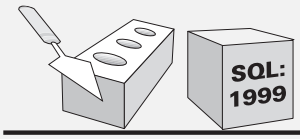

5

SQL: CONSULTAS, RESTRICCIONES Y DISPARADORES

- ¿Qué incluye el lenguaje SQL? ¿Qué es SQL:1999?
- ¿Cómo expresan las consultas SQL?
  ¿Cuál es el significado de las consultas expresadas según la norma de SQL?
- ¿Cómo aprovecha SQL el álgebra y el cálculo relacionales y cómo los amplía?
- ¿Qué es la agrupación? ¿Cómo se emplea en las operaciones de agregación?
- ¿Qué son las consultas anidadas?
- ¿Qué son los valores nulos (*null*)?
- ¿Cómo se pueden utilizar las consultas para escribir restricciones de integridad complejas?
- ¿Qué son los disparadores y por qué resultan útiles? ¿Qué relación tienen con las restricciones de integridad?
- **Conceptos fundamentales:** consultas SQL, conexión con el álgebra y el cálculo relacionales; características adicionales al álgebra, la cláusula `DISTINCT` y la semántica multiconjunto, agrupación y agregación; consultas anidadas, correlación; operadores para la comparación de conjuntos; valores nulos (*null*), reuniones externas; restricciones de integridad especificadas mediante consultas; disparadores y bases de datos activas, reglas evento-condición-acción.

¿Qué hombres o dioses son éstos? ¿Qué doncellas reticentes?
¿Qué loco empeño? ¿Qué lucha por escapar?
¿Qué flautas y panderetas? ¿Qué salvaje éxtasis?

— John Keats, *Oda a una urna griega*

El lenguaje estructurado de consulta (Structured Query Language, SQL) es el lenguaje comercial para bases de datos relacionales más utilizado. Se desarrolló originalmente en IBM en los

Conformidad con las normas de SQL. SQL:1999 tiene un conjunto de características denominado Core SQL (SQL Fundamental) que los fabricantes deben implementar para poder afirmar que cumplen la norma SQL:1999. Se estima que todos los principales fabricantes pueden cumplir Core SQL con poco esfuerzo. Gran parte del resto de características se organiza en **paquetes**.

Por ejemplo, los paquetes tratan las siguientes características se tratan mediante (se indican los capítulos correspondientes entre paréntesis): *tratamiento mejorado de la fecha y de la hora*, *gestión mejorada de la integridad y bases de datos activas* (este capítulo), *interfaces para lenguajes externos* (Capítulo 6), *OLAP* (Capítulo 15) y *características orientadas a objetos* (Capítulo 15). La norma SQL/MM complementa SQL:1999 mediante la definición de paquetes adicionales que soportan la *minería de datos* (Capítulo 17), los *datos espaciales* (no tratado en este texto) y los *documentos de texto* (Capítulo 18). El soporte para los datos y las consultas XML es inminente.

proyectos SEQUEL-XRM y System-R (1974-1977). Casi inmediatamente, otros fabricantes introdujeron productos de SGBD basados en SQL, y ahora es la norma de facto. SQL sigue evolucionando en respuesta a las necesidades cambiantes del área de las bases de datos. La norma ANSI/ISO para SQL que se trata en este libro es SQL:1999. Aunque no todos los productos de SGBD soportan todavía toda la norma SQL:1999, los fabricantes trabajan para conseguir ese objetivo, y la mayor parte de los productos ya soportan las características principales. La norma SQL:1999 es muy similar a la norma anterior, SQL-92, con respecto a las características que se estudian en este capítulo. La presentación que se ofrece es consistente tanto con SQL-92 como con SQL:1999, y los aspectos que no son iguales en las dos versiones de la norma se destacan de manera explícita.

5.1 INTRODUCCIÓN

El lenguaje SQL tiene varios aspectos diferentes.

- **Lenguaje de manipulación de datos (LMD).** Este subconjunto de SQL permite a los usuarios formular consultas e insertar, eliminar y modificar filas. Las consultas son el principal objeto de atención de este capítulo. Las órdenes del LMD para insertar, eliminar y modificar filas se trataron en el Capítulo 3.
- **Lenguaje de definición de datos (LDD).** Este subconjunto de SQL soporta la creación, eliminación y modificación de definiciones de tablas y vistas. Se pueden definir *restricciones de integridad* para las tablas, bien en el momento de crearlas, bien posteriormente. Las características del LDD de SQL se trataron en el Capítulo 3. Aunque la norma no estudia los índices, las implementaciones comerciales ofrecen también órdenes para la creación y eliminación de índices.
- **Disparadores y restricciones de integridad avanzadas.** La nueva norma SQL:1999 incluye soporte para los *disparadores*, que son acciones ejecutadas por el SGBD siempre

que las modificaciones de la base de datos cumplen las condiciones especificadas en el disparador. Los disparadores se tratan en este capítulo. SQL permite el empleo de consultas para definir especificaciones complejas de restricciones de integridad. También se examinan esas restricciones en este capítulo.

- **SQL incorporado y SQL dinámico.** Las características de SQL incorporado permiten llamar al código SQL desde lenguajes anfitriones como C o COBOL. Las características de SQL dinámico permiten que se creen (y ejecuten) consultas en el momento de la ejecución. Estas características se tratan en el Capítulo 6.
- **Ejecución cliente-servidor y acceso a bases de datos remotas.** Estas órdenes controlan el modo en que los programas de aplicación *clientes* pueden conectarse con los *servidores* de bases de datos de SQL o tener acceso a los datos de las bases de datos a través de la red. Estas órdenes se tratan en el Capítulo 7.
- **Gestión de transacciones.** Diversas órdenes permiten que los usuarios controlen de manera explícita aspectos del modo en que se deben ejecutar las transacciones. Estas órdenes se tratan en el Capítulo 14.
- **Seguridad.** SQL ofrece mecanismos para control el acceso de los usuarios a los objetos de datos, como tablas y vistas. Esto se trata en el Capítulo 14.
- **Características avanzadas.** La norma SQL:1999 incluye características orientadas a objetos (Capítulo 15), consultas de apoyo a las decisiones (Capítulo 16) y también aborda áreas emergentes como la minería de datos (Capítulo 17), los datos espaciales y la gestión de datos de texto y de XML (Capítulo 18).

5.1.1 Organización del capítulo

El resto de este capítulo se organiza de la manera siguiente. Las consultas SQL se presentan en el Apartado 5.2 y los operadores para conjuntos de SQL en el Apartado 5.3. Las consultas anidadas, en las que relaciones a las que se hace referencia en la consulta se definen a su vez en la propia consulta, se tratan en el Apartado 5.4. Los operadores de agregación, que permiten escribir consultas SQL que no se pueden expresar en el álgebra relacional, se tratan en el Apartado 5.5. Los valores *null*, que son valores especiales empleados para indicar que el valor de un campo es desconocido o no existe, se estudian en el Apartado 5.6. Las restricciones de integridad complejas que se pueden especificar mediante el LDD de SQL se estudian en el Apartado 5.7, lo que amplía el estudio del LDD de SQL del Capítulo 3; las nuevas especificaciones de las restricciones permiten aprovechar por completo las capacidades de lenguaje de consulta SQL.

Finalmente, se estudia el concepto de *base de datos activa* en los Apartados 5.8 y 5.9. Una **base de datos activa** tiene un conjunto de **disparadores** que especifica el DBA. Cada disparador describe acciones que se deben llevar a cabo cuando se dan determinadas circunstancias. El SGBD vigila la base de datos, detecta esas situaciones e invoca al disparador. La norma SQL:1999 exige el soporte de los disparadores, y varios productos de SGBD relacionales ya soportan alguna forma de disparador.

idm	*nombrem*	*categoría*	*edad*
22	Domínguez	7	45,0
29	Bravo	1	33,0
31	Lorca	8	55,5
32	Alández	8	25,5
58	Rubio	10	35,0
64	Horacio	7	35,0
71	Zuazo	10	16,0
74	Horacio	9	35,0
85	Arturo	3	25,5
95	Benito	3	63,5

Figura 5.1 El ejemplar $M3$ de Marineros

idm	*idb*	*fecha*
22	101	10/10/98
22	102	10/10/98
22	103	10/8/98
22	104	10/7/98
31	102	11/10/98
31	103	11/6/98
31	104	11/12/98
64	101	9/5/98
64	102	9/8/98
74	103	9/8/98

Figura 5.2 El ejemplar $R2$ de Reservas

Acerca de los ejemplos

Se van a presentar varias consultas de ejemplo que emplean la siguiente tabla de definiciones:

Marineros(*idm:* `integer`, *nombrem:* `string`, *categoría:* `integer`, *edad:* `real`)
Barcos(*idb:* `integer`, *nombreb:* `string`, *color:* `string`)
Reservas(*idm:* `integer`, *idb:* `integer`, *fecha:* `date`)

A cada consulta se le concede un número único, que continúa el esquema de numeración empleado en el Capítulo 4. La primera consulta nueva de este capítulo tiene el número C15. Las consultas C1 a C14 se introdujeron en el Capítulo 4[1]. Las consultas se ilustran mediante los ejemplares $M3$ de Marineros, $R2$ de Reservas y $B1$ de Barcos introducidas en el mismo capítulo, que se reproducen en las Figuras 5.1, 5.2 y 5.3, respectivamente.

Todas las tablas y consultas de ejemplo que aparecen en este capítulo están disponibles en inglés en la página web del libro en

`http://www.cs.wisc.edu/~dbbook`

El material en línea incluye instrucciones sobre la configuración de Oracle, IBM DB2, Microsoft SQL Server y MySQL, y secuencias de comandos para la creación de tablas y consultas de ejemplo.

5.2 FORMA DE LAS CONSULTAS SQL BÁSICAS

Este apartado presenta la sintaxis de las consultas SQL sencillas y explica su significado mediante una *estrategia de evaluación conceptual*. Las estrategias de evaluación conceptual son un medio de evaluar las consultas que se pretende que sean fáciles de comprender antes que eficientes. Los SGBD suelen ejecutar cada consulta de manera diferente y del modo más eficiente.

La forma básica de las consultas SQL es la siguiente:

[1]Todas las referencias a cada una de las consultas se pueden hallar en el índice del libro.

idb	*nombreb*	*color*
101	Intrépido	azul
102	Intrépido	rojo
103	Campeón	verde
104	Místico	rojo

Figura 5.3 El ejemplar $B1$ de Barcos

```
SELECT [ DISTINCT ] lista-de-selección
FROM   lista-de-tablas
WHERE  condición
```

Todas las consultas deben tener una cláusula `SELECT`, que especifica las columnas que se deben conservar en el resultado, y una cláusula `FROM`, que especifica un producto cartesiano de tablas. La cláusula opcional `WHERE` especifica las condiciones de selección para las tablas indicadas en la cláusula `FROM`.

Una consulta así, intuitivamente, corresponde a una expresión del álgebra relacional que implica selecciones, proyecciones y productos cartesianos. La estrecha relación entre SQL y el álgebra relacional es la base de la optimización de las consultas en los SGBD relacionales. En realidad, los planes de ejecución de las consultas SQL se representan mediante una variación de las expresiones del álgebra relacional.

Considérese un ejemplo sencillo.

(C15) Averiguar el nombre y la edad de todos los marineros.

```
SELECT DISTINCT M.nombrem, M.edad
FROM   Marineros M
```

La respuesta es un *conjunto* de filas, cada una de ellas es un par $\langle nombrem,\ edad \rangle$. Si dos o más marineros tienen el mismo nombre y la misma edad, la respuesta sigue teniendo una sola pareja con ese nombre y edad. Esta consulta es equivalente a la aplicación del operador proyección del álgebra relacional.

Si se omite la palabra clave `DISTINCT`, se obtendrá una copia de la fila $\langle n,e \rangle$ por cada marinero de nombre n y edad e; la respuesta sería un *multiconjunto* de filas. Los **multiconjuntos** se parecen a los conjuntos en que son colecciones desordenadas de elementos, pero en los multiconjuntos puede haber varias copias de cada elemento y el número de copias es significativo —dos multiconjuntos pueden tener los mismos elementos y ser diferentes porque el número de copias de algún elemento sea diferente—. Por ejemplo, {a, b, b} y {b, a, b} denotan el mismo multiconjunto, y son diferentes del multiconjunto {a, a, b}.

La respuesta a esta consulta, con o sin la palabra clave `DISTINCT`, para el ejemplar $M3$ de Marineros puede verse en las Figuras 5.4 y 5.5. La única diferencia es que la tupla de Horacio aparece dos veces si se omite `DISTINCT`; esto se debe a que hay dos marineros que se llaman Horacio y tienen 35 años.

La siguiente consulta es equivalente a una aplicación del operador selección del álgebra relacional.

(C11) Averiguar todos los marineros de categoría superior a 7.

nombrem	*edad*
Domínguez	45,0
Bravo	33,0
Lorca	55,5
Alández	25,5
Rubio	35,0
Horacio	35,0
Zuazo	16,0
Arturo	25,5
Benito	63,5

Figura 5.4 Respuesta a C15

nombrem	*edad*
Domínguez	45,0
Bravo	33,0
Lorca	55,5
Alández	25,5
Rubio	35,0
Horacio	35,0
Zuazo	16,0
Horacio	35,0
Arturo	25,5
Benito	63,5

Figura 5.5 Respuesta a C15 sin `DISTINCT`

```
SELECT  M.idm, M.nombrem, M.categoría, M.edad
FROM    Marineros AS M
WHERE   M.categoría > 7
```

Esta consulta emplea la palabra clave opcional `AS` para introducir una variable de rango. Por cierto, cuando se desea recuperar todas las columnas como en esta consulta, SQL ofrece una cómoda abreviatura: basta con escribir `SELECT *`. Esta notación resulta útil para las consultas interactivas, pero no es una buena práctica para las consultas que se pretende volver a utilizar y conservar, ya que el esquema del resultado no queda claro desde la propia consulta; hay que hacer referencia al esquema de la tabla Marineros subyacente.

Como ilustran estos dos ejemplos, la cláusula `SELECT` se emplea realmente para hacer *proyecciones*, mientras que las *selecciones* en el sentido del álgebra relacional se expresan mediante la cláusula `WHERE`. Este desajuste entre la denominación de los operadores selección y proyección del álgebra relacional y la sintaxis de SQL es un accidente histórico desafortunado.

Ahora se considerará la sintaxis de una consulta básica de SQL con más detalle.

- La **lista-de-tablas** de la cláusula `FROM` es una lista de nombres de tablas. El nombre de cada tabla puede ir seguido de una **variable de rango**; las variables de rango resultan especialmente útiles cuando el mismo nombre de tabla aparece más de una vez en esta lista.

- La **lista-de-selección** es una lista de (expresiones que implican a) nombres de columna de tablas que se mencionan en la lista-de-selección. A los nombres de columna se les puede anteponer una variable de rango.

- La **condición** de la cláusula `WHERE` es una combinación booleana (es decir, una expresión que emplea las conectivas lógicas `AND`, `OR` y `NOT`) de condiciones de la forma *expresión* `op` *expresión*, donde `op` es alguno de los operadores de comparación $\{<, <=, =, <>, >=, >\}$[2].

[2]Las expresiones con `NOT` siempre se pueden sustituir por expresiones equivalentes sin `NOT` dado el conjunto de operaciones de comparación que se acaba de mencionar.

Una *expresión* es un nombre de *columna*, una *constante* o una expresión (aritmética o cadena de caracteres).

- La palabra clave `DISTINCT` es opcional. Indica que la tabla calculada como respuesta a la consulta no debe contener *duplicados*, es decir, dos copias de la misma fila. El valor predeterminado es que los duplicados no se eliminan.

Aunque las reglas anteriores describen (de manera informal) la sintaxis de las consultas básicas de SQL, no indican el *significado* de las consultas. La respuesta a cada consulta es en sí misma una relación —que es un *multiconjunto* de filas en SQL— cuyo contenido se puede comprender considerando la siguiente estrategia de evaluación conceptual:

1. Calcular el producto cartesiano de las tablas de la **lista-de-tablas**.
2. Eliminar filas del producto cartesiano que no cumplan las condiciones de **condición**.
3. Eliminar todas las columnas que no aparezcan en la **lista-de-selección**.
4. Si se especifica `DISTINCT`, eliminar las filas repetidas.

Esta sencilla estrategia de evaluación conceptual explicita las filas que deben encontrarse en la respuesta a la consulta. Sin embargo, es probable que sea bastante ineficiente. En capítulos posteriores se considerará la manera en que los SGBD evalúan realmente las consultas; por ahora, nuestro objetivo no es más que explicar el significado de las consultas. La estrategia de evaluación conceptual se ilustrará mediante la consulta siguiente:

(C1) Averiguar el nombre de los marineros que han reservado el barco 103.

Se puede expresar en SQL de la manera siguiente.

```
SELECT M.nombrem
FROM   Marineros M, Reservas R
WHERE  M.idm = R.idm AND R.idb=103
```

Calculemos la respuesta a esta consulta para los ejemplares $R3$ de Reservas y $M4$ de Marineros de las Figuras 5.6 y 5.7, ya que el cálculo para los ejemplares de ejemplo habituales ($R2$ y $M3$) resultaría innecesariamente tedioso.

idm	*idb*	*fecha*
22	101	10/10/96
58	103	11/12/96

Figura 5.6 Ejemplar $R3$ de Reservas

idm	*nombrem*	*categoría*	*edad*
22	domínguez	7	45,0
31	lorca	8	55,5
58	rubio	10	35,0

Figura 5.7 Ejemplar $M4$ de Marineros

El primer paso es crear el producto cartesiano $M4 \times R3$, que puede verse en la Figura 5.8.

El segundo paso es aplicar la condición *M.idm = R.idm* `AND` *R.idb=103*. (Obsérvese que la primera parte de esta condición necesita una operación reunión.) Este paso elimina todas las filas del ejemplar mostrado en la Figura 5.8 menos la última. El tercer paso es eliminar las columnas no deseadas; sólo aparece *nombrem* en la cláusula `SELECT`. Este paso deja con el resultado mostrado en la Figura 5.9, que es una tabla con una sola columna y, por lo que se ve, una sola fila.

idm	*nombrem*	*categoría*	*edad*	*idm*	*idb*	*fecha*
22	domínguez	7	45,0	22	101	10/10/96
22	domínguez	7	45,0	58	103	11/12/96
31	lorca	8	55,5	22	101	10/10/96
31	lorca	8	55,5	58	103	11/12/96
58	rubio	10	35,0	22	101	10/10/96
58	rubio	10	35,0	58	103	11/12/96

Figura 5.8 *M4 × R3*

nombrem
rubio

Figura 5.9 Respuesta a la consulta C1 para R3 y M4

5.2.1 Ejemplos de consultas básicas de SQL

A continuación se presentarán varias consultas de ejemplo, muchas de las cuales ya se expresaron anteriormente en el álgebra y el cálculo relacionales (Capítulo 4). El primer ejemplo ilustra que el empleo de variables de rango es opcional, a menos que sean necesarias para resolver alguna ambigüedad. La consulta C1, que ya se examinó en el apartado anterior, también puede expresarse de la manera siguiente:

```
SELECT nombrem
FROM   Marineros M, Reservas R
WHERE  M.idm = R.idm AND idb=103
```

Sólo hay que cualificar las apariciones de *idm*, ya que esta columna aparece tanto en la tabla Marineros como en la tabla Reservas. Una manera equivalente de escribir esta consulta es:

```
SELECT nombrem
FROM   Marineros, Reservas
WHERE  Marineros.idm = Reservas.idm AND idb=103
```

Esta consulta muestra que los nombres de las tablas se pueden emplear de manera implícita como variables de fila. Sólo hace falta introducir variables de rango de manera explícita cuando la cláusula `FROM` contiene más de una aparición de una misma relación[3]. No obstante, se recomienda el empleo explícito de las variables de rango y la condición completa de todas las apariciones de las columnas con una variable de rango para mejorar la legibilidad de las consultas. En todos los ejemplos del libro se seguirá este convenio.

(C16) Averiguar el idm de los marineros que han reservado barcos rojos.

[3]El nombre de la tabla no se puede emplear como variable de rango implícita una vez se ha introducido una variable de rango para la relación.

```
SELECT   R.idm
FROM     Barcos B, Reservas R
WHERE    B.idb = R.idb AND B.color = 'rojo'
```

Esta consulta contiene una reunión de dos tablas, seguida de una selección del color de los barcos. Se puede pensar en B y en R como en filas de las tablas correspondientes que "prueban" que un marinero con idm R.idm reservó un barco rojo B.idb. En los ejemplares de ejemplo $R2$ y $M3$ (Figuras 5.1 y 5.2), la respuesta consiste en los *idm*s 22, 31 y 64. Si se desean los nombres de los marineros en el resultado, también se debe considerar la relación Marineros, ya que Reservas no contiene esta información, como ilustra el ejemplo siguiente.

(C2) Averiguar el nombre de los marineros que han reservado barcos rojos.

```
SELECT   M.nombrem
FROM     Marineros M, Reservas R, Barcos B
WHERE    M.idm = R.idm AND R.idb = B.idb AND B.color = 'rojo'
```

Esta consulta contiene una reunión de tres tablas seguida de una selección del color de los barcos. La reunión con Marineros permite averiguar el nombre del marinero que, según la tupla R de Reservas, ha reservado el barco rojo descrito por la tupla B.

(C3) Averiguar el color de los barcos alquilados por Lorca.

```
SELECT B.color
FROM   Marineros M, Reservas R, Barcos B
WHERE  M.idm = R.idm AND R.idb = B.idb AND M.nombrem = 'Lorca'
```

Esta consulta es muy parecida a la anterior. Obsérvese que, en general, puede que haya más de un marinero llamado Lorca (ya que *nombrem* no es clave de Marineros); esta consulta sigue siendo correcta en el sentido de que devolverá el color de los barcos reservados por *algún* Lorca, si es que hay varios marineros llamados Lorca.

(C4) Averiguar el nombre de los marineros que han reservado, como mínimo, un barco.

```
SELECT M.nombrem
FROM   Marineros M, Reservas R
WHERE  M.idm = R.idm
```

La reunión de Marineros y Reservas garantiza que, para cada *nombrem* seleccionado, el marinero haya hecho alguna reserva. (Si algún marinero no ha hecho ninguna reserva, el segundo paso de la estrategia de evaluación conceptual eliminará todas las filas del producto cartesiano que impliquen a ese marinero.)

5.2.2 Expresiones y cadenas de caracteres en la orden SELECT

SQL soporta una versión más general de la **lista-de-selección** que una mera lista de columnas. Cada elemento de una **lista-de-selección** puede ser de la forma *expresión* `AS` *nombre_columna*, donde *expresión* es cualquier expresión aritmética o de cadenas de caracteres para los nombres de las columnas (posiblemente con variables de rango antepuestas) y constantes, y *nombre_columna* es un nombre nuevo para esa columna en el resultado de la consulta.

Las expresiones regulares en SQL. Para reflejar la creciente importancia de los datos de texto, SQL:1999 incluye una versión más potente del operador `LIKE` denominada `SIMILAR`. Este operador permite emplear una amplia variedad de expresiones regulares como estructuras mientras se busca texto. Las expresiones regulares son parecidas a las soportadas por el sistema operativo Unix para la búsqueda de cadenas de caracteres, aunque la sintaxis es un poco diferente.

También puede contener *agregados* como *sum* y *count*, que se tratarán en el Apartado 5.5. La norma de SQL también incluye expresiones para los valores de fecha y de hora, que no se tratarán aquí. Aunque no forme parte de la norma de SQL, muchas implementaciones soportan también el empleo de funciones predefinidas como *sqrt*, *sen* y *mod*.

(C17) Calcular el incremento de la categoría de las personas que han navegado en dos barcos diferentes el mismo día.

```
SELECT M.nombrem, M.categoría+1 AS categoría
FROM   Marineros M, Reservas R1, Reservas R2
WHERE  M.idm = R1.idm AND M.idm = R2.idm
       AND R1.fecha = R2.fecha AND R1.idb <> R2.idb
```

Además, cada elemento de la *condición* puede ser tan general como *expresión1 = expresión2*.

```
SELECT M1.nombrem AS nombre1, M2.nombrem AS nombre2
FROM   Marineros M1, Marineros M2
WHERE  2*M1.categoría = M2.categoría-1
```

Para la comparación de cadenas de caracteres se pueden emplear los operaciones de comparación (=, <, >, etc.) con el orden de las cadenas de caracteres determinado alfabéticamente, como de costumbre. Si hay que ordenar las cadenas de caracteres de manera distinta a la alfabética (por ejemplo, ordenar las cadenas de caracteres que denotan el nombre de los meses según su orden en el calendario enero, febrero, marzo, etc.), SQL soporta el concepto general de **ordenación**, u orden de colocación, para los conjuntos de caracteres. La ordenación permite que el usuario especifique los caracteres que son "menores que" otros y ofrece gran flexibilidad para la manipulación de cadenas de caracteres.

Además, SQL ofrece soporte para la comparación de estructuras mediante el operador `LIKE`, junto con el uso de los símbolos comodín % (que sustituye a cero o más caracteres arbitrarios) y _ (que sustituye exactamente a un carácter arbitrario). Así, '_AB%' denota una estructura que coincide con todas las cadenas de caracteres que contienen, como mínimo, tres caracteres, en las que el segundo y el tercer carácter son, respectivamente, A y B. Obsérvese que, a diferencia de los demás operadores de comparación, los espacios en blanco pueden resultar significativos para el operador `LIKE` (dependiendo de la ordenación del conjunto de caracteres subyacente). Por tanto, *'Jesús' = 'Jesús '* es verdadero, mientras que *'Jesús'* `LIKE` *'Jesús '* es falso. A continuación se da un ejemplo del empleo de `LIKE` en una consulta.

(C18) Averiguar la edad de los marineros cuyo nombre comienza por B, acaba por O y tiene, como mínimo, seis caracteres.

El álgebra relacional y SQL. Las operaciones de conjuntos de SQL están disponibles en el álgebra relacional. La principal diferencia, por supuesto, es que en SQL se trata de operaciones sobre *multiconjuntos*, ya que las tablas son multiconjuntos de tuplas.

```
SELECT M.edad
FROM   Marineros M
WHERE  M.nombrem LIKE 'B_%___O'
```

El único marinero así es Benito, y su edad es 63,5.

5.3 UNION, INTERSECT Y EXCEPT

SQL ofrece tres estructuras para la manipulación de conjuntos que amplían la forma básica de las consultas presentada anteriormente. Dado que la respuesta a cada consulta es un multiconjunto de filas, resulta natural considerar el empleo de operaciones como la unión, la intersección y la diferencia. SQL soporta estas operaciones con el nombre de `UNION, INTERSECT` y `EXCEPT`[4]. SQL ofrece también otras operaciones con conjuntos: `IN` (para comprobar si un elemento pertenece a un conjunto dado), `op ANY`, `op ALL` (para comparar un valor con los elementos de un conjunto dado, mediante el operador comparación `op`) y `EXISTS` (para comprobar si un conjunto está vacío). `IN` y `EXISTS` pueden llevar antepuesto `NOT`, con la modificación obvia de su significado. En este apartado se tratarán `UNION, INTERSECT` y `EXCEPT`, y las demás operaciones se tratarán en el Apartado 5.4.

Considérese la consulta siguiente:

(C5) Averiguar el nombre de los marineros que han reservado barcos rojos o verdes.

```
SELECT M.nombrem
FROM   Marineros M, Reservas R, Barcos B
WHERE  M.idm = R.idm AND R.idb = B.idb
       AND (B.color = 'rojo' OR B.color = 'verde')
```

Esta consulta se expresa fácilmente mediante la conectiva `OR` de la cláusula `WHERE`. Sin embargo, la consulta siguiente, que es idéntica salvo por el empleo de "y" en lugar de "o" en su versión en castellano, resulta ser mucho más difícil:

(C6) Averiguar el nombre de los marineros que han reservado tanto barcos rojos como barcos verdes.

Si simplemente hubiera que sustituir el empleo de `OR` en la consulta anterior por el de `AND`, en analogía con la expresión en castellano de las dos consultas, se recuperaría el nombre de los marineros que han reservado barcos que son a la vez rojos y verdes. La restricción de integridad de que *idb* es clave de Barcos indica que el mismo barco no puede tener dos colores y, por tanto, la variante de la consulta anterior con `AND` en lugar de `OR` devolverá siempre un conjunto de respuestas vacío. La expresión correcta de la consulta C6 mediante `AND` es la siguiente:

[4]Obsérvese que, aunque la norma de SQL incluye estas operaciones, muchos sistemas actualmente sólo soportan `UNION`. Además, muchos sistemas reconocen la palabra clave `MINUS` en lugar de `EXCEPT`.

```
SELECT M.nombrem
FROM   Marineros M, Reservas R1, Barcos B1, Reservas R2, Barcos B2
WHERE  M.idm = R1.idm AND R1.idb = B1.idb
       AND M.idm = R2.idm AND R2.idb = B2.idb
       AND B1.color='rojo' AND B2.color = 'verde'
```

Se puede pensar en R1 y en B1 como en filas que prueban que el marinero M.idm ha reservado barcos rojos. R2 y B2, de manera parecida, prueban que el mismo marinero ha reservado barcos verdes. M.nombrem no se incluye en el resultado a menos que se encuentren cinco filas como M, R1, B1, R2 y B2.

La consulta anterior es difícil de comprender (y también ineficiente de ejecutar, como puede verse). En especial, el parecido con la consulta anterior `OR` (la consulta C5) se pierde completamente. Una solución más adecuada para estas dos consultas es emplear `UNION` e `INTERSECT`.

La consulta `OR` (la consulta C5) se puede reescribir de la manera siguiente:

```
SELECT M.nombrem
FROM   Marineros M, Reservas R, Barcos B
WHERE  M.idm = R.idm AND R.idb = B.idb AND B.color = 'rojo'
UNION
SELECT M2.nombrem
FROM   Marineros M2, Barcos B2, Reservas R2
WHERE  M2.idm = R2.idm AND R2.idb = B2.idb AND B2.color = 'verde'
```

Esta consulta indica que se desea la unión del conjunto de marineros que han reservado barcos rojos con el conjunto de marineros que han reservado barcos verdes. En completa simetría, la consulta `AND` (la consulta C6) se puede reescribir de la manera siguiente:

```
SELECT M.nombrem
FROM   Marineros M, Reservas R, Barcos B
WHERE  M.idm = R.idm AND R.idb = B.idb AND B.color = 'rojo'
INTERSECT
SELECT M2.nombrem
FROM   Marineros M2, Barcos B2, Reservas R2
WHERE  M2.idm = R2.idm AND R2.idb = B2.idb AND B2.color = 'verde'
```

Esta consulta contiene realmente un fallo sutil —si hay dos marineros como Horacio en los ejemplares de ejemplo $B1$, $R2$ y $M3$, uno de los cuales ha reservado un barco rojo, mientras que el otro ha reservado un barco verde, se devolverá el nombre de Horacio aunque ninguna persona llamada Horacio haya reservado tanto barcos rojos como verdes—. Por tanto, la consulta calcula realmente nombres de marineros tales que algún marinero con ese nombre haya reservado un barco rojo y algún marinero con el mismo nombre (acaso un marinero diferente) haya reservado un barco verde.

Como se observó en el Capítulo 4, el problema surge porque se emplea *nombrem* para identificar a los marineros, y *nombrem* no es clave de Marineros. Si se selecciona *idm* en lugar de *nombrem* en la consulta anterior, se calculará el conjunto de *idm*s de marineros que han reservado tanto barcos rojos como barcos verdes. (Para calcular el nombre de esos marineros hace falta una consulta anidada; se volverá a este ejemplo en el Apartado 5.4.4.)

La siguiente consulta ilustra la operación diferencia de conjuntos en SQL.

(C19) Averiguar el idm *de todos los marineros que han reservado barcos rojos pero no verdes.*

```
SELECT M.idm
FROM   Marineros M, Reservas R, Barcos B
WHERE  M.idm = R.idm AND R.idb = B.idb AND B.color = 'rojo'
EXCEPT
SELECT M2.idm
FROM   Marineros M2, Reservas R2, Barcos B2
WHERE  M2.idm = R2.idm AND R2.idb = B2.idb AND B2.color = 'verde'
```

Los marineros 22, 64 y 31 han reservado barcos rojos. Los marineros 22, 74 y 31 han reservado barcos verdes. Por tanto, la respuesta contiene sólo el *idm* 64.

En realidad, dado que la relación Reservas contiene información sobre los idms, no hay necesidad alguna de consultar la relación Marineros, y se puede emplear la consulta siguiente, que es más sencilla:

```
SELECT R.idm
FROM   Barcos B, Reservas R
WHERE  R.idb = B.idb AND B.color = 'rojo'
EXCEPT
SELECT R2.idm
FROM   Barcos B2, Reservas R2
WHERE  R2.idb = B2.idb AND B2.color = 'verde'
```

Obsérvese que esta consulta confía en la integridad referencial; es decir, no hay reservas de marineros que no existan. Téngase en cuenta que `UNION`, `INTERSECT` y `EXCEPT` se pueden emplear en *cualesquiera* dos tablas que sean compatibles para la unión, es decir, tienen el mismo número de columnas y esas columnas, tomadas en orden, tienen el mismo tipo. Por ejemplo, se puede escribir la consulta siguiente:

(C20) Averiguar el idm de los marineros que tengan una categoría de 10 o hayan reservado el barco 104.

```
SELECT M.idm
FROM   Marineros M
WHERE  M.categoría = 10
UNION
SELECT R.idm
FROM   Reservas R
WHERE  R.idb = 104
```

La primera parte de la unión devuelve los *idm*s 58 y 71. La segunda parte devuelve 22 y 31. La respuesta es, por tanto, el conjunto de *idm*s 22, 31, 58 y 71. Un último punto a destacar sobre `UNION`, `INTERSECT` y `EXCEPT` es el siguiente. A diferencia del criterio predeterminado de que en la forma básica de las consultas no se eliminan los duplicados a menos que se especifique `DISTINCT`, el criterio predeterminado para las consultas con `UNION` es que los duplicados *sí* se eliminan. Para conservar los duplicados se debe emplear `UNION ALL`; en ese

El álgebra relacional y SQL. El anidamiento de consultas es una característica que no está disponible en el álgebra relacional, pero las consultas anidadas se pueden traducir al álgebra. El anidamiento en SQL está más inspirado por el cálculo relacional que por el álgebra. Junto con algunas otras características de SQL, como los operadores para (multi)conjuntos y la agregación, el anidamiento es una estructura muy expresiva.

caso, el número de copias de cada fila del resultado es siempre $m + n$, donde m y n son el número de veces que la fila aparece en las dos partes de la unión. De manera parecida, `INTERSECT ALL` conserva los duplicados —el número de copias de cada fila del resultado es $min(m, n)$— y `EXCEPT ALL` también conserva los duplicados —el número de copias de cada fila del resultado es $m - n$, donde m corresponde a la primera relación—.

5.4 CONSULTAS ANIDADAS

Una de las características más potentes de SQL son las consultas anidadas. Una **consulta anidada** es una consulta que tiene otra consulta incorporada en su interior; la consulta incorporada se denomina **subconsulta**. Por supuesto, la propia consulta incorporada puede ser una consulta anidada; por tanto, se pueden encontrar consultas que tengan estructuras anidadas muy profundas. Al escribir una consulta a veces hay que expresar una condición que hace referencia a alguna tabla que, a su vez, se debe calcular. Las consultas empleadas para calcular esas tablas subsidiarias son subconsultas y aparecen como parte de las consultas principales. Las subconsultas suelen aparecer en el interior de las cláusulas `WHERE` de las consultas. Las subconsultas pueden aparecer a veces en las cláusulas `FROM` o `HAVING` (que se presentarán en el Apartado 5.5). Este apartado sólo estudia las subconsultas que aparecen en las cláusulas `WHERE`. El tratamiento de las subconsultas que aparecen en otras partes de las consultas es muy parecido. Algunos ejemplos de subconsulta que aparecen en la cláusula `FROM` se estudian posteriormente en el Apartado 5.5.1.

5.4.1 Introducción a las consultas anidadas

A manera de ejemplo se reescribirá la consulta siguiente, examinada previamente, empleando una subconsulta anidada:

(C1) Averiguar el nombre de los marineros que han reservado el barco 103.

```
SELECT M.nombrem
FROM   Marineros M
WHERE  M.idm IN ( SELECT R.idm
                  FROM   Reservas R
                  WHERE  R.idb = 103 )
```

La subconsulta anidada calcula el (multi)conjunto de *idm*s de los marineros que han reservado el barco 103 (el conjunto contiene 22, 31 y 74 para los ejemplares $R2$ y $M3$) y la consulta de nivel superior recupera el nombre de los marineros cuyo *idm* se halla en ese conjunto. El operador `IN` permite comprobar si un valor pertenece a un conjunto de elementos

dado; para generar el conjunto que se desea comprobar se utiliza una consulta SQL. Obsérvese que resulta muy sencillo modificar esta consulta para averiguar todos los marineros que *no* han reservado el barco 103 —basta con sustituir `IN` por `NOT IN`—.

La mejor manera de comprender una consulta anidada es pensar en ella en términos de una estrategia de evaluación conceptual. En este ejemplo, la estrategia consiste en examinar las filas de Marineros y para cada una de ellas evaluar la subconsulta para Reservas. En general, la estrategia de evaluación conceptual que se ha presentado para definir la semántica de una consulta se puede ampliar para que abarque las consultas anidadas de la manera siguiente: se crea el producto cartesiano de las tablas de la cláusula `FROM` de la consulta de nivel anterior como se hizo anteriormente. Por cada fila del producto cartesiano, mientras se comprueba la condición en la cláusula `WHERE`, se (vuelve a) calcular la subconsulta[5]. Por supuesto, la propia subconsulta podría contener otra subconsulta anidada, en cuyo caso se aplicaría nuevamente la misma idea, lo que llevaría a una estrategia de evaluación con varios niveles de bucles anidados.

Como ejemplo de consulta con varios anidamientos, se reescribirá la consulta siguiente.

(C2) Averiguar el nombre de los marineros que han reservado barcos rojos.

```
SELECT M.nombrem
FROM   Marineros M
WHERE  M.idm IN ( SELECT R.idm
                  FROM   Reservas R
                  WHERE  R.idb IN ( SELECT B.idb
                                    FROM   Barcos B
                                    WHERE  B.color = 'rojo' )
```

La subconsulta más interna busca el conjunto de *idb*s de los barcos rojos (102 y 104 para el ejemplar $B1$). La subconsulta que se halla un nivel por encima de ella busca el conjunto de *idm*s de marineros que han reservado alguno de esos barcos. Para los ejemplares $B1$, $R2$ y $M3$ ese conjunto de *idm*s contiene 22, 31 y 64. La consulta de nivel superior busca el nombre de los marineros cuyo *idm* pertenece a ese conjunto de *idm*s; se obtiene Domínguez, Lorca y Horacio.

Para averiguar el nombre de los marineros que no han reservado ningún barco rojo, se sustituye la aparición más externa de `IN` por `NOT IN`, como se ilustra en la consulta siguiente.

(C21) Averiguar el nombre de los marineros que no *han reservado ningún barco rojo.*

```
SELECT M.nombrem
FROM   Marineros M
WHERE  M.idm NOT IN ( SELECT R.idm
                      FROM   Reservas R
                      WHERE  R.idb IN ( SELECT B.idb
                                        FROM   Barcos B
                                        WHERE  B.color = 'rojo' )
```

[5]Dado que la subconsulta interior del ejemplo no depende de la fila "actual" de la consulta externa de ninguna manera, uno se podría preguntar el motivo de tener que calcular de nuevo la subconsulta para cada fila externa. Para entender el porqué véase el Apartado 5.4.2.

Esta consulta calcula el nombre de los marineros cuyo *idm no* pertenece al conjunto 22, 31 y 64.

A diferencia de la consulta C21, se puede modificar la consulta anterior (la versión anidada de C2) sustituyendo la aparición interior (en vez de la aparición exterior) de `IN` por `NOT IN`. Esta consulta modificada calculará el nombre de los marineros que han reservado barcos que no son rojos, es decir, si han hecho alguna reserva, no es de ningún barco rojo. Considérese la manera de hacerlo. En la consulta interior se comprueba que *R.idb no* es ni 102 ni 104 (los *idb*s de los barcos rojos). La consulta exterior busca entonces los *idm*s de las tuplas de Reservas en las que el *idb* no es ni 102 ni 104. Para los ejemplares $B1$, $R2$ y $M3$, la consulta exterior calcula el conjunto de *idm*s 22, 31, 64 y 74. Finalmente, se averigua el nombre de los marineros cuyo *idm* pertenece a ese conjunto.

También se puede modificar la consulta anidada C2 sustituyendo las dos apariciones de `IN` por `NOT IN`. Esta variante busca el nombre de los marineros que no han reservado barcos que no sean rojos, es decir, que sólo han reservado barcos rojos (si es que han reservado algún barco). Actuando como en el párrafo anterior, para los ejemplares $B1$, $R2$ y $M3$, la consulta exterior calcula el conjunto de *idm*s (de Marineros) que no son 22, 31, 64 ni 74. Se trata del conjunto 29, 32, 58, 71, 85 y 95. Luego se busca el nombre de los marinerios cuyo *idm* pertenece a ese conjunto.

5.4.2 Consultas anidadas correlacionadas

En las consultas anidadas vistas hasta el momento la subconsulta interior ha sido completamente independiente de la consulta exterior. En general, la subconsulta interior puede depender de la fila que se está examinando en cada momento en la consulta exterior (en términos de la estrategia de evaluación conceptual). Reescribamos una vez más la siguiente consulta.

(C1) Averiguar el nombre de los marineros que han reservado el barco número 103.

```
SELECT M.nombrem
FROM   Marineros M
WHERE  EXISTS ( SELECT *
                FROM   Reservas R
                WHERE  R.idb = 103
                       AND R.idm = M.idm )
```

El operador `EXISTS` es otro operador para la comparación de conjuntos, como `IN`. Permite comprobar si un conjunto no está vacío, es decir, se trata de una comparación implícita de igualdad con el conjunto vacío. Por tanto, para cada fila M de Marineros, se comprueba si el conjunto de filas de Reservas R tal que *R.idb = 103* `AND` *M.idm = R.idm* no está vacío. Si no lo está, el marinero M ha reservado el barco 103 y se devuelve su nombre. La subconsulta depende claramente de la fila actual M y se debe volver a evaluar para cada fila de Marineros. La aparición de M en la subconsulta (en forma del literal *M.idm*) se denomina *correlación*, y estas consultas se denominan *consultas correlacionadas*.

Esta consulta también ilustra el empleo del símbolo especial * en situaciones en las que todo lo que se desea hacer es comprobar si existe una fila que cumpla la condición y no se desea realmente recuperar ninguna columna de esa fila. Éste es uno de los dos empleos de *

en la cláusula SELECT que se considera buen estilo de programación; el otro es un argumento de la operación de agregación COUNT, que se describirá en breve.

Como ejemplo adicional, mediante el empleo de NOT EXISTS en lugar de EXISTS, se puede calcular el nombre de los marineros que no han reservado barcos rojos. Estrechamente relacionado con EXISTS está el predicado UNIQUE. Cuando se aplica UNIQUE a una subconsulta, la condición resultante devuelve verdadero si ninguna fila aparece dos veces en la respuesta de la subconsulta, es decir, si no hay ningún duplicado; en especial, devuelve verdadero si la respuesta está vacía. (También hay una versión NOT UNIQUE.)

5.4.3 Operadores para la comparación de conjuntos

Ya se han visto los operadores para comparación de conjuntos EXISTS, IN y UNIQUE, junto con sus versiones negadas. SQL también soporta op ANY y op ALL, donde op es alguno de los operadores de comparación $\{<, <=, =, <>, >=, >\}$. (También está disponible SOME, pero no es más que un sinónimo de ANY.)

(C22) Averiguar los marineros cuya categoría es superior a la de algún marinero llamado Horacio.

```
SELECT  M.idm
FROM    Marineros M
WHERE   M.categoría > ANY ( SELECT  M2.categoría
                            FROM    Marineros M2
                            WHERE   M2.nombrem = 'Horacio' )
```

Si hay varios marineros llamados Horacio, esta consulta halla todos los marineros cuya categoría es superior que la de *algún* marinero llamado Horacio. Para el ejemplar $M3$, esto supone los *idm*s 31, 32, 58, 71 y 74. ¿Qué ocurriría si *no* hubiera ningún marinero llamado Horacio? En ese caso la comparación *M.categoría* > ANY ... devuelve por definición falso, y la consulta devuelve un conjunto de respuestas vacío. Para comprender las comparaciones que emplean ANY resulta útil considerar que la comparación se ejecuta de manera reiterada. En este ejemplo, *M.categoría* se compara con éxito con cada valor de la categoría que constituye una respuesta de la consulta anidada. De manera intuitiva, la subconsulta debe devolver una fila que haga que la comparación sea verdadera, con objeto de que *M.categoría* > ANY ... devuelva verdadero.

(C23) Averiguar los marineros cuya categoría es superior a la de todos los marineros llamados Horacio.

Todas esas consultas se pueden conseguir con una sencilla modificación de la consulta C22: basta con sustituir ANY por ALL en la cláusula WHERE de la consulta exterior. Para el ejemplar $M3$ se obtienen los *idm*s 58 y 71. Si no hubiera ningún marinero llamado Horacio, la comparación *M.categoría* > ALL ... devuelve por definición verdadero, por lo que la consulta devolvería el nombre de todos los marineros. Una vez más resulta útil pensar que la comparación se lleva a cabo de manera reiterada. De manera intuitiva, la comparación debe ser verdadera para todas las filas devueltas para que *M.categoría* > ALL ... devuelva verdadero.

Como ilustración adicional de ALL considérese la consulta siguiente.

(C24) Averiguar los marineros que tienen la máxima categoría.

```
SELECT M.idm
FROM   Marineros M
WHERE  M.categoría >= ALL ( SELECT M2.categoría
                            FROM   Marineros M2 )
```

La subconsulta calcula el conjunto de todos los valores de categoría de Marineros. La condición exterior `WHERE` sólo se satisface cuando *M.categoría* es mayor o igual que cada uno de esos valores de categoría, es decir, cuando es el valor de categoría más elevado. En el ejemplar $M3$ la condición sólo se satisface para la *categoría* 10, y la respuesta incluye los *idm*s de los marineros con esa categoría, es decir, 58 y 71.

Obsérvese que `IN` y `NOT IN` son equivalentes a = `ANY` y <> `ALL`, respectivamente.

5.4.4 Más ejemplos de consultas anidadas

Revisemos una consulta que se tomó en consideración anteriormente empleando el operador `INTERSECT`.

(C6) Averiguar el nombre de los marineros que han reservado tanto barcos rojos como barcos verdes.

```
SELECT M.nombrem
FROM   Marineros M, Reservas R, Barcos B
WHERE  M.idm = R.idm AND R.idb = B.idb AND B.color = 'rojo'
       AND M.idm IN ( SELECT M2.idm
                      FROM   Marineros M2, Barcos B2, Reservas R2
                      WHERE  M2.idm = R2.idm AND R2.idb = B2.idb
                             AND B2.color = 'verde' )
```

Esta consulta se puede entender de la manera siguiente: "Averiguar todos los marineros que han reservado barcos rojos y, además, tienen *idm*s que están incluidos en el conjunto de *idm*s de marineros que han reservado barcos verdes." Esta formulación de la consulta ilustra la manera en que las consultas que implican a `INTERSECT` se pueden reescribir empleando `IN`, lo que resulta útil saber si el sistema no soporta `INTERSECT`. Las consultas que emplean `EXCEPT` se pueden reescribir de manera parecida empleando `NOT IN`. Para averiguar los *idm*s de los marineros que han reservado barcos rojos pero no barcos verdes basta con sustituir la palabra clave `IN` de la consulta anterior por `NOT IN`.

Como puede verse, escribir la consulta (C6) empleando `INTERSECT` resulta más complicado porque hay que utilizar los *idm*s para identificar a los marineros (durante la intersección) y hay que devolver el nombre de los marineros:

```
SELECT M.nombrem
FROM   Marineros M
WHERE  M.idm IN (( SELECT R.idm
                   FROM   Barcos B, Reservas R
                   WHERE  R.idb = B.idb AND B.color = 'rojo' )
                   INTERSECT
                   (SELECT R2.idm
                   FROM   Barcos B2, Reservas R2
```

Funciones de agregación de SQL:1999. El conjunto de funciones de agregación se amplía enormemente en la nueva norma, que incluye varias funciones estadísticas como la desviación normal, la covarianza y los percentiles. No obstante, las nuevas funciones de agregación se encuentran en el paquete SQL/OLAP y puede que no todos los fabricantes las soporten.

```
                    WHERE  R2.idb = B2.idb AND B2.color = 'verde' ))
```

El siguiente ejemplo ilustra la manera en que se puede expresar en SQL la operación *división* del álgebra relacional.

(C9) Averiguar el nombre de los marineros que han reservado todos los barcos.

```
SELECT M.nombrem
FROM   Marineros M
WHERE  NOT EXISTS (( SELECT B.idb
                     FROM   Barcos B )
                     EXCEPT
                     (SELECT R.idb
                     FROM   Reservas R
                     WHERE  R.idm = M.idm ))
```

Obsérvese que esta consulta está correlacionada —para cada marinero M se comprueba si el conjunto de barcos reservado por M incluye todos los barcos—. Una manera alternativa de llevar a cabo esta consulta sin emplear `EXCEPT` es la siguiente:

```
SELECT M.nombrem
FROM   Marineros M
WHERE  NOT EXISTS ( SELECT B.idb
                    FROM   Barcos B
                    WHERE  NOT EXISTS ( SELECT R.idb
                                        FROM   Reservas R
                                        WHERE  R.idb = B.idb
                                               AND R.idm = S.idm ))
```

De manera intuitiva, para cada marinero se comprueba que no hay ningún barco que no haya sido reservado por ese marinero.

5.5 OPERADORES DE AGREGACIÓN

Además de recuperar simplemente datos, a menudo se desea llevar a cabo algún cálculo o resumen. Como ya se ha indicado en este capítulo, SQL permite el empleo de expresiones aritméticas. Consideremos ahora una potente clase de estructuras para el cálculo de *valores de agregación* como `MIN` y `SUM`. Estas características representan una ampliación significativa del álgebra relacional. SQL soporta cinco operaciones de agregación, que se pueden aplicar a cualquier columna, como A, de una relación dada:

1. `COUNT` ([`DISTINCT`] A): el número de valores (únicos) de la columna A.
2. `SUM` ([`DISTINCT`] A): la suma de todos los valores (únicos) de la columna A.
3. `AVG` ([`DISTINCT`] A): el promedio de todos los valores (únicos) de la columna A.
4. `MAX` (A): el valor máximo de la columna A.
5. `MIN` (A): el valor mínimo de la columna A.

Obsérvese que no tiene sentido especificar `DISTINCT` en conjunción con `MIN` o `MAX` (aunque SQL no lo impida).

(C25) Averiguar el promedio de edad de todos los marineros.

```
SELECT AVG (M.edad)
FROM   Marineros M
```

Para el ejemplar $M3$ el promedio de edad es 37,4. Por supuesto, se puede utilizar la cláusula `WHERE` para restringir los marineros que se emplean en el cálculo de la edad promedio.

(C26) Averiguar el promedio de edad de los marineros con categoría 10.

```
SELECT AVG (M.edad)
FROM   Marineros M
WHERE  M.categoría = 10
```

Hay dos marineros así, y su promedio de edad es 25,5. Se pueden emplear `MIN` (o `MAX`) en lugar de `AVG` en las consultas anteriores para averiguar la edad del marinero más joven (o más viejo). Sin embargo, hallar tanto el nombre como la edad del marinero más anciano resulta más complicado, como ilustra la consulta siguiente.

(C27) Averiguar el nombre y la edad del marinero más anciano.

Considérese el siguiente intento de responder a esta consulta:

```
SELECT M.nombrem, MAX (M.edad)
FROM   Marineros M
```

La pretensión es que esta consulta no sólo devuelva la edad máxima, sino también el nombre de los marineros que tienen esa edad. Sin embargo, esta consulta es ilegal en SQL —si la cláusula `SELECT` emplea alguna operación de agregación, *only* debe emplear operaciones de agregación, a menos que la consulta contenga alguna cláusula `GROUP BY`—. (La intuición subyacente a esta restricción debería quedar clara cuando se estudie la cláusula `GROUP BY` en el Apartado 5.5.1.) Por tanto, no se puede emplear `MAX` (M.edad) ni M.nombrem en la cláusula `SELECT`. Hay que emplear consultas anidadas para calcular la respuesta deseada a C27:

```
SELECT M.nombrem, M.edad
FROM   Marineros M
WHERE  M.edad = ( SELECT MAX (M2.edad)
                  FROM Marineros M2 )
```

Obsérvese que se ha empleado el resultado de una operación de agregación en la subconsulta como argumento de una operación de comparación. Hablando estrictamente, estamos comparando un valor de edad con el resultado de la subconsulta, que es una relación. Sin embargo, debido al empleo de la operación de agregación, se garantiza que la subconsulta devuelva una sola tupla con un solo campo, y SQL convierte esa relación en un valor de campo con vistas a la comparación. La siguiente consulta equivalente a C27 es legal en la norma de SQL pero, desafortunadamente, muchos sistemas no la soportan:

```
SELECT M.nombrem, M.edad
FROM   Marineros M
WHERE  ( SELECT MAX (M2.edad)
         FROM Marineros M2 ) = M.edad
```

Se puede contar el número de marineros con `COUNT`. Este ejemplo ilustra el empleo de * como argumento de `COUNT`, que resulta útil cuando se desea contar todas las filas.

(C28) Contar el número de marineros.

```
SELECT COUNT (*)
FROM   Marineros M
```

Se puede considerar * una abreviatura de todas las columnas (del producto cartesiano de la **lista-de-tablas** de la cláusula `FROM`). Compárese esta consulta con la siguiente, que calcula el número de nombres de marinero diferentes. (Recuérdese que *nombrem* no es clave.)

(C29) Contar el número de nombres de marinero diferentes.

```
SELECT COUNT ( DISTINCT M.nombrem )
FROM   Marineros M
```

Para el ejemplar $M3$ la respuesta a C28 es 10, mientras que la respuesta a C29 es 9 (ya que dos marineros tienen el mismo nombre, Horacio). Si se omite `DISTINCT`, la respuesta a C29 es 10, ya que el nombre Horacio se cuenta dos veces. Si `COUNT` no incluye `DISTINCT`, entonces `COUNT(*)` da la misma respuesta que `COUNT(`x`)`, donde x es cualquier conjunto de atributos. En el ejemplo anterior y sin `DISTINCT`, C29 es equivalente a C28. No obstante, el empleo de `COUNT` (*) supone un mejor estilo de consulta, ya que inmediatamente queda claro que todos los registros contribuyen a la cuenta total.

Las operaciones de agregación ofrecen una alternativa a las estructuras `ANY` y `ALL`. Por ejemplo, considérese la consulta siguiente:

(C30) Averiguar el nombre de los marineros de más edad que el marinero más viejo de categoría 10.

```
SELECT M.nombrem
FROM   Marineros M
WHERE  M.edad > ( SELECT MAX ( M2.edad )
                  FROM   Marineros M2
                  WHERE  M2.categoría = 10 )
```

Para el ejemplar $M3$ el marinero de más edad de categoría 10 es el marinero 58, cuya edad es 35. Los nombres de los marineros de más edad son Benito, Domínguez, Horacio y Lorca. Empleando `ALL` esta consulta se podría escribir también de la manera siguiente:

El álgebra relacional y SQL. La agregación es una operación fundamental que no se puede expresar en el álgebra relacional. De manera parecida, la estructura de agrupamiento de SQL no se puede expresar en el álgebra.

```
SELECT M.nombrem
FROM   Marineros M
WHERE  M.edad > ALL ( SELECT M2.edad
                      FROM   Marineros M2
                      WHERE  M2.categoría = 10 )
```

Sin embargo, la consulta `ALL` es más susceptible de sufrir errores —fácil (e incorrectamente) se podría emplear `ANY` en lugar de `ALL` y recuperar marineros de más edad que *algún* marinero de categoría 10—. El empleo de `ANY` se corresponde intuitivamente con el de `MIN`, en lugar de `MAX`, en la consulta anterior.

5.5.1 Las cláusulas GROUP BY y HAVING

Hasta ahora se han aplicado las operaciones de agregación a todas las filas (que tenían las condiciones adecuadas) de la relación. A menudo se desea aplicar operaciones de agregación a cada uno de los **grupos** de filas de una relación, donde el número de grupos depende del ejemplar de esa relación (es decir, no se conoce con antelación). Por ejemplo, considérese la consulta siguiente.

(C31) Averiguar la edad del marinero más joven de cada categoría.

Si se sabe que las categorías son enteros que van del 1 al 10, se pueden escribir diez consultas de esta forma:

```
SELECT MIN (M.edad)
FROM   Marineros M
WHERE  M.categoría = i
```

donde $i = 1, 2, \ldots, 10$. Escribir diez consultas así resulta tedioso. Lo que es más importante, puede que no se conozca con antelación las categorías que hay.

Para escribir estas consultas hace falta una importante extensión de la forma básica de consulta SQL: la cláusula `GROUP BY`. De hecho, la ampliación también incluye la cláusula opcional `HAVING`, que se puede emplear para especificar condiciones para los grupos (por ejemplo, puede que sólo interesen categorías > 6). La forma general de una consulta SQL con estas extensiones es:

```
SELECT   [ DISTINCT ] lista-de-selección
FROM     lista-de-tablas
WHERE    condición
GROUP BY lista-para-formar-grupos
HAVING   condición-sobre-grupos
```

Mediante la cláusula `GROUP BY` se puede escribir C31 de la manera siguiente:

```
SELECT    M.categoría, MIN (M.edad)
FROM      Marineros M
GROUP BY  M.categoría
```

Consideremos algunos aspectos importantes de las nuevas cláusulas:

- La **lista-de-selección** de la cláusula `SELECT` consta de (1) una lista de nombres de columna y (2) una lista de términos que tienen la forma **opagr** (*nombre-columna*) `AS` *nombre-nuevo*. Ya se vió el empleo de `AS` para cambiar el nombre a las columnas del resultado. Las columnas que forman el resultado de los operadores de agregación no tienen todavía nombre de columna y, por tanto, resulta especialmente útil darles nombre con `AS`.

  Todas las columnas que aparecen en (1) deben aparecer también en **lista-para-formar-grupos**. El motivo es que cada fila del resultado de la consulta se corresponde con un *grupo*, que es un conjunto de filas que concuerdan con los valores de las columnas de **lista-para-formar-grupos**. En general, si una columna aparece en la lista (1), pero no en **lista-para-formar-grupos**, puede haber varias filas de un mismo grupo que tengan valores diferentes en esa columna, y no resulta evidente el valor que se le debe asignar en la fila de la respuesta.

  A veces se puede utilizar información de la clave primaria para comprobar que una columna dada tiene un valor único en todas las filas de cada grupo. Por ejemplo, si **lista-para-formar-grupos** contiene la clave primaria de una tabla de la **lista-de-tablas**, todas las columnas de esta tabla tendrán un valor único dentro de cada grupo. En SQL:1999 también se permite que esas columnas aparezcan en la parte (1) de la **lista-de-selección**.

- Las expresiones que aparecen en **grupo-condición** de la cláusula `HAVING` deben tener un *solo* valor por grupo. La intuición es que la cláusula `HAVING` determina si hay que generar una fila de respuesta para un grupo dado. Para satisfacer este requisito en SQL-92, las columnas que aparecen en **grupo-condición** deben aparecer como argumentos del operador de agregación, o deben aparecer también en **lista-para-formar-grupos**. En SQL:1999 se han introducido dos nuevas funciones para conjuntos que permiten comprobar si *todas* o *alguna* de las filas de un grupo dado satisfacen una condición determinada; esto permite emplear condiciones parecidas a las de la cláusula `WHERE`.

- Si se omite `GROUP BY`, se considera a toda la tabla como un solo grupo.

La semántica de estas consultas explica mediante un ejemplo.

(C32) Averiguar la edad del marinero más joven que tiene derecho a voto (es decir, tiene, como mínimo, 18 años) para cada categoría que tenga, como mínimo, dos marineros con derecho a voto.

```
SELECT    M.categoría, MIN (M.edad) AS edadmín
FROM      Marineros M
WHERE     M.edad >= 18
GROUP BY  M.categoría
HAVING    COUNT (*) > 1
```

Esta consulta se evaluará con el ejemplar $M3$ de Marineros que se reproduce por comodidad en la Figura 5.10. Para la ampliación de la estrategia de evaluación conceptual del Apartado 5.2 se procede de la manera siguiente. El primer paso es construir el producto cartesiano de las tablas de la **lista-de-tablas**. Como la única relación de la **lista-de-tablas** de la consulta C32 es Marineros, el resultado es el ejemplar mostrado en la Figura 5.10.

idm	*nombrem*	*categoría*	*edad*
22	Domínguez	7	45,0
29	Bravo	1	33,0
31	Lorca	8	55,5
32	Alández	8	25,5
58	Rubio	10	35,0
64	Horacio	7	35,0
71	Zuazo	10	16,0
74	Horacio	9	35,0
85	Arturo	3	25,5
95	Benito	3	63,5
96	Francisco	3	25,5

Figura 5.10 El ejemplar $M3$ de Marineros

El segundo paso es aplicar la condición de la cláusula `WHERE`, $M.edad >= 18$. Este paso elimina la fila $\langle 71, zuazo, 10, 16\rangle$. El tercer paso es eliminar las columnas no deseadas. Sólo son necesarias las columnas mencionadas en la cláusula `SELECT`, la cláusula `GROUP BY` o la cláusula `HAVING`, lo que significa que se pueden eliminar *idm* y *nombrem* del ejemplo. El resultado puede verse en la Figura 5.11. Obsérvese que hay dos filas idénticas con *categoría* 3 y *edad* 25,5 —SQL no elimina los duplicados excepto cuando se le obliga a hacerlo mediante el empleo de la palabra clave `DISTINCT`—. El número de copias de cada fila en la tabla intermedia de la Figura 5.11 viene determinado por el número de filas de la tabla original que tenían esos valores en las columnas proyectadas.

El cuarto paso es ordenar la tabla de acuerdo con la cláusula `GROUP BY` para identificar los grupos. El resultado de este paso puede verse en la Figura 5.12.

El quinto paso es aplicar la condición de grupos de la cláusula `HAVING`, es decir, la condición `COUNT` $(*) > 1$. Este paso elimina los grupos con *categoría* igual a 1, 9 y 10. Obsérvese que el orden en que se toman en consideración las cláusulas `WHERE` y `GROUP BY` es significativo: si no se tomara en consideración en primer lugar la cláusula `WHERE`, el grupo con *categoría=10* habría cumplido la condición para grupos de la cláusula `HAVING`. El sexto paso es generar una fila de respuesta para cada grupo restante. La fila de respuesta correspondiente a cada grupo consiste en un subconjunto de las columnas de agrupación, y una o más columnas generadas mediante la aplicación de un operador de agregación. En este ejemplo cada fila de respuesta tiene una columna *categoría* y una columna *edadmín*, que se calcula aplicando `MIN` a los valores de la columna *edad* del grupo correspondiente. El resultado de este paso puede verse en la Figura 5.13.

categoría	*edad*
7	45,0
1	33,0
8	55,5
8	25,5
10	35,0
7	35,0
9	35,0
3	25,5
3	63,5
3	25,5

Figura 5.11 Tras la evaluación del paso 3

categoría	*edad*
1	33,0
3	25,5
3	25,5
3	63,5
7	45,0
7	35,0
8	55,5
8	25,5
9	35,0
10	35,0

Figura 5.12 Tras la evaluación del paso 4

categoría	*edadmín*
3	25,5
7	35,0
8	25,5

Figura 5.13 Resultado final de la evaluación del ejemplo

Ampliaciones de SQL:1999. Se han añadido dos nuevas funciones para conjuntos, `EVERY` y `ANY`. Cuando se emplean en la cláusula `HAVING`, la intuición básica de que la cláusula especifica una condición que debe satisfacer cada grupo, considerado en conjunto, no se modifica. Sin embargo, ahora la condición puede implicar pruebas para cada tupla del grupo, mientras que anteriormente se basaba de manera exclusiva en las funciones de agregación aplicadas al grupo de tuplas.

Si la consulta contiene `DISTINCT` en la cláusula `SELECT`, los duplicados se eliminan en un paso final adicional.

SQL:1999 ha introducido dos nuevas funciones para conjuntos, `EVERY` y `ANY`. Para ilustrarlas, se puede sustituir la cláusula `HAVING` del ejemplo por

```
HAVING   COUNT (*) > 1 AND EVERY ( M.edad <= 60 )
```

El quinto paso de la evaluación conceptual es el afectado por la modificación de la cláusula `HAVING`. Considérese el resultado del cuarto paso, que puede verse en la Figura 5.12. La palabra clave `EVERY` exige que todas las filas de cada grupo satisfagan la condición adjunta para superar la condición para grupos. El grupo de la *categoría* 3 no cumple este criterio y se descarta; el resultado se muestra en la Figura 5.14.

Merece la pena constrastar la consulta anterior con la siguiente, en la que la condición para *edad* se halla en la cláusula `WHERE` en lugar de en la cláusula `HAVING`:

```
SELECT    M.categoría, MIN (M.edad) AS edadmín
FROM      Marineros M
WHERE     M.edad >= 18 AND M.edad <= 60
GROUP BY  M.categoría
HAVING    COUNT (*) > 1
```

Ahora, el resultado tras el tercer paso de la evaluación conceptual ya no contiene la fila con *edad* 63,5. No obstante, el grupo para la *categoría* 3 satisface la condición `COUNT (*) > 1`, ya que sigue teniendo dos filas, y cumple el criterio de condición de grupos aplicado en el quinto paso. El resultado final de esta consulta puede verse en la Figura 5.15.

categoría	*edadmín*
7	45,0
8	55,5

Figura 5.14 Resultado final de la consulta `EVERY`

categoría	*edadmín*
3	25,5
7	45,0
8	55,5

Figura 5.15 Resultado de la consulta alternativa

5.5.2 Más ejemplos de consultas de agregación

(C33) Para cada barco rojo, averiguar el número de reservas realizadas.

```
SELECT    B.idb, COUNT (*) AS númreservas
FROM      Barcos B, Reservas R
WHERE     R.idb = B.idb AND B.color = 'rojo'
GROUP BY  B.idb
```

Para los ejemplares $B1$ y $R2$, la respuesta a esta consulta contiene las tuplas $\langle 102, 3\rangle$ y $\langle 104, 2\rangle$.

Obsérvese que esta versión de la consulta anterior es ilegal:

```
SELECT    B.idb, COUNT (*) AS númreservas
FROM      Barcos B, Reservas R
WHERE     R.idb = B.idb
GROUP BY  B.idb
HAVING    B.color = 'rojo'
```

Aunque la condición para grupos *B.color = 'rojo'* tiene un solo valor para cada grupo, ya que el atributo de agrupación *idb* es clave de Barcos (y, por tanto, determina *color*), SQL no permite esta consulta[6]. Sólo pueden aparecer en la cláusula `HAVING` las columnas que aparecen en la cláusula `GROUP BY`, a menos que aparezcan como argumentos de algún operador de agregación de la cláusula `HAVING`.

(C34) Averiguar la edad media de los marineros de cada categoría que tenga, como mínimo, dos marineros.

[6]Esta consulta se puede reescribir fácilmente para que sea legal en SQL:1999 empleando `EVERY` en la cláusula `HAVING`.

```
SELECT   M.categoría, AVG (M.edad) AS edadmed
FROM     Marineros M
GROUP BY M.categoría
HAVING   COUNT (*) > 1
```

Tras identificar los grupos basados en la *categoría*, sólo se conservan los grupos con dos marineros como mínimo. La respuesta a esta consulta para el ejemplar $M3$ puede verse en la Figura 5.16.

categoría	*edadmed*
3	44,5
7	40,0
8	40,5
10	25,5

Figura 5.16 Respuesta a C34

categoría	*edadmed*
3	45,5
7	40,0
8	40,5
10	35,0

Figura 5.17 Respuesta a C35

categoría	*edadmed*
3	45,5
7	40,0
8	40,5

Figura 5.18 Respuesta a C36

La siguiente formulación alternativa de la consulta C34 ilustra que la cláusula `HAVING` puede tener subconsultas anidadas, al igual que la cláusula `WHERE`. Obsérvese que se puede emplear *M.categoría* dentro de la consulta anidada de la cláusula `HAVING`, ya que tiene un solo valor para el grupo actual de marineros:

```
SELECT   M.categoría, AVG ( M.edad ) AS edadmed
FROM     Marineros M
GROUP BY M.categoría
HAVING   1 < ( SELECT COUNT (*)
               FROM   Marineros M2
               WHERE  M.categoría = M2.categoría )
```

(C35) Averiguar la edad media de los marineros con derecho a voto (es decir, con 18 años de edad como mínimo) para cada categoría que tenga, como mínimo, dos marineros.

```
SELECT   M.categoría, AVG ( M.edad ) AS edadmed
FROM     Marineros M
WHERE    M. edad >= 18
GROUP BY M.categoría
HAVING   1 < ( SELECT COUNT (*)
               FROM   Marineros M2
               WHERE  M.categoría = M2.categoría )
```

En esta variante de la consulta C34 primero se eliminan las tuplas con *edad* <= 18 y se agrupan las tuplas restantes por *categoría*. La subconsulta de la cláusula `HAVING` calcula para cada grupo el número de tuplas de Marineros (sin aplicar la selección *edad* <= 18) con el mismo valor de *categoría* que el grupo en cuestión. Si algún grupo tiene menos de dos marineros, se descarta. Para cada grupo restante se obtiene como resultado la edad media. La respuesta a esta consulta con el ejemplar $M3$ puede verse en la Figura 5.17. Obsérvese que la respuesta es muy parecida a la de la consulta C34, con la única diferencia de que para el grupo de categoría 10 ahora se ignora al marinero de 16 años de edad al calcular la media.

(C36) Averiguar la edad media de los marineros que tienen derecho a voto (es decir, tienen, como mínimo, 18 años) para cada categoría que tiene, como mínimo, dos marineros así.

```
SELECT    M.categoría, AVG ( M.edad ) AS edadmed
FROM      Marineros M
WHERE     M. edad > 18
GROUP BY  M.categoría
HAVING    1 < ( SELECT COUNT (*)
                FROM   Marineros M2
                WHERE  M.categoría = M2.categoría AND M2.edad >= 18 )
```

Esta formulación de la consulta muestra su parecido con C35. La respuesta a C36 para el ejemplar $M3$ puede verse en la Figura 5.18. Se diferencia de la respuesta a C35 en que no hay ninguna tupla para la categoría 10, ya que sólo hay una tupla de categoría 10 y $edad \geq 18$.

La consulta C36 es realmente muy parecida a C32, como muestra la siguiente formulación más sencilla:

```
SELECT    M.categoría, AVG ( M.edad ) AS edadmed
FROM      Marineros M
WHERE     M. edad > 18
GROUP BY  M.categoría
HAVING    COUNT (*) > 1
```

Esta formulación de C36 se aprovecha del hecho de que la cláusula `WHERE` se aplica antes de llevar a cabo la agrupación; por tanto, sólo quedan los marineros de $edad > 18$ cuando se agrupa. Resulta instructivo considerar otra manera más de escribir esta consulta:

```
SELECT Temp.categoría, Temp.edadmed
FROM   ( SELECT    M.categoría, AVG ( M.edad ) AS edadmed,
                   COUNT (*) AS númcategorías
         FROM      Marineros M
         WHERE     M. edad > 18
         GROUP BY  M.categoría ) AS Temp
WHERE Temp.númcategorías > 1
```

Esta alternativa trae a colación varios puntos interesantes. En primer lugar, la cláusula `FROM` también puede contener subconsultas anidadas de acuerdo con la norma de SQL[7]. En segundo lugar, la cláusula `HAVING` no es necesaria en absoluto. Cualquier consulta con cláusulas `HAVING` se puede reescribir sin ellas, pero muchas consultas resultan más sencillas de expresar con la cláusula `HAVING`. Finalmente, cuando aparecen subconsultas en la cláusula `FROM`, es necesario emplear la palabra clave `AS` para darles nombre (ya que, en caso contrario, no podrían expresar, por ejemplo, la condición *Temp.númcategorías > 1*).

(C37) Averiguar las categorías para las que la edad media de los marineros es mínima.

Esta consulta se emplea para ilustrar que las operaciones de agregación no se pueden anidar. Se podría considerar escribirla de la manera siguiente:

[7] No todos los sistemas comerciales de bases de datos soportan actualmente consultas anidadas en la cláusula `FROM`.

El modelo relacional y SQL. Los valores nulos no forman parte del modelo relacional básico. Al igual que el tratamiento de las tablas como multiconjuntos de tuplas en SQL, se trata de una extensión del modelo básico.

```
SELECT   M.categoría
FROM     Marineros M
WHERE    AVG (M.edad) = ( SELECT   MIN (AVG (M2.edad))
                          FROM     Marineros M2
                          GROUP BY M2.categoría )
```

Una pequeña reflexión muestra que esta consulta no funcionaría aunque la expresión `MIN (AVG (M2.edad))` estuviera permitida, la cual es ilegal. En la consulta anidada, Marineros se divide en grupos por categorías y la edad media se calcula para cada valor de la categoría. Para cada grupo, la aplicación de `MIN` a este valor medio de la edad del grupo devuelve el mismo valor. A continuación se ofrece una versión correcta de esta consulta. Fundamentalmente calcula una tabla temporal que contiene la edad media de cada valor de categoría y luego halla la(s) categoría(s) para las que esa edad media es mínima.

```
SELECT Temp.categoría, Temp.edadmed
FROM   ( SELECT   M.categoría, AVG (M.edad) AS edadmed,
         FROM     Marineros M
         GROUP BY M.categoría) AS Temp
WHERE  Temp.edadmed = ( SELECT MIN (Temp.edadmed) FROM Temp )
```

La respuesta a esta consulta con el ejemplar $M3$ es $\langle 10, 25{,}5 \rangle$.
Como ejercicio, considérese si la consulta siguiente calcula la misma respuesta.

```
SELECT   Temp.categoría, MIN ( Temp.edadmed )
FROM     ( SELECT   M.categoría, AVG (M.edad) AS edadmed,
           FROM     Marineros M
           GROUP BY M.categoría ) AS Temp
GROUP BY Temp.categoría
```

5.6 VALORES NULOS

Hasta ahora se ha asumido que los valores de las columnas para una fila dada son siempre conocidos. En la práctica, los valores de las columnas pueden ser `desconocidos`. Por ejemplo, cuando un marinero, por ejemplo Dani, se hace socio de un club náutico, puede que no tenga todavía categoría asignada. Dado que la definición de la tabla Marineros tiene una columna *categoría*, ¿qué fila hay que insertar para Dani? Lo que hace falta es un valor especial que denote *desconocido*. Supóngase que la definición de la tabla Marineros se modifica para incluir la columna *segundo-apellido*. Sin embargo, sólo las personas de origen hispano tienen segundo apellido. Para las personas de otras nacionalidades la columna *segundo apellido* es *inaplicable*. Una vez más, ¿qué valor se incluye en esa columna para la fila que representa a Dani?

SQL ofrece un valor especial para las columnas denominado *null* (nulo) para emplearlo en estas situaciones. Se emplea *null* cuando el valor de la columna es *desconocido* o *inaplicable*.

Empleando la definición de la tabla Marineros se puede introducir la fila $\langle 98, Dani, null, 39\rangle$ para representar a Dani. La presencia de valores *null* complica muchas situaciones y en este apartado se considera su impacto sobre SQL.

5.6.1 Comparaciones que emplean valores nulos

Considérese una comparación como *categoría = 8*. Si se aplica a la fila de Dani, ¿esta condición es verdadera o falsa? Dado que la categoría de Dani es desconocida, resulta razonable decir que esta consideración debería tomar el valor `desconocido`. De hecho, éste es el caso para las comparaciones *categoría* > 8 y también para *categoría* < 8. Acaso de manera menos obvia, si se comparan dos valores *null* mediante $<, >, =$, etcétera, el resultado siempre es `desconocido`. Por ejemplo, si se tiene *null* en dos filas distintas de la relación Marineros, cualquier comparación devuelve `desconocido`.

SQL también ofrece el operador de comparación especial `IS NULL` para comprobar si el valor de alguna columna es *null*; por ejemplo, se puede decir que *categoría* `IS NULL`, que tomaría el valor `verdadero` para la fila que representa a Dani. También se puede decir que *categoría* `IS NOT NULL`, que tomaría el valor `falso` para la fila de Dani.

5.6.2 Las conectivas lógicas AND, OR y NOT

Considérense ahora expresiones booleanas como *categoría* $= 8$ `OR` *edad* < 40 y *categoría* $= 8$ `AND` *edad* < 40. Tomando nuevamente en consideración la fila de Dani, como *edad* < 40, la primera expresión toma el valor de verdadera independientemente del valor de *categoría*, pero ¿qué ocurre con la segunda? Sólo se puede decir `desconocido`.

Pero este ejemplo destaca un problema importante —una vez se tienen valores *null* hay que definir los operadores lógicos `AND`, `OR` y `NOT` mediante una lógica *de tres valores* en la que las expresiones toman el valor de `verdadero`, `falso` o `desconocido`—. Se amplían las interpretaciones habituales de `AND`, `OR` y `NOT` para que cubran el caso en el que uno de los argumentos es `desconocido` de la manera siguiente. La expresión `NOT desconocido` se define como `desconocido`. `OR` para dos argumentos adopta el valor `verdadero` si cualquiera de los dos tiene el valor de `verdadero` y como `desconocido` si uno de los argumentos toma el valor `falso` y el otro toma el valor `desconocido`. (Si los dos argumentos son `falsos`, por supuesto, `OR` toma el valor `falso`.) `AND` para dos argumentos adopta el valor `falso` si cualquiera de los argumentos es `falso`, y `desconocido` si uno de ellos toma el valor `desconocido` y el otro toma el valor `verdadero` o `desconocido`. (Si los dos argumentos son `verdadero`, `AND` toma el valor `verdadero`.)

5.6.3 Consecuencias para las estructuras de SQL

Las expresiones booleanas aparecen en muchos contextos de SQL, y el efecto de los valores *null* se debe reconocer. Por ejemplo, la condición de la cláusula `WHERE` elimina filas (del producto cartesiano de tablas llamadas en la cláusula `FROM`) para las cuales la condición no toma el valor `verdadero`. Por tanto, en presencia de valores *null*, cualquier fila que tome el valor `falso` o `desconocido` se elimina. La eliminación de filas que toman el valor `desconocido` tiene un

efecto sutil pero significativo en las consultas, especialmente en las consultas anidadas que implican a `EXISTS` o a `UNIQUE`.

Otro problema con la presencia de valores *null* es la definición de la consideración como *duplicadas* de dos filas del mismo ejemplar de una relación dada. La definición de SQL es que dos filas están duplicadas si las columnas correspondientes son iguales o contienen valores *null*. Compárese esta definición con el hecho de que, si se comparan dos valores *null* con =, el resultado es `desconocido`. En el contexto de los duplicados, esta comparación se trata de manera implícita como `verdadera`, lo que constituye una anomalía.

Como cabía esperar, todas las operaciones aritméticas $+, -, *$ y / devuelven *null* si uno de sus argumentos es *null*. Sin embargo, los valores *null* pueden provocar comportamientos inesperados de las operaciones de agregación. `COUNT`(*) trata los valores *null* igual que a los demás valores; es decir, los cuenta. Todas las demás operaciones de agregación (`COUNT`, `SUM`, `AVG`, `MIN`, `MAX` y las variaciones con `DISTINCT`) se limitan a descartar los valores *null* —así, `SUM` no puede comprenderse sólo como la suma de todos los valores del (multi)conjunto de valores al que se aplica; también se debe tener en cuenta el paso previo de descarte de los valores *null*—. Como caso especial, si uno de estos operadores —que no sea `COUNT`— se aplica *sólo* a valores null, el resultado es también *null*.

5.6.4 Reuniones externas

SQL soporta algunas variedades interesantes de la operación reunión que aprovechan los valores *null*, las denominadas **reuniones externas**. Considérese la reunión de dos tablas, por ejemplo, Marineros $\bowtie_c$ Reservas. Las tuplas de Marineros que no coinciden con ninguna fila de Reservas según la condición de reunión c no aparecen en el resultado. En las reuniones externas, por otro lado, las filas de Marineros sin filas correspondientes en Reservas aparecen en el resultado exactamente una vez y con las columnas heredadas de Reservas asignadas a valores *null*.

De hecho, hay diversas variedades de la idea de reunión externa. En las **reuniones externas por la izquierda** las filas de Marineros sin fila correspondiente de Reservas aparecen en el resultado, pero no al contrario. En las **reuniones externas por la derecha** las filas de Reservas sin fila correspondiente de Marineros aparecen en el resultado, pero no al contrario. En las **reuniones externas completas** tanto las filas de Marineros como las de Reservas sin fila correspondiente aparecen en el resultado. (Por supuesto, las filas con correspondencias aparecen en el resultado en todas las variedades, igual que en las reuniones habituales, a veces denominadas reuniones *internas*, presentadas en el Capítulo 4.)

SQL permite especificar el tipo de reunión deseado en la cláusula `FROM`. Por ejemplo, la siguiente consulta muestra pares $\langle idm,\ idb \rangle$ correspondientes a marineros y a los barcos que han reservado:

```
SELECT  M.idm, R.idb
FROM    Marineros M NATURAL LEFT OUTER JOIN Reservas R
```

La palabra clave `NATURAL` especifica que la condición de reunión es la igualdad para todos los atributos comunes (en este ejemplo, *idm*) y no se exige la cláusula `WHERE` (a menos que se desee especificar condiciones adicionales que no afecten a la reunión). Para los ejemplares de

Marineros y de Reservas mostrados en la Figura 5.6, esta consulta calcula el resultado que puede verse en las Figuras 5.6 y 5.7.

idm	*idb*
22	101
31	*null*
58	103

Figura 5.19 Reunión externa por la izquierda de *R3* y *M4*

5.6.5 Desactivación de los valores nulos

Se pueden impedir los valores *null* especificando `NOT NULL` como parte de la definición del campo; por ejemplo, *nombrem* `CHAR(20) NOT NULL`. Además, no se permite que los campos de la clave primaria adopten valores *null*. Por tanto, hay una restricción `NOT NULL` implícita para todos los campos que aparecen en la restricción `PRIMARY KEY`.

Esta cobertura de los valores *null* dista mucho de estar completa. El lector interesado deberá consultar alguno de los muchos libros dedicados a SQL para obtener un tratamiento más detallado de este tema.

5.7 RESTRICCIONES DE INTEGRIDAD COMPLEJAS EN SQL

En este apartado se estudia la especificación de restricciones de integridad complejas que aprovechan toda la potencia de las consultas SQL. Las características estudiadas en este apartado complementan las características de las restricciones de integridad de SQL presentadas en el Capítulo 3.

5.7.1 Restricciones sobre una sola tabla

Se pueden especificar restricciones complejas para una sola tabla mediante **restricciones de tabla**, que tienen la forma `CHECK` *expresión-condicional*. Por ejemplo, para garantizar que la *categoría* sea un entero en el rango de 1 a 10, se puede emplear:

```
CREATE TABLE Marineros ( idm       INTEGER,
                         nombrem   CHAR(10),
                         categoría INTEGER,
                         edad      REAL,
                         PRIMARY KEY (idm),
                         CHECK ( categoría >= 1 AND categoría <= 10 ))
```

Para hacer que se cumpla la restricción de que no se puede reservar el barco Intrépido se puede utilizar:

```
CREATE TABLE Reservas ( idm       INTEGER,
                        idb       INTEGER,
                        fecha     DATE,
                        FOREIGN KEY (idm) REFERENCES Marineros
                        FOREIGN KEY (idb) REFERENCES Barcos
                        CONSTRAINT noResIntrépido
                        CHECK ( 'Intrépido' <>
                                ( SELECT B.nombreb
                                  FROM   Barcos B
                                  WHERE  B.idb = Reservas.idb )))
```

Cuando se inserta alguna fila en Reservas o se modifica alguna fila ya existente, se evalúa la *expresión condicional* de la restricción `CHECK`. Si adopta el valor `falso`, se rechaza la orden.

5.7.2 Restricciones de dominio y tipos distintos

Los usuarios pueden definir nuevos dominios mediante la instrucción `CREATE DOMAIN`, que utiliza restricciones `CHECK`.

```
CREATE DOMAIN valcategoría INTEGER DEFAULT 1
                           CHECK ( VALUE >= 1 AND VALUE <= 10 )
```

`INTEGER` es el tipo subyacente, o *fuente*, del dominio `valcategoría`, y todos los valores de `valcategoría` deben ser de este tipo. Los valores de `valcategoría` quedan aún más restringidos por el empleo de la restricción `CHECK`; para definir esa restricción se utiliza la palabra clave `VALUE` para hacer referencia a los valores del dominio. Mediante esta característica se pueden restringir los valores que pertenecen a un dominio aprovechando toda la potencia de las consultas SQL. Una vez definido el dominio se puede utilizar su nombre para restringir los valores de las columnas de la tabla; por ejemplo, se puede emplear la línea siguiente para la declaración del esquema:

```
categoría     valcategoría
```

La palabra clave opcional `DEFAULT` se utiliza para asociar un valor predeterminado con el dominio. Si se utiliza el dominio `valcategoría` para una columna de alguna relación y no se introduce ningún valor para esa columna en la tupla insertada, se utiliza el valor predeterminado 1 asociado con `valcategoría`.

El apoyo de SQL al concepto de dominio queda limitado en un aspecto importante. Por ejemplo, se pueden definir dos dominios denominados `IdMarinero` e `IdBarco`, cada uno de los cuales emplea `INTEGER` como tipo subyacente. La intención es obligar a que falle siempre la comparación entre los valores de `IdMarinero` y los de `IdBarco` (ya que se extraen de dominios diferentes); sin embargo, dado que los dos tienen el mismo tipo base, `INTEGER`, la comparación tendrá éxito en SQL. Este problema se aborda en SQL:1999 mediante la introducción de **tipos distintos**:

```
CREATE TYPE tipocategoría AS INTEGER
```

Esta instrucción define un nuevo tipo *distinto* denominado `tipocategoría`, con `INTEGER` como tipo fuente. Los valores del tipo `tipocategoría` se pueden comparar entre sí, pero no

Los tipos distintos en SQL:1999. Muchos sistemas, como UDS de Informix y DB2 de IBM, ya soportan esta característica. Con su introducción se espera que se *abandone* el soporte de los dominios y, finalmente, se elimine en versiones futuras de la norma SQL. Realmente no es más que una parte de un amplio conjunto de características orientadas a objetos de SQL:1999, que se tratan en el Capítulo 15.

se pueden comparar con valores de otros tipos. En concreto, los valores de `tipocategoría` se tratan como diferentes de los valores del tipo fuente, `INTEGER` —no se pueden comparar ni combinar con enteros (es decir, añadir enteros a valores `tipocategoría`)—. Si se desea definir operaciones para el tipo nuevo, por ejemplo, una función *promedio*, hay que hacerlo de manera explícita; no se puede aprovechar ninguna de las operaciones existentes del tipo origen. En el Apartado 15.4.1 se estudiará la manera en que se pueden definir estas funciones.

5.7.3 Asertos: RI para varias tablas

Las restricciones de tabla se asocian con una sola tabla, aunque la expresión condicional de la cláusula `CHECK` pueda hacer referencia a otras tablas. Las restricciones de tabla *sólo* son necesarias si la tabla no está vacía. Por tanto, cuando una restricción implica a dos o más tablas, a veces el mecanismo de las restricciones de tabla resulta engorroso y no responde exactamente a lo deseado. Para tratar estas situaciones SQL soporta la creación de **asertos**, que son restricciones que no están asociadas con ninguna tabla en concreto.

Como ejemplo, supóngase que se desea hacer que se cumpla la restricción de que el número de barcos más el número de marineros debe ser menor de 100. (Esta condición puede exigirse, por ejemplo, para entrar en la categoría de club náutico "pequeño".) Se puede probar con la siguiente restricción de tabla:

```
CREATE TABLE Marineros ( idm       INTEGER,
                         nombrem   CHAR(10),
                         categoría INTEGER,
                         edad      REAL,
                         PRIMARY KEY (idm),
                         CHECK ( categoría >= 1 AND categoría <= 10)
                         CHECK ( ( SELECT COUNT (M.idm) FROM Marineros M )
                                 + ( SELECT COUNT (B.idb) FROM Barcos B )
                                 < 100 ))
```

Esta solución tiene dos inconvenientes. Está asociada con Marineros, aunque implica a Barcos de manera completamente simétrica. Lo que es más importante, si la tabla Marineros está vacía, esta restricción (debido a la semántica de las restricciones de tabla) siempre se cumple por definición, aunque haya más de 100 filas en Barcos. La especificación de esta restricción se puede ampliar para que compruebe que Marineros no esté vacía, pero ese enfoque se vuelve engorroso. La mejor solución es crear un aserto de la manera siguiente:

```
CREATE ASSERTION Clubpequeño
CHECK ( ( SELECT COUNT (M.idm) FROM Marineros M )
```

```
+ ( SELECT COUNT (B.idb) FROM Barcos B)
< 100 )
```

5.8 DISPARADORES Y BASES DE DATOS ACTIVAS

Los **disparadores** son procedimientos que el SGBD invoca automáticamente en respuesta a cambios concretos de la base de datos, que suele especificar el DBA. Las bases de datos con un conjunto de disparadores asociados se denominan **bases de datos activas**. La descripción de cada disparador contiene tres partes:

- **Evento**: una modificación de la base de datos que **activa** el disparador.
- **Condición**: una consulta o prueba que se ejecuta cuando se activa el disparador.
- **Acción**: un procedimiento que se ejecuta cuando se activa el disparador y su condición es verdadera.

Se puede considerar a los disparadores como "demonios" que controlan la base de datos y se ejecutan cuando se modifica la base de datos de modo que coincida con la especificación del *evento*. Los disparadores pueden ser activados por instrucciones de inserción, eliminación o actualización independientemente del usuario o de la aplicación que invocara la instrucción que lo activó; puede que los usuarios ni siquiera sean conscientes de que se ha ejecutado un disparador como efecto secundario de su programa.

La *condición* del disparador puede ser una instrucción verdadero/falso (por ejemplo, todos los sueldos de los empleados son menores de 100.000 €) o una consulta. La consulta se interpreta como *verdadera* si el conjunto de respuestas no está vacío y como *falsa* si no tiene ninguna respuesta. Si la condición adopta el valor verdadero, se ejecuta la acción asociada con el disparador.

La *acción* del disparador puede examinar las respuestas a la consulta de la condición del disparador, hacer referencia a los valores antiguos y nuevos de tuplas modificadas por la instrucción que activa el dispardor, ejecutar consultas nuevas o realizar modificaciones de la base de datos. De hecho, la acción puede incluso ejecutar una serie de órdenes para la definición de datos (por ejemplo, crear tablas nuevas o modificar autorizaciones) u orientadas a las transacciones (por ejemplo, comprometer) o llamar a procedimientos del lenguaje anfitrión.

Un aspecto importante de los disparadores es el momento en que se ejecuta la acción en relación con la instrucción que ha activado el disparador. Por ejemplo, una instrucción que inserte registros en la tabla Alumnos puede activar un disparador que se emplee para conservar estadísticas sobre el número de alumnos menores de 18 años insertados de una vez por una instrucción de inserción típica. En función de lo que haga exactamente el disparador, puede que se desee que la acción se ejecute *antes* de que se apliquen las modificaciones a la tabla Alumnos o *después*. Los disparadores que inicialicen una variable empleada para contar el número de inserciones que cumplen la condición establecida deben ejecutarse antes, mientras que los disparadores que se ejecuten una vez por cada registro insertado que cumpla la condición establecida e incrementen la variable deben ejecutarse después de la inserción de cada registro (ya que puede que deseemos examinar los valores del nuevo registro para determinar la acción correspondiente).

5.8.1 Ejemplos de disparadores en SQL

Los ejemplos de la Figura 5.20, escritos empleando la sintaxis de Oracle Server para definir disparadores, ilustran los conceptos básicos subyacentes a los disparadores. (La sintaxis de SQL:1999 para estos disparadores es parecida; en breve se verá un ejemplo que emplea la sintaxis de SQL:1999.) El disparador denominado *inic_cuenta* inicializa una variable contadora antes de cada ejecución de la instrucción `INSERT` que añade tuplas a la relación Alumnos. El disparador denominado *aum_cuenta* incrementa el contador por cada tupla insertada que satisface la condición *edad* < 18.

```
CREATE TRIGGER inic_cuenta BEFORE INSERT ON Alumnos                    /* Evento */
    DECLARE
        cuenta INTEGER;
    BEGIN                                                              /* Acción */
        cuenta := 0;
    END

CREATE TRIGGER aum_cuenta AFTER INSERT ON Alumnos                      /* Evento */
    WHEN (new.edad < 18)          /* Condición; 'new' es la tupla recién insertada */
    FOR EACH ROW
    BEGIN          /* Acción; procedimiento con la sintaxis de PL/SQL de Oracle */
        cuenta := cuenta + 1;
    END
```

Figura 5.20 Ejemplos que ilustran los disparadores

Uno de los disparadores de ejemplo de la Figura 5.20 se ejecuta antes que la instrucción de activación y el otro, después. También se pueden programar los disparadores para que se ejecuten *en lugar de* la instrucción de activación; o de manera *diferida*, al final de la transacción que contiene la instrucción de activación; o de manera *asíncrona*, como parte de una transacción diferente.

El ejemplo de la Figura 5.20 ilustra otro aspecto importante de la ejecución de los disparadores: los usuarios deben poder especificar si cada disparador se debe ejecutar una vez por registro modificado o una vez por instrucción de activación. Si la acción depende de cada registro modificado, por ejemplo, hay que examinar el campo *edad* del registro insertado de Alumnos para decidir si hay que incrementar la cuenta, se debe definir que el evento de disparo tenga lugar para cada registro modificado; para hacer esto se emplea la cláusula `FOR EACH ROW`. Estos disparadores se denominan **disparadores por filas**. Por otro lado, el disparador *inic_cuenta* sólo se ejecuta una vez por instrucción `INSERT`, independientemente del número de registros insertados, ya que se ha omitido la expresión `FOR EACH ROW`. Estos disparadores se denominan **disparadores por instrucciones**.

En la Figura 5.20 la palabra clave `new` hace referencia a la tupla recién insertada. Si se modificara alguna tupla ya existente se podrían utilizar las palabras clave `old` y `new` para hacer referencia a los valores anteriores y posteriores a la modificación. SQL:1999 también

permite que la acción del disparador haga referencia al *conjunto* de registros modificados, en vez de limitarse a un registro modificado cada vez. Por ejemplo, sería útil poder hacer referencia al conjunto de registros insertados de Alumnos en un disparador que se ejecute una vez tras la instrucción `INSERT`; se podría contar el número de registros insertados con *edad* < 18 mediante una consulta SQL para ese conjunto. En la Figura 5.21 se muestra un disparador de ese tipo, que es una alternativa a los disparadores de la Figura 5.20.

La definición de la Figura 5.21 utiliza la sintaxis de SQL:1999 con objeto de ilustrar los parecidos y diferencias con respecto a la sintaxis empleada en un SGBD típico actual. La cláusula de la palabra clave `NEW TABLE` permite dar nombre de tabla (TuplasInsertadas) al conjunto de tuplas recién insertadas. La cláusula `FOR EACH STATEMENT` especifica un disparador por instrucciones y se puede omitir, ya que es el valor predeterminado. Esta definición no tiene cláusula `WHEN`; si se incluye una cláusula de este tipo, va detrás de la cláusula `FOR EACH STATEMENT`, justo antes de la especificación de la acción correspondiente.

El disparador se evalúa una vez por cada instrucción de SQL que inserte tuplas en Alumnos, e inserta una sola tupla en la tabla que contiene estadísticas de las modificaciones de las tablas de la base de datos. Los dos primeros campos de la tupla contienen constantes (que identifican a la tabla modificada, Alumnos, y el tipo de instrucción que la ha modificado, una instrucción `INSERT`), y el tercer campo es el número de tuplas de Alumnos insertadas con *edad* < 18. (El disparador de la Figura 5.20 sólo calcula el recuento; hace falta un disparador adicional para insertar la tupla correspondiente en la tabla de estadísticas.)

```
CREATE TRIGGER definir_cuenta AFTER INSERT ON Alumnos                /* Evento */
REFERENCING NEW TABLE AS TuplasInsertadas
FOR EACH STATEMENT
    INSERT                                                           /* Acción */
        INTO TablaEstadísticas(TablaModificada, TipoModificación, Cuenta)
        SELECT 'Alumnos', 'Insert', COUNT *
        FROM TuplasInsertadas T
        WHERE T.edad < 18
```

Figura 5.21 Disparador orientado a conjuntos

5.9 DISEÑO DE BASES DE DATOS ACTIVAS

Los disparadores ofrecen un potente mecanismo para tratar las modificaciones de la base de datos, pero se deben emplear con precaución. El efecto de un conjunto de disparadores puede ser muy complejo, y el mantenimiento de una base de datos activa puede volverse muy difícil. A menudo, el empleo juicioso de las restricciones de integridad puede sustituir el de los disparadores.

5.9.1 ¿Por qué los disparadores pueden resultar difíciles de comprender?

En los sistemas de bases de datos activas, cuando el SGBD va a ejecutar alguna instrucción que modifique la base de datos, comprueba si esa instrucción activa algún disparador. En caso positivo, el SGBD procesa el disparador mediante la evaluación de su condición y, luego, (si la condición toma el valor verdadero) la ejecución de su acción.

Si alguna instrucción activa más de un disparador, el SGBD suele procesar todos, en un orden arbitrario. Es importante recordar que la ejecución de la acción de un disparador puede, a su vez, activar otro disparador. En especial, la ejecución de la acción de un disparador puede activar nuevamente ese mismo disparador; estos disparadores se denominan **disparadores recursivos**. La posibilidad de esas activaciones *en cadena* y el orden impredecible en que el SGBD procesa los disparadores activados pueden hacer difícil de comprender el efecto de los conjuntos de disparadores.

5.9.2 Restricciones y disparadores

Un empleo habitual de los disparadores es el mantenimiento de la consistencia de la base de datos y, en esos casos, siempre se debe tomar en consideración si el empleo de restricciones de integridad (por ejemplo, una restricción de clave externa) consigue los mismos objetivos. El significado de las restricciones no se define operativamente, a diferencia del efecto de los disparadores. Esta propiedad facilita la comprensión de las restricciones, y también da más opertunidades al SGBD para optimizar la ejecución. Las restricciones también evitan que los datos se vuelvan inconsistentes por *alguna* clase de instrucción, mientras que los disparadores los activa una clase concreta de instrucción (`INSERT, DELETE` o `UPDATE`). Una vez más, esta restricción hace que las restricciones sean más sencillas de comprender.

Por otro lado, los disparadores permiten conservar la integridad de la base de datos de maneras más flexibles, como ilustran los ejemplos siguientes.

- Supóngase que se tiene una tabla llamada Pedidos con los campos *idartículo*, *cantidad*, *idcliente* y *preciounitario*. Cuando un cliente formula un pedido, el valor de los tres primeros campos lo rellena el usuario (en este ejemplo, un comercial). El valor del cuarto campo se puede obtener de una tabla denominada Artículos, pero es importante incluirlo en la tabla Pedidos para tener un registro completo del pedido, por si el precio del artículo cambia posteriormente. Se puede definir un disparador que examine ese valor y lo incluya en el cuarto campo del registro recién insertado. Además de reducir el número de campos que el comecial tiene que rellenar, el disparador elimina la posibilidad de que se introduzca un error que provoque un precio inconsistente en la tabla Pedidos.

- Siguiendo con este ejemplo, puede que se desee llevar a cabo alguna acción adicional cuando se reciba un pedido. Por ejemplo, si la compra se carga a una línea de crédito concedida por la empresa, puede que se desee comprobar si el coste total de la adquisición se halla dentro del límite de crédito. Se puede emplear un disparador para que haga esa comprobación; en realidad, incluso se puede utilizar una restricción `CHECK`. Sin embargo, el empleo de un disparador permite implementar políticas más sofisticadas pa-

ra el tratamiento de adquisiciones que superen el límite de crédito. Por ejemplo, puede que se permitan adquisiciones que superen el límite en un máximo del 10% si el cliente ha trabajado con la empresa un mínimo de un año, y añadir el cliente a una tabla de candidatos a aumentos del límite de crédito.

5.9.3 Otros usos de los disparadores

Muchos posibles usos de los disparadores van más allá del mantenimiento de la integridad. Los disparadores pueden alertar a los usuarios de eventos infrecuentes (como se refleja en las actualizaciones de la base de datos). Por ejemplo, puede que se desee comprobar si el cliente que formula un pedido ha realizado suficientes adquisiciones el mes anterior como para tener derecho a un descuento adicional; en caso positivo, se debe informar al comercial de que puede indicárselo al cliente y, posiblemente, generar ventas adicionales. Esta información se puede transmitir mediante un disparador que compruebe las adquisiciones recientes e imprima un mensaje si el cliente tiene derecho a ese descuento.

Los disparadores pueden generar un registro de los eventos para apoyar las auditorías y los controles de seguridad. Por ejemplo, cada vez que un cliente formula un pedido, se puede crear un registro con el identificador del cliente y su límite de crédito actual e insertarlo en una tabla con el historial de los clientes. El análisis posterior de esta tabla pudiera sugerir candidatos para un límite de crédito ampliado (por ejemplo, clientes que nunca han dejado de pagar a tiempo una factura y que se han quedado con menos del 10% de su crédito al menos tres veces durante el mes anterior).

Como ilustran los ejemplos del Apartado 5.8, se pueden emplear disparadores para reunir datos estadísticos de acceso a las tablas y de sus modificaciones. Algunos sistemas de bases de datos incluso emplean internamente los disparadores como base de la gestión de réplicas de las relaciones. Esta lista de usos posibles de los disparadores no es exhaustiva; por ejemplo, también se ha considerado el empleo de disparadores para la gestión de flujos de trabajo y para hacer que se cumplan las reglas de negocio.

5.10 PREGUNTAS DE REPASO

Las respuestas a las preguntas de repaso se pueden hallar en los apartados mencionados.

- ¿Cuáles son las partes de una consulta básica de SQL? ¿Son conjuntos o multiconjuntos las tablas de datos y de resultados de las consultas SQL? ¿Cómo se puede obtener un conjunto de tuplas como resultado de una consulta? **(Apartado 5.2)**
- ¿Qué son las variables de rango de SQL? ¿Cómo se puede poner nombre a las columnas de resultados de las consultas que se definen mediante expresiones aritméticas o de cadenas de caracteres? ¿Qué soporte ofrece SQL a la coincidencia estricta de patrones? **(Apartado 5.2)**
- ¿Qué operaciones ofrece SQL para los (multi)conjuntos de tuplas y cómo se pueden aprovechar para escribir consultas? **(Apartado 5.3)**
- ¿Qué son las consultas anidadas? ¿Qué es la *correlación* de las consultas anidadas? ¿Cómo se pueden utilizar los operadores `IN`, `EXISTS`, `UNIQUE`, `ANY` y `ALL` para escribir consultas

anidadas? ¿Por qué resultan útiles? Ilústrese la respuesta mostrando la manera de escribir el operador *division* en SQL. **(Apartado 5.4)**

- ¿Qué operadores de agregación soporta SQL? **(Apartado 5.5)**
- ¿Qué es la *agrupación*? ¿Existe un equivalente en el álgebra relacional? Explíquese esta característica y discútase la interacción de las cláusulas `HAVING` y `WHERE`. Menciónese cualquier restricción que deban cumplir los campos que aparecen en la cláusula `GROUP BY`. **(Apartado 5.5.1)**
- ¿Qué son los valores *null*? ¿Están soportados en el modelo relacional, tal y como se describe en el Capítulo 3? ¿Cómo afectan al significado de las consultas? ¿Pueden contener valores *null* los campos de la clave principal de las tablas? **(Apartado 5.6)**
- ¿Qué tipos de restricciones de SQL se pueden especificar mediante el lenguaje de consulta? ¿Se pueden expresar las restricciones de clave principal mediante alguna de estos nuevos tipos de restricciones? En caso positivo, ¿por qué ofrece SQL una sintaxis diferente para la restricción de la clave primaria? **(Apartado 5.7)**
- ¿Qué son los *disparadores* y cuáles son las tres partes en que se dividen? ¿Cuáles son las diferencias entre los disparadores por filas y los disparadores por instrucciones? **(Apartado 5.8)**
- ¿Por qué los disparadores pueden resultar difíciles de comprender? Explíquense las diferencias entre los disparadores y las restricciones de integridad y descríbase cuándo se preferirán unos u otras. ¿Para qué se emplean los disparadores? **(Apartado 5.9)**

EJERCICIOS

Se dispone de material en línea para todos los ejercicios de este capítulo en la página Web del libro en

`http://www.cs.wisc.edu/~dbbook`

En esa página se incluyen secuencias de comandos para crear tablas para cada ejercicio para usarlas con Oracle, DB2 de IBM, SQL Server de Microsoft, Microsoft Access y MySQL.

Ejercicio 5.1 Considérense las relaciones siguientes:

Alumnos(*numa:* `integer`, *nombrea:* `string`, *carrera:* `string`, *nivel:* `string`, *edad:* `integer`)
Cursos(*nombre:* `string`, *impartido_en:* `string`, *aula:* `string`, *idp:* `integer`)
Matriculado(*numa:* `integer`, *nombrec:* `string`)
Profesores(*idp*: `integer`, *nombrep:* `string`, *iddep:* `integer`)

El significado de estas relaciones es evidente; por ejemplo, Matrículado tiene un registro para cada par alumno-curso tal que el alumno está matriculado en ese curso.

Escríbanse las consultas siguientes en SQL. No se permiten duplicados en ninguna de las respuestas.

1. Averiguar el nombre de todos los alumnos de tercer año (nivel = JR) que están matriculados en alguna clase impartida por El Profe.
2. Averiguar la edad del alumno de más edad que tiene un título de Historia o está matriculado en algún curso impartido por El Profe.

3. Averiguar el nombre de todos los cursos que se imparten en el aula A128 o tienen matriculados cinco o más alumnos.
4. Averiguar el nombre de todos los alumnos que están matriculados en dos cursos que se impartan al mismo tiempo.
5. Averiguar el nombre de los profesores que dan clase en todas las aulas en las que se imparte alguna clase.
6. Averiguar el nombre de los profesores para los que la matrícula total de las asignaturas que imparten es menor de cinco.
7. Para cada nivel, determinar el nivel y la edad media de los alumnos de ese nivel.
8. Para todos los niveles excepto el tercer año, determinar el nivel y la edad media de los alumnos de ese nivel.
9. Para cada profesor que sólo haya dado clase en el aula A128, determinar el nombre del profesor y el número total de cursos que ha dado.
10. Averiguar el nombre de los alumnos matriculados en mayor número de cursos.
11. Averiguar el nombre de los alumnos no matriculados en ningún curso.
12. Para cada valor de edad que aparezca en Alumnos, averiguar el valor de nivel que aparece más a menudo. Por ejemplo, si hay más alumnos de primer año con 18 de edad que alumnos de cuarto (SR), tercero (JR) o segundo año (SO) con esa edad, hay que determinar la pareja (18, FR).

Ejercicio 5.2 Considérese el esquema siguiente:

Proveedores(*idp:* `integer`, *nombrep:* `string`, *domicilio:* `string`)
Repuestos(*idr:* `integer`, *nombrer:* `string`, *color:* `string`)
Catálogo(*idp:* `integer`, *idr:* `integer`, *coste:* `real`)

La relación Catálogo muestra el precio que cobran los proveedores por los repuestos. Escríbanse las consultas siguientes en SQL:

1. Averiguar el *nombrer* de los repuestos para los que hay algún proveedor.
2. Averiguar el *nombrep* de los proveedores que suministran todos los repuestos.
3. Averiguar el *nombrep* de los proveedores que suministran todos los repuestos rojos.
4. Averiguar el *nombrer* de los repuestos suministrados por el proveedor Repuestos Acme y por nadie más.
5. Averiguar el *idp* de los proveedores que cobran más por algún repuesto que el coste medio de ese repuesto (promediado sobre todos los proveedores que suministran ese repuesto).
6. Para cada repuesto, averiguar el *nombrep* del proveedor que cobra más por ese repuesto.
7. Averiguar el *idp* de los proveedores que sólo suministran repuestos rojos.
8. Averiguar el *idp* de los proveedores que suministran un repuesto rojo y otro verde.
9. Averiguar el *idp* de los proveedores que suministran un repuesto rojo o uno verde.
10. Para cada proveedor que sólo suministra repuestos verdes, determinar el nombre del proveedor y el número total de repuestos que suministra.
11. Para cada proveedor que suministra un repuesto verde y uno rojo, determinar el nombre y el precio del repuesto más caro que suministre.

Ejercicio 5.3 Las relaciones siguientes realizan el seguimiento de la información de vuelos de una línea aérea:

Vuelos(*nuvu:* `integer`, *de:* `string`, *a:* `string`, *distancia:* `integer`,
salida: `time`, *llegada:* `time`, *precio:* `real`)
Aviones(*ida:* `integer`, *nombrea:* `string`, *autonomía:* `integer`)
Certificado(*ide:* `integer`, *ida:* `integer`)
Empleados(*ide:* `integer`, *nombree:* `string`, *sueldo:* `integer`)

Obsérvese que la relación Empleados describe a los pilotos y también a otros tipos de empleados; todos los pilotos están homologados para determinados tipos de aviones, y sólo los pilotos están homologados para volar. Escríbase cada una de las consultas siguientes en SQL. (*Hay más consultas que utilizan el mismo esquema en los ejercicios del Capítulo 4.*)

1. Averiguar el nombre de los aviones tales que todos los pilotos homologados para operar con ellos tienen sueldos superiores a 80.000 €.
2. Para cada piloto homologado para más de tres aviones, averiguar el *ide* y la *autonomía* máxima de los aviones para los que está homologado.
3. Averiguar el nombre de los pilotos cuyo *sueldo* es menor que el precio de la ruta más barata de Las Palmas a Hamburgo.
4. Para todos los aviones con *autonomía* superior a 2.000 kilómetros, averiguar el nombre del avión y el sueldo medio de todos los pilotos homologados para ese modelo.
5. Averiguar el nombre de los pilotos homologados para algún avión de Airbus.
6. Averiguar el *ida* de todos los aviones que se pueden utilizar en rutas de Las Palmas a Cracovia.
7. Identificar las rutas que pueden pilotar todos los pilotos que ganan más de 100.000 €.
8. Determinar el *nombree* de los pilotos que pueden operar con aviones de *autonomía* superior a 5.000 kilómetros pero que no están homologados para ningún avión de Airbus.
9. Un cliente desea viajar de Málaga a Nancy con no más de dos transbordos. Indíquese la selección de horas de salida desde Málaga si el cliente desea llegar a Nancy hacia las seis de la tarde.
10. Calcúlese la diferencia entre el sueldo medio de los pilotos y el sueldo medio de todos los empleados (incluidos los pilotos).
11. Escríbase el nombre y el sueldo de todos los empleados que no son pilotos cuyo sueldo es superior al sueldo medio de los pilotos.
12. Determinar el nombre de los empleados que sólo están homologados para aviones con autonomía superior a 2.000 kilómetros.
13. Determinar el nombre de los empleados que sólo están homologados para aviones con autonomía superior a 2.000 kilómetros, pero, como mínimo, para dos aviones.
14. Determinar el nombre de los empleados que sólo están homologados para aviones con autonomía superior a 2.000 kilómetros y para algún avión de Airbus.

Ejercicio 5.4 Considérese el siguiente esquema relacional. Cada empleado puede trabajar en más de un departamento; el campo *tiempo_parcial* de la relación Trabaja muestra el porcentaje de tiempo que cada empleado trabaja en un departamento dado.

Emp(*ide:* `integer`, *nombree:* `string`, *edad:* `integer`, *sueldo:* `real`)
Trabaja(*ide:* `integer`, *idd:* `integer`, *tiempo_parcial:* `integer`)
Dep(*idd:* `integer`, *nombred:* `string`, *presupuesto:* `real`, *idencargado:* `integer`)

Escribir las consultas siguientes en SQL:

1. Determinar el nombre y la edad de cada empleado que trabaje tanto en el departamento de Ferretería como en el de Software.
2. Para cada departamento con más de 20 empleados equivalentes a tiempo completo (es decir, en el que los empleados a tiempo parcial y a tiempo completo suman, como mínimo, el equivalente a esos empleados a tiempo completo), determinar el *idd* junto con el número de empleados que trabajan en ese departamento.
3. Determinar el nombre de cada empleado cuyo sueldo supere el presupuesto de todos los departamentos en los que trabaje.
4. Averiguar el *idencargado* de los encargados que sólo dirigen departamentos con presupuestos mayores de 1 millón de euros.

idm	*nombrem*	*categoría*	*edad*
18	jiménez	3	30,0
41	jaraiz	6	56,0
22	ahab	7	44,0
63	moby	*null*	15,0

Figura 5.22 Un ejemplar de Marineros

5. Averiguar el *nombree* de los encargados que dirigen los departamentos de mayor presupuesto.
6. Supóngase que si un encargado dirige más de un departamento entonces *controla* la suma de los presupuestos de esos departamentos. Averiguar el *idencargado* de los encargados que controlan más de 5 millones de euros.
7. Averiguar el *idencargado* de los encargados que controlan los mayores importes.
8. Averiguar el *nombree* de los encargados que sólo dirigen departamentos con presupuestos de más de 1 millón de euros pero, como mínimo, un departamento con presupuesto menor de 5 millones de euros.

Ejercicio 5.5 Considérese el ejemplar de la relación Marineros que puede verse en la Figura 5.22.

1. Escríbanse consultas de SQL para calcular la categoría media, empleando `AVG`; la suma de categorías, utilizando `SUM`; y el número de categorías, usando `COUNT`.
2. Si se divide la suma recién calculada por la cuenta, ¿el resultado será igual que el promedio? ¿Cómo cambiaría la respuesta si este procedimiento se llevara a cabo para el campo *edad* en lugar de para *categoría*?
3. Considérese la consulta siguiente: *Averiguar el nombre de los marineros de categoría superior a todos los marineros de edad* < 21. Las dos consultas de SQL siguientes intentan obtener la respuesta a esta pregunta. ¿Calculan las dos el resultado buscado? En caso de no hacerlo, explíquese el motivo. ¿En qué condiciones calcularían el mismo resultado?

```
SELECT M.nombrem
FROM   Marineros M
WHERE  NOT EXISTS ( SELECT *
                    FROM   Marineros M2
                    WHERE  M2.edad < 21
                           AND M.categoría <= M2.categoría )

SELECT *
FROM   Marineros M
WHERE  M.categoría > ANY ( SELECT M2.categoría
                           FROM   Marineros M2
                           WHERE  M2.edad < 21 )
```

4. Considérese el ejemplar de Marineros de la Figura 5.22. Definamos el ejemplar M1 de Marineros como consistente en las dos primeras tuplas, el ejemplar M2 como consistente en las dos últimas tuplas y M como el ejemplar dada.
   (a) Mostrar la reunión externa por la izquierda de M consigo misma, siendo la condición de reunión *idm=idm*.
   (b) Mostrar la reunión externa por la derecha de M consigo misma, siendo la condición de reunión *idm=idm*.
   (c) Mostrar la reunión externa completa de M consigo misma, siendo la condición de reunión *idm=idm*.

(d) Mostrar la reunión externa por la izquierda de M1 con M2, siendo la condición de reunión *idm=idm*.

(e) Mostrar la reunión externa por la derecha de M1 con M2, siendo la condición de reunión *idm=idm*.

(f) Mostrar la reunión externa completa de M1 con M2, siendo la condición de reunión *idm=idm*.

Ejercicio 5.6 Considérese este esquema relacional y respóndase brevemente a las preguntas siguientes:

Emp(*ide:* `integer`, *nombree:* `string`, *edad:* `integer`, *sueldo:* `real`)
Trabaja(*ide:* `integer`, *idd:* `integer`, *tiempo_parcial:* `integer`)
Dep(*idd:* `integer`, *presupuesto:* `real`, *idencargado:* `integer`)

1. Defínase una restricción de tabla para Emp que garantice que todos los empleados ganan, como mínimo 10.000 €.
2. Defínase una restricción de tabla para Dep que garantice que todos los encargados tienen *edad* > 30.
3. Defínase un aserto para Dep que garantice que todos los empleados tengan *edad* > 30. Compárese este aserto con la restricción de tabla equivalente. Explíquese cuál es mejor.
4. Escríbanse instrucciones de SQL para eliminar toda la información sobre los empleados superen los del encargado de uno o varios de los departamentos en los que trabajan. Hay que asegurarse de que se satisfacen todas las restricciones de integridad relevantes se cumplen tras las actualizaciones.

Ejercicio 5.7 Considérense las relaciones siguientes:

Alumnos(*numa:* `integer`, *nombrea:* `string`, *carrera:* `string`,
nivel: `string`, *edad:* `integer`)
Cursos(*nombre:* `string`, *impartido_en:* `time`, *aula:* `string`, *idp:* `integer`)
Matriculado(*numa:* `integer`, *nombrec:* `string`)
Profesores(*idp*: `integer`, *nombrep:* `string`, *iddep:* `integer`)

El significado de estas relaciones es evidente; por ejemplo, Matriculado tiene un registro por cada par alumno-curso tal que el alumno está matriculado en ese curso.

1. Escríbanse las instrucciones de SQL necesarias para crear estas relaciones, incluidas las versiones adecuadas de todas las restricciones de integridad de clave principal y de clave externa.
2. Exprésense cada una de las restricciones de integridad siguientes en SQL, a menos que alguna esté implicita en la restricción de clave primaria y de clave externa; en ese caso, explíquese la manera en que está implícita. Si la restricción no se puede expresar en SQL, indíquese. Para cada restricción, indíquese qué operaciones (inserciones, eliminaciones y actualizaciones de relaciones concretas) se deben controlar para hacer que se cumpla esa restricción.

   (a) Todos los cursos tienen una matrícula mínima de 5 alumnos y máxima de 30.

   (b) Como mínimo se imparte un curso en cada aula.

   (c) Todos los profesores deben impartir, como mínimo, dos asignaturas.

   (d) Sólo los profesores del departamento con *iddep=33* imparten más de tres asignaturas.

   (e) Todos los alumnos deben estar matriculados en la asignatura denominada Mat101.

   (f) El aula en la que se imparte el curso programado en primer lugar (es decir, el curso con el valor mínimo de *impartido_en*) no debe ser la misma en que se imparte el curso programado en último lugar.

   (g) No se pueden impartir dos cursos en la misma aula al mismo tiempo.

   (h) El departamento con más profesores debe tener menos del doble que el departamento con menos profesores.

   (i) Ningún departamento puede tener más de 10 profesores.

(j) Cada alumno no puede añadir más de dos asignaturas a la vez (es decir, en una sola actualización).

(k) El número de alumnos de cuarto curso de Informática debe ser mayor que el de alumnos de cuarto curso de Matemáticas.

(l) El número de asignaturas diferentes en las que hay matriculados alumnos de cuarto curso de Informática es mayor que el de asignaturas diferentes en que hay matriculados alumnos de cuarto curso de Matemáticas.

(m) La matrícula total de las asignaturas impartidas por los profesores del departamento con *iddep=33* es mayor que el número de alumnos de cuarto curso de Matemáticas.

(n) Debe haber, como mínimo, un alumno de cuarto curso de Informática si es que hay algún alumno.

(ñ) Los profesores de departamentos diferentes no pueden dar clase en la misma aula.

Ejercicio 5.8 Discútanse los puntos fuertes y débiles del mecanismo de los disparadores. Compárense los disparadores con otras restricciones de integridad soportadas por SQL.

Ejercicio 5.9 Considérese el siguiente esquema relacional. Cada empleado puede trabajar en más de un departamento; el campo *tiempo_parcial* de la relación Trabaja muestra el porcentaje de tiempo que cada empleado trabaja en los diferentes departamentos.

Emp(*ide:* `integer`, *nombree:* `string`, *edad:* `integer`, *sueldo:* `real`)
Trabaja(*ide:* `integer`, *idd:* `integer`, *tiempo_parcial:* `integer`)
Dep(*idd:* `integer`, *presupuesto:* `real`, *idencargado:* `integer`)

Escríbanse restricciones de integridad en SQL-92 (de dominio, de clave, de clave externa o `CHECK`; o asertos) o disparadores en SQL:1999 para garantizar cada uno de los requisitos siguientes, considerados de manera independiente.

1. Los empleados deben ganar un sueldo mínimo de 1.000 €.
2. Todos los encargados deben ser también empleados.
3. El porcentaje total de nombramientos para cada empleado debe ser inferior al 100%.
4. Los encargados siempre deben tener un sueldo más elevado que cualquier empleado que dirijan.
5. Siempre que se conceda un aumento a un empleado se debe incrementar el sueldo del encargado en el mismo importe, como mínimo.
6. Siempre que se conceda un aumento a un empleado se debe incrementar el sueldo del encargado para que sea, como mínimo, igual. Además, siempre que se conceda un aumento a un empleado se debe incrementar el presupuesto del departamento para que sea mayor que la suma de los sueldos de todos los empleados del departamento.

EJERCICIOS BASADOS EN PROYECTOS

Ejercicio 5.10 Identifíquese el subconjunto de consultas de SQL soportadas en Minibase.

NOTAS BIBLIOGRÁFICAS

La versión original de SQL se desarrolló como lenguaje de consulta para el proyecto System R de IBM, y se puede seguir su primer desarrollo en [63, 94]. Desde entonces, SQL se ha transformado en el lenguaje relacional de consultas más utilizado y su desarrollo se halla sometido actualmente a un proceso internacional de normalización.

Melton y Simon presentan un tratamiento muy legible y completo de SQL-92 en Melton [326], y las características principales de SQL:1999 se tratan en [327]. Se remite a los lectores a estos libros si desean

un tratamiento autorizado de SQL. [155] presenta un breve resumen de la norma SQL:1999. Date ofrece una penetrante crítica de SQL en [134]. Aunque parte de los problemas se han abordado en SQL-92 y en revisiones posteriores, otros continúan. En [340] se presenta una semántica formal para un gran subconjunto de consultas de SQL. SQL:1999 es la norma actual de la Organización Internacional para la Normalización (International Organization for Standardization, ISO) y del Instituto Nacional Americano de Normalización (American National Standards Institute, ANSI). Melton es el editor de la norma ANSI e ISO SQL:1999, documento ANSI/ISO/IEC 9075-:1999. El documento ISO correspondiente es ISO/IEC 9075-:1999. Un sucesor, planeado para 2003, evolucionará a partir de SQL:1999. SQL:2003 se halla cercano a la ratificación (a junio de 2002). Los borradores de las deliberaciones sobre SQL:2003 están disponibles en el siguiente URL:

`ftp://sqlstandards.org/SC32/`

[471] contiene un conjunto de trabajos que tratan el campo de las bases de datos activas. [485] incluye una buena introducción en profundidad a las reglas activas, que trata los aspectos de la semántica, las aplicaciones y el diseño. [161] analiza las ampliaciones de SQL para la especificación de las comprobaciones de las restricciones de integridad mediante disparadores. [72] también estudia un mecanismo procedimental, denominado *alertador*, para el control de bases de datos. [117] es un trabajo reciente que sugiere la manera en que se pueden incorporar disparadores a las ampliaciones de SQL. Entre los prototipos influyentes de bases de datos están Ariel [240], HiPAC [321], ODE [9], Postgres [439] RDL [415], y Sentinel [22]. [91] compara diversas arquitecturas para los sistemas de bases de datos activas.

[19] considera las condiciones en las que un conjunto de reglas activas tiene el mismo comportamiento, independientemente del orden de evaluación. La semántica de las bases de datos activas se estudia también en [187] y en [483]. El diseño y administración de sistemas complejos de bases de datos se trata en [40, 150]. [88] estudia la gestión de las reglas mediante Chimera, un lenguaje y modelo de datos para los sistemas de bases de datos activas.