

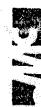
McGraw-Hill

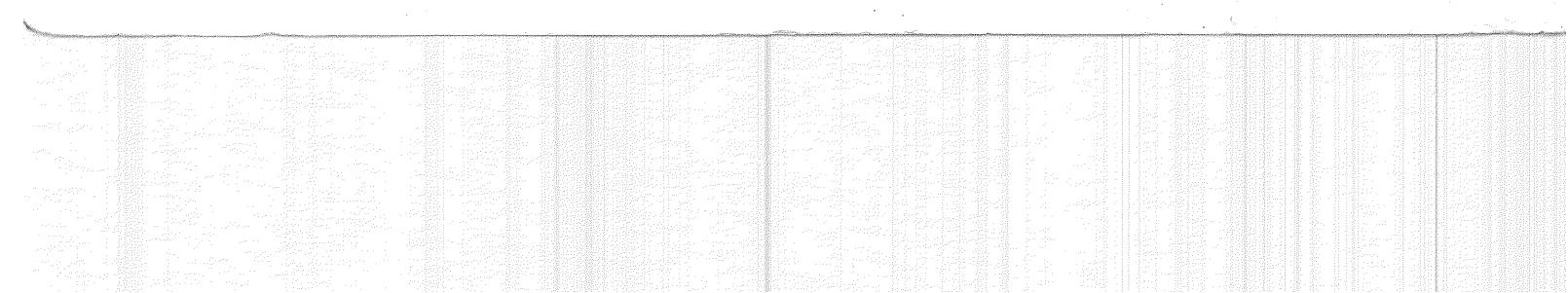
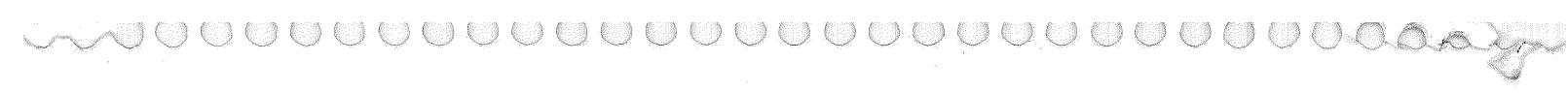
APPLIQUE

SQL

James R. Groff
Paul N. Weinberg

Incluye Oracle, DB2, SQL Server,
dBASE IV, SQL/DS, Ingres, OS/2 Extended Edition,
Informix, Sybase, SQLBase, VAX SQL





APLIQUE SQL

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100
---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	-----

**CONSULTORES EDITORIALES
AREA DE INFORMATICA Y COMPUTACION**

Antonio Vaquero Sánchez
Catedrático de Informática
Facultad de Ciencias Físicas
Universidad Complutense de Madrid
ESPAÑA

Raymundo Hugo Rangel G.
Físico, Facultad de Ciencias, UNL
Profesor, Carrera Ing. en Computación
Facultad de Ingeniería, UNAM

Gerardo Quiroz Veyra
Ingeniero en Comunicaciones y Electrónica
Escuela Superior de Ingeniería Mecánica y Electrónica, IPN
Carter Wallace, S. A.
Universidad Autónoma Metropolitana
Docente DCSA
MEXICO

Willy Vega Gálvez
Universidad Nacional de Ingeniería
PERU

Luis Ernesto Ramírez
Coordinador de Informática
Escuela de Administración y Contaduría
Universidad Católica Andrés Bello, UCAB
VENEZUELA

APLIQUE SQL

James R. Groff
Paul N. Weinberg

Traductor:

ALFREDO BAUTISTA PALOMA
Departamento de Informática y Automática
Facultad de Ciencias Físicas
Universidad Complutense de Madrid

Revisor técnico:

ANTONIO VAQUERO SANCHEZ
Catedrático de Informática
Facultad de Ciencias Físicas
Universidad Complutense de Madrid

Osborne/McGraw-Hill

MADRID • BUENOS AIRES • CARACAS • GUATEMALA • LISBOA • MEXICO
NUEVA YORK • PANAMA • SAN JUAN • SANTA FE DE BOGOTA • SANTAGO • SAO PAULO
AUCKLAND • HAMBURGO • LONDRES • MILAN • MONTREAL • NUEVA DELHI • PARIS
SAN FRANCISCO • SIDNEY • SINGAPUR • ST. LOUIS • TOKIO • TORONTO

Un vistazo al contenido

<i>Primerª parte. Panorámica general de SQL</i>	1
1. <i>Introducción</i>	3
2. <i>Rápido repaso de SQL</i>	13
3. <i>SQL en perspectiva</i>	23
4. <i>Bases de datos relacionales</i>	43
<i>Segunda parte. Recuperación de datos</i>	63
5. <i>Conceptos básicos de SQL</i>	65
6. <i>Consultas simples</i>	85
7. <i>Consultas multitable (composiciones)</i>	127
8. <i>Consultas sumarias</i>	163

APLIQUE SQL

No está permitida la reproducción total o parcial de este libro, ni su tratamiento informático, ni la transmisión de ninguna forma o por cualquier medio, ya sea electrónico, mecánico, por fotocopia, por registro u otros métodos, sin el permiso previo y por escrito de los titulares del Copyright.

DERECHOS RESERVADOS © 1991 respecto a la primera edición en español por

McGRAW-HILL/INTERAMERICANA DE ESPAÑA, S. A.
Edificio Oasis-A
Bárcuri, s/n, 1.º planta
28023 Aravaca (Madrid)

Traducido de la primera edición en inglés de USING SQL
Copyright © MCMXC, por McGRAW-HILL, Inc.
ISBN: 0-07-881524-X

ISBN: 84-7615-571-9
Depósito legal: M. 9.367-1992

Compruebo en: MonoComp, S. A. Conde de Vilches, 31. 28028 Madrid
Impreso en EDIGRAFOS, S. A. c/Edison, B-22. Pol. Ind. San Marcos. 28906 GETAFE (Madrid)

IMPRESO EN ESPAÑA - PRINTED IN SPAIN

9.	Subconsultas	191
<i>Tercera parte. Actualización de datos</i>		
10.	<i>Actualizaciones de bases de datos</i>	219
11.	<i>Integridad de datos</i>	241
12.	<i>Procesamiento de transacciones</i>	265
<i>Cuarta parte. Estructura de una base de datos</i>		
13.	<i>Creación de una base de datos</i>	295
14.	<i>Vistas</i>	331
15.	<i>Seguridad SQL</i>	351
16.	<i>El catálogo de sistema</i>	375
<i>Quinta parte. Programación con SQL</i>		
17.	<i>SQL incorporado</i>	393
18.	<i>SQL dinámico</i>	395
19.	<i>API de SQL</i>	447
<i>Sexta parte. Direcciones futuras</i>		
20.	<i>Gestión de bases de datos distribuidas</i>	529
21.	<i>El futuro de SQL</i>	531
A.	<i>La base de datos ejemplo</i>	555
B.	<i>Perfiles de vendedores DBMS</i>	573
C.	<i>Lista de empresas y productos</i>	591
D.	<i>Sintaxis de SQL ANSI/ISO</i>	597
	<i>Índice</i>	607

Contenido

<i>Prólogo</i>	XXIII
<i>Por qué este libro es para usted</i>	XXVII
<i>Primera parte. Panorámica general de SQL</i>	1
<i>1. Introducción</i>	3
El lenguaje SQL	3
El papel de SQL	6
Características y beneficios de SQL	7
Independencia de los vendedores	8
Portabilidad a través de sistemas informáticos	8
Estándares SQL	9
Apoyo de IBM	9
Fundamento relacional	9
Estructura de alto nivel en inglés	9
Consultas interactivas ad hoc	10
Acceso a la base de datos mediante programas	10
Vistas múltiples de datos	10
Lenguaje completo de base de datos	10
Definición dinámica de datos	11
Arquitectura cliente/servidor	11

2.	Rápido repaso de SQL	13
	Una base de datos sencilla	13
	Recuperación de datos	15
	Sumarización de datos	15
	Adición de datos a la base de datos	17
	Supresión de datos	18
	Actualización de la base de datos	19
	Protección de datos	19
	Creación de una base de datos	19
	Resumen	20
		21
3.	SQL en perspectiva	23
	SQL y la gestión de base de datos	23
	Breve historia de SQL	24
	Los primeros años	25
	Primeros productos relacionales	25
	Productos IBM	26
	Aceptación comercial	27
	Estandáres SQL	28
	El estándar ANSI/ISO	28
	Otros estándares SQL	30
	El mito de la portabilidad	30
	SQL y la conexión por red	32
	Arquitectura centralizada	32
	Arquitectura de servidor de archivos	33
	Arquitectura cliente/servidor	34
	El impacto de SQL	35
	SQL y la SAA de IBM	36
	SQL en minicomputadores	37
	SQL en sistemas UNIX	37
	SQL y el procesamiento de transacciones	38
	SQL en computadores personales	38
	SQL y OS/2	40
	Resumen	41
4.	Bases de datos relacionales	43
	Modelos de datos primitivos	43
	Sistemas de gestión de archivos	44
	Bases de datos jerárquicas	44
	Bases de datos en red	47
	El modelo de datos relacional	49

5.	Conceptos básicos de SQL	65
	Sentencias	65
	Nombres	69
	Nombres de tabla	69
	Nombres de columna	70
	Tipos de datos	71
	Tipos de datos extendidos	71
	Diferencias de tipos de datos	74
	Constantes	76
	Constantes numéricas	77
	Constantes de cadena	77
	Constantes de fecha y hora	78
	Constantes simbólicas	80
	Expresiones	80
	Funciones internas	82
	Falta de datos (valores NULL)	82
	Resumen	84
6.	Consultas simples	85
	La sentencia SELECT	85
	La cláusula SELECT	88
	La cláusula FROM	88
	Resultados de consultas	89
	Consultas sencillas	91
	Columnas calculadas	92
	Selección de todas las columnas (SELECT *)	95
	Filas duplicadas (DISTINCT)	96
	Selección de fila (cláusula WHERE)	98
	Condiciones de búsqueda	100
	Test de comparación (=, < >, < =, > , > =)	100
		101
		102
		103
		104
		105
		106
		107
		108
		109
		110
		111
		112
		113
		114
		115
		116
		117
		118
		119
		120
		121
		122
		123
		124
		125
		126
		127
		128
		129
		130
		131
		132
		133
		134
		135
		136
		137
		138
		139
		140
		141
		142
		143
		144
		145
		146
		147
		148
		149
		150
		151
		152
		153
		154
		155
		156
		157
		158
		159
		160
		161
		162
		163
		164
		165
		166
		167
		168
		169
		170
		171
		172
		173
		174
		175
		176
		177
		178
		179
		180
		181
		182
		183
		184
		185
		186
		187
		188
		189
		190
		191
		192
		193
		194
		195
		196
		197
		198
		199
		200
		201
		202
		203
		204
		205
		206
		207
		208
		209
		210
		211
		212
		213
		214
		215
		216
		217
		218
		219
		220
		221
		222
		223
		224
		225
		226
		227
		228
		229
		230
		231
		232
		233
		234
		235
		236
		237
		238
		239
		240
		241
		242
		243
		244
		245
		246
		247
		248
		249
		250
		251
		252
		253
		254
		255
		256
		257
		258
		259
		260
		261
		262
		263
		264
		265
		266
		267
		268
		269
		270
		271
		272
		273
		274
		275
		276
		277
		278
		279
		280
		281
		282
		283
		284
		285
		286
		287
		288
		289
		290
		291
		292
		293
		294
		295
		296
		297
		298
		299
		300
		301
		302
		303
		304
		305
		306
		307
		308
		309
		310
		311
		312
		313
		314
		315
		316
		317
		318
		319
		320
		321
		322
		323
		324
		325
		326
		327
		328
		329
		330
		331
		332
		333
		334
		335
		336
		337
		338
		339
		340
		341
		342
		343
		344
		345
		346
		347
		348
		349
		350
		351
		352
		353
		354
		355
		356
		357
		358
		359
		360
		361
		362
		363
		364
		365
		366
		367
		368
		369
		370
		371
		372
		373
		374
		375
		376
		377
		378
		379
		380
		381
		382
		383
		384
		385
		386
		387
		388
		389
		390
		391
		392
		393
		394
		395
		396
		397
		398
		399
		400
		401
		402
		403
		404
		405
		406
		407
		408
		409
		410
		411
		412
		413
		414
		415
		416
		417
		418
		419
		420
		421
		422
		423
		424
		425
		426
		427
		428
		429
		430
		431
		432
		433
		434
		435
		436
		437
		438
		439
		440
		441
		442
		443
		444
		445
		446
		447
		448
		449
		450
		451
		452
		453
		454
		455
		456
		457
		458
		459
		460
		461
		462
		463
		464
		465
		466
		467
		468
		469
		470
		471
		472
		473
		474
		475
		476
		477
		478
		479
		480
		481
		482
		483
		484
		485
		486
		487</

Test de rango (BETWEEN)	103
Test de pertenencia a conjunto (IN)	106
Test de correspondencia con patrón (LIKE)	108
Test de valor nulo (IS NULL)	110
Condiciones de búsqueda compuestas (AND, OR y NOT)	112
Ordenación de los resultados de una consulta (cláusula ORDER BY)	115
Reglas para procesamiento de consultas de tabla única	117
Combinación de resultados de consulta (UNION *)	118
Uniones y filas duplicadas *	121
Uniones y ordenación *	122
Uniones múltiples*	123
Resumen	124

7. Consultas multitable (composiciones)	127
Ejemplo de consulta de dos tablas	127
Composiciones simples (equicomposiciones)	129
Consultas padre/hijo	132
Composiciones con criterios de selección de fila	134
Múltiples columnas de emparejamiento	135
Consultas de tres o más tablas	135
Otras equicomposiciones	138
Composiciones basadas en desigualdad	141
Consideraciones SQL para consultas multitable	142
Nombres de columna cualificados	142
Selecciones de todas las columnas	144
Autoacomposiciones	145
Alias de tablas	148
Rendimiento de consultas multitable	149
La estructura de una composición	150
Multiplicación de tablas	150
Reglas para procesamiento de consultas multitable	152
Composiciones externas*	153
Composiciones externas izquierda y derecha*	158
Notación de composición externa*	159
Resumen	161

Determinación de valores extremos (MIN y MAX)	167
Cuenta de valores de datos (COUNT)	169
Funciones de columna en la lista de selección	170
Valores NULL y funciones de columna	172
Eliminación de filas duplicadas (DISTINCT)	174
Consultas agrupadas (cláusula GROUP BY)	175
Múltiples columnas de agrupación	178
Restricciones en consultas agrupadas	182
Valores NULL en columnas de agrupación	183
Condiciones de búsqueda de grupos (cláusula HAVING)	185
Restricciones en condiciones de búsqueda de grupos	188
Valores NULL y condiciones de búsqueda de grupos	189
HAVING sin GROUP BY	190
Resumen	190

9. Subconsultas	191
Utilización de subconsultas	191
¿Qué es una subconsulta?	193
Subconsultas en la cláusula WHERE	194
Referencias externas	194
Condiciones de búsqueda en subconsultas	196
Test de comparación subconsulta (=, <, >, <=, >, > =)	196
Test de pertenencia a conjunto (IN)	199
Test de existencia (EXISTS)	200
Test cuantificados (ANY y ALL)*	202
Subconsultas y composiciones	208
Subconsultas anidadas	210
Subconsultas correlacionadas*	211
Subconsultas en la cláusula HAVING*	214
Resumen	216
Consultas SQL. Resumen final	217

Tercera parte. Actualización de datos	219
--	------------

10. Actualizaciones de bases de datos	221
Adición de datos a la base de datos	222
La sentencia INSERT de una fila	222
La sentencia INSERT multilibra	226
Utilidades de carga masiva	229
Supresión de datos de la base de datos	230

8. Consultas sumarias	163
Funciones de columna	163
Cálculo del total de una columna (SUM)	165
Cálculo del promedio de una columna (AVG)	166

La sentencia DELETE	230	Contenido	xv
Supresión de todas las filas	232		
DELETE con subconsulta *	232		
Modificación de datos en la base de datos	235		
La sentencia UPDATE	235		
Actualización de todas las filas	238		
UPDATE con subconsulta*	238		
Resumen	239		
11. Integridad de datos	241		
¿Qué es la integridad de datos?	241		
Datos requeridos	243		
Comprobación de validez	244		
Integridad de entidad	245		
Otras restricciones de unicidad	246		
Unicidad y valores NULL	247		
Integridad referencial	247		
Problemas de integridad referencial	248		
Reglas de supresiones	250		
Supresiones en cascada *	252		
Círculos referenciales*	254		
Claves foráneas y valores NULL*	258		
Reglas comerciales	260		
¿Qué es un disparador?	261		
Disparadores e integridad referencial	262		
El futuro de los disparadores	263		
Resumen	264		
12. Procesamiento de transacciones	265		
¿Qué es una transacción?	265		
COMMIT y ROLLBACK	267		
El modelo de transacción ANSI/ISO	269		
Otros modelos de transacciones	271		
Transacciones: tras el telón*	274		
Transacciones y procesamiento multiusuario	276		
El problema de la actualización perdida	276		
El problema de los datos no cumplimentados	277		
El problema de los datos inconsistentes	279		
Transacciones concurrentes	280		
Cerramiento (<i>locking</i>)*	282		
Niveles de cerramiento	282		
Cuarta parte. Estructura de una base de datos	295		
13. Creación de una base de datos	297		
El lenguaje de definición de datos	297		
Creación de una base de datos	299		
Definiciones de tablas	300		
Creación de una tabla (CREATE TABLE)	300		
Eliminación de una tabla (DROP TABLE)	308		
Modificación de una definición de tabla (ALTER TABLE)	309		
Sinónimos (CREATE/DROP SYNONYM)	313		
Índices CREATE/DROP INDEX)	314		
Otros objetos de bases de datos	318		
Estructura de base de datos	318		
Arquitectura de base de datos única	320		
Arquitectura de bases de datos múltiples	322		
Arquitectura de ubicación múltiple	324		
DDL y el estándar ANSI/ISO	326		
Resumen	329		
14. Vistas	331		
¿Qué es una vista?	331		
Cómo maneja el DBMS las vistas	333		
Ventajas de las vistas	334		
Desventajas de las vistas	335		
Creación de una vista (CREATE VIEW)	335		
Vistas horizontales	336		
Vistas verticales	338		
Vistas con subconjuntos fila/columna	340		
Vistas agrupadas	340		
Vistas compuestas	342		
Actualización de una vista	344		
Actualizaciones de vistas y el estándar ANSI/ISO	346		
Actualizaciones de vistas en productos SQL comerciales	346		
Comprobación de actualizaciones de vistas (CHECK OPTION)	347		
Resumen	350		

15. Seguridad SQL	351	Utilización de variables principales 414 Recuperación de datos en SQL incorporado 422 Consultas monofila 423 Consultas multifila 430 Supresiones y actualizaciones basadas en cursor 439 Cursos y procesamiento de transacciones 443 Resumen 444
16. El catálogo de sistema	375	¿Qué es el catálogo de sistema? 375 El catálogo y sus herramientas de consulta 376 El catálogo y el estándar ANSI/ISO 378 Contenidos del catálogo 378 Información sobre tablas 380 Información sobre columnas 382 Información sobre vistas 384 Comentarios y etiquetas 385 Información sobre relaciones 386 Información sobre usuarios 389 Información sobre privilegios 390 Otras informaciones 392 Resumen 392
17. SQL incorporado	395	Técnicas de SQL programado 395 Procesamiento de sentencias DBMS 397 Conceptos de SQL incorporado 399 Desarrollo de un programa de SQL incorporado 400 Ejecución de un programa de SQL incorporado 404 Sentencias simples de SQL incorporado 406 Declaración de tablas 410 Gestión de errores 410
Quinta parte. Programación con SQL	393	489
18. SQL dinámico*	447	Limitaciones de SQL estático 447 Conceptos de SQL dinámico 449 Ejecución dinámica de sentencia (EXECUTE IMMEDIATE) 451 Ejecución dinámica en dos pasos 453 La sentencia PREPARE 457 La sentencia EXECUTE 466 La sentencia DECLARE STATEMENT 467 Consultas dinámicas 472 La sentencia DESCRIBE 474 La sentencia DECLARE CURSOR 475 La sentencia OPEN dinámica 475 La sentencia FETCH dinámica 477 La sentencia CLOSE dinámica 478 Dialectos de SQL dinámico 479 SQL dinámico en SQL/DS 479 SQL dinámico en Oracle* 482 Resumen 486
19. API de SQL	489	
		Conceptos API 489 El API de SQL Server 490 Técnicas básicas de SQL Server 493 Consultas en SQL Server 499 Procedimientos almacenados 505 Actualizaciones posicionadas 513 Consultas dinámicas 515 Otras interfaces de llamada de función 520 El API de llamadas Oracle 520 El API de SQLBase 522 Resumen 528

Sexta parte. Direcciones futuras	529	
20. Gestión de bases de datos distribuidas	531	
El reto de la gestión de datos distribuidos	531	
Etapas del acceso a datos distribuidos	536	
Peticiones remotas	536	
Transacciones remotas	538	
Transacciones distribuidas	540	
Peticiones distribuidas	542	
Tablas distribuidas	542	
Divisiones horizontales de tabla	544	
Divisiones verticales de tabla	546	
Tablas reflejadas	548	
El protocolo de cumplimentación en dos fases*	549	
Resumen	553	
21. El futuro de SQL	555	
Tendencia del mercado de bases de datos	556	
Tendencias del rendimiento hardware	557	
La guerra de los bancos de prueba	559	
Productos DBMS en grupos	560	
Estándares SQL	561	
Extensões del lenguaje SQL	563	
Tipos de datos complejos	564	
Aplicaciones cliente/servidor	566	
Acceso SQL desde aplicaciones PC	566	
Herramientas de bases de datos	568	
Bases de datos distribuidas	569	
Bases de datos orientadas a objetos	570	
A. La base de datos ejemplo	573	
B. Perfiles de vendedores DBMS	579	
Ashton-Tate Corporation	580	
Digital Equipment Corporation	581	
Gupta Technologies, Inc.	581	
Hewlett-Packard Company	582	
IBM Corporation	583	
Informix Software, Inc.	584	
C. Lista de empresas y productos	591	
D. Sintaxis de SQL ANSI/ISO	597	
Sentencias de definición de datos	598	
Sentencias básicas de manipulación de datos	598	
Sentencias de procesamiento de transacciones	599	
Sentencias basadas en cursor	599	
Expresiones de consulta	600	
Condiciones de búsqueda	600	
Expresiones	600	
Elementos simples	601	
E. Marcas registradas	603	
Indice	607	

Agradecimientos

Este libro no se habría producido sin la dedicación y el duro trabajo de muchas personas en Osborne/McGraw-Hill. En particular, nos gustaría agradecer a Cindy Hudson su paciencia, a Liz Fisher su perseverante, a Judy Wohlfstrom su tolerancia, a Ilene Shapera por mantener el proceso en acción y a Judith Brown, Nancy Beckus y Ann Spivack por su atención meticolosa a los detalles. Además, gracias especiales a Joe Brilando de Ashton-Tate por formular las preguntas que han inspirado este libro.

Prólogo

Aplique SQL proporciona un tratamiento completo y en profundidad del lenguaje SQL destinado a usuarios tanto técnicos, programadores, profesionales y gestores de procesamiento de datos que desean entender el impacto de SQL en el mercado informático. Este libro ofrece un marco conceptual para comprender y utilizar SQL, describe la historia de SQL y los estándares SQL y explica el papel de SQL en la industria informática actual. Este libro mostrará, paso a paso, cómo utilizar las características SQL, con muchas ilustraciones y ejemplos realistas para clarificar los conceptos SQL. Además el libro compara los productos SQL de los principales vendedores DBMS, describiendo sus ventajas, beneficios y compromisos, para ayudar a seleccionar el producto adecuado a una aplicación.

En algunos de los capítulos de este libro, el material se explora a dos niveles diferentes —una descripción fundamental del tema y una discusión avanzada destinada a profesionales informáticos que necesitan entender parte de las «interioridades» de SQL—. La información más avanzada se trata en las secciones marcadas con un asterisco (*). No es necesario leer estas secciones para obtener una comprensión de lo que SQL es y de lo que hace.

Cómo está organizado este libro

El libro está dividido en seis partes que cubren diferentes aspectos del lenguaje SQL:

- La Primera parte, «Panorámica general de SQL», proporciona una introducción a SQL y una perspectiva de mercado de su papel como lenguaje de base

■ La Segunda parte, «Recuperación de datos», describe las características de SQL que permiten efectuar consultas a la base de datos. El primer capítulo de esta parte describe la estructura básica del lenguaje SQL. Los cuatro capítulos siguientes comienzan con las consultas SQL más sencillas, y progresivamente construyen consultas más complejas, incluyendo consultas multitable, consultas sumarias y consultas que utilizan subconsultas.

■ La Tercera parte, «Actualización de datos», muestra cómo se puede utilizar SQL para añadir nuevos datos a una base, suprimir datos de una base y modificar los datos de una base ya existente. También describe las cuestiones de integridad que se presentan cuando los datos se actualizan y cómo SQL trata estas cuestiones. El último de los tres capítulos de esta parte discute el concepto de transacción SQL y el soporte SQL para el procesamiento de transacciones multiusuario.

■ La Cuarta parte, «Estructura de una base de datos», se ocupa de la creación y administración de una base de datos basada en SQL. Sus cuatro capítulos informan sobre cómo crear tablas, vistas e índices que forman la estructura de una base de datos relacional. También describe el esquema de seguridad SQL que impide el acceso no autorizado a los datos y el catálogo de sistema SQL que describe la estructura de una base de datos. Esta parte también discute las diferencias significativas entre las estructuras de bases de datos soportadas por varios productos DBMS basados en SQL.

■ La Quinta parte, «Programación con SQL», describe cómo utiliza SQL los programas de aplicación para acceder a una base de datos. Discute el SQL incorporado especificado por el estándar ANSI y utilizado por IBM, Oracle, Ingres, Informix y la mayoría de los productos DBMS basados en SQL. También describe el interfaz de SQL dinámico utilizado para construir herramientas de bases de datos de propósito general tales como escritores y programadores de inspección de bases de datos. Finalmente, esta parte describe los APIs de SQL proporcionados por SQL Server, Oracle, SQLBase y los compara con los interfaces IBM y ANSI incorporados.

■ La Sexta parte, «Direcciones futuras», examina el estado actual de los productos DBMS basados en SQL, las direcciones que SQL seguirá en la próxima década y el probable impacto SQL en los diferentes segmentos del mercado informático. Describe la intensa actividad actual en bases de datos distribuidos, la continua evolución de los estándares SQL y el papel de las bases de

datos basadas en SQL en las aplicaciones de procesamiento de transacciones en línea. Esta parte también discute la batalla entre IBM, Oracle, Microsoft y Ashton-Tate en el mercado de servidores de bases de datos OS/2, y el impacto que las bases de datos orientadas a objetos pueden tener en la evolución de SQL en los noventa.

Convenciones utilizadas en este libro

Aplique SQL describe las características y funciones SQL que están disponibles en la mayoría de los productos DBMS basados en SQL más populares y aquéllas que son descritas en el estándar SQL ANSI/ISO. Cuando es posible, la sintaxis de sentencias SQL descrita en el libro y utilizada en los ejemplos se aplica a todos los dialectos de SQL. Cuando los dialectos se diferencian, las diferencias se señalan en el texto y los ejemplos siguen la práctica más común. En este caso usted puede tener que modificar ligeramente las sentencias SQL de los ejemplos para acomodarlas a su DBMS particular.

A lo largo del libro, los términos técnicos aparecen en cursiva la primera vez que son utilizados y definidos. Los elementos del lenguaje SQL, incluyendo las palabras clave SQL, los nombres de columna y tabla y las sentencias SQL de funciones API SQL aparecen en una fuente monoespacio mayúscula. Los listados de programa también aparecen en fuente monoespacio, y utilizan los convenios de caja normal para el lenguaje de programación particular (mayúsculas para COBOL y FORTRAN, minúsculas para C). Observe que estos convenios se utilizan únicamente para mejorar la legibilidad; la mayoría de las implementaciones SQL aceptarán sentencias tanto en mayúsculas como en minúsculas. Muchos de los ejemplos SQL incluyen resultados de consultas que aparecen inmediatamente a continuación de las sentencias SQL tal como lo harían en una sesión SQL interactiva. En algunos casos, si los resultados de las consultas son largos se truncan después de unas pocas filas. Esto se indica con una elipsis vertical (...) a continuación de la última fila de los resultados de la consulta.

Sobre los autores

James R. Groff

Jim Groff es cofundador y Presidente de la Network Innovations Corporation, ha desarrollado un software de red basado en SQL que enlaza computadores personales a bases de datos corporativas. Fundada en 1984, Network Innovations fue adquirida por Apple Computer en 1988. Antes de fundar Network

Innovations, Gross trabajó en Hewlett-Packard y en Plexus Computers, un fabricante de sistemas microcomputadores basados en UNIX. Es conferenciente habitual y autor de artículos técnicos sobre temas de SQL y UNIX. Gross obtuvo un título B.S. en matemáticas por el Massachusetts Institute of Technology y un M.B.A. por la Universidad de Harvard.

Paul N. Weinberg

Paul Weinberg es cofundador y Vicepresidente de Network Innovations Corporation. Antes de fundar Network Innovations, se ocupó del desarrollo de software y de trabajos comerciales en Bell Laboratories, Hewlett-Packard y Plexus Computers. Weinberg es coautor, junto a Jim Gross, de *Understanding UNIX: A Conceptual Guide*, un libro en ventas sobre el sistema operativo UNIX destinado a directores y profesionales del procesamiento de datos. Obtuvo un título B.S. por la Universidad de Michigan y un M.S. por la Universidad de Stanford, ambos en informática. Mientras estuvo en Standford, Weinberg colaboró en *The Simple Solution to Rubik's Cube*, el líder en ventas de 1981, con más de siete millones de copias en prensa.

Por qué este libro es para usted

Aplique SQL es el libro adecuado para quien desea entender y aprender SQL, incluyendo usuarios de bases de datos, profesionales del procesamiento de datos, programadores, estudiantes y gestores. Describe —en un lenguaje sencillo y comprensible extensamente ilustrado con figuras y ejemplos— lo que SQL es, por qué es importante y cómo utilizarlo. Este libro no es específico de ningún producto o dialecto particular de SQL. Al contrario, describe el núcleo central, estándar del lenguaje SQL y luego examina las diferencias entre los productos SQL más populares, incluyendo DB2, SQL/DS, OS/2 Extended Edition, Oracle, Ingres, Sybase, SQL Server, SQLBase, dBASE IV y otros.

Si usted es un usuario nuevo de SQL, este libro le ofrece un tratamiento completo paso a paso del lenguaje, que va desde consultas simples hasta los conceptos más avanzados. La estructura del libro le permitirá comenzar rápidamente a utilizar SQL, pero el libro continuará siendo valioso cuando empiece a utilizar características más complejas del lenguaje.

Si usted es un profesional o un director de procesamiento de datos, este libro le dará una perspectiva sobre el impacto que SQL está teniendo en todos los segmentos del mercado informático —desde los computadores personales a los grandes computadores, las redes de área local y los sistemas OLTP—. Los primeros capítulos del libro describen la historia de SQL, su papel en el mercado y su evolución desde las tecnologías de bases de datos primitivas. Los capítulos finales describen el futuro de SQL y el desarrollo de bases de datos distribuidos y otras tecnologías de bases de datos de los noventa.

Si usted es un programador, este libro le ofrece un tratamiento completo de la programación con SQL. A diferencia de los manuales de referencia de muchos productos DBMS, éste ofrece un marco conceptual para la programación SQL, explicando el *porqué* además del *cómo* del desarrollo de una aplicación basada

en SQL. En el libro se contrastan los interfaces de programación SQL ofrecidos por todos los principales productos SQL (SQL incorporado, SQL dinámico, dblib de Server SQL y otros APIs de SQL) proporcionando una perspectiva que no se halla en ningún otro libro.

Si usted está seleccionando un producto DBMS, este libro le ofrece una comparación detallada de las características, ventajas y beneficios SQL ofrecidos por los diferentes vendedores de DBMS. Las diferencias entre los principales productos DBMS se explican, no solamente en términos técnicos, sino también en términos de su impacto sobre las aplicaciones y su posición competitiva en el mercado.

En resumen, tanto los usuarios técnicos como los no técnicos pueden beneficiarse de este libro. Es la fuente más completa de información disponible referente al lenguaje SQL, características y beneficios SQL, productos populares basados en SQL, historia de SQL e impacto de SQL en la dirección futura del mercado informático.

Panorámica general de SQL

Primera parte

Los cuatro primeros capítulos de este libro proporcionan una perspectiva y una introducción rápida a SQL. El Capítulo 1 describe lo que SQL es y explica sus principales características y beneficios. En el Capítulo 2, un rápido repaso de SQL muestra muchas de sus capacidades con ejemplos sencillos y breves. El Capítulo 3 ofrece una perspectiva de mercado de SQL repasando su historia, describiendo los estándares SQL y los principales vendedores de productos basados en SQL e identificando las razones para la importancia actual de SQL. El Capítulo 4 describe el modelo de datos relacional sobre el cual se basa SQL y lo compara con los modelos de datos más primitivos.

1 *Introducción*

La explosiva popularidad de SQL es una de las tendencias más importantes en la industria informática hoy día. Durante los últimos años, SQL se ha convertido en el lenguaje de base de datos estándar. Alrededor de 100 productos de gestión de base de datos soportan ahora SQL, ejecutándose en sistemas informáticos que van desde computadoras personales a mainicomputadores. Se ha adoptado un estándar SQL internacional oficial, y SQL juega un papel clave en el estándar Systems Application Architecture de IBM. Los artículos en las revistas informáticas celebran a SQL como una nueva *lingua franca* que proporciona una partición de datos sin fisuras entre computadores en red procedentes de muchos vendedores diferentes. Desde sus oscuros comienzos como un proyecto de investigación en IBM, SQL ha saltado a un lugar prominente como tecnología informática importante y como fuerza de mercado poderosa.

¿Qué es exactamente SQL? ¿Por qué es importante? ¿Qué puede hacer y cómo hace su trabajo? Si SQL es realmente un estándar, ¿por qué hay tantas versiones y dialectos diferentes? ¿En qué se asemejan y se diferencian productos SQL populares como dBASE IV, OS/2 Extended Edition, SQL Server, SQLBase, Oracle, Ingres, Sybase y DB2? ¿Es SQL realmente importante en computadores personales? ¿Puede satisfacer la demanda de procesamiento de transacciones de elevado volumen? ¿Cómo impactará SQL en la manera de utilizar los computadores y cómo se puede conseguir el máximo de esta herramienta de gestión de datos emergente?

El lenguaje SQL

SQL es una herramienta para organizar, gestionar y recuperar datos almacenados en una base de datos informática. El nombre «SQL» es una abreviatura de

Structured Query Language (Lenguaje de Estructuras de Consultas). Por razones históricas, SQL se pronuncia generalmente «sequel», pero también se emplea la pronunciación alternativa «S.Q.L.». Como su nombre implica, SQL es un *lenguaje de informática* que se puede utilizar para interactuar con una base de datos. En efecto, SQL trabaja con un tipo específico de base de datos, llamada *base de datos relacional*.

La Figura 1.1 muestra cómo funciona SQL. El sistema informático de la figura tiene una *base de datos* que almacena información importante. Si el sistema informático estuviere en una empresa comercial, la base de datos podría almacenar datos de inventario, producción, ventas o nómina. En un computador personal, la base de datos podría almacenar datos referentes a los cheques que usted haya firmado, listas de personas y sus números de teléfono, o datos extraídos de un sistema informático mayor. El *programa informático* que controla la base de datos se denomina *sistema de gestión de base de datos (database management system)* o DBMS.

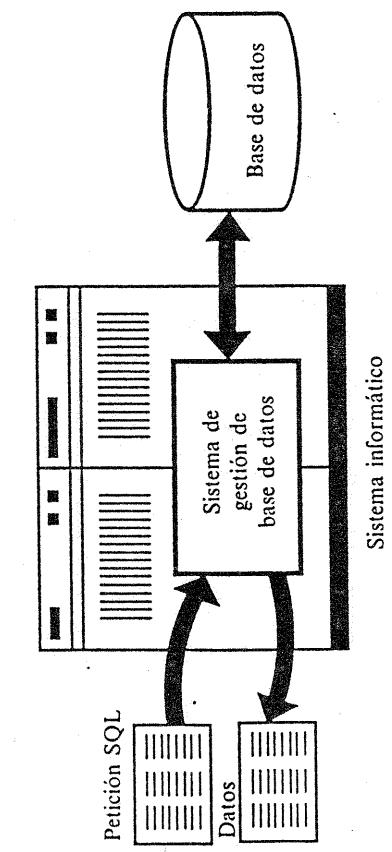


Figura 1.1. Uso de SQL para acceso a base de datos.

Cuando se necesita recuperar datos de la base de datos, se utiliza el lenguaje SQL para efectuar la petición. El DBMS procesa la petición SQL, recupera los datos solicitados y los devuelve. Este proceso de solicitar datos de la base de datos y de recibir los resultados se denomina *consulta (query)* a la base de datos; de aquí el nombre Structured Query Language.

El nombre Structured Query Language es realmente y en cierta medida inapropiado. En primer lugar, SQL es mucho más que una herramienta de consulta, aunque ese fue su propósito original y recuperar datos sigue siendo una de sus funciones más importantes. SQL se utiliza para controlar todas las funciones que un DBMS proporciona a sus usuarios, incluyendo:

■ *Definición de datos.* SQL permite a un usuario definir la estructura y organización de los datos almacenados y de las relaciones entre ellos.

■ *Recuperación de datos.* SQL permite a un usuario o a un programa de aplicación recuperar los datos almacenados de la base de datos y utilizarlos.

■ *Manipulación de datos.* SQL permite a un usuario o a un programa de aplicación actualizar la base de datos añadiendo nuevos datos, suprimiendo datos antiguos y modificando datos previamente almacenados.

■ *Control de acceso.* SQL puede ser utilizado para restringir la capacidad de un usuario para recuperar, añadir y modificar datos, protegiendo así los datos almacenados frente a accesos no autorizados.

■ *Compartición de datos.* SQL se utiliza para coordinar la compartición de datos por parte de usuarios concurrentes, asegurando que no interfieren unos con otros.

■ *Integridad de datos.* SQL define restricciones de integridad en la base de datos, protegiéndola contra corrupciones debidas a actualizaciones inconsistentes o a fallos del sistema.

Por tanto SQL es un lenguaje completo de control e interactuación con un sistema de gestión de base de datos.

En segundo lugar, SQL no es realmente un lenguaje informático completo tal como COBOL, FORTRAN o C. SQL no dispone de la sentencia IF para examinar condiciones, ni de la sentencia GOTO para bifurcaciones, ni de las sentencias DO o FOR para iteraciones. En vez de ello, SQL es un *sublenguaje* de base de datos, consistente en unas treinta sentencias especializadas para tareas de gestión de bases de datos. Estas sentencias SQL se *incorporan* a otro lenguaje, como COBOL, FORTRAN o C para extender ese lenguaje y permitirle utilizar el acceso a la base de datos.

Finalmente, SQL no es un lenguaje particularmente estructurado, especialmente cuando se compara con lenguajes altamente estructurados tales como C o Pascal. En vez de ello, las sentencias SQL se asemejan a frases en inglés, completadas con «palabras de relleno» que no añaden nada al significado de la frase pero que hace que se lea más naturalmente. Hay unas cuantas inconsistencias en el lenguaje SQL, y también existen algunas reglas especiales para impedir la construcción de sentencias SQL que parecen perfectamente legales, pero que no tienen sentido.

A pesar de la imprecisión de su nombre, SQL ha emergido como el lenguaje estándar para la utilización de bases de datos relacionales. SQL es a la vez un potente lenguaje y un lenguaje relativamente fácil de aprender. El rápido repaso de SQL en el próximo capítulo proporcionará una buena panorámica general del lenguaje y sus capacidades.

SQL no es en sí mismo un sistema de gestión de base de datos, ni un producto autónomo. Usted no puede ir a una tienda de informática y «comprar SQL». En su lugar, SQL es parte integral de un sistema de gestión de base de datos, un lenguaje y una herramienta para comunicarse con el DBMS. La Figura 1.2 muestra algunos de los componentes de un DBMS típico, y cómo SQL actúa como el «adhesivo» que los une.

La *máquina de base de datos* es el corazón del DBMS, responsable de estructurar, almacenar y recuperar realmente los datos en el disco. Acepta peticiones SQL procedentes de otros componentes DBMS, tales como una facilidad de formularios, un escritor de informes o una facilidad de consultas interactivas, desde los programas de aplicación escritos por el usuario e incluso desde otros sistemas informáticos. Como muestra la figura, SQL juega muchos papeles diferentes:

■ SQL es un *lenguaje de consultas interactivas*. Los usuarios escriben órdenes SQL en un programa SQL interactivo para recuperar datos y mostrarlos en la pantalla, proporcionando una herramienta conveniente y fácil de utilizar para consultas ad hoc de la base de datos.

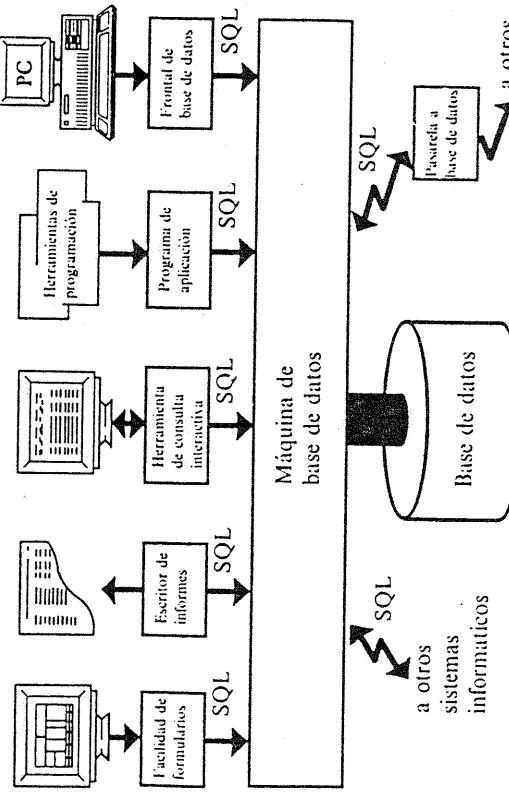


Figura 1.2. Componentes de un sistema típico de gestión de base de datos.

■ SQL es un *lenguaje de programación de base de datos*. Los programadores insertan órdenes SQL en sus programas de aplicación para acceder a los datos de la base. Tanto los programas escritos por el usuario como los programas de utilidad de la base de datos (tales como los escritores de informe y las herramientas de entrada de datos) utilizan esta técnica para acceso a la base de datos.

■ SQL es un *lenguaje de administración de base de datos*. El administrador de la base de datos responsable de gestionar una base de datos en un minicomputador o en un mainframe utiliza SQL para definir la estructura de la base de datos y para controlar el acceso a los datos almacenados.

■ SQL es un *lenguaje cliente/servidor*. Los programas de computador personal utilizan SQL para comunicarse sobre una red de área local con servidores de base de datos que almacenan los datos compartidos. Muchas aplicaciones novedosas están utilizando esta arquitectura de cliente/servidor, que minimiza el tráfico por la red y permite que tanto los PCs como los servidores efectúen mejor su trabajo.

■ SQL es un *lenguaje de datos distribuidos*. Los sistemas de gestión de base de datos distribuidos utilizan SQL para ayudar a distribuir datos a través de muchos sistemas informáticos conectados. El software DBMS de cada sistema utiliza SQL para comunicarse con los otros sistemas, enviando peticiones para acceso a datos.

■ SQL es un *lenguaje de pasarela de base de datos*. En una red informática con una mezcla de diferentes productos DBMS, SQL se utiliza a menudo en una *pasarela (gateway)* que permite que nuestro producto de DBMS se comunique con otro producto.

SQL ha emergido por tanto como una herramienta potente y útil para enlazar personas y sistemas informáticos a los datos almacenados en una base de datos relacional.

Características y beneficios de SQL

SQL es a la vez un lenguaje fácil de entender y una herramienta completa para gestionar datos. He aquí algunas de las principales características de SQL y las fuerzas del mercado que le han hecho tener éxito:

■ Su independencia de los vendedores.

■ Su portabilidad a través de sistemas informáticos.

■ Los estándares SQL.

Estándares SQL

- Su fundamento relacional.
- Su estructura de alto nivel semejante al inglés.
- Las consultas interactivas ad hoc.
- Su acceso a la base de datos mediante programas.
- Las vistas múltiples de datos.
- El ser un lenguaje de base de datos.
- Su definición dinámica de datos.
- La arquitectura cliente/servidor.

Estas son las razones por las que SQL ha emergido como la herramienta estándar para gestionar datos en computadores personales, minicomputadores y maxicomputadores. Cada una de ellas se describe en las secciones siguientes.

Independencia de los vendedores

SQL es ofrecido por todos los principales vendedores de DBMS, y ningún producto nuevo de base de datos puede tener éxito sin el soporte de SQL. Una base de datos basada en SQL y los programas que la utilizan pueden transferirse de un DBMS al DBMS de otro vendedor con mínimo esfuerzo de conversión y poco reentrenamiento del personal. Herramientas de base de datos basadas en SQL que funcionan con varios productos de DBMS, tales como escritores de informe y generadores de aplicación, están comenzando a aparecer. La independencia del vendedor proporcionada así por SQL es una de las razones más importantes de su popularidad.

Portabilidad a través de sistemas informáticos

Los vendedores de DBMS en SQL ofrecen sus productos sobre sistemas informáticos que van desde computadores personales y estaciones de trabajo hasta redes de área local, minicomputadores y maxicomputadores. Las aplicaciones basadas en SQL que comienzan en sistemas monousuario pueden ser transferidas a sistemas mayores de minicomputadores o maxicomputadores cuando crecen. Los datos procedentes de bases de datos corporativas basadas en SQL pueden ser extraídas y remitidas a bases de datos departamentales o personales. Finalmente, los económicos computadores personales pueden ser utilizados para construir prototipos de aplicaciones de bases de datos basadas en SQL antes de transferirlas a un sistema multiusuario costoso.

- El American National Standards Institute (ANSI) y la International Standards Organization (ISO) han publicado conjuntamente un estándar oficial para SQL.
- SQL se ha convertido también en un estándar del U.S. Federal Information Processing Standard (FIPS), lo que le convierte en un requerimiento esencial para los grandes contratos informáticos del gobierno. En Europa, X/Open, un estándar para un entorno de aplicación por cable basado en UNIX, ha añadido SQL como el estándar para acceso a base de datos. La Open Software Foundation (OSF), un grupo de vendedores de UNIX, planea basar su estándar de acceso a base de datos en SQL. Estos estándares sirven como sello oficial de aprobación para SQL y han acelerado su aceptación en el mercado.

Apoyo de IBM

SQL fue inventado originalmente por investigadores de IBM, y desde entonces se ha convertido en un producto estratégico para IBM. SQL es un componente esencial de la Systems Application Architecture (SAA), la marca de IBM para la compatibilidad de sus diversas líneas de productos. El soporte SQL está disponible en todas las cuatro familias de sistemas incluidas bajo el paraguas SAA; los computadores personales PS/2, los sistemas de medio rango AS/400 y los maxi-computadores de IBM que ejecutan los sistemas operativos MVS y VM. Este extenso soporte de IBM ha acelerado la aceptación del mercado de SQL, y proporcionado una clara señal de las intenciones de IBM para que otros vendedores de sistemas y bases de datos las sigan.

Fundamento relacional

SQL es un lenguaje para base de datos relacional, y se ha popularizado juntamente con el modelo de base de datos relacional. La estructura tabular, de filas y columnas de una base de datos relacional, es intuitiva para los usuarios, y hace que el lenguaje SQL se mantenga simple y fácil de entender. El modelo relacional tiene también un fuerte fundamento teórico que ha guiado la evolución y la implementación de las bases de datos relacionales. Cabalgando sobre una ola de implementación de las bases de datos relacionales, SQL se ha convertido en la aceptación provocada por el éxito del modelo relacional, SQL se ha convertido en el lenguaje de base de datos para las bases de datos relacionales.

Estructura de alto nivel en inglés

Las sentencias SQL parecen sencillas frases en inglés, lo que hace que SQL sea fácil de aprender y entender. Esto es en parte debido a que las sentencias de

SQL describe los *datos* a recuperar, en lugar de especificar *cómo* hallar los datos. Las tablas y columnas de una base de datos SQL pueden tener nombres largos y descriptivos. Como consecuencia, la mayoría de las sentencias SQL «dicen lo que significan», y pueden ser leídas como frases claras y naturales.

Consultas interactivas ad hoc

SQL es un lenguaje de consultas interactivas que proporciona a los usuarios acceso ad hoc a los datos almacenados. Utilizando SQL interactivamente, un usuario puede obtener respuestas incluso a cuestiones complejas en minutos o segundos, en fuerte contraste con los días o semanas que le llevaría a un programador escribir un programa de informe a medida. Debido a la potencia de consulta ad hoc de SQL, los datos son más accesibles y pueden ser utilizados para ayudar a una organización a tomar decisiones mejores y más informadas.

Acceso a la base de datos mediante programas

SQL es también un lenguaje de base de datos utilizado por los programadores para escribir aplicaciones que acceden a una base de datos. Las mismas sentencias SQL se utilizan para acceso interactivo y programado, de modo que las partes de acceso a base de datos de un programa pueden ser comprobadas primero con SQL interactivo y luego insertadas dentro del programa. En contraste, las bases de datos tradicionales proporcionan un conjunto de herramientas para acceso mediante programas y una facilidad de consulta aparte para peticiones ad hoc, sin ninguna sinergia entre los dos modos de acceso.

Vistas múltiples de datos

Utilizando SQL, el creador de una base de datos puede dar a diferentes usuarios de la base de datos *vistas* diferentes de su estructura y contenidos. Por ejemplo, la base de datos puede ser construida de modo que cada usuario sólo vea datos de su propio departamento o de su propia región de ventas. Además, los datos procedentes de diferentes partes de la base de datos pueden combinarse y presentarse al usuario como una simple fila/columna de una tabla. Las vistas de SQL pueden ser utilizadas de este modo para mejorar la seguridad de una base de datos y para acomodarla a las necesidades particulares de los usuarios individuales.

Lenguaje completo de base de datos

SQL fue inicialmente desarrollado como un lenguaje de consulta ad hoc, pero su potencia va ahora más allá de la recuperación de datos. SQL proporciona un

lenguaje complejo y consistente para crear una base de datos, gestionar su seguridad, actualizar sus contenidos, recuperar los datos y compartirlos entre muchos usuarios concurrentes. Los conceptos de SQL que son aprendidos en una parte del lenguaje pueden ser aplicados a otras órdenes de SQL, haciendo más productivos a sus usuarios.

Definición dinámica de datos

Utilizando SQL, la estructura de una base de datos puede ser modificada y ampliada dinámicamente, incluso mientras los usuarios están accediendo a los contenidos de la base de datos. Este es un avance importante sobre los lenguajes de definición de datos estáticos, que impiden el acceso a la base de datos mientras su estructura está siendo modificada. SQL proporciona de este modo máxima flexibilidad, permitiendo que una base de datos se adapte a exigencias cambiantes mientras continúan sin ser interrumpidas las aplicaciones en línea.

Arquitectura cliente/servidor

SQL es un vehículo natural para implementar aplicaciones utilizando una arquitectura cliente/servidor distribuida. En este papel, SQL sirve como enlace entre los sistemas informáticos «frontales» (*front-end*) optimizados para interacción con el usuario y los sistemas «de apoyo» (*back-end*) especializados para gestión de bases de datos, permitiendo que cada sistema rinda lo mejor posible. SQL también permite que los computadores personales funcionen como frontales de bases de datos mayores dispuestas en minicomputadores y mainframes, proporcionando acceso a datos corporativos desde aplicaciones informáticas personales.

2 Rápido repaso de SQL

Antes de sumergirnos en los detalles de SQL, es buena idea desarrollar una perspectiva global del lenguaje y de cómo funciona. Este capítulo contiene un rápido repaso de SQL que ilustra sus características y funciones más importantes. El objetivo de este rápido repaso no es convertirle a usted en un experto en la escritura de sentencias SQL; ese es el objetivo de la Segunda parte de este libro. En su lugar, para cuando haya finalizado este capítulo, usted tendrá una familiaridad básica con el lenguaje SQL y una panorámica general de sus capacidades.

Una base de datos sencilla

Los ejemplos expuestos en este repaso rápido se basan en un sencilla base de datos relacional para una pequeña empresa de distribución. La base de datos, mostrada en la Figura 2.1, almacena la información necesaria para implementar una pequeña aplicación de procesamiento de pedidos. Específicamente, almacena la información siguiente:

- los clientes que compran los productos de la empresa,
- los pedidos remitidos por esos clientes,
- los vendedores que venden los productos a los clientes, y
- las oficinas de ventas en donde trabajan los vendedores.

Tabla PEDIDOS

NUM_PEDIDO	CLIE	PRODUCTO	CANT	IMPORTE
112961	2117	2A44L	7	\$31,500.00
113012	2111	41003	35	\$3,745.00
112989	2101	114	6	\$1,558.00
113051	2118	XK47	4	\$1,420.00
112968	2102	41004	34	\$3,978.00
113036	2107	41002	9	\$32,500.00
113045	2112	2A44R	10	\$45,000.00
112963	2103	41004	28	\$3,276.00
113013	2118	41003	1	\$652.00
113058	2108	112	10	\$1,480.00
112997	2124	41003	1	\$852.00
112983	2103	41004	6	\$762.00
113024	2114	XK47	20	\$7,100.00
113062	2124	114	10	\$2,430.00
112979	2114	41002	6	\$15,000.00
113027	2103	41002	54	\$4,104.00
113007	2112	737C	3	\$2,925.00
113069	2109	737C	22	\$31,350.00
113034	2107	2A45C	8	\$632.00
112992	2118	41002	10	\$760.00
112975	2111	2A44G	6	\$2,100.00
113055	2108	4100X	6	\$150.00
113048	2120	779C	2	\$3,750.00
112993	2106	2A45C	24	\$1,896.00
113065	2106	XK47	6	\$2,130.00
113003	2108	737C	3	\$5,625.00
113049	2118	XK47	2	\$776.00
112987	2103	4100Y	11	\$27,500.00
113057	2111	4100X	24	\$600.00
113042	2113	2A44R	5	\$22,500.00

Tabla CLIENTES

NUM_CLIE	EMPRESA	REP_CLIE	LIMITE_CREDITO
2111	JCP Inc.	103	\$50,000.00
2102	First Corp.	101	\$55,000.00
2103	Acme Mfg.	105	\$50,000.00
2123	Carter & Sons	102	\$40,000.00
2107	Ace International	110	\$35,000.00
2115	Smithson Corp.	101	\$20,000.00
2101	Jones Mfg.	106	\$65,000.00
2112	Zetacorp	108	\$50,000.00
2121	GMA Assoc.	103	\$45,000.00
2114	Orion Corp.	102	\$20,000.00
2124	Peter Brothers	107	\$40,000.00
2108	Holm & Landis	109	\$55,000.00
2117	J.P. Sinclair	106	\$35,000.00
2122	Three-Way Lines	105	\$30,000.00
2120	Rico Enterprises	102	\$50,000.00
2106	Fred Lewis Corp.	102	\$85,000.00
2119	Solomon Inc.	109	\$25,000.00
2118	Midwest Systems	108	\$60,000.00
2113	Ian & Schmidt	104	\$20,000.00
2109	Then Associates	107	\$25,000.00
2105	AAA Investments	101	\$45,000.00

Tabla REPVENTAS

NOMBRE	OFICINA REP	CUOTA	VENTAS
Bill Adams	13	\$350,000.00	\$367,911.00
Mary Jones	11	\$300,000.00	\$392,725.00
Sue Smith	21	\$350,000.00	\$474,050.00
Sam Clark	11	\$275,000.00	\$299,912.00
Bob Smith	12	\$200,000.00	\$142,596.00
Dan Roberts	12	\$300,000.00	\$305,673.00
Tom Snyder	NULL	NULL	\$35,985.00
Larry Fitch	21	\$350,000.00	\$361,865.00
Paul Cruz	12	\$275,000.00	\$286,775.00
Nancy Angelli	22	\$300,000.00	\$186,042.00

Tabla OFICINAS

OFICINA	CIUDAD	REGION	OBJETIVO	VENTAS
22	Denver	Oeste	\$300,000.00	\$186,042.00
11	New York	Oeste	\$575,000.00	\$492,637.00
12	Chicago	Oeste	\$80,000.00	\$735,042.00
13	Atlanta	Oeste	\$350,000.00	\$367,911.00
21	Los Angeles	Oeste	\$725,000.00	\$835,915.00

Esta base de datos, como la mayoría de las demás, es un modelo del «mundo real». Los datos almacenados en la base de datos representan entidades reales: clientes, pedidos, vendedores y oficinas. Hay una tabla separada de datos para cada clase diferente de entidad. Las peticiones de base de datos que usted hace utilizando el lenguaje SQL se corresponden con actividades del mundo real, tales como emisión, cancelación y cambio de pedidos por parte de los clientes, contratación y despido de vendedores, etc. Veámos cómo se puede utilizar SQL para manipular los datos.

Recuperación de datos

Primero, listemos las oficinas de ventas, mostrando la ciudad en donde cada una está localizada y sus ventas anuales hasta la fecha. La sentencia SQL que recupera datos de la base de datos se denomina SELECT. Esta sentencia SQL recupera los datos que se deseen:

```
SELECT CIUDAD, OFICINA, VENTAS
FROM OFICINAS
```

La sentencia SELECT solicita tres datos —la ciudad, el número de oficina y las ventas— por cada oficina. También especifica que los datos provienen de la tabla OFICINAS, la cual almacena datos referentes a las oficinas de ventas. Los resultados de la consulta aparecen, en forma tabular, inmediatamente después de la solicitud.

La sentencia SELECT es utilizada por todas las consultas SQL. Por ejemplo, he aquí una consulta que lista los nombres y las ventas anuales hasta la fecha para cada vendedor de la base de datos. También muestra datos referentes a las oficinas de ventas y el número de oficina en donde cada persona trabaja. En este caso, los datos provienen de la tabla REPVENTAS:

Figura 2.1 Una sencilla base de datos relacional.

```

SELECT NOMBRE, OFICINA REP, VENTAS, CUOTA
FROM REVENTAS
      ----- -----
NOMBRE      OFICINA REP    VENTAS      CUOTA
----- -----
Bill Adams   13   $367,911.00  $350,000.00
Mary Jones   11   $392,725.00  $300,000.00
Sue Smith   21   $474,050.00  $350,000.00
Sam Clark    11   $299,912.00  $275,000.00
Bob Smith   12   $142,594.00  $200,000.00
Dan Roberts 12   $305,673.00  $300,000.00
Tom Snyder   NULL  $75,985.00  NULL
Larry Fitch  21   $361,865.00  $350,000.00
Paul Cruz    12   $286,775.00  $275,000.00
Nancy Angelli 22   $186,042.00  $300,000.00

```

SQL también permite solicitar resultados calculados. Por ejemplo, se puede solicitar a SQL que calcule la cuantía en la cual cada vendedor está por encima o por debajo de su cuota:

```

SELECT NOMBRE, VENTAS, CUOTA, (VENTAS-CUOTA)
FROM REVENTAS
      ----- -----
NOMBRE      VENTAS      CUOTA      (VENTAS-CUOTA)
----- -----
Bill Adams   $367,911.00  $350,000.00  $17,911.00
Mary Jones   $392,725.00  $300,000.00  $92,725.00
Sue Smith   $474,050.00  $350,000.00  $124,050.00
Sam Clark    $299,912.00  $275,000.00  $24,912.00
Bob Smith   $142,594.00  $200,000.00  -$57,406.00
Dan Roberts $305,673.00  $300,000.00  $5,673.00
Tom Snyder   $75,985.00  NULL        NULL
Larry Fitch  $361,865.00  $350,000.00  $11,865.00
Paul Cruz    $286,775.00  $275,000.00  $11,775.00
Nancy Angelli $186,042.00  $300,000.00  $113,958.00

```

Los datos solicitados (incluyendo la diferencia calculada entre ventas y cuota para cada vendedor) aparecen una vez más en la fila/columna de una tabla. Quizás usted preferiría centrarse en el vendedor cuyas ventas son inferiores a sus cuotas. SQL permite recuperar esta clase de información selectiva muy fácilmente, añadiendo una comparación matemática a la petición previa:

```

SELECT NOMBRE, VENTAS, CUOTA, (VENTAS-CUOTA)
FROM REVENTAS
WHERE VENTAS < CUOTA
      ----- -----
NOMBRE      VENTAS      CUOTA      (VENTAS-CUOTA)
----- -----
Bob Smith   $142,594.00  $200,000.00  -$57,406.00
Nancy Angelli $186,042.00  $300,000.00  -$113,958.00

```

La misma técnica puede ser utilizada para listar grandes pedidos en la base de datos y para hallar qué cliente remitió el pedido, qué producto fue pedido y en qué cantidad. También se puede pedir a SQL que ordene los pedidos en base a su importe:

```

SELECT NUM_PEDIDO, CLIE, PRODUCTO, CANT, IMPORTE
FROM PEDIDOS
WHERE IMPORTE > 25000.00
ORDER BY IMPORTE
      ----- -----
NUM_PEDIDO  CLIE  PRODUCTO  CANT  IMPORTE
----- -----
112887     2103  4100Y    11   $27,500.00
113069     2109  775C     22   $31,350.00
112961     2117  2A44L    7    $31,500.00
113045     2112  2A44R    10   $45,000.00

```

Sumarización de datos

SQL no solamente recupera datos de la base de datos, también puede ser utilizado para sumarizar los contenidos de la base de datos. ¿Cuál es el tamaño medio de un pedido en la base de datos? Esta petición solicita a SQL que examine todos los pedidos y encuentre el importe medio:

```

SELECT AVG(IMPORTE)
FROM PEDIDOS
      ----- -----
AVG(IMPORTE)
----- -----
$8,256.37

```

También puede solicitársele el importe medio de todos los pedidos remitidos por un cliente particular:

```

SELECT AVG(IMPORTE)
FROM PEDIDOS
WHERE CLIE = 2103
      ----- -----
AVG(IMPORTE)
----- -----
$8,895.50

```

Finalmente, hallemos el importe total de los pedidos remitidos por cada cliente.

para hacer esto, se puede solicitar a SQL que agrupe los pedidos por número de cliente, y que luego totalice los pedidos de cada cliente:

```
SELECT CLIE, SUM(CImporte)
  FROM PEDIDOS
 GROUP BY CLIE
CLIE  SUM(CImporte)
-----+
2101   $1,458.00
2102   $3,978.00
2103   $35,582.00
2106   $4,026.00
2107   $23,132.00
2108   $7,255.00
2109   $31,350.00
2111   $6,445.00
2112   $47,925.00
2113   $22,500.00
2114   $22,100.00
2117   $31,500.00
2118   $3,608.00
2120   $3,750.00
2124   $3,082.00
```

Supresión de datos

Del mismo modo que la sentencia INSERT de SQL añade nuevos datos a la base de datos, la sentencia DELETE suprime datos de la base de datos. Si Acme Industries decide unos pocos días más tarde pasarse a la competencia, usted puede eliminarla de la base de datos con esta sentencia:

```
DELETE FROM CLIENTES
 WHERE EMPRESA = 'Acme Industries'
 1 fila suprimida.
```

Y si usted decide despedir a todos los vendedores cuyas ventas sean inferiores a sus cuotas, puede suprimirlos de la base de datos con esta sentencia DELETE:

```
DELETE FROM REPVENTAS
 WHERE VENTAS < CUOTA
 2 filas suprimidas.
```

Actualización de la base de datos

El lenguaje SQL se utiliza también para modificar datos que ya están almacenados en la base de datos. Por ejemplo, para incrementar el límite de crédito de First Corp. a \$75,000 se utilizaría la sentencia UPDATE:

```
UPDATE CLIENTES
 SET LIMITE_CREDITO = 75000.00
 WHERE EMPRESA = 'First Corp.'
 1 fila actualizada.
```

La sentencia UPDATE también puede efectuar muchas modificaciones en la base de datos de una vez. Por ejemplo, esta sentencia UPDATE eleva la cuota de todos los vendedores en \$15,000:

```
UPDATE REPVENTAS
 SET CUOTA = CUOTA + 15000.00
 8 filas actualizadas.
```

Protección de datos

Un papel importante de una base de datos es proteger los datos almacenados frente al acceso por parte de usuarios no autorizados. Por ejemplo, supongamos

que su secretaria, llamada Mary, no estuviera previamente autorizada a insertar datos referentes a nuevos clientes en la base de datos. Esta sentencia SQL le concede ese permiso:

```
GRANT INSERT  
ON CLIENTES  
TO MARY
```

Privilegio concedido.

Análogamente, la siguiente sentencia SQL da a Mary permiso para actualizar datos referentes a los clientes, y para recuperar datos de clientes con la sentencia SELECT:

```
GRANT UPDATE, SELECT  
ON CLIENTES  
TO MARY
```

Privilegio concedido.

Si a Mary ya no se le permite añadir nuevos clientes a la base de datos, esta sentencia REVOKE lo impedirá:

```
REVOKE INSERT  
ON CLIENTES  
FROM MARY
```

Privilegio revocado.

Análogamente, esta sentencia REVOKE revocará los privilegios de Mary impidiéndole acceder a los datos de los clientes bajo ningún concepto:

```
REVOKE ALL  
ON CLIENTES  
FROM MARY
```

Privilegio revocado.

Creación de una base de datos

Antes de que usted pueda almacenar datos en una base de datos, debe primero definir la estructura de los datos. Supongamos que desea ampliar la base de datos ejemplo añadiendo una tabla de datos referente a los productos vendidos por su empresa. Por cada producto, los datos a almacenar incluyen:

- un código ID del fabricante de tres caracteres,

- un código ID del producto de cinco caracteres,
- una descripción de hasta treinta caracteres,
- el precio del producto, y
- la cantidad actualmente disponible.

Esta sentencia de SQL, CREATE TABLE, define una nueva tabla para almacenar los datos de los productos:

```
CREATE TABLE PRODUCTOS  
(ID_FAB CHAR(3),  
ID_PRODUCTO CHAR(5),  
DESCRIPCION VARCHAR(20),  
PRECIO MONEY,  
EXISTENCIAS INTEGER)
```

Tabla creada.

Aunque más críptica que las sentencias de SQL anteriores, la sentencia CREATE TABLE sigue siendo bastante sencilla. Asigna el nombre "PRODUCTOS" a la nueva tabla y especifica el nombre y tipo de los datos almacenados en cada una de sus cinco columnas.

Una vez que la tabla ha sido creada, usted puede rellenarla con datos. He aquí una sentencia INSERT para un nuevo envío de 250 baratijas de tamaño 7 (producto ACI-41007), que cuestan \$225,00 la pieza:

```
INSERT INTO PRODUCTOS (ID_FAB, ID_PRODUCTO, DESCRIPCION, PRECIO, EXISTENCIAS)  
VALUES ('ACI', '41007', 'Artículo Tipo 7', 225.00, 250)
```

1 fila insertada.

Finalmente, si usted descubre posteriormente que ya no necesita almacenar datos de los productos en la base de datos, puede eliminar la tabla (y todos los datos que contiene) con la sentencia DROP TABLE:

```
DROP TABLE PRODUCTOS  
Tabla eliminada.
```

Resumen

Este rápido repaso de SQL ha demostrado lo que SQL puede hacer y ha ilustrado el estilo del lenguaje SQL, utilizando ocho de las sentencias más comúnmente utilizadas en SQL. Para resumir:

- SQL se utiliza para *recuperar* datos de la base de datos, utilizando la sentencia `SELECT`. Se pueden recuperar todos o parte de los datos almacenados, ordenarlos y solicitar a SQL que sumarice los datos, utilizando totales y promedios.
- SQL se utiliza para *actualizar* la base de datos, añadiendo nuevos datos con la sentencia `INSERT`, suprimiendo datos con la sentencia `DELETE` y modificando datos existentes con la sentencia `UPDATE`.
- SQL se utiliza para *controlar el acceso* a la base de datos, concediendo y revocando privilegios específicos a usuarios específicos.
- SQL se utiliza para *crear* la base de datos definiendo la estructura de nuevas tablas y borrando tablas cuando ya no se vayan a necesitar.

3 SQL en perspectiva

SQL está surgiendo rápidamente y a la vez como un lenguaje estándar de hecho y oficial para gestión de base de datos. ¿Qué significa para SQL ser estándar? ¿Qué papel juega SQL como lenguaje de base de datos? ¿Cómo se convirtió SQL en un estándar y qué impacto está teniendo el estándar SQL en ordenadores personales, redes, minicomputadores y mainframes? Para responder a estas cuestiones, este capítulo traza la historia de SQL y descubre su papel actual en el mercado informático.

SQL y la gestión de base de datos

Una de las principales tareas de un sistema informático es almacenar y gestionar datos. Para ocuparse de esta tarea, programas computadores especializados conocidos como sistemas de gestión de bases de datos comenzaron a aparecer a finales de los sesenta y comienzo de los setenta. Un sistema de gestión de base de datos, o DBMS (*database management system*), ayudaba a los usuarios del computador a organizar y estructurar sus datos, y permitía al sistema informático jugar un papel más activo en la gestión de los datos. Aunque los sistemas de gestión de bases de datos se desarrollaron inicialmente en grandes sistemas de mainframes, su popularidad se ha extendido a minicomputadores, computadores personales y estaciones de trabajo.

Hoy día, la gestión de base de datos es un gran negocio. Las compañías de software independientes y los vendedores de computadores facturan alrededor de 3.000 millones de dólares en productos de gestión de base de datos anualmente. Los expertos de la industria informática predicen que durante los próxi-

mos años los productos de bases de datos para maxi y minicomputadores contabilizarán entre 20 y 25 por 100 del mercado de base de datos y los computadores personales contabilizarán el 40 por 100 ó más. La gestión de base de datos afecta por tanto a todos los segmentos del mercado informático.

Durante los últimos años se ha popularizado un tipo específico de DBMS, llamado sistema de gestión de base de datos *relacional* (RDBMS). Las bases de datos relacionales organizan los datos en una forma tabular sencilla y proporcionan muchas ventajas sobre los tipos anteriores de bases de datos. SQL es específicamente un lenguaje relacional de base de datos utilizado para trabajar con bases de datos relacionales.

Breve historia de SQL

La historia del lenguaje SQL está íntimamente interrelacionada con el desarrollo de las bases de datos relacionales. La Tabla 3.1 muestra algunos de los hitos en sus veinte años de historia. El concepto de base de datos relacional fue desarrollado originalmente por el Dr. E. F. «Ted» Codd, un investigador de IBM. En junio de 1970, el Dr. Codd publicó un artículo titulado «Un modelo relacional de datos para grandes bancos de datos compartidos» que esquematizaba una teoría matemática de cómo los datos podrían ser almacenados y manipulados utilizando una estructura tabular. Las bases de datos y SQL tienen sus orígenes en este artículo que apareció en la revista científica *Communications of the Association for Computing Machinery*.

Los primeros años

El artículo desencadenó una racha de investigaciones en base de datos relacional, incluyendo un importante proyecto de investigación dentro de IBM. El objetivo del proyecto, llamado System/R, fue demostrar la operabilidad del concepto relacional y proporcionar alguna experiencia en la implementación efectiva de un DBMS relacional. El trabajo en el System/R comenzó a mediados de los setenta en los laboratorios Santa Teresa de IBM en San Jose, California. En 1974 y 1975 la primera fase del proyecto System/R produjo un mínimo prototipo de un DBMS relacional. Además del propio DBMS, el proyecto System/R incluía trabajos sobre lenguajes de consulta de bases de datos. Uno de estos lenguajes fue denominado SEQUEL, un acrónimo de Structured English Query Language (Lenguaje Estructurado de Consultas Inglés). En 1976 y 1977 el prototipo de investigación System/R fue reescrito desde el principio. La nueva implementación soportaba consultas multitable y permitía que varios usuarios compartieran el acceso a los datos.

La implementación de System/R fue distribuida a una serie de instalaciones de clientes de IBM para evaluación en 1978 y 1979. Estas primeras instalaciones de usuario proporcionaron cierta experiencia efectiva en el uso de System/R y de su lenguaje de base de datos, que había sido renombrada como SQL, o Structured Query Language (Lenguaje Estructurado de Consultas), por razones legales. A pesar del cambio de nombre, la pronunciación SEQUEL permaneció, y continúa hoy día. En 1979 el proyecto de investigación System/R llegó al final, e IBM concluyó que las bases de datos relacionales no solamente eran factibles, sino que podrían ser la base de un producto comercial útil.

Primeros productos relacionales

El proyecto System/R y su lenguaje de base de datos SQL recibieron buenos comentarios en las revistas técnicas durante los setenta. Seminarios en tecnología de base de datos celebraron debates sobre los méritos del nuevo y «herético» modelo relacional. Para 1976 era evidente que IBM se estaba entusiasmado con la tecnología de base de datos relacional, y que estaba adquiriendo un compromiso importante con el lenguaje SQL.

La publicidad referente al System/R atrajo la atención de un grupo de ingenieros en Menlo Park, California, que decidieron que la investigación de IBM presagiaba un mercado comercial para las bases de datos relacionales. En 1977 formaron una compañía llamada Relational Software, Inc., para construir un DBMS basado en SQL. El producto, de nombre Oracle, apareció en 1979, y se convirtió en el primer DBMS relacional comercialmente disponible. Oracle se adelantó comercialmente al primer producto de IBM en dos años completos, y se ejecutaba en minicomputadores VAX de Digital, que eran menos caros que

Fecha	Acontecimiento
1970	Codd define el modelo de base de datos relacional.
1974	Comienza el proyecto System/R de IBM.
1974	Primer artículo que describe el lenguaje SEQUEL.
1978	Test de clientes del System/R.
1979	Oracle introduce el primer RDBMS comercial.
1981	Relational Technology introduce Ingres.
1981	IBM anuncia SQL/DS.
1982	ANSI forma el comité de estándares SQL.
1983	IBM anuncia DB2.
1986	Se ratifica el estándar ANSI SQL.
1986	Sybase introduce RDBMS para procesamiento de transacciones.
1987	Se ratifica el estándar ISO SQL.
1988	Ashton-Tate y Microsoft anuncian SQL Server para OS/2
1988	IBM anuncia la versión 2 de DB2.
1989	Primera entrega de servidores de bases de datos SQL para OS/2

Tabla 3.1. Hitos en el desarrollo de SQL.

los mainframe de IBM. Hoy la empresa, renombrada Oracle Corporation, es un vendedor importante de sistemas de gestión de bases de datos relacional, con ventas anuales que se aproximan a los 1.000 millones de dólares.

Profesores en los laboratorios informáticos de Berkeley de la Universidad de California estuvieron también investigando las bases de datos relacionales a mediados de los setenta. Al igual que el equipo investigador de IBM, también ellos construyeron un prototipo de un DBMS relacional, y llamaron a su sistema Ingres. El proyecto Ingres incluía un lenguaje de consulta llamado QUEL que, aunque más «estructurado» que SQL, era menos parecido al inglés. Muchos de los expertos de base de datos actuales comenzaron a interesarse en las bases de datos relacionales a consecuencia del proyecto Ingres de Berkeley, incluyendo entre ellos a los fundadores de Britton-Lee, un suministrador de máquinas de bases de datos, y Sybase, el creador del producto SQL Server de Ashton-Tate/Microsoft.

En 1980 varios profesores abandonaron Berkeley y fundaron Relational Technology, Inc. para construir una versión comercial de Ingres, anunciada en 1981. Relational Technology, renombrada Ingres Corporation en 1989, sigue siendo un importante vendedor de DBMS relacional, con ventas anuales de más de 150 millones de dólares. El lenguaje de consulta original QUEL fue efectivamente sustituido por SQL en 1986, testimonio de la potencia en el mercado del estándar SQL.

Productos IBM

Con Oracle e Ingres en la carrera por convertirse en productos comerciales, el proyecto System/R de IBM se había transformado también en un esfuerzo por construir un producto comercial, llamado SQL/Data System (SQL/DS). IBM anunció el SQL/DS en 1981, y comenzó a entregar el producto en 1982. En 1983 IBM anunció una versión de SQL/DS para VM/CMS, un sistema operativo que se utiliza frecuentemente en mainframe IBM en aplicaciones corporativas de «centros de información».

En 1983 IBM introdujo también Database 2 (DB2), otro DBMS relacional para sus sistemas mainframe. DB2 operaba bajo el sistema operativo MVS de IBM, el sistema operativo principal utilizado en los centros de datos de grandes mainframe. La primera revisión de DB2 comenzó a entregarse en 1985, y los directivos de IBM lo celebraron como una pieza estratégica en la tecnología software de IBM. DB2 se ha convertido desde entonces en el principal producto DBMS relacional de IBM, y con el peso de IBM detrás de él, el lenguaje SQL de DB2 se ha convertido de hecho en el estándar entre los lenguajes de bases de datos.

Aceptación comercial

Durante la primera mitad de los ochenta, los vendedores de bases de datos relacionales lucharon por la aceptación comercial de sus productos. Los productores relacionales tenían varias desventajas cuando se comparaban con las arquitecturas tradicionales de bases de datos. El rendimiento de las bases de datos relacionales era inferior al de las bases de datos tradicionales. Excepto en los productos IBM, las bases de datos relacionales provenían de pequeños vendedores «novatos». Y, excepto los productos IBM, las bases de datos relacionales tendían a ejecutarse en minicomputadores en lugar de en mainframe.

Sin embargo, los productos relacionales tenían ciertamente una ventaja importante. Sus lenguajes de consulta relacional (SQL, QUEL y otros) permitían a los usuarios realizar consultas ad hoc a la base de datos, y obtener respuestas inmediatas, sin escribir programas. Como resultado, las bases de datos relacionales comenzaron lentamente a introducirse en las aplicaciones de centros de información como herramientas de soporte de decisiones. En mayo de 1985 Oracle proclamó orgullosoamente disponer de «más de 1.000» instalaciones. Ingres fue instalado en un número comparable de localizaciones. DB2 y SQL/DS también estaban siendo lentamente aceptados, y contabilizaban sus instalaciones combinadas ligeramente por encima de las 1.000 localizaciones.

Durante la última mitad de los ochenta, SQL y las bases de datos relacionales fueron rápidamente aceptadas como la tecnología de base de datos del futuro. El rendimiento de los productos de bases de datos relacionales mejoró enormemente. Ingres y Oracle, en particular, mejoraron sustancialmente con cada nueva versión, reclamando superioridad sobre el competidor y el aumento de dos o tres veces el rendimiento de la revisión anterior. Las mejoras en la potencia de procesamiento del hardware informático subyacente también ayudaban a reforzar el rendimiento.

Las fuerzas del mercado también acrecentaban la popularidad de SQL a finales de los ochenta. IBM aumentaba la defensa y divulgación de SQL, posicionando DB2 como la solución de gestión de datos para los noventa. La publicación del estándar ANSI/ISO para SQL en 1986, dio estatus «oficial» a SQL como estándar. SQL también emergió como un estándar para sistemas informáticos basados en UNIX, cuya popularidad se aceleró en los ochenta. Finalmente, al hacerse los computadores personales más potentes y conectarse en redes de área local, comenzaron a necesitar una gestión de base de datos más sofisticada. Los vendedores de base de datos para PC adoptaron SQL como la solución a sus necesidades. La Tabla 3.2 muestra algunos de los sistemas de gestión de base de datos basados en SQL más populares sobre diferentes tipos de sistemas de computadores.

Cuando la industria informática entró en los noventa, las instalaciones SQL se contaban por cientos de miles. SQL estaba claramente establecido como el

lenguaje estándar de base de datos. Los vendedores de base de datos que aún no soportaban SQL se están preocupando de hacerlo, y cualquier nuevo producto de base de datos requería a SQL como reclamo para ser tomado en serio. SQL se había convertido de hecho en el estándar oficial para las bases de datos relacionales.

Estándares SQL

Uno de los desarrollos más importantes en la aceptación del mercado de SQL es el surgimiento de los estándares SQL. Las referencias al «estándar SQL» significa generalmente el estándar oficial adoptado por la American National Standards Institute (ANSI) y la International Standards Organization (ISO). Sin embargo, existen otros dos estándares SQL importantes que incluyen el SQL definido por la Systems Application Architecture (SAA) de IBM y el estándar X/OPEN para SQL bajo UNIX.

DMBS Sistemas informáticos

DMBS	Sistemas informáticos
DB2	Maxicomputadores IBM bajo MVS.
SQL/DS	Maxicomputadores IBM bajo VM y DOS/VSE.
Rdb/VMS	Minicomputadores VAX/VMS de Digital.
Oracle	Maxicomputadores, minicomputadores y PCs.
Ingres	Minicomputadores y PCs.
Sybase	Minicomputadores y LANs.
Informix-SQL	Minicomputadores y PCs basados en UNIX.
Unify	Sistemas PS/2 de IBM bajo OS/2.
OS/2 Extended Edition	PC LANs basados en OS/2.
SQL Server	PC LANs basados en DOS y OS/2.
SQL/Base	PC LANs y PC LANs.
dBASE IV	PC LANs y PC LANs.

Tabla 3.2. Principales sistemas de gestión de base de datos basados en SQL.

El estándar ANSI/ISO

El trabajo en el estándar SQL oficial comenzó en 1982, cuando ANSI encargó a su comité X3H2 que definiera un lenguaje de base de datos relacional. Al principio el comité debatió los méritos de los diferentes lenguajes de base de datos propuestos. Sin embargo, cuando el compromiso de IBM con SQL se incrementó y SQL emergió como el estándar de hecho en el mercado, el comité

seleccionó SQL como su lenguaje de base de datos relacional, y se aplicó a estandarizarlo.

El estándar ANSI para SQL resultante está basado en gran medida en el SQL de DB2, aunque contiene algunas diferencias importantes con respecto a él. Después de varias revisiones, el estándar fue oficialmente adoptado como estándar ANSI X3.135 en 1986, y como estándar ISO en 1987. El estándar ANSI/ISO ha sido adoptado desde entonces como estándar del Federal Information Processing Standard (FIPS) por el gobierno de los Estados Unidos.

Muchos de los miembros del comité de estándares ANSI e ISO eran representantes de vendedores de bases de datos que tenían productos SQL existentes, cada uno implementando un dialecto SQL ligeramente diferente. Al igual que los dialectos de los lenguajes humanos, los dialectos SQL eran generalmente muy similares los unos a los otros, pero incompatibles en sus detalles. En muchas áreas el comité simplemente esquivó esas diferencias omitiendo algunas partes del lenguaje del estándar y especificando otras como «definidas por el constructor». Estas decisiones permitieron que las implementaciones SQL existentes reclamaran una amplia adherencia al estándar ANSI/ISO resultante, pero hizo el estándar relativamente débil.

A pesar de la existencia de un estándar, ningún producto SQL comercial disponible hoy se conforma exactamente a él, y no hay dos productos comerciales SQL que soporten exactamente el mismo dialecto de SQL. Además, como los vendedores de base de datos introducen nuevas capacidades, amplían sus dialectos SQL y se apartan aún más del estándar. El núcleo central del lenguaje SQL muestra algunos signos de normalizarse cada vez más, sin embargo. Cuando pueden hacerlo sin dañar a los clientes o a las características ya existentes, los vendedores llevan lentamente sus productos de conformidad con el estándar ANSI/ISO. Por ejemplo, las revisiones más recientes de DB2 permiten las sintaxis alternativas de varias palabras clave para que se conformen específicamente con el estándar.

Para tratar las insuficiencias del estándar original, los comités ANSI e ISO han continuado su trabajo, y han estado circulando trabajos para un nuevo y más riguroso estándar SQL2. A diferencia del estándar actual, las propuestas de SQL2 especifican claramente las características considerablemente más avanzadas de las que se encuentran en productos SQL comerciales actuales. Incluso se han propuesto cambios importantes para un estándar SQL3. Como resultado, los estándares propuestos SQL2 y SQL3 son bastante más controvertidos que el estándar SQL inicial.

El estándar SQL «real», naturalmente, es el SQL implementado en los productos que están ampliamente aceptados por el mercado. Si este estándar provendrá de los comités estándares oficiales, de la potencia de mercado de IBM o de los esfuerzos empresariales de los vendedores de base de datos independientes está aún por ver.

Otros estándares SQL

Aunque es el más ampliamente reconocido, el estándar ANSI/ISO no es el único estándar para SQL. X/Open, un grupo de vendedores europeos, ha adoptado también SQL como parte de su grupo de estándares para un «entorno de aplicaciones portables» basado en UNIX. Los estándares X/Open juegan un papel principal en el mercado informático europeo, donde la portabilidad entre sistemas informáticos de diferentes vendedores es una cuestión esencial. Desgraciadamente, el estándar X/Open difiere del estándar ANSI/ISO en varios aspectos.

IBM ha incluido también SQL en la especificación de su Systems Application Architecture, prometiendo que todos sus productos SQL se trasladarán eventualmente a este dialecto SAA SQL. Finalmente, DB2, como insignia DBMS relacional de IBM, ejerce una fuerte influencia como estándar para SQL. Donde el estándar ANSI/ISO guarda silencio, o donde hay diferencias entre los estándares y la implementación DB2 de SQL, es en el dialecto DB2 el que con más frecuencia copian el resto de vendedores.

El mito de la portabilidad

La existencia de un estándar SQL publicado ha generado unas cuantas aseveraciones exageradas acerca de la portabilidad de SQL y sus aplicaciones. Diagramas tales como el de la Figura 3.1 muestran cómo una aplicación que utiliza

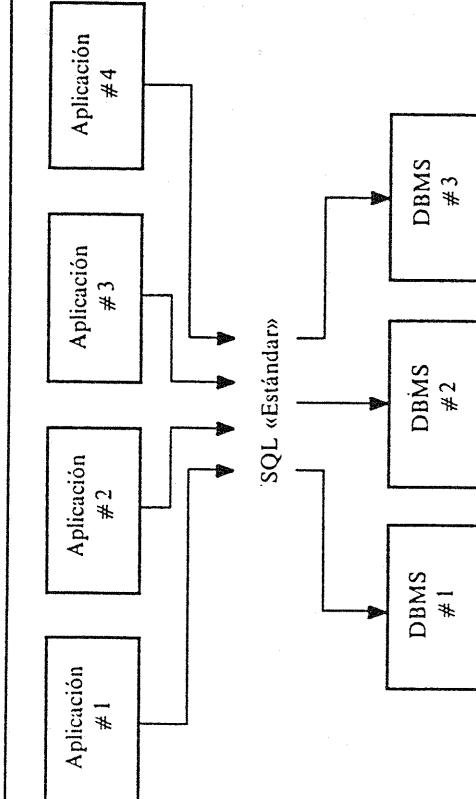


Figura 3.1. El mito de la portabilidad de SQL.

SQL puede funcionar indistintamente con cualquier sistema de gestión de base de datos basado en SQL. De hecho, las insuficiencias en el estándar SQL ANSI/ISO y las diferencias entre los dialectos SQL son suficientemente significativas para que una aplicación deba ser *siempre* modificada cuando se pasa de una base de datos SQL a otra. Estas diferencias incluyen:

- **Códigos de error.** El estándar no especifica los códigos de error a devolver cuando SQL detecta un error, y todas las implementaciones comerciales utilizan su propio conjunto de códigos de error.
- **Tipos de datos.** El estándar define un conjunto mínimo de tipos de datos, pero omite alguno de los tipos más populares y útiles, tales como las cadenas de caracteres de longitud variables, las fechas y horas y los datos monetarios.
- **Tablas de sistema.** El estándar no dice nada acerca de las tablas de sistema que proporcionan información referente a la estructura de la propia base de datos. Cada vendedor tiene su propia estructura para estas tablas, e incluso las cuatro implementaciones SQL de IBM se diferencian unas de otras.
- **SQL interactivo.** El estándar solamente especifica el *SQL programado* utilizado por un programa de aplicación, y no el SQL interactivo. Por ejemplo, la sentencia SELECT utilizada para consultar la base de datos en SQL interactivo está ausente del estándar.
- **Interfaz de programa.** El estándar especifica una técnica abstracta para utilizar SQL desde dentro de un programa de aplicaciones escrito en COBOL, C, FORTRAN y otros lenguajes de programación. Ningún producto SQL comercial utiliza esta técnica, y hay variaciones considerables en los interfaces reales de programas utilizados.
- **SQL dinámico.** El estándar no incluye las características requeridas para desarrollar frontales de bases de datos de propósito general, tales como herramientas de consulta y escritores de informes. Estas características, conocidas como *SQL dinámico*, se encuentran en virtualmente todos los sistemas de bases de datos SQL, pero varían significativamente de un producto a otro.
- **Diferencias semánticas.** Debido a que el estándar especifica ciertos detalles como «definidos por el fabricante», es posible ejecutar la misma consulta con dos implementaciones SQL diferentes, y producir dos conjuntos diferentes de resultados. Estas diferencias ocurren en el manejo de los valores NULL, las funciones de columna y la eliminación de filas duplicadas.
- **Secuencias de cotejo.** El estándar no define la secuencia de cotejo (ordenación) de los caracteres almacenados en la base de datos. Los resultados de una consulta ordenada serán diferentes si la consulta se ejecuta en un computador personal (con caracteres ASCII) o en un mainframe (con caracteres EBCDIC).

Estructura de base de datos. El estándar especifica el lenguaje SQL a utilizar una vez que una base de datos particular ha sido abierta y está lista para procesar. Los detalles de la denominación de la base de datos y de cómo se establece la conexión inicial con la base de datos varían ampliamente y no son portables.

A pesar de estas diferencias, en 1989 comenzaron a emerger herramientas de base de datos comerciales que proclaman su portabilidad a través de diferentes productos de bases de datos SQL. En todo caso, sin embargo, las herramientas requieren un adaptador especial para cada DBMS soportado, que genera el dialecto SQL adecuado, maneja la conversión de tipos de datos, traduce códigos de errores, etc. El día de la portabilidad transparente a través de diferentes productos DBMS basados en el SQL estándar aún no ha llegado.

SQL y la conexión por red

La creciente popularidad de la conexión de computadores por red durante los últimos años ha tenido un fuerte impacto en la gestión de base de datos y dado a SQL una nueva prominencia. Conforme las redes pasan a ser más comunes, las aplicaciones que han corrido tradicionalmente en un minicomputador o maxicomputador central se están transfiriendo a redes de área local con estaciones de trabajo de sobremesa y servidores. En estas redes SQL juega un papel crucial como vínculo entre una aplicación que corre en una estación de trabajo y el DBMS que gestiona los datos compartidos en el servidor.

Arquitectura centralizada

La arquitectura de base de datos tradicional utilizada por DB2, SQL/DS y las bases de datos sobre minicomputadores tales como Oracle e Ingres se muestra en la Figura 3.2. En esta arquitectura el DBMS y los datos físicos residen ambos

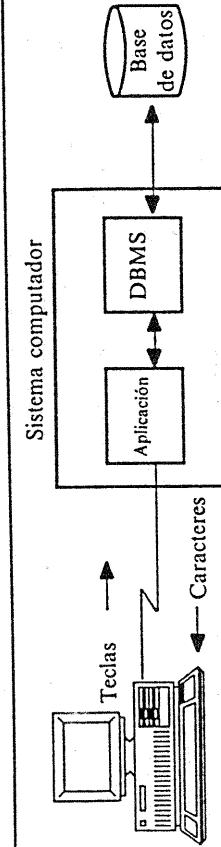


Figura 3.2. Gestión de base de datos en una arquitectura centralizada.

en un sistema mainframe o minicomputador central, junto con el programa de aplicación que acepta entradas desde el terminal de usuario y muestra los datos en la pantalla del usuario.

Supongamos que el usuario teclea una consulta que requiere una búsqueda secuencial en la base de datos, tal como petición de hallar la cantidad media de mercancías de todos los pedidos. El DBMS recibe la consulta, explora la base de datos para acceder a cada uno de los registros de datos del disco, calcula el promedio y muestra el resultado en la pantalla del terminal. Tanto el procesamiento de la aplicación como el procesamiento de la base de datos se producen en el computador central, y como el sistema es compartido por muchos usuarios, cada usuario experimenta una degradación del rendimiento cuando el sistema tiene una carga fuerte.

Arquitectura de servidor de archivos

La introducción de los computadores personales y de las redes de área local condujo al desarrollo de la arquitectura *servidora de archivos*, mostrada en la Figura 3.3. En esta arquitectura, una aplicación que corre en un computador personal puede acceder de forma transparente a datos localizados en un servidor de archivos, que almacena los archivos compartidos. Cuando una aplicación en PC solicita datos de un archivo compartido, el software de red recupera automáticamente los datos y los devolverá a la aplicación.

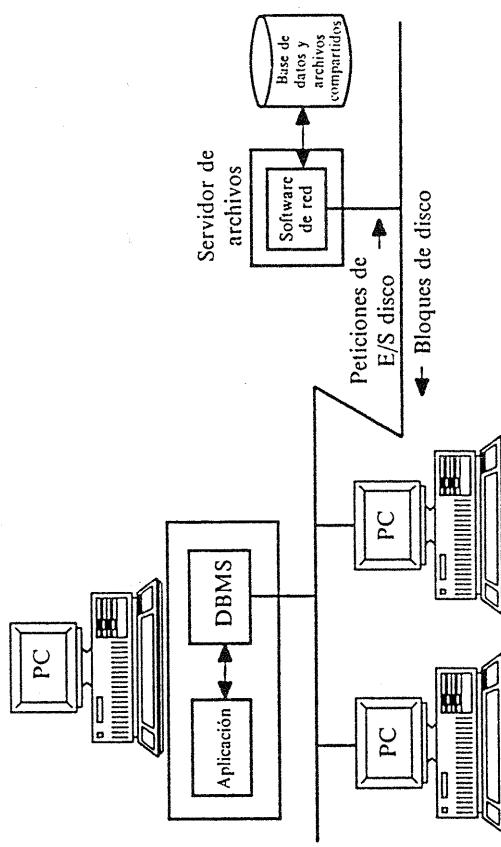


Figura 3.3. Gestión de base de datos en una arquitectura servidora de archivos.

máticamente el bloque solicitado del archivo en el servidor. Varias bases de datos PC populares, entre las que se incluyen dBASE III, R:BASE y Paradox, soportan esta estrategia servidora de archivos, donde cada computador personal ejecuta su propia copia del software DBMS.

Para consultas típicas esta arquitectura proporciona un rendimiento excelente, ya que cada usuario dispone de la potencia completa de un computador personal ejecutando su propia copia del DBMS. Sin embargo, consideremos la consulta efectuada en el ejemplo anterior. Puesto que la consulta requiere una exploración secuencial de la base de datos, el DBMS solicita repentinamente bloques de datos de la base de datos, la cual está localizada físicamente a través de la red en el servidor. Eventualmente *todos* los bloques del archivo están solicitados y enviados a través de la red. Obviamente esta arquitectura produce un fuerte tráfico de red y un bajo rendimiento para consultas de este tipo.

Arquitectura cliente/servidor

La Figura 3.4 muestra la emergente arquitectura *cliente/servidor* para gestión de bases de datos. En esta arquitectura, los computadores personales están combinados en una red de área local junto con un *servidor de base de datos* que almacena las bases de datos compartidas. Las funciones del DBMS están divididas en dos partes. Los «front-ends» (*frontales*) de base de datos, tales como

herramientas de consulta interactiva, escritores de informe y programas de aplicación, se ejecutan en el computador personal. La máquina de soporte (*back-end*) de la base de datos que almacena y gestiona los datos se ejecuta en el servidor. SQL se ha convertido en el lenguaje de base de datos estándar para comunicación entre las herramientas frontales y la máquina de soporte en esta arquitectura.

Consideremos una vez más la consulta que solicita el tamaño medio de los pedidos. En la arquitectura cliente/servidor, la consulta viaja a través de la red hasta el servidor de base de datos como una petición SQL. La máquina de base de datos en el servidor procesa la petición y explora la base de datos, que también reside en el servidor. Cuando calcula el resultado, la máquina de base de datos envía de vuelta a través de la red una única contestación a la petición inicial, y la aplicación frontal la muestra en pantalla del PC.

La arquitectura cliente/servidor reduce el tráfico de red y divide la carga de trabajo de la base de datos. Las funciones de intensiva relación con el usuario, tales como el manejo de la entrada y la visualización de los datos, se concentran en el PC. Las funciones intensivas en proceso de datos, tales como la entrada/salida de archivos y el procesamiento de consultas, se concentran en el servidor de la base de datos. Lo que es más importante, el lenguaje SQL proporciona una interfaz bien definida entre los sistemas frontales y de soporte, comunicando las peticiones de acceso a la base de datos de una manera eficiente.

La arquitectura cliente/servidor ha recibido gran atención con la introducción de las redes PC basadas en OS/2, SQL Server, el Servidor Oracle para

El impacto de SQL

Como emergente estándar de las bases de datos relacionales, SQL ha tenido un fuerte impacto en todas las áreas del mercado informático. IBM ha adoptado SQL como tecnología unificadora de bases de datos para su línea de productos. Todos los vendedores de minicomputadores ofrecen bases de datos basadas en SQL, y las bases de datos basadas en SQL dominan el mercado de los sistemas informáticos basados en UNIX. SQL está también influyendo en el mercado de las bases de datos sobre computadores personales a la vez que los computadores personales y a OS/2. SQL está emergiendo incluso como tecnología para procesamiento de transacciones en línea, refutando la creencia convencional de que las bases de datos relacionales nunca ofrecerían rendimiento suficiente para las aplicaciones de procesamiento de transacciones.

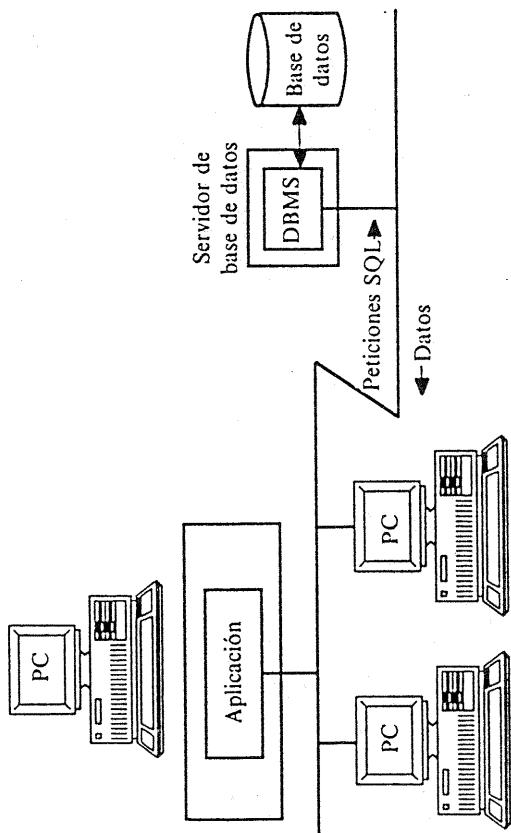


Figura 3.4. Gestión de base de datos en una arquitectura cliente/servidor.

SQL y la SAA de IBM

SQL juega un papel importante como lenguaje de acceso de base de datos en la Systems Application Architecture de IBM. Anunciada en marzo de 1987, la SAA es el plan maestro de IBM para proporcionar compatibilidad entre sus cuatro familias de computadores incompatibles. La SAA especifica cuatro categorías de compatibilidad:

- **Acceso común de usuario.** Los elementos del interfaz de usuario, incluyendo menús, visualizaciones y órdenes, serán normalizados de acuerdo con un conjunto de normas de interfaz de usuario.
- **Interfaz de programación común.** Los lenguajes e interfaces utilizados por los programadores de aplicación serán normalizados para todos los sistemas.
- **Sopor te común de comunicaciones.** Los protocolos de comunicación utilizados en todos los sistemas serán compatibles, y se utilizará un interfaz común para enviar y recibir mensajes.
- **Programas comunes de aplicación.** Aplicaciones selectas de IBM, tales como el paquete Office Vision de herramientas de automatización de oficinas, estarán disponibles en todos los sistemas.

El interfaz común de programación incluye los lenguajes de programación COBOL, FORTRAN, C y SQL como lenguaje de base de datos incorporado. IBM ha asegurado que su producto estratégico de base de datos en cada una de las cuatro familias de sistemas evolucionará para soportar el SQL de SAA. Los productos estratégicos son:

- **DB2.** El gestor de base de datos relacional insignia de IBM es el SQL de SAA estándar para los maxicomputadores IBM que corren MVS.
- **SQL/DS.** SQL/DS es el sistema de base de datos relacional de SAA para VM, otro sistema operativo de los maxicomputadores IBM.
- **SQL/400.** Esta implementación SQL para sistemas de gama media de IBM soporta la base de datos relacional interna de AS/400.
- **OS/2 Extended Edition.** Esta versión mejorada del sistema operativo estratégico para computadores personales de IBM incluye un gestor de base de datos interno basado en SQL.

Los cuatro productos se comercializan actualmente, pero el soporte SQL en cada uno de ellos difiere en ciertos aspectos de la definición del SQL de SAA. De los cuatro, el gestor de base de datos OS/2 Extended Edition es el que más se acerca al estándar SQL de SAA.

SQL en minicomputadores

Los minicomputadores fueron uno de los mercados inicialmente más fériles para los sistemas de bases de datos basados en SQL. Oracle e Ingres fueron comercializados originalmente en sistemas minicomputadores VAX/VMS de Digital. Aunque ambos productos se han portado desde entonces a muchas otras plataformas, VAX/VMS sigue siendo el corazón de su negocio. Sybase, un sistema de base de datos más reciente especializado en procesamiento de transacciones en línea, también se destinó al VAX como una de sus plataformas primarias.

Los vendedores de minicomputadores han desarrollado también sus propias bases de datos relacionales propietarias caracterizadas por SQL. Digital considera que las bases relacionales son tan importantes que está comercializando una versión en tiempo de ejecución de su base de datos Rdb/VMS. Hewlett-Packard ofrece Allbase, una base de datos que soporta su dialecto HPSQL y un interfaz no relacional. La base de datos DG/SQL de Data General ha reemplazado a sus bases de datos más antiguas no relacionales como herramienta estratégica de gestión de datos de DG. Además, muchos de los vendedores de minicomputadores revenden bases de datos relativacionales procedentes de vendedores de software de base de datos independientes. Por ejemplo, Digital oferta Ingres para su línea de productos VAX basados en UNIX, y Oracle está disponible en varias docenas de vendedores de minicomputadores.

SQL en sistemas UNIX

SQL se ha establecido firmemente como solución de gestión de datos a elegir para sistemas informáticos basados en UNIX. Originalmente desarrollado en Bell Laboratories, UNIX se popularizó en los ochenta como sistema operativo estándar independiente de los vendedores. Corre en un amplio rango de sistemas informáticos, desde estaciones de trabajo a mainframes y se ha convertido en el sistema operativo estándar para aplicaciones científicas y técnicas.

A comienzo de los ochenta había realmente cuatro bases de datos importantes disponibles para sistemas UNIX. Dos de ellas, Ingres y Oracle, eran versiones UNIX de los productos que corrían en los minicomputadores propietarios de DEC. Las otras dos, Informix y Unify, fueron escritas específicamente para UNIX. Ninguna de ellas ofreció originalmente soporte SQL, pero en 1985 Unify ofrecía un lenguaje de consulta SQL e Informix había sido reescrito como Informix-SQL, con total soporte SQL. Los cuatro sistemas de base de datos están ahora disponibles en la mayoría de los sistemas UNIX importantes, con Sybase como quinta oferta disponible en sistemas seleccionados.

SQL y el procesamiento de transacciones

SQL y las bases de datos relacionales habían tenido históricamente muy poco impacto en las aplicaciones de procesamiento de transacciones en línea (OLTP). Al hacer énfasis en las consultas, las bases de datos relacionales se confinaron al soporte de decisiones y a aplicaciones en línea de bajo volumen, en las cuales su rendimiento más lento no era una desventaja. Para aplicaciones OLTP, donde cientos de usuarios necesitan acceso en línea a los datos y tiempos de respuesta por debajo del segundo, el Information Management System (IMS) no relacional de IBM reinaba como DBMS dominante.

En 1986 un nuevo vendedor DBMS, Sybase, introdujo una nueva base de datos basada en SQL diseñada especialmente para aplicaciones OLTP. El DBMS Sybase corría en minicomputadores VAX/VMS y en estaciones de trabajo Sun, y se centraba en obtener un rendimiento en línea máximo. Oracle Corporation y Relational Technology siguieron en breve con anuncios de que también ellos ofrecerían versiones OLTP de sus populares sistemas de base de datos Oracle e Ingres. En el mercado UNIX, Informix anunció una versión OLTP de su DBMS, llamada Informix-Turbo.

En abril de 1988 IBM supo subirse al tren del OLTP relacional con DB2 Versión 2, cuyos programas de prueba mostraban que la nueva versión operaba por encima de 250 transacciones por segundo en grandes maxicomputadores. IBM proclamó que el rendimiento de DB2 era ahora adecuado para casi todas las aplicaciones OLTP excepto las más exigentes, y animó a los clientes a considerarla como una serie alternativa a IMS. Los bancos de prueba de OLTP se han convertido ahora en una herramienta estándar de ventas para base de datos relacionales, a pesar de serias cuestiones acerca de lo adecuado de esos programas para medir efectivamente el rendimiento de aplicaciones reales.

La conveniencia de SQL para OLTP continúa mejorando, debido a los avances en la tecnología relacional y a la mayor potencia del hardware computador que conducen a tasas de transacciones cada vez más altas. Los clientes parecen considerar ahora seriamente a DB2 y a las bases de datos relacionales para su utilización en aplicaciones OLTP.

SQL en computadores personales

Las bases de datos han sido populares en computadores personales prácticamente desde la introducción del IBM PC. El producto dBASE de Ashton-Tate ha sido instalado en más de un millón de PCs basados en MS-DOS, y otros productos tales como R:BASE, PFS:FILE y Paradox han conseguido también éxito significativo. En el Macintosh, las bases de datos tales como 4th Dimension combinan la gestión de datos y un interfaz gráfico de usuario. Aunque estas bases de datos PC presentan con frecuencia los datos en forma tabular, les falta

la potencia completa de un DBMS relacional y un lenguaje de base de datos relacional tal como SQL.

SQL tuvo poco impacto en los computadores personales hasta finales de los ochenta. Para entonces ya eran comunes los PCs potentes que soportaban decenas o centenares de megabytes de almacenamiento en disco. Los usuarios se conectaban, además, mediante redes de área local y deseaban compartir bases de datos. En resumen, los PCs comenzaron a necesitar las características que SQL y las bases de datos relacionales podían proporcionarles.

Las primeras bases de datos basadas en SQL para computadores personales fueron versiones de los productos populares para minicomputadores que difícilmente se ajustaban a los computadores personales. Professional Oracle, anunciado en 1984, requería dos megabytes de memoria en un IBM PC, y Oracle para Macintosh, anunciado en 1988, tenía exigencias análogas. Una versión PC de Ingres, anunciada en 1987, entraba (aunque muy justamente) dentro del límite de 640KB de MS-DOS. Informix-SQL para MS-DOS fue anunciado en 1986, proporcionando una versión PC de la popular base de datos UNIX. También en 1986, Gupta Technologies, una empresa fundada por un ex-diretor de Oracle, anunció SQLBase, una base de datos para redes de área local PC. SQLBase estuvo entre los primeros productos PC en ofrecer una arquitectura cliente/servidor, un avance de los anuncios de bases de datos OS/2 por llegar.

El impacto real de SQL en los computadores personales comenzó con el anuncio de OS/2 por parte de IBM y Microsoft en abril de 1987. Además del producto estándar OS/2, IBM anunció un OS/2 Extended Edition (OS/2 EE) propietario con una base de datos SQL interna y soporte de comunicaciones. Con esta introducción, IBM afirmó de nuevo su fuerte compromiso con SQL, manifestando en efecto que SQL era tan importante que pertenecía al sistema operativo del computador.

OS/2 Extended Edition supuso un problema para Microsoft. Como fabricante y distribuidor de OS/2 estándar para otros fabricantes de computadores personales, Microsoft necesitaba una alternativa a Extended Edition. Microsoft respondió adquiriendo la licencia del DBMS Sybase, que había sido desarrollado para el VAX, y comenzó a portarlo al OS/2. En enero de 1988, en un acuerdo sorpresa, Microsoft y Ashton-Tate anunciaron que venderían conjuntamente el producto resultante basado en OS/2, renombrado SQL Server. Microsoft vendría SQL Server junto con OS/2 a los fabricantes de computadores; Ashton-Tate vendería el producto a través de sus canales de venta a usuarios de PC. En septiembre de 1989 Lotus Development añadió su apoyo al SQL Server invitando en Sybase. Posteriormente ese año, Ashton-Tate renunció a sus derechos de distribución exclusiva al por menor y vendió su inversión a Lotus.

La introducción de OS/2 Extended Edition y del SQL Server de Ashton-Tate/Microsoft centró la atención en el potencial de SQL sobre redes de área local basadas en OS/2. Los clientes comenzaron a examinar seriamente las redes de área local y la arquitectura cliente/servidor como alternativa a un minicomputador o maxicomputador centralizado para aplicaciones de bases de datos.

A finales de 1989 aparecieron una gran cantidad de anuncios de servidores de bases de datos OS/2, y cuatro de los productos emergieron como líderes en el mercado de servidores de bases de datos OS/2:

- *OS/2 Extended Edition.* Aunque su versión servidora aún no estaba comercializada, todos esperaban que el gestor de base de datos OS/2 de IBM jugara un papel importante en el mercado de servidores OS/2.
- *SQL Server de Microsoft.* Respaldado por dos de los tres principales desarrolladores de software PC, este servidor aparecía como el jugador independiente más fuerte. Tenía la ventaja de una buena tecnología (procedente de Sybase), publicidad excelente y una distribución establecida.
- *Oracle Server para OS/2 de Oracle.* La versión OS/2 del DBMS relacional independiente más importante reforzaba la buena conexión con las bases de datos Oracle sobre minicomputador y maxicomputador y con la potencia de la fuerza de ventas directas de Oracle.

■ *SQLBase de Gupta.* El primer vendedor con una base de datos en PC LAN fue el más pequeño de los cuatro y un contendiente oscuro, pero el servidor era real, tenía buenas herramientas frontales gráficas, estaba comercializándose (lo que no era necesariamente cierto de los otros tres) y funcionaba.

Además de estos cuatro, existían productos servidores de bases de datos de Novell (NetWare SQL), XDB Systems (XDB-Server) y otras varias compañías más pequeñas. La arquitectura cliente/servidor también estimuló el desarrollo de herramientas de bases de datos frontales que podrían funcionar con los nuevos servidores. Quizás el producto frontal más significativo fue dBASE IV, que Ashton-Tate anunció funcionaría como frontal de SQL Server y de OS/2 Extended Edition. Paradox y DataEase, otras dos bases de datos en PCs, también anunciaron frontales para tres o cuatro de los principales servidores. Cada vendedor de servidor también ofreció sus propios productos frontales, incluyendo herramientas de desarrollo de aplicaciones, consultas orientadas a ventanas y facilidades de entradas de datos, herramientas de inspección de la base de datos y otras.

El modelo servidor/cliente de OS/2 también significó un nuevo incentivo para las bases de datos basadas en MS-DOS. Aunque los servidores requerían típicamente la potencia del sistema OS/2, un PC basado en MS-DOS era adecuado como sistema frontal para muchas aplicaciones, tales como entrada de datos, elaboración de informes y consultas interactivas. El problema principal, como siempre, era la limitada memoria disponible bajo MS-DOS. La combinación del sistema operativo, el software de red y el interfaz de base de datos dejaban con frecuencia poco espacio para el programa de aplicación frontal. Para resolver este problema, Ashton-Tate decidió dividir dBASE IV en dos versiones: un dBASE IV estándar para PC autónomo, y un dBASE IV Server Edition, que soportaba los enlaces cliente/servidor con SQL Server, pero que requiere una extensión especial de MS-DOS.

Conforme los años ocurrían se acercaban a su final, el papel de los servidores de base de datos basados en SQL y OS/2 se convertía en uno de los temas más calientes en la prensa informática. Sin embargo, el tamaño íntimo del mercado, la aceptación por parte de los clientes de los sistemas y el relativo éxito de los fabricantes siguen siendo cuestiones abiertas a resolver en los noventa.

Resumen

En este capítulo se ha descrito el desarrollo de SQL y su papel como lenguaje estándar para gestión de bases de datos relationales:

- SQL fue originalmente desarrollado por investigadores de IBM, y el fuerte apoyo de IBM a SQL es una razón clave de su éxito.
- Existe un estándar oficial SQL ANSI/ISO y otros varios estándares SQL, cada uno en cierto modo diferente al estándar ANSI/ISO. Existen propuestas para extender los estándares ANSI/ISO y enmendar así algunas de las deficiencias del primer estándar.
- A pesar de la existencia de un estándar, hay muchas pequeñas variaciones entre los dialectos SQL comerciales; no hay dos SQL exactamente iguales.
- SQL está teniendo impacto en la gestión de base de datos en muchos segmentos diferentes del mercado informático, incluyendo mainframe, sistemas OLTP, estaciones de trabajo, computadores personales y redes de área local.

4 Bases de datos relacionales

Los sistemas de gestión de base de datos organizan y estructuran los datos de tal modo que puedan ser recuperados y manipulados por usuarios y programas de aplicación. Las estructuras de los datos y las técnicas de acceso proporcionadas por un DBMS particular se denominan su *modelo de datos*. El modelo de datos determina la «personalidad» de un DBMS, y las aplicaciones para las cuales está particularmente bien conformado.

SQL es un lenguaje de base de datos para bases de datos relacionales, y utiliza el *modelo de datos relacional*. ¿Qué es exactamente una base de datos relacional? ¿Cómo se almacenan los datos en una base de datos relacional? ¿Cómo se comparan las bases de datos relacionales con las tecnologías primitivas, tales como las bases de datos jerárquicos y en red? ¿Cuáles son las ventajas y desventajas del modelo relacional? Este capítulo describe el modelo de datos relacional soportado por SQL y lo compara con las estrategias primitivas de organización de base de datos.

Modelos de datos primarios

Cuando la gestión de base de datos se popularizó durante los setenta y los ochenta emergieron un puñado de modelos de datos populares. Cada uno de estos primeros modelos de datos tenían ventajas y desventajas que jugaron papeles importantes en el desarrollo del modelo de datos relacional. En muchos sentidos el modelo de datos relacional representó un intento de simplificar los modelos de datos anteriores. Para comprender el papel y la contribución de SQL y el modelo relacional, sería útil examinar brevemente algunos modelos de datos que precedieron al desarrollo de SQL.

Sistemas de gestión de archivos

Antes de la introducción de los sistemas de gestión de la base de datos, todos los datos permanentemente almacenados en un sistema informático, tales como la nómina y los registros de contabilidad, se almacenaban en archivos individuales. Un *sistema de gestión de archivos*, generalmente proporcionado por el fabricante del computador como parte del sistema operativo, llevaba la cuenta de los nombres y ubicaciones de los archivos. El sistema de gestión de archivos básicamente no tenía un modelo de datos; no sabía nada acerca de los contenidos internos de los archivos. Para el sistema de gestión de archivos, un archivo que contuviera un documento de procesamiento de textos y un archivo que contuviera datos de nóminas aparecían igual.

El conocimiento acerca del contenido de un archivo —qué datos contuviera y cómo estuvieran organizados— estaba incorporado a los programas de aplicación que utilizaban el archivo, tal como se muestra en la Figura 4.1. En esta aplicación de nóminas, cada uno de los programas COBOL que procesaban el archivo maestro de empleados contenía una *descripción de archivo* (DA) que describía la composición de los datos en el archivo. Si la estructura de los datos cambiaba —por ejemplo, si un ítem adicional de datos fuera a ser almacenado por cada empleado— todos los programas que accedían al archivo tenían que ser modificados. Como el número de archivos y programas crecía con el tiempo, todo el esfuerzo de procesamiento de datos de un departamento se perdía en mantener aplicaciones existentes en lugar de desarrollar otras nuevas.

Los problemas de mantener grandes sistemas basados en archivos condujeron a finales de los sesenta al desarrollo de los sistemas de gestión de base de datos. La idea detrás de estos sistemas era sencilla: tomar la definición de los contenidos de un archivo y la estructura de los programas individuales, y almacenarla, junto con los datos, en una base de datos. Utilizando la información de la base de datos, el DBMS que la controlaba podría tomar un papel mucho más activo en la gestión de los datos y en los cambios a la estructura de la base de datos.

Bases de datos jerárquicas

Una de las aplicaciones más importantes de los sistemas de gestión de base de datos primitivos era el planeamiento de la producción para empresas de facturación. Si un fabricante de automóviles decidía producir 10.000 unidades de un modelo de coche y 5.000 unidades de otro modelo, necesitaba saber cuántas piezas pedir a sus suministradores. Para responder a la cuestión, el producto (un coche) tenía que descomponerse en ensamblajes (motor, cuerpo, chasis), que a su vez se descomponían en subensamblajes (válvulas, cilindros, bujías) y luego en sub-subensamblajes, etc. El manejo de estas listas de piezas, conocido como una *cuenta de materiales*, era un trabajo a la medida para los computadores.

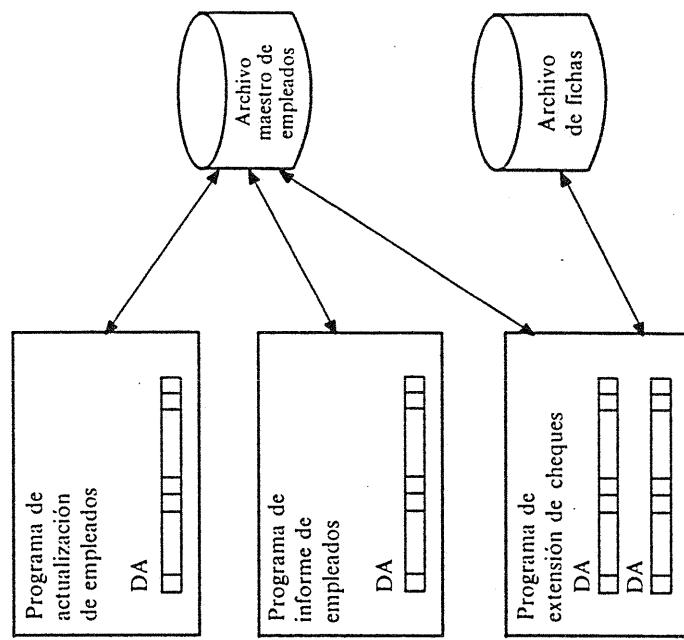


Figura 4.1. Aplicación de nómina utilizando un sistema de gestión de archivos.

La cuenta de materiales para un producto tenía una estructura jerárquica natural. Para almacenar estos datos, se desarrolló el modelo de datos *jerárquico*, ilustrado en la Figura 4.2. En este modelo, cada *registro* de la base de datos representa una pieza específica. Los registros tenían *relaciones padre/hijo*, que ligaba cada pieza a su subpieza, y así sucesivamente.

Para acceder a los datos en la base de datos, un programa podía:

- hallar un pieza particular mediante su número (como por ejemplo la puerta izquierda),
- descender al primer hijo (el tirador de la puerta),
- ascender hasta su padre (el cuerpo),
- moverse de lado hasta el siguiente hijo (la puerta derecha).

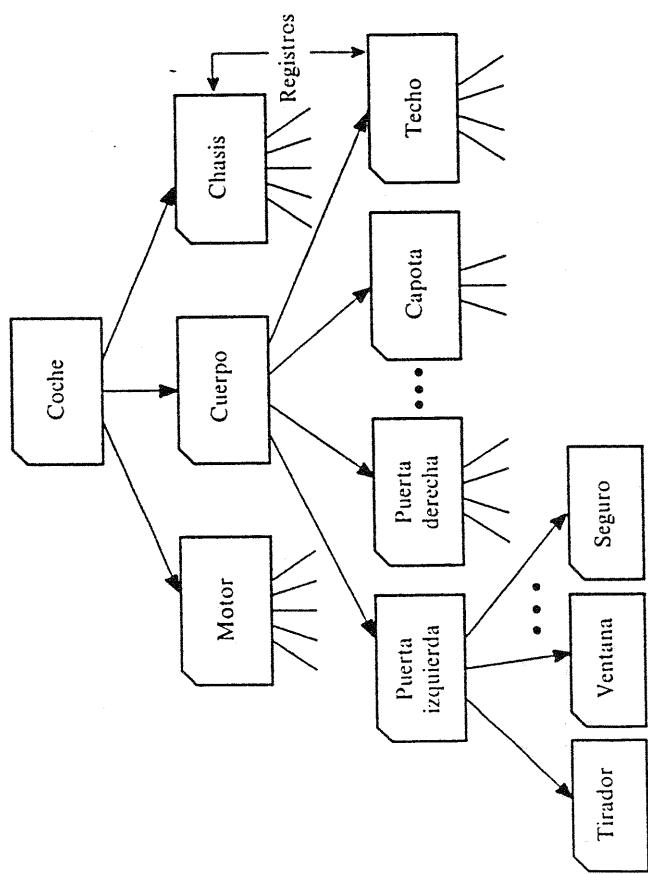


Figura 4.2. Una base de datos de lista de materiales jerárquica.

La recuperación de los datos en una base de datos jerárquica requería, por tanto, *navegar* a través de los registros, moviéndose hacia arriba, hacia abajo y hacia los lados un registro cada vez.

Uno de los sistemas de gestión de base de datos jerárquica más populares fue el Information Management System (IMS) de IBM, introducido primariamente en 1968. Las ventajas del IMS y su modelo jerárquico son las siguientes:

- **Estructura simple.** La organización de una base de datos IMS era fácil de entender. La jerarquía de la base de datos se asemejaba al diagrama de organización de una empresa o a un árbol familiar.

- **Organización padre/hijo.** Una base de datos IMS era excelente para representar relaciones padre/hijo, tales como «A es una pieza de B» o «A es propiedad de B».

- **Rendimiento.** IMS almacenaba las relaciones padre/hijo como punteros físicos de un registro de datos a otro, de modo que el movimiento a través de la

base de datos era rápido. Puesto que la estructura era sencilla, IMS podía colocar los registros padre e hijo cercanos unos a otros en el disco, minimizando la entrada/salida de disco.

IMS sigue siendo el DBMS más ampliamente instalado en los mainframe IBM. Se utiliza en más del 25 por 100 de las instalaciones de mainframe IBM.

Bases de datos en red

La estructura sencilla de una base de datos jerárquicos se convertía en una desventaja cuando los datos tenían una estructura más compleja. En una base de datos de procesamiento de pedidos, por ejemplo, un simple pedido podría participar en tres relaciones padre/hijo *diferentes*, ligando el pedido al cliente que lo remitió, al vendedor que lo aceptó y al producto ordenado, tal como se muestra en la Figura 4.3. La estructura de ese tipo de datos simplemente no se ajustaría a la jerarquía estricta de IMS.

Para manejar aplicaciones tales como el procesamiento de pedidos, se desarrolló un nuevo modelo de datos en red. El modelo de datos en red extendió el modelo jerárquico permitiendo que un registro participe en múltiples relaciones padre/hijo, como se muestra en la Figura 4.4. Estas relaciones eran conocidas como *conjuntos* en el modelo en red. En 1971 la Conferencia sobre Lenguajes de Sistemas de Datos publicó un estándar oficial para bases de datos en red, que se hizo conocido como el modelo CODASYL. IBM nunca desarrolló un DBMS en red por sí mismo, eligiendo en su lugar extender el IMS a lo largo de

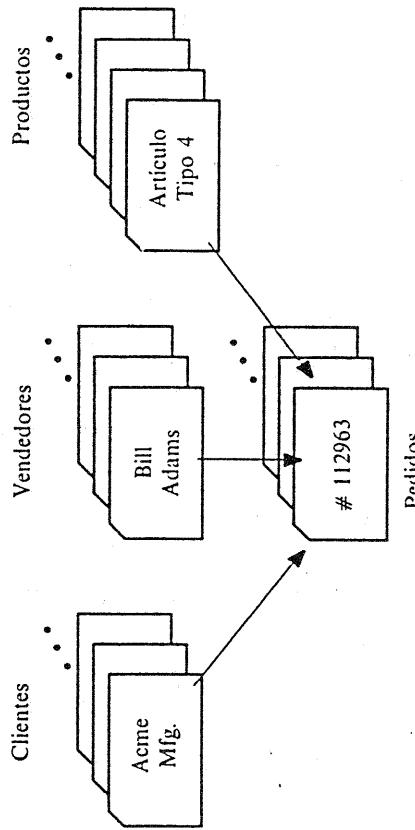


Figura 4.3. Múltiples relaciones padre/hijo.

los años. Pero durante los años setenta, compañías de software independientes se apresuraron a adoptar el modelo en red, creando productos tales como el IDMS de Cullinet, el Total de Cincom y el DBMS Adabas que se hizo muy popular.

Para un programador, acceder a una base de datos en red era muy similar a acceder a una base de datos jerárquicos. Un programa de aplicación podía:

- hallar un registro padre específico mediante clave (como por ejemplo un número de cliente),
- descender al primer hijo en un conjunto particular (el primer pedido remitido por este cliente),
- moverse lateralmente de un hijo al siguiente dentro del conjunto (la orden siguiente remitida por el mismo cliente),
- ascender desde un hijo a su padre en otro conjunto (el vendedor que aceptó el pedido).

Una vez más el programador tenía que recorrer la base de datos registro a registro, especificando esta vez qué relación recorrer además de indicar la dirección.

Las bases de datos en red tenían varias ventajas:

- **Flexibilidad.** Las múltiples relaciones padre/hijo permitían a una base de datos en red representar datos que no tuvieran una estructura jerárquica sencilla.
- **Normalización.** El estándar CODASYL respaldó la popularidad del modelo de red, y los vendedores de minicomputadores tales como Digital Equipment Corporation y Data General implementaron bases de datos en red.
- **Rendimiento.** A pesar de su superior complejidad, las bases de datos en red reforzaron el rendimiento aproximándolo al de las bases de datos jerárquicos. Los conjuntos se representaron mediante punteros a registros de datos físicos, y en algunos sistemas, el administrador de la base de datos podía especificar la agrupación de datos basada en una relación de conjunto.

Las bases de datos en red tenían sus desventajas también. Igual que las bases de datos jerárquicos, resultaban muy rígidas. Las relaciones de conjunto y la estructura de los registros tenían que ser especificadas de antemano. Modificar la estructura de la base de datos requería típicamente la reconstrucción de la base de datos, y en su lugar representaba todos los datos en la base de datos como sencillas tablas fila/columna de valores de datos. La Figura 4.5 muestra una versión relacional de la base de datos en red para procesamiento de pedidos de la Figura 4.4.

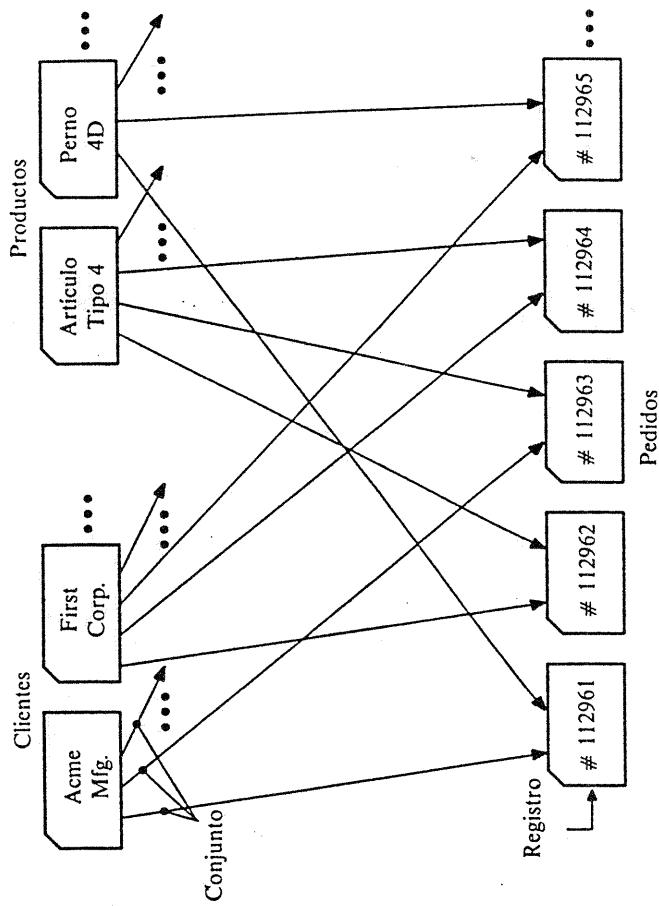


Figura 4.4. Una base de datos en red (CODASYL) para procesamiento de pedidos.

escribir un programa que recorriera su camino a través de la base de datos. La anotación de las peticiones para informes a medida duraba con frecuencia semanas o meses, y para el momento en que el programa estaba escrito la información que se entregaba con frecuencia ya no merecía la pena.

El modelo de datos relacional

Las desventajas de los modelos jerárquicos y en red condujeron a un intenso interés en el nuevo modelo de datos *relacional*, cuando fue escrito por primera vez por el Dr. Codd en 1970. El modelo relacional era un intento de simplificar la estructura de las bases de datos. Eliminaba las estructuras explícitas padre/hijo de la base de datos, y en su lugar representaba todos los datos en la base de datos como sencillas tablas fila/columna de valores de datos. La Figura 4.5 muestra una versión relacional de la base de datos en red para procesamiento de pedidos de la Figura 4.4.

La base de datos ejemplo

Tabla PRODUCTOS		
DESCRIPCION	PRECIO	EXISTENCIAS
Artículo tipo 3	\$107.00	207
Artículo tipo 4	\$117.00	139
Pasador Bisagra	\$350.00	14
:	:	
:	:	

Tabla PEDIDOS			
NUMERO_PEDIDO	EMPRESA	PRODUCTO	CANT
112963	Acme Mfg. JCP Inc.	41004	28
112975	Acme Mfg. JCP Inc.	24446	6
112983	Acme Mfg. JCP Inc.	41004	6
113012	Acme Mfg. JCP Inc.	41003	35
:	:	:	
:	:	:	

Tabla CLIENTES

EMPRESA	REP_CLIE	LIMITE_CREDITO
Acme Mfg. JCP Inc.	105	\$50,000.00
:	103	\$50,000.00
:		

Figura 4.5. Una base de datos relacional para procesamiento de pedidos.

Desgraciadamente, la definición práctica de «¿Qué es una base de datos relacional?» resultaba mucho menos clara que la definición matemática precisa recogida en el artículo de Codd de 1970. Los primeros sistemas de gestión de base de datos relacionales fallaron en implementar algunas partes clave del modelo de Codd, que sólo ahora están encontrando su acomodo en productos comerciales. Conforme el concepto relacional crecía en popularidad, muchas bases de datos que se llamaban a sí mismas «relacionales» no lo eran de hecho. En respuesta a la corrupción del término «relacional» el Dr. Codd escribió un artículo en 1985 estableciendo doce reglas a seguir por cualquier base de datos que se llamaría «verdaderamente relacional». Las doce reglas de Codd han sido aceptadas desde entonces como la definición de un DBMS verdaderamente relacional. Sin embargo, es más fácil comenzar con una definición más informal.

Una base de datos relacional es una base de datos en donde todos los datos visibles al usuario están organizados estrictamente como tablas de valores, y en donde todas las operaciones de la base de datos operan sobre estas tablas.

La definición está destinada específicamente a eliminar estructuras tales como los punteros incorporados de una base de datos jerárquico o en red. Un DBMS relacional puede representar relaciones padre/hijo, pero éstas se representan estrictamente por los valores contenidos en las tablas de la base de datos.

La Figura 4.6 muestra una pequeña base de datos relacional para una aplicación de procesamiento de pedidos. Esta base de datos ejemplo se utiliza a lo largo de todo este libro, y proporciona la base para la mayoría de los ejemplos. El Apéndice A contiene una descripción completa de la estructura de la base de datos y de su contenido.

La base de datos ejemplo contiene cinco tablas. Cada tabla almacena información referente a un *tipo* particular de entidad:

- La tabla CLIENTES almacena datos acerca de cada cliente, tales como el nombre de la empresa, el límite de crédito y el vendedor que atiende al cliente.
- La tabla REPVENTAS almacena el número de empleado, el nombre, la edad, las ventas anuales hasta la fecha y otros datos referentes a cada vendedor.

Tabla PEDIDOS

NUM_PEDIDO	FECHA_PEDIDO	CLIE	REP	FAB	PRODUCTO	CANT	IMPORTE
112961	12-17-1989	2117	106	REI	2444L	7	\$31,500.00
113012	01-11-1990	2111	ACI	41003	35		\$3,745.00
112989	01-03-1990	2101		Tabla PRODUCTOS			\$1,458.00
113051	02-10-1990	2118	1	ID_FAB	ID_PRODUCTO	DESCRIPCION	PRECIO
112968	10-12-1989	2102	1	RET	2AA5C	Vstagio Triángulo	\$9,00
112968	10-12-1989	2107	1	ACI	4100Y	Extractor	\$2,750.00
113036	01-30-1990	2107	1	QSA	XK47	Reducutor	\$355.00
113045	02-02-1990	2112	1	BIC	41672	Placa	\$180.00
112	Tabla CLIENTES	1					
113	NUM_CLIE	EMPRESA		REP_CLIE	LIMITE_CREDITO	Riostrag	
113	111	JCP Inc.	103	50,000.00	Widget	\$107.00	9
112	2111	First Corp.	101	65,000.00	Widget	\$117.00	207
112	2102	Acme Mfg.	105	50,000.00	in	\$552.00	139
113	2103	Carter & Sons	102	40,000.00		\$250.00	3
112	2107	Ace International	110	35,000.00	ngue	\$134.00	24
113	2115	Smithson Corp.	101	20,000.00		\$4,500.00	203
113	2101	Jones Mfg.	106	65,000.00	older	\$148.00	115
113	52	Tabla REPVENTAS	108	50,000.00		\$54.00	223
112	NOMBRE	EDAD	OFICINA REP	TITULO	CONTRATO	DIF CUOTA	VENTAS
112	Bill Adams	37	13	Rep Ventas	02-12-1988	104	\$350,000.00
	Mary Jones	31	11	Rep Ventas	10-12-1989	106	\$300,000.00
	Sue Smith	48	21	Rep Ventas	12-10-1986	108	\$350,000.00
		52	11	VP Ventas	01-14-1988	NULL	\$275,000.00
						106	\$200,000.00
						104	\$350,000.00
						101	NULL
						106	\$350,000.00

Tabla CLIENTES

CLUID	REGION	DIR	OBJETIVO	VENTAS
22	Oeste	108	\$186,042.00	\$367,911.00
11	Este	106	\$575,000.00	\$305,673.00
12	Este	104	\$800,000.00	\$75,985.00
13	Este	105	\$350,000.00	\$361,865.00
				\$367,911.00

Figura 4.6. La base de datos ejemplo.

- La tabla **OFICINAS** almacena datos acerca de cada una de las cinco oficinas de ventas incluyendo la ciudad en donde está localizada la oficina, la región de ventas a la que pertenece, etc.
- La tabla **PEDIDOS** lleva la cuenta de cada pedido remitido por un cliente, identificando al vendedor que aceptó el pedido, el producto solicitado, la cantidad y el importe del pedido, etc. Por simplicidad, cada pedido atañe a un solo producto.
- La tabla **PRODUCTOS** almacena datos acerca de cada producto disponible para venta, tal como el fabricante, el número del producto, su descripción y su precio.

- La tabla **VENTAS** almacena datos acerca de cada producto disponible para venta, tal como el fabricante, el número del producto, su descripción y su precio.
- El principio de organización de una base de datos relacional es la *tabla*, una disposición rectangular fila/columna de los valores de datos. Cada tabla de una

Tablas

El principio de organización de una base de datos relacional es la *tabla*, una disposición rectangular fila/columna de los valores de datos. Cada tabla de una

Tabla OFICINAS

OFICINA	CIUDAD	REGION	DIR	OBJETIVO	VENTAS
22	Denver	Oeste	108	\$300,000.00	\$186,042.00
11	New York	Este	106	\$575,000.00	\$692,637.00
12	Chicago	Este	104	\$800,000.00	\$735,042.00
13	Atlanta	Este	105	\$350,000.00	\$367,911.00
21	Los Angeles	Oeste	108	\$725,000.00	\$835,915.00

Ciudad donde cada
oficina está
localizada

Número
de empleado
del director
de la oficina

Ventas anuales
hasta la fecha
para la oficina

Los datos de
esta fila se
refieren a
esta oficina

Los datos de
esta fila se
refieren a
esta oficina

Los datos de
esta fila se
refieren a
esta oficina

ya que la compañía sólo tiene estas dos regiones de ventas.

Cada columna de una tabla tiene un *nombre de columna*, que se escribe generalmente como encabezamiento en la parte superior de la columna. Las columnas de una tabla deben tener todos nombres diferentes, pero no está prohibido que columnas de tablas diferentes tengan nombres idénticos. De hecho, nombres frecuentemente utilizados, tales como NOMBRE, DIRECCIÓN, CNT, PRECIO y VENTAS, se encuentran a menudo en muchas tablas diferentes de una base de datos de producción.

Las columnas de una tabla tienen un orden de izquierda a derecha, que se define cuando la tabla se crea por primera vez. Una tabla siempre tiene al menos una columna. El estándar SQL ANSI/ISO no especifica un número máximo de columnas en una tabla, pero casi todos los productos comerciales SQL imponen un límite. Generalmente el límite es de 255 columnas por tabla o más. A diferencia de las columnas, las filas de una tabla no tienen orden particular. De hecho, si se utilizan dos consultas de bases de datos consecutivos para

base de datos tiene un *nombre de tabla* único que identifica sus contenidos. (En realidad, cada usuario puede elegir sus propios nombres de tablas sin preocuparse acerca de los nombres elegidos por otros usuarios, como se explicará en el Capítulo 5).

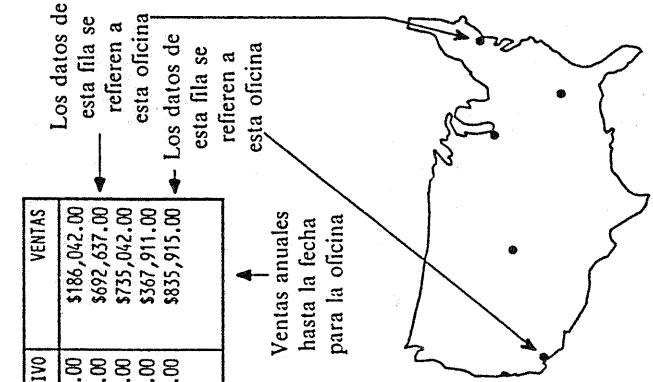
La estructura fila/columna de una tabla se muestra más claramente en la Figura 4.7, que es una vista ampliada de la tabla OFICINAS. Cada fila horizontal de la tabla OFICINA representa una única entidad física —una única oficina de ventas—. Juntas las cinco filas de la tabla representan a las cinco oficinas de ventas de la empresa. Todos los datos de una fila particular de la tabla se aplican a la oficina representada mediante esa fila.

Cada columna vertical de la tabla OFICINAS representa un elemento de datos que está almacenado en la base de datos para cada oficina. Por ejemplo, la columna VENTAS contiene el total de ventas anuales hasta la fecha para cada oficina. La columna DIR muestra el número de empleado de la persona que dirige la oficina.

Cada fila de una tabla contiene exactamente un valor en cada columna. En la fila que representa la oficina de New York, por ejemplo, la columna CIUDAD contiene el valor «New York». La columna VENTAS contiene el valor \$692,637.00, que es el total de las ventas anuales hasta la fecha para la oficina de Nueva York.

Para cada columna de una tabla, todos los valores de esa columna contienen el mismo tipo de datos. Por ejemplo, todos los valores de la columna CIUDAD son palabras, todos los valores de VENTAS son cantidades de dinero y todos los valores DIR son enteros (que representan números de empleados). El conjunto de valores que una columna puede contener se denomina el *dominio* de la columna. El dominio de la columna CIUDAD es el conjunto de todos los nombres de ciudades. El dominio de la columna VENTAS es cualquier cantidad de dinero. El dominio de la columna REGION es uno de dos valores de datos, «Este» y «Oeste», ya que la compañía sólo tiene estas dos regiones de ventas.

Figura 4.7. La estructura fila/columna de una tabla relacional.



visualizar los contenidos de una tabla no hay garantía de que las filas sean listadas en el mismo orden dos veces. Naturalmente se puede solicitar a SQL que ordene las filas antes de visualizarlas, pero el orden de clasificación no tiene nada que ver con la disposición efectiva de las filas dentro de la tabla.

Una tabla puede tener cualquier número de filas. Una tabla de cero filas es perfectamente legal, y se denomina una tabla *vacía* (por razones obvias). Una tabla vacía sigue teniendo una estructura, impuesta por sus columnas; simplemente no contiene datos. El estándar ANSI/ISO no limita el número de filas de una tabla, y muchos productos SQL permiten que una tabla crezca hasta que agote el espacio de disco disponible en el computador. Otros productos SQL imponen un límite máximo, pero éste es siempre muy generoso —dos millones de filas o más es común.

Claves primarias

Puesto que las filas de una tabla relacional no están ordenadas, no se puede seleccionar una fila específica por su posición en la tabla. No hay «primera fila», «última fila», o «decimotercera fila» de una tabla. ¿Cómo se puede especificar entonces una fila particular, por ejemplo la fila correspondiente a la oficina de ventas de Denver?

En una base de datos relacional bien diseñada cada tabla tiene una columna o combinación de columnas cuyos valores identifican únicamente cada fila en la tabla. Esta columna (o columnas) se denomina *clave primaria* de la tabla. Miremos una vez más la tabla OFICINAS en la Figura 4.7. A primera vista, tanto la columna OFICINA como la columna CIUDAD podrían servir como clave primaria para la tabla, pero si la empresa se amplia y abre dos oficinas de ventas en la misma ciudad, la columna CIUDAD ya no podría servir como clave primaria. En la práctica, «números de ID», tales como el número de oficina (OFICINA en la tabla OFICINAS), el número de empleado (NUM_EMP en la tabla REPVENTAS) y los números de clientes (NUM_CLIE en la tabla CLIENTES) se eligen con frecuencia como claves primarias. En el caso de la tabla PEDIDOS no hay elección —lo único que identifica únicamente un pedido es el número de pedido (NUM_PEDIDO).

La tabla PRODUCTOS, parte de la cual se muestra en la Figura 4.8, es un ejemplo de una tabla en donde la clave primaria debe ser una *combinación* de columnas. La columna ID_FAB identifica al fabricante de cada producto en la tabla y la columna ID_PRODUCTO especifica el número de producto del fabricante. La columna ID_PRODUCTO podría ser una buena clave primaria, pero no hay nada que impida que dos fabricantes diferentes utilicen el mismo número para sus productos. Por tanto, debe utilizarse una combinación de las columnas ID_FAB e ID_PRODUCTO como clave primaria de la tabla PRODUCTOS. Cada producto de la tabla se garantiza que tiene una combinación única de valores en estas dos columnas.

Tabla PRODUCTOS					
ID_FAB	ID_PRODUCTO	DESCRIPCION		PRECIO	EXISTENCIAS
•	•	Artículo Tipo 3		\$107.00	207
ACI	41003	Artículo Tipo 4		\$117.00	139
ACI	41004	Manivela		\$652.00	3
BIC	41003				
•	•				

Clave Primaria

Figura 4.8. Una tabla con una clave primaria compuesta.

La clave primaria tiene un valor único diferente para cada fila de una tabla, de modo que no hay dos filas de una tabla con clave primaria que sean duplicados exactos la una de la otra. Una tabla en donde cada fila es diferente de todas las demás se llama una *relación* en términos matemáticos. El nombre «base de datos relacional» proviene de este término, ya que las relaciones (las tablas con filas distintas) son el corazón de una base de datos relacional.

Aunque las claves primarias son parte esencial del modelo de datos relacional, los primeros sistemas de gestión de base de datos relacionales (System/R, DB2, Oracle y otros), no proporcionaban soporte explícito para claves primarias. Los diseñadores de bases de datos aseguraban generalmente que todas las tablas de sus bases de datos tuvieran una clave primaria, pero el propio DBMS no proporcionaba un modo de identificar la clave primaria de la tabla. DB2 Versión 2, introducido en abril de 1988, añadió finalmente soporte de clave primaria a los productos SQL comerciales de IBM. El estándar ANSI/ISO fue posteriormente ampliado para incluir soporte para claves primarias. Sin embargo, el soporte DBMS explícito de claves primarias sigue siendo raro en los productos SQL comerciales.

Relaciones

Una de las principales diferencias entre el modelo relacional y los modelos de datos primitivos es que los punteros explícitos, tales como las relaciones padre/hijo de una base de datos jerárquicos, están prohibidas en las bases de datos relacionales. Obviamente estas relaciones siguen existiendo en una base de datos

relacional. Por ejemplo, en la base de datos de muestra, cada uno de los vendedores tiene asignada una oficina de ventas particular, por lo que hay una relación obvia entre las filas de la tabla OFICINAS y las filas de la tabla REPVENTAS. ¿No « pierde información» el modelo relacional al prohibir estas relaciones en la base de datos?

Como se muestra en la Figura 4.9, la respuesta a la pregunta es «no». La figura muestra una vista en detalle de unas cuantas filas de las tablas OFICINAS y REPVENTAS. Observe que la columna OFICINA.REP de la tabla REPVENTAS contiene el número de oficina de la oficina de ventas en donde trabaja cada vendedor. El dominio de esta columna (el conjunto de valores legales que puede contener) es precisamente el conjunto de números de oficina que se hallan en la columna OFICINA de la tabla OFICINAS. De hecho, usted puede hallar la oficina de ventas en donde trabaja Mary Jones encontrando el valor de la columna OFICINA.REP de Mary (11), y determinando la fila de la tabla OFICINAS que tiene un valor coincidente en la columna OFICINA (en la fila para la oficina de New York). Análogamente, para encontrar a todos los vendedores que trabajan en New York, podría anotarse el valor OFICINA para la fila New York (11) y luego repasar la columna OFICINA.REP de la tabla REPVENTAS buscando los valores coincidentes (en las filas de Mary Jones y Sam Clark).

La relación padre/hijo entre una oficina de ventas y la gente que trabaja en ella no se pierde en el modelo relacional, tan solo no se representa por un

puntero explícito almacenado en la base de datos. En vez de ello, la relación está representada por *valores de datos comunes* almacenados en las dos tablas. Todas las relaciones de una base de datos relacional están representadas de este modo. Uno de los objetivos principales del lenguaje SQL es permitir recuperar de la base de datos, datos relacionados manipulando estas relaciones de un modo sencillo y directo.

Claves foráneas

Una columna de una tabla cuyo valor coincide con la clave primaria de alguna otra tabla se denomina una *clave foránea*. En la Figura 4.9 la columna OFICINA.REP es una clave foránea para la tabla OFICINAS. Aunque es una columna en la tabla REPVENTAS, los valores que esta columna contiene son números de oficina. Coincidirán con valores en la columna OFICINA, que es la clave primaria para la tabla OFICINAS. Juntas, una clave primaria y una clave foránea crean una relación padre/hijo entre las tablas que las contienen, del mismo modo que las relaciones padre/hijo de una base de datos jerárquicos.

Lo mismo que una combinación de columnas puede servir como clave primaria de una tabla, una clave foránea puede ser también una combinación de columnas. De hecho, la clave foránea será *siempre* una clave compuesta (multicolumna) cuando referencia a una tabla con una clave primaria compuesta. Obviamente el número de columnas y los tipos de datos de las columnas en la clave foránea y en la clave primaria deben ser idénticos unos a otros.

Una tabla puede contener más de una clave foránea si está relacionada con más de una tabla adicional. La Figura 4.10 muestra las tres claves foráneas de la tabla PEDIDOS en la base de datos ejemplo:

- La columna CLIE es una clave foránea para la tabla CLIENTES, que relaciona cada pedido con el cliente que lo remitió.

- La columna REP es un clave foránea para la tabla REPVENTAS que relaciona cada pedido con el vendedor que lo tomó.

- Las columnas DIR y PRODUCTO juntas son una clave foránea compuestas para la tabla PRODUCTOS, que relacionan cada pedido con el producto solicitado.

Las múltiples relaciones padre/hijo creadas por las tres claves foráneas en la tabla PEDIDOS pueden parecerle familiar a usted, y así debería ser. Son precisamente las mismas relaciones que se presentaban en la base de datos en red de la Figura 4.4. Como muestra el ejemplo, el modelo de datos relacional tiene toda la potencia del modelo en red para expresar relaciones complejas.

Las claves foráneas son parte fundamental del modelo relacional ya que crean relaciones entre tablas en la base de datos. Desgraciadamente, como con las claves primarias, el soporte de claves foráneas falta en los sistemas de gestión

Tabla OFICINAS			
OFICINA	CIUDAD	REGION	
22	Denver	Oeste	
11	New York	Este	
	Chicago	Este	
	:		
	:		

Tabla REPVENTAS			
NUM.EMPLE	NOMBRE	EDAD	OFICINA.REP
105	Bill Adams	37	13
109	Mary Jones	31	11
102	Sue Smith	48	21
106	Sam Clark	52	11
	:		
	:		

Figura 4.9. Una relación padre/hijo en una base de datos relacional.

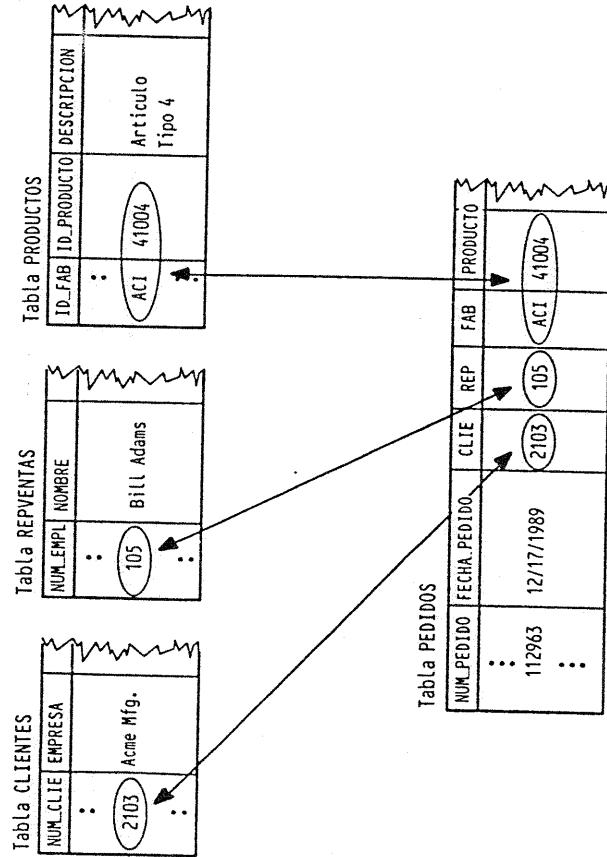


Figura 4.10. *Múltiples relaciones padre/hijo en una base de datos relacional.*

de base de datos relacional primitivo. Fueron añadidos a DB2 Versión 2 y desde entonces se han añadido al estándar ANSI/ISO. Al igual que las claves primarias, sin embargo, el soporte explícito de las claves foráneas está ausente en la mayoría de los productos SQL comerciales actuales.

Las doce reglas de Codd*

En su artículo de 1985 en *Computerworld*, Ted Codd presentó doce reglas que una base de datos debe obedecer para que sea considerada verdaderamente relacional. Las doce reglas de Codd se muestran en la Tabla 4.1, y desde entonces se han convertido en una definición semioficial de una base de datos relacional. Las reglas se derivan del trabajo teórico de Codd sobre el modelo relacional, y representan realmente más un objetivo ideal que una definición de una base de datos relacional.

Ningún DBMS relacional actualmente disponible satisface totalmente las doce reglas de Codd. De hecho, se está convirtiendo en una práctica popular

elaborar «tarjetas de tanteo» para productos DBMS comerciales, que muestran lo bien o mal que éstos satisfacen cada una de las reglas. Desgraciadamente, las reglas son subjetivas de modo que los calificadores están generalmente llenos de notas a pie de página y calificadores, y no revelan demasiado acerca de los productos.

La Regla 1 es básicamente la definición informal de una base de datos relacional presentada al comienzo de esta sección. La Regla 2 refuerza la importancia de las claves primarias para localizar datos en la base de datos. El nombre de la tabla localiza la tabla correcta, el nombre de la columna encuentra la columna correcta y el valor de clave primaria encuentra la fila que contiene un dato individual de interés. La Regla 3 requiere soporte para falta de datos mediante el uso de valores NULL, que se describirán en el Capítulo 5.

La Regla 4 requiere que una base de datos relacional sea autodescriptiva. En otras palabras, la base de datos debe contener ciertas *tabelas de sistema* cuyas columnas describan la estructura de la propia base de datos. Estas tablas se describirán en el Capítulo 16.

La Regla 5 ordena la utilización de un lenguaje de base de datos relacional como SQL, aunque no se requiera específicamente SQL. El lenguaje debe ser capaz de soportar todas las funciones básicas de un DBMS —creación de una base de datos, recuperación y entrada de datos, implementación de la seguridad de la base de datos, etc.

La Regla 6 trata de las vistas, que son *tabelas virtuales* utilizadas para dar a diferentes usuarios de una base de datos diferentes vistas de su estructura. Es una de las reglas más difíciles de implementar en la práctica, y ningún producto comercial la satisface totalmente hoy día. Las vistas y sus problemas de actualización se describirán en el Capítulo 13.

La Regla 7 refuerza la naturaleza orientada a conjuntos de una base de datos relacional. Requiere que las filas sean tratadas como conjuntos en operaciones de inserción, supresión y actualización. La regla está diseñada para prohibir implementaciones que sólo soportan la modificación o recorrido fila a fila de la base de datos.

La Regla 8 y la Regla 9 aislan al usuario o al programa de aplicación de la implementación de bajo nivel de la base de datos. Especifican que las técnicas de acceso a almacenamiento específicas utilizadas por el DBMS, e incluso los cambios a la estructura de las tablas en la base de datos, no deberían afectar a la capacidad del usuario de trabajar con los datos.

La Regla 10 dice que el lenguaje de base de datos debería soportar las restricciones de integridad que restringen los datos que pueden ser introducidos en la base de datos y las modificaciones que puedan ser efectuadas en ésta. Esta es otra de las reglas que no soportan la mayoría de los productos comerciales DBMS.

La Regla 11 dice que el lenguaje de base de datos debe ser capaz de manipular datos distribuidos localizados en otros sistemas informáticos. Los

1. *La regla de información.* Toda la información de una base de datos relacional está representada explícitamente a nivel lógico y exactamente de un modo —mediante valores en tablas.
2. *Regla de acceso garantizado.* Todos y cada uno de los datos (valor atómico) de una base de datos relacional se garantiza que sean lógicamente accesibles recurriendo a una combinación de nombre de tabla, valor de clave primaria y nombre de columna.
3. *Tratamiento sistemático de valores nulos.* Los valores nulos (distinto de la cadena de caracteres vacía o de una cadena de caracteres en blanco y distinta del cero o de cualquier otro número) se soportan en los DBMS completamente relacionales para representar la falta de información y la información inaplicable de un modo sistemático e independiente del tipo de datos.
4. *Catálogo en línea dinámico basado en el modelo relacional.* La descripción de la base de datos se representa a nivel lógico del mismo modo que los datos ordinarios, de modo que los usuarios autorizados puedan aplicar a su interrogación el mismo lenguaje relacional que aplican a los datos regulares.
5. *Regla de sublenguaje completo de datos.* Un sistema relacional puede soportar varios lenguajes y varios modo de uso terminal (por ejemplo, el modo de rellenar con blancos). Sin embargo, debe haber al menos un lenguaje cuyas sentencias sean expresables, mediante alguna sintaxis bien definida, como cadenas de caracteres, y que sea completa en cuanto al soporte de todos los puntos siguientes.
 - Definición de datos.
 - Definición de vista.
 - Manipulación de datos (interactiva y por programa).
 - Restricciones de integridad.
 - Autorización.
 - Fronteras de transacciones (comienzo, cumplimentación y vuelta atrás).
6. *Regla de actualización de vista.* Todas las vistas que sean teóricas actualizables son también actualizables por el sistema.
7. *Inserción, actualización y supresión de alto nivel.* La capacidad de manejar una relación de base de datos o una relación derivada como un único operando se aplica no solamente a la recuperación de datos, sino también a la inserción, actualización y supresión de los datos.
8. *Independencia física de los datos.* Los programas de aplicación y las actividades terminales permanecen lógicamente inalterados cuando se efectúan cambios en la base de datos, ya sea a las representaciones de almacenamiento o a los métodos de acceso.

9. *Independencia lógica de los datos.* Los programas de aplicación y las actividades terminales permanecen lógicamente inalterados cuando se efectúan cambios sobre las tablas de base cambios preservadores de la información de cualquier tipo que teóricamente permita alteraciones.
10. *Independencia de integridad.* Las restricciones de integridad específicas para una base de datos relacional particular deben ser definibles en el sublenguaje de datos relacional y almacenables en el catálogo, no en los programas de aplicación.
11. *Independencia de distribución.* Un DBMS relacional tiene independencia de distribución.
12. *Regla de no subversión.* Si un sistema relacional tiene un lenguaje de bajo nivel (un solo registro cada vez), ese bajo nivel no puede ser utilizado para subvertir o suprimir las reglas de integridad y las restricciones expresadas en el lenguaje relacional de nivel superior (multiples registros a la vez).

Tabla 4.1. Las 12 reglas de Codd para DBMS relacional (continuación).

SQL está basado en el modelo relacional de datos que organiza los datos en una base de datos como una colección de tablas:

- Cada tabla tiene un nombre que la identifica únicamente.
- Cada tabla tiene una o más columnas nominadas, que están dispuestas en un orden específico de izquierda a derecha.
- Cada tabla tiene cero o más filas, conteniendo cada una un único valor en cada columna. Las filas están desordenadas.
- Todos los valores de una columna determinada tienen el mismo tipo de datos, y éstos están extraídos de un conjunto de valores legales llamado el dominio de la columna.

Las tablas están relacionadas unas con otras por los datos que contienen. El modelo de datos relacional utiliza claves primarias y claves foráneas para representar estas relaciones entre tablas.

Tabla 4.1. Las 12 reglas de Codd para DBMS relacional.

- Una clave primaria es una columna o combinación de columnas dentro de una tabla cuyo(s) valor(es) identifica(n) únicamente a cada fila de la tabla. Una tabla tiene una única clave primaria.
- Una clave foránea es una columna o combinación de columnas en una tabla cuyo(s) valor(es) es(son) un valor de clave primaria para alguna otra tabla. Una tabla puede contener más de una clave foránea, enlazándola a una o más tablas.
- Una combinación clave primaria/clave foránea crea una relación padre/hijo entre las tablas que las contienen.

Recuperación de datos

Segunda parte

Las consultas son el corazón del lenguaje SQL, y mucha gente utiliza SQL exclusivamente como herramienta de consulta de base de datos. Los cinco capítulos siguientes describen las consultas SQL en profundidad. El Capítulo 5 describe las estructuras básicas del lenguaje SQL que se emplean para formar sentencias SQL. El Capítulo 6 discute consultas simples que extraen datos de una única tabla de datos. El Capítulo 7 amplía la discusión a consultas multitable. Las consultas que resumen datos se describen en el Capítulo 8. Finalmente, el Capítulo 9 explica la capacidad subconsulta de SQL que se utiliza para manejar consultas complejas.

5 Conceptos básicos de SQL

Este capítulo comienza con una descripción detallada de las características de SQL. Describe la estructura básica de una sentencia SQL y los elementos básicos del lenguaje, tales como las palabras clave, los tipos de datos y las expresiones. El modo en que SQL gestiona la falta de datos mediante valores NULL también se describe aquí. Aunque éstas son características básicas de SQL, hay algunas diferencias sutiles en el modo en que son implementadas por parte de diferentes productos SQL, y en muchos casos los productos SQL proporcionan extensiones significativas a las capacidades especificadas en el estándar SQL ANSI/ISO. Estas diferencias y extensiones también se describen en este capítulo.

Sentencias

El lenguaje SQL consta de unas treinta sentencias, que a continuación se resumen en la Tabla 5.1. Cada sentencia demanda una acción específica por parte del DBMS, tal como la creación de una nueva tabla, la recuperación de datos o la inserción de nuevos datos en la base. Todas las sentencias SQL tienen la misma forma básica, ilustrada en la Figura 5.1.

Sentencia	Descripción
<i>Manipulación de datos</i>	
SELECT	Recupera datos de la base de datos.
INSERT	Añade nuevas filas de datos a la base de datos.
DELETE	Suprime filas de datos de la base de datos.
UPDATE	Modifica datos existentes en la base de datos.
<i>Definición de datos</i>	
CREATE TABLE	Añade una nueva tabla a la base de datos.
DROP TABLE*	Suprime una tabla de la base de datos.
ALTER TABLE*	Modifica la estructura de una tabla existente.
CREATE VIEW*	Añade una nueva vista a la base de datos.
DROP VIEW*	Suprime una lista de la base de datos.
CREATE INDEX*	Construye un índice para una columna.
DROP INDEX*	Suprime el índice para una columna.
CREATE SYNONYM*	Define un alias para un nombre de tabla.
DROP SYNONYM*	Suprime un alias para un nombre de tabla.
COMMENT*	Define comentarios para una tabla.
LABEL*	Define el título de una columna.
<i>Control de acceso</i>	
GRANT	Concede privilegios de acceso a usuarios.
REVOKE	Suprime privilegios de acceso a usuarios.
<i>Control de transacciones</i>	
COMMIT	Finaliza la transacción actual.
ROLLBACK	Aborta la transacción actual.
<i>SQL programático</i>	
DECLARE	Define un cursor para una consulta.
EXPLAIN*	Describe el plan de acceso a datos para una consulta.
OPEN	Abre un cursor para recuperar resultados de consulta.
FETCH	Recupera una fila de resultados de consulta.
CLOSE	Cierra un cursor.
PREPARE*	Prepara una sentencia SQL para ejecución dinámica.
EXECUTE*	Ejecuta dinámicamente una sentencia SQL.
DESCRIBE*	Describe una consulta preparada.

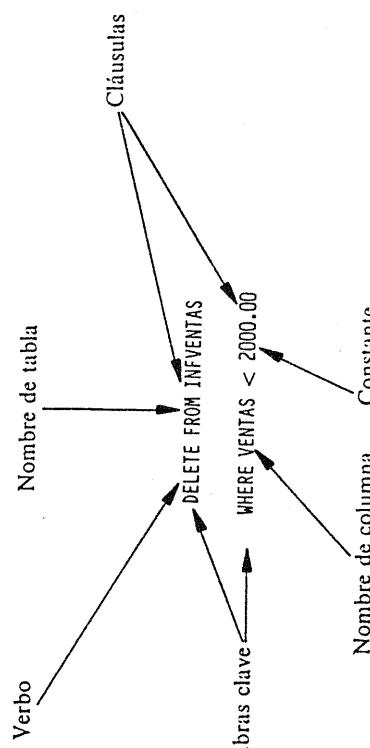
Tabla 5.1. Principales sentencias SQL.

Figura 5.1. La estructura de una sentencia SQL.

Todas las sentencias SQL comienzan con un *verbo*, una palabra clave que describe lo que la sentencia hace. CREATE, INSERT, DELETE y COMMