

## A.2 Modelo Relacional

Tanto los identificadores de relaciones como los de atributos se escriben en minúsculas, y en fuente de máquina de escribir. Los valores constantes, por ejemplo los datos, se escriben en *cursiva*.

## A.3 Lenguaje Procedural

Los nombres de las funciones tienen el formato `funcion_de_variable(v)`. Los parámetros de las funciones, así como las variables locales tienen nombres cortos, normalmente formados por las iniciales de lo que describen.

## A.4 Versiones de los Sistemas Utilizados

Para el desarrollo de la mayor parte del libro se ha utilizado el sistema operativo Windows 10. A menudo, se hacen referencias a aspectos exclusivos de este sistema sin hacer explícito el hecho. Igualmente, todo el código implementado en los últimos capítulos trabaja con la versión 9.3 de PostgreSQL cuando se usa en MS-DOS, y la versión 8.1 en Linux. Sin embargo, para que no resulte tediosa la lectura con esos detalles, se le supone al lector la capacidad de comprender lo que se explica aunque realice sus experimentos en otras versiones, así como discernir este hecho y atribuir a la diferencia de versiones los funcionamientos en los que aparezca alguna diferencia en el diálogo o en el comportamiento.



# Apéndice B

## Códigos ASCII

En la Tabla B.1 se muestra el conjunto de 95 caracteres imprimibles, con los códigos correspondientes, establecido en la American Standard Code for Interchange Information. El primero de todos ellos es el espacio en blanco con el código 32.

		32	!33	"34	#35	\$36	%37	&38	'39
(40	)41	*42	+43	,44	-45	.46	/47	048	149
250	351	452	553	654	755	856	957	:58	;59
<60	=61	>62	?63	@64	A65	B66	C67	D68	E69
F70	G71	H72	I73	J74	K75	L76	M77	N78	O79
P80	Q81	R82	S83	T84	U85	V86	w87	X88	Y89
Z90	[91	\92	]93	^94	_95	'96	a97	b98	c99
d100	e101	f102	g103	h104	i105	j106	k107	l108	m109
n110	o111	p112	q113	r114	s115	t116	u117	v118	w119
X120	Y121	Z122	{123	124	}125	~126			

Tabla B.1: *Codis Imprimibles en ASCII de set bits.*

Los primeros treinta y dos caracteres, del cero al treinta y uno, son caracteres de control, o en general no imprimibles. Por ejemplo el carácter 7 es el bip sonoro. El 9, el tabulador, el 10 línea abajo, y el 13 el retorno de carro.

Teniendo en cuenta que los caracteres se guardan en un byte, que es un número en binario de ocho cifras y por tanto puede representar los códigos desde el cero al 255, conviene tener en cuenta que la mitad superior, es decir, los códigos que empiezan en 1 que en decimal significa a partir del 128, se llaman caracteres de ocho bits, y el signo que codifican depende de la parte del mundo donde nos encontremos. Esta parte de la tabla se llama ASCII extendido. En este margen tenemos las vocales acentuadas

así como la ce cedilla o la ñe. Es peligroso usar este tipo de caracteres en textos que hayan de ser compilados o interpretados por algún sistema informático, ya que requiere que el intérprete en cuestión comparta la página de códigos con los datos. Aún así, si se desea disponer de un mayor número de códigos de caracteres, entonces se puede usar la codificación UTF-8, de *Unicode Transformation Format*, que permite códigos que ocupen varios bytes.

# Apéndice C

## Datos para el Ejemplo del Club Deportivo

En este apéndice se proporcionan los datos para poder seguir los ejemplos de los capítulos de SQL. Las tablas se dan sin ningún orden para representar más adecuadamente un caso real. Los datos de número de habitantes de las ciudades se han conseguido de la wikipedia.

### C.1 Tabla provincia

provincia
Napo
Los Ríos
El Oro
Chimborazo
Azuay
Pichincha
Tungurahua
Carchi
Guayas
Pastaza

## C.2 Tabla ciudad

ciudad	habitantes	provincia
Babahoyo	90191	Los Ríos
Tena	23307	Napo
San Francisco	805235	
Berlín	3499879	
Rio de Janeiro	6320446	
Machala	331260	El Oro
París	2249975	
Baños	12995	Tungurahua
New York	18897109	
Quito	1919432	Pichincha
Cuenca	331000	Azuay
Guayaquil	3500000	Guayas
Balao	9220	Guayas
Zaruma	22222	El Oro
Atenas	664046	
Pallatanga	3160	Chimborazo
Archidona	5478	Napo
Chilla	2665	El Oro
Roma	2796102	
Pueblo Viejo	24865	Los Ríos
Mira	3096	Carchi

## C.3 Tabla persona

pasaporte	nombres	apellidos	mail	ciudad
0127673812	Carmen Esmeralda	Peralta Gutiérrez	cperalta@ionos.gov	Babahoyo
0123478937	Carlos Sayaro	Sanabria Oña	sayarona1998_2@hotmail.com	Tena
1047548338	Ana Estefanía	Sanabria Oña	anasteona1997yahoo.ec	Tena
1145493393	Jesús Marcelo	Hortesa Orellana	rexstat143@rediris.es	Machala
CKUS01549	Michael	Bros	mbros1989@aol.com	San Francisco
FFJ904992	Klauss	Stallman	kstallman@dwv.tum.de	Berlín
1239238229	Samantha Anabel	Aragall Tumbaco	steatumonia1995@mismail.com	
HAT481338	Rita	Derbeken	matrit@iic.esd.edu	Berlín
1124637238	Rosario Carmela	Puente Pizarro	rp1995sip12@claro.com	Balao
C01X01TN	Roberto	Rietto	rrrietto@gmail.com	Roma
1238433548	Anabel Madelyn	Zurita Margalef	amargale554f@gmail.com	Balao
1232234958	José Marcelo	Sanlúcar Flavia	josesanmitraet@gmail.com	Machala
2038474483	Miquel Pablo	Vila Valverde	mvvalverde@gmail.com	Pallatanga
2142065765	Camila Odalys	Noriega Pastuña	cnoriega@ics.cat	Machala
1759119283	Pedro Fernando	Camprubí Villasana	perec@dpto.erf.org	
1827961020	Pedro David	García Miranda	prx132@yahoo.com	Archidona
1624865898	Maria Carmen	Martín Filoia	mariamartin@dia.udr.edu	Archidona
1574394950	Elena Angustias	Fernández Mierdaza	efmierdaza1996.com	Cuenca
1337228901	Sebastián Luber	Fonollá Orellana	sfmiranda@miliwat.cat	Quito
2029874567	Leonardo Domingo	Soler Alvarado	rexstan12@gmail.com	Baños
05CK02337	Jean Marie	Godard	jmgo.dard@ille.fr	París
C00021549	Mick	Brown	mibr@esportespot.cat	New York
1736940559	Magdalena Karen	Pinós Champa	mapi@esportespot.cat	Quito
1938223890	Jordy Francesco	Parmalat Montaluia	jparmalat@uab.edu	Tena
1451234329	Mireia Erika	Matas Montenegro	mima@esportespot.cat	Archidona
1545847558	José Antonio	González Huertas	jogo@esportespot.cat	Guayaquil
1848377283	Daniel Gonzalo	González Andrade	dago@esportespot.cat	Guayaquil
1637866969	Carmen Esmeralda	Ferrer Veciano	cafe@esportespot.cat	Mira
1827827228	Sonia Erika	Colmena Robles	soco@esportespot.cat	Pueblo Viejo
194534332C	Gabriel Luis	Cobos Barrionuevo	gaco@esportespot.cat	Pueblo Viejo
193439185	Boris Mauro	Santos Auyacu	bosa@esportespot.cat	Chilla

## C.4 Tabla teléfonos

<b>pasaporte</b>	<b>telefono</b>
0127673812	0969123439
05CK02337	0011833493388
05CK02337	655944839
05CK02337	972102293
C01X01TN	968339229
1624865898	0943199191
1938223890	092349329
1938223890	001434944989
1938223890	091229203
193439185	095559239
1574394950	00343992303920
1337228901	093483494
2029874567	097243758
05CK02337	00012932304493
C00021549	0915855948
1736940559	0972395921
1451234329	0977602330
1545847558	3431232205
1848377283	0972343679
1637866969	0934405550
1827827228	0639032291
194534332C	0936189283
193439185	0905449338

## C.5 Tabla conoce

<b>conoce</b>	<b>es_conocida</b>
0127673812	0123478937
0123478937	0127673812
0127673812	1047548338
0127673812	FFJ904992
FFJ904992	0127673812
FFJ904992	CKUS01549
1145493393	1232234958
1232234958	2142065765
194534332C	1827827228
1827827228	2142065765

## C.6 Tabla socio

<b>pasaporte</b>	<b>registro</b>
0127673812	21/03/2010
0123478937	12/09/2011
1047548338	01/04/2013
2038474483	02/03/2016
1145493393	31/05/2011
CKUS01549	30/06/2009
FFJ904992	14/06/2012
1239238229	14/11/2011
HAT481338	17/07/2013
1124637238	15/12/2009
C01X01TN	09/01/2009
1238433548	18/12/2012
1232234958	16/07/2010
2142065765	27/04/2013
1759119283	26/10/2012
1827961020	09/10/2016
1624865898	09/12/2016

## C.7 Tabla trabajador

<b>pasaporte</b>	<b>departamento</b>	<b>obedece</b>
1574394950	administrador	
1337228901	administrador	1574394950
2029874567	administrador	1574394950
05CK02337	administrador	1574394950
C00021549	administrador	05CK02337
1736940559	comercial	
1938223890	comercial	1736940559
1451234329	comercial	1736940559
1545847558	administrador	05CK02337
1848377283	administrador	1337228901
1637866969	comercial	1736940559
1827827228	entrenador	
194534332C	entrenador	
193439185	entrenador	

## C.8 Tabla deporte

<b>nombre</b>	<b>precio</b>	<b>jugadores</b>
natación	18.35	1
tenis	21.50	1
tenis dobles	21.50	2
ping-pong	10.30	1
básquet	8.60	5
fútbol	15.40	11
handbol	9.50	7
handbol	14.50	6
golf	24.15	1
vela	22.60	2



## C.9 Tabla hace

socio	deporte	cuota
0127673812	natación	18.35
0123478937	natación	18.35
1047548338	tenis	21.50
2038474483	tenis	21.50
1145493393	tenis dobles	21.50
CKUS01549	básquet	8.60
FFJ904992	básquet	8.60
1239238229	natación	18.35
HAT481338	básquet	8.60
1124637238	natación	18.35
C01X01TN	ping-pong	10.30
1238433548	ping-pong	10.30
1232234958	ping-pong	10.30
2142065765	tenis dobles	21.50
1759119283	natación	18.35
1827961020	handbol	14.50
1624865898	tenis	21.50
0127673812	fútbol	15.40
0123478937	fútbol	15.40
1047548338	básquet	8.60
2038474483	básquet	8.60
0127673812	ping-pong	10.30
0123478937	ping-pong	10.30
1047548338	golf	24.15
2038474483	vela	22.60
1145493393	vela	22.60
CKUS01549	vela	22.60
FFJ904992	fútbol	15.40
1239238229	fútbol	15.40
HAT481338	fútbol	15.40
1124637238	fútbol	15.40
C01X01TN	natación	18.35
1238433548	natación	18.35
1232234958	handbol	14.50
2142065765	tenis	21.50
1759119283	handbol	14.50
1827961020	natación	18.35
1624865898	ping-pong	10.30

## C.10 Tabla pagos

<b>pasaporte</b>	<b>periodo</b>	<b>salario_base</b>	<b>retencion</b>
1337228901	28/01/2017	1525.67	8.15
1337228901	29/12/2016	1530.80	8.15
1337228901	29/11/2016	1523.42	8.15
1337228901	29/10/2016	1523.42	8.15
2029874567	28/01/2017	1213.93	10.25
2029874567	29/12/2016	1213.93	10.25
2029874567	29/11/2016	1213.93	10.25
2029874567	29/10/2016	1201.46	10.25
2029874567	28/09/2016	1201.46	6.80
2029874567	30/08/2016	1201.46	6.80
2029874567	29/07/2016	1201.46	6.80
2029874567	29/06/2016	1213.93	6.80
2029874567	31/05/2016	1213.93	6.80
05CK02337	31/01/2016	1450.69	2.00
C00021549	28/01/2017	812.30	3.15
C00021549	29/12/2016	812.30	3.15
C00021549	29/11/2016	915.35	3.15
C00021549	29/10/2016	915.35	3.15
C00021549	28/09/2016	915.35	3.15
C00021549	30/08/2016	915.35	3.15
C00021549	29/07/2016	812.30	3.15
1736940559	28/01/2017	1090.23	5.00
1736940559	29/12/2016	1090.23	5.00
1736940559	29/11/2016	1090.23	5.00
1938223890	28/01/2017	1102.43	9.10
1938223890	29/12/2016	1102.43	9.10
1938223890	29/11/2016	1102.43	7.35
1938223890	29/10/2016	1060.90	7.35
1938223890	28/09/2016	1060.90	2.00
1451234329	28/01/2017	1232.02	2.00
1545847558	28/01/2017	729.45	3.59
1545847558	29/12/2016	711.34	3.59
1545847558	29/11/2016	708.10	3.59
1545847558	29/10/2016	703.56	3.59
1848377283	28/01/2017	698.32	11.50
1848377283	29/12/2016	698.32	11.50
1637866969	28/01/2017	1115.45	9.80
1637866969	29/12/2016	1102.76	9.80
1637866969	29/11/2016	1094.07	9.80
1827827228	28/01/2017	1050.30	9.80
1827827228	29/12/2016	1050.30	9.80
194534332C	28/01/2017	1050.30	9.10
194534332C	29/12/2016	1050.30	9.10
194534332C	29/11/2016	1050.30	7.35
194534332C	29/10/2016	1050.30	7.35
193439185	28/01/2017	3750.12	1.00
193439185	29/12/2016	3753.60	1.00
193439185	29/11/2016	3752.00	1.00

# Bibliografía

- [1] C. J. Date and Sergio Luis María Ruiz Faudón. *Introducción a los Sistemas de Bases de Datos*. 7 edició. Prentice Hall, 2001.
- [2] diccionari.cat. *Diccionari de l'Enciclopèdia Catalana*.
- [3] C. Franquesa. *Algorísmia Comentada*. Publicacions UB, 2012.
- [4] T. Hastie, R. Tibshirani, and J Friedman. *The Elements of Statistical Learning*. 4a. Edició. Springer-Verlag. DOI 10.1007/978-1-4614-7138-7, 2012.
- [5] <http://hibernate.org/>. *Hibernate. Everything data*.
- [6] <http://www14.gencat.cat/llc/AppJava/index.html>. *Optimot Consultes Lingüístiques*.
- [7] <http://www.postgresql.org/>. PostgreSQL. *The most advanced open source database*. 2014.
- [8] <http://www.postgresql.org/docs/>. Documentació i Manuals de PostgreSQL. En francès i anglès. 2014.
- [9] G. James, D. Witten, T. Hastie, and R. Tibshirani. An introduction to statistical learning with applications in r. *4a. Edició*, 2014.
- [10] A. Shilberschatz, H. F. Korth, and S. Surdarshan. *Fundamentos de Bases de Datos*. 6a. Edició. Mc Graw Hill, 2014.

# Cajas

1.1	<i>Definición de subconjunto propio</i>	12
-----	---	----

1.2	<i>Definición de compatibilidad entre conjuntos.</i>	13
1.3	<i>Definición de la unión de conjuntos.</i>	17
1.4	<i>Definición de la intersección de conjuntos.</i>	19
1.5	<i>Definición de la diferencia de conjuntos.</i>	20
1.6	<i>Definición del producto cartesiano de conjuntos.</i>	22
1.7	<i>Leyes de De Morgan.</i>	27
1.8	<i>Relación entre la lógica de predicados y la teoría de conjuntos.</i>	28
4.1	<i>Pseudocódigo para establecer las cardinalidades de una base de datos.</i>	85
5.1	<i>Esquema de una relación en el álgebra relacional.</i>	104
5.2	<i>Ejemplo de esquema de relación en el álgebra relacional.</i>	104
5.3	<i>Definición formal de una relación.</i>	105
5.4	<i>Definición de compatibilidad entre las relaciones <math>r</math> y <math>s</math>.</i>	106
5.5	<i>Definición de la relación para la visión cartesiana.</i>	108
5.6	<i>Expresión relacional de una selección.</i>	127
5.7	<i>Expresión relacional de una proyección.</i>	129
5.8	<i>Expresión relacional del producto cartesiano entre dos relaciones.</i>	132
5.9	<i>Expresión relacional de la unión de dos relaciones.</i>	135
5.10	<i>Expresión relacional de la diferencia de relaciones.</i>	137
5.11	<i>Expresión relacional para el renombramiento de cualquier expresión relacional.</i>	137
5.12	<i>Expresión relacional de la intersección de relaciones.</i>	142
5.13	<i>La intersección no es una operación básica.</i>	143
5.14	<i>Expresión relacional de la reunión natural entre dos relaciones.</i>	143
5.15	<i>Expresión relacional de la reunión interna entre dos relaciones.</i>	145
5.16	<i>Expresión relacional de las reuniones externas. (a) por la izquierda. (b) completa. (c) por la derecha.</i>	147
5.17	<i>Expresión relacional de la división entre dos relaciones.</i>	149
5.18	<i>La división no es una operación básica.</i>	149
5.19	<i>Expresión para la asignación de relaciones.</i>	150
5.20	<i>Asignación de una relación con una sola tupla a la relación <b>persona</b>.</i>	151
5.21	<i>Proyección generalizada de los atributos de una relación.</i>	152
5.22	<i>Nueva versión de la asignación de la Caja 5.20.</i>	152
5.23	<i>Expresión relacional mínima para las funciones de agregación.</i>	155
5.24	<i>Expresión relacional para las funciones de agregación.</i>	157
6.1	<i>Contenido inicial del archivo club.sql, escript principal.</i>	175
6.2	<i>Dominio para el atributo departamento de la tabla trabajador.</i>	177
6.3	<i>Creación de un dominio para los números positivos.</i>	177
6.4	<i>Dominio para el atributo mail de la tabla persona.</i>	178
6.5	<i>Archivo provincia.sql.</i>	179
6.6	<i>Primera propuesta para el archivo ciudad.sql.</i>	181
6.7	<i>Restricción de eliminación para la clave apuntada.</i>	182
6.8	<i>Restricción de participación total de CIUDAD en PROVINCIA.</i>	182
6.9	<i>Archivo ciudad.sql.</i>	184
6.10	<i>Archivo persona.sql.</i>	186
6.11	<i>Contenido actual del escript principal, archivo club.sql.</i>	187
6.12	<i>Archivo telefonos.sql.</i>	188
6.13	<i>Archivo conoce.sql.</i>	189

6.14	<i>Archivo socio.sql.</i>	190
6.15	<i>Archivo trabajador.sql.</i>	191
6.16	<i>Archivo deporte.sql.</i>	192
6.17	<i>Archivo hace.sql.</i>	193
6.18	<i>Archivo pagos.sql.</i>	193
6.19	<i>Contenido final del escript principal, archivo club.sql.</i>	194
6.20	<i>Estructura fundamental de una consulta de lectura en SQL.</i>	198
6.21	<i>Estructura habitual de una consulta de lectura en SQL.</i>	199
6.22	<i>Archivo club/provincia/inserts.sql.</i>	200
6.23	<i>Archivo club/ciudad/inserts.sql.</i>	201
6.24	<i>Consulta de todo el contenido de la tabla persona.</i>	202
6.25	<i>Archivo club/conoce/inserts.sql.</i>	210
6.26	<i>Formato de una consulta con criterios de agregación.</i>	216
6.27	<i>Sentencia de creación de una vista.</i>	223
6.28	<i>Archivo club/hace/facturas_socio.sql.</i>	224
6.29	<i>Forma básica del comando de inserción.</i>	225
6.30	<i>Forma alternativa del comando de inserción.</i>	226
6.31	<i>Forma más compleja del comando de inserción.</i>	226
6.32	<i>Sintaxis del comando de actualización.</i>	228
6.33	<i>Sintaxis del comando de eliminación.</i>	229
6.34	<i>Sintaxis de la operación de unión de relaciones.</i>	232
6.35	<i>Sintaxis de la operación de diferencia de relaciones.</i>	237
6.36	<i>Sintaxis de la operación de intersección de relaciones.</i>	238
6.37	<i>Expresión alternativa a la consulta de intersección del ejemplo 6.37.</i>	239
6.38	<i>Sintaxis de la reunión natural.</i>	241
6.39	<i>Sintaxis de la reunión interna con la condición USING.</i>	246
6.40	<i>Sintaxis de la reunión interna con la condición ON.</i>	247
6.41	<i>Consulta anidada.</i>	254
6.42	<i>Cláusula lógica IN.</i>	256
6.43	<i>Definición de la cláusula SOME con el comparador genérico.</i>	257
6.44	<i>Cláusula lógica SOME.</i>	258
6.45	<i>Definición de la cláusula ALL con el comparador genérico.</i>	259
6.46	<i>Cláusula lógica ALL.</i>	259
6.47	<i>Definición de la cláusula EXISTS.</i>	260
6.48	<i>Cláusula lógica EXISTS.</i>	261
7.1	<i>Archivo 1-init.sql.</i>	269
7.2	<i>Archivo 2-tablas.sql.</i>	269
7.3	<i>Archivo principal, club.sql.</i>	270
7.4	<i>Creación de bases de datos.</i>	272
7.5	<i>Creación de usuarios.</i>	272
7.6	<i>Creación de grupos de usuarios.</i>	273
7.7	<i>Versiones más actuales para la creación de usuarios y grupos.</i>	274
7.8	<i>Archivo usuarios.sql.</i>	277
7.9	<i>Garantizar privilegios.</i>	277
7.10	<i>Revocar privilegios.</i>	278
7.11	<i>Dos transacciones en dos comandos.</i>	283
7.12	<i>Transacción de dos comandos.</i>	283

7.13	Archivo <code>siete_bits.sql</code> . Ejemplo de bucle sencillo. . . . .	297
7.14	Archivo <code>ejemplos_raises.sql</code> . Uso de la instrucción <code>RAISE</code> . . . . .	299
7.15	Archivo <code>registrar_socio.sql</code> . Consultas de actualización. . . . .	301
7.16	Archivo <code>registrar_trabajador.sql</code> . Consultas de actualización. . . . .	303
7.17	Adición de privilegios sobre las funciones en el archivo <code>4-usuarios.sql</code> . . . . .	304
7.18	Primera versión archivo <code>nombres_y_apellidos.sql</code> . Selección de un solo valor. . . . .	305
7.19	Archivo <code>nombres_y_apellidos.sql</code> . Selección y verificación de un solo valor. . . . .	306
7.20	Archivo <code>canoniza.sql</code> . Función auxiliar. . . . .	307
7.21	Archivo <code>provincia_parecida.sql</code> . Función auxiliar. . . . .	309
7.22	Archivo <code>deportes_socios.sql</code> . Uso de la instrucción <code>RETURN QUERY</code> . . . . .	312
7.23	Archivo <code>ciudades_deportes_socios.sql</code> . Recorridos de consultas. . . . .	314
7.24	Cabecera de una función de tipo <code>TRIGGER</code> . . . . .	316
7.25	Disparador antes de la inserción, para cada fila, en la tabla <code>provincia</code> . . . . .	318
7.26	Declaración de un disparador para la tabla <code>provincia</code> . . . . .	318
7.27	Consulta del valor y el origen de un parámetro. . . . .	329

## Diagramas

5.1	Diagrama bivalente como ejemplo de uso de las coordenadas cartesianas. . . . .	107
5.2	Ejemplo de diagrama de esquema de relación. . . . .	123
5.3	Diagrama de esquemas de relación del Modelo ER del ejemplo. . . . .	125

## Figuras

1.1	Conjuntos $C$ y $D$ utilizados en los ejemplos. . . . .	17
1.2	Mnemograma para la unión de conjuntos. . . . .	17
1.3	Conjunto Unión $C \cup D$ de los conjuntos de la Figura 1.1. . . . .	18
1.4	Mnemograma para la intersección de conjuntos. . . . .	19
1.5	Conjunto Intersección $C \cap D$ de los conjuntos de la Figura 1.1. . . . .	19
1.6	Mnemograma para la diferencia de conjuntos. . . . .	20
1.7	Conjunto Diferencia $C \setminus D$ de los conjuntos de la Figura 1.1. . . . .	21
1.8	Mnemograma para el producto cartesiano de conjuntos. . . . .	22
1.9	Producto Cartesiano $C \times D$ de los conjuntos de la Figura 1.1. . . . .	23

2.1	<i>Arquitectura Cliente-Servidor.</i> . . . . .	33
2.2	<i>Modelo en forma de cebolla de un sistema informático.</i> . . . . .	34
2.3	<i>Arquitectura a tres niveles.</i> . . . . .	35
3.1	<i>Grafo de 5 nodos y 7 arcos</i> . . . . .	44
3.2	<i>(a) Fuertemente conexo; (b) Unilateralmente conexo; (c) Débilmente conexo.</i> . . . . .	45
3.3	<i>Grafo acíclico conexo, o sea árbol.</i> . . . . .	45
3.4	<i>Ejemplo de modelo de dominio para el club de deportes.</i> . . . . .	47
4.1	<i>Representación de una entidad en un modelo ER.</i> . . . . .	52
4.2	<i>Representación de un atributo en una entidad de un modelo ER.</i> . . . . .	54
4.3	<i>Representación de un atributo clave en una entidad de un modelo ER.</i> . . . . .	57
4.4	<i>Atributo multivalorado en un modelo ER.</i> . . . . .	58
4.5	<i>Atributo compuesto en una entidad de un modelo ER.</i> . . . . .	60
4.6	<i>Atributo calculado en una entidad de un modelo ER.</i> . . . . .	62
4.7	<i>Participación parcial de <math>E_1</math> y total de <math>E_2</math> en la relación <math>R</math>.</i> . . . . .	64
4.8	<i>Relación 1:1 en un modelo ER.</i> . . . . .	66
4.9	<i>Relación 1:N en un modelo ER.</i> . . . . .	68
4.10	<i>Entidad débil <math>E_2</math> identificada vía <math>E_1</math>.</i> . . . . .	72
4.11	<i>Relación jerárquica entre relaciones 1:N.</i> . . . . .	75
4.12	<i>Relación M:N en un modelo ER.</i> . . . . .	76
4.13	<i>Potencia expresiva de las autorrelaciones 1:1.</i> . . . . .	81
4.14	<i>Autorelación 1:1 en un modelo ER.</i> . . . . .	81
4.15	<i>Vectorización de un árbol. (a) Árbol; (b) Redisposición secuencial; (c) Vector de predecesores.</i> . . . . .	82
4.16	<i>Autorelación 1:N en un modelo ER.</i> . . . . .	83
4.17	<i>Autorelación M:N en un modelo ER.</i> . . . . .	84
4.18	<i>Relación ternaria en un modelo ER.</i> . . . . .	88
4.19	<i>Relación ternaria con cardinalidad 1:M:N.</i> . . . . .	89
4.20	<i>Alternativa incorrecta a una relación ternaria.</i> . . . . .	90
4.21	<i>Especialización o generalización en un modelo ER.</i> . . . . .	91
4.22	<i>Agregación de una relación. (a) Relación M:N. (b) Entidad agregada.</i> . . . . .	95
5.1	<i>Una relación es un conjunto de tuplas.</i> . . . . .	100
5.2	<i>Representación de la parábola que describe la función <math>f(x) = x^2</math>.</i> . . . . .	105
5.3	<i>Instancia de la relación de la Caja 5.5: (a) Descripción tabular. (b) Descripción cartesiana.</i> . . . . .	109
5.4	<i>Implementación de una relación 1:N del modelo ER con claves foráneas en el modelo relacional.</i> . . . . .	114
5.5	<i>Relación persona de los ejemplos. (a) Descripción tabular. (b) Descripción cartesiana.</i> . . . . .	126
5.6	<i>Selección de personas de tena. (a) Descripción tabular. (b) Descripción cartesiana.</i> . . . . .	128
5.7	<i>Selección de personas de 35 años, del tena. (a) Descripción tabular. (b) Descripción cartesiana.</i> . . . . .	129
5.8	<i>Selección de personas de 35 años, o de tena. (a) Descripción tabular. (b) Descripción cartesiana.</i> . . . . .	130

5.9	edad y ciudad de las personas. (a) Descripción tabular. (b) Descripción cartesiana. . . . .	131
5.10	edad de todas las personas. (a) Descripción tabular. (b) Descripción cartesiana. . . . .	132
5.11	edad y ciudad de las personas que se llaman alexis. (a) Descripción tabular. (b) Descripción cartesiana. . . . .	133
5.12	Producto cartesiano. (a) Relaciones de entrada. (b) Descripción tabular. (c) Descripción cartesiana. . . . .	134
5.13	La historieta del álgebra relacional. . . . .	135
5.14	Inserción de la tupla ('paola',25,'ambato'). (a) Descripción tabular. (b) Descripción cartesiana. . . . .	136
5.15	Eliminación de las personas de 35 años. (a) Descripción tabular. (b) Descripción cartesiana. . . . .	138
5.16	Renombramiento de los atributos de la relación persona. (a) Descripción tabular. (b) Descripción cartesiana. . . . .	139
5.17	Última etapa de la resolución del ejemplo 5.10. . . . .	141
5.18	La intersección es uno menos la diferencia. . . . .	143
6.1	Arquitectura del entorno de trabajo . . . . .	162
6.2	Clasificación de las relaciones de una base de datos. . . . .	167
6.3	Estructura actual del espacio de trabajo. . . . .	180
6.4	Estructura del espacio con la tabla ciudad. . . . .	185
6.5	Estructura de directorios del proyecto. . . . .	196
6.6	Incorporación de escripts de inserción. . . . .	197
6.7	Implicación sintáctica de la cláusula GROUP BY. . . . .	219
6.8	Ejemplos de la cláusula SOME. . . . .	258
6.9	Ejemplos de la cláusula ALL. . . . .	259
7.1	Página web del manual. . . . .	266
7.2	Columna de privilegios de acceso a una tabla . . . . .	281
7.3	(a) Parte de una relación; (b) Estructura para un atributo índice. . . . .	288
7.4	Archivo hola_mundo.sql . . . . .	293

## Modelos

4.1	Ejemplo de una entidad en un modelo ER. . . . .	53
4.2	Ejemplo del uso de atributos en la entidad del Modelo 4.1. . . . .	54
4.3	Ejemplo más preciso del uso de atributos en la entidad del Modelo 4.1. . . . .	55
4.4	Representación de una entidad completa en un modelo ER. . . . .	57
4.5	Ejemplo de atributo multivalorado en el Modelo 4.4. . . . .	58



4.6	<i>Ejemplo de atributo compuesto en el Modelo 4.4.</i>	60
4.7	<i>Ejemplo de atributo calculado en el Modelo 4.4.</i>	63
4.8	<i>Ejemplo de relación 1:1 en un modelo ER.</i>	67
4.9	<i>Simplificación del Modelo 4.8.</i>	67
4.10	<i>Entidad con atributos incorrectos en un modelo ER.</i>	68
4.11	<i>Modelo correcto para el ejemplo del Modelo 4.10.</i>	69
4.12	<i>Modelo correcto simplificado para el ejemplo del Modelo 4.10.</i>	70
4.13	<i>Ejemplo del Modelo 4.11 con una relación nueva.</i>	70
4.14	<i>Entidad de soporte PROVINCIA.</i>	71
4.15	<i>Ejemplo de uso de una entidad débil.</i>	73
4.16	<i>Modelo inicial antes de introducir una relación M:N.</i>	77
4.17	<i>Ejemplo de relación M:N en un modelo ER.</i>	78
4.18	<i>Ejemplo de atributos en una relación M:N de un modelo ER.</i>	79
4.19	<i>Ejemplo de autorelación 1:1 en el Modelo 4.16.</i>	81
4.20	<i>Ejemplo de autorelación 1:N en el Modelo 4.16.</i>	83
4.21	<i>Ejemplo de autorelación M:N en el Modelo 4.16.</i>	84
4.22	<i>Ejemplo de relación ternaria añadido al Modelo 4.18.</i>	88
4.23	<i>Ejemplo de especialización o generalización en un modelo ER.</i>	92
4.24	<i>Buñuelo inadmisibles en un modelo ER.</i>	93
5.1	<i>Ejemplo de esquema de relación.</i>	112
5.2	<i>Ejemplo de esquema de relación con clave primaria binomial.</i>	112
5.3	<i>Modelo ER para la aplicación del club deportivo.</i>	116
5.4	<i>Primeras relaciones de la versión relacional del ejemplo 5.3.</i>	117
5.5	<i>Incorporación de la relación persona.</i>	117
5.6	<i>Transformación de un atributo multivalorado.</i>	118
5.7	<i>Transformación de una autorelación M:N al modelo relacional.</i>	119
5.8	<i>Modelo ER para los ejemplos, repetición del 5.3.</i>	119
5.9	<i>Transformación de entidades especializadas y de autorelación 1:N.</i>	120
5.10	<i>Transformación de relaciones M:N previa descripción de las entidades relacionadas.</i>	121
5.11	<i>Modelo Relacional del diseño del Modelo Entidad Relación 5.3.</i>	122
5.12	<i>Fragmento Modelo 5.11 relacional.</i>	147
5.13	<i>Fragmento del Modelo 5.11 para los ejemplos de las funciones de agregación.</i>	156
6.1	<i>Modelo ER de la base de datos.</i>	173
6.2	<i>Modelo Relacional del diseño del Modelo Entidad Relación 6.1.</i>	174
7.1	<i>Permisos y privilegios.</i>	270
7.2	<i>Permisos y Privilegios.</i>	274
7.3	<i>Fragmento superior del Modelo 7.2.</i>	279
7.4	<i>Modelo ER para la aplicación del club deportivo.</i>	291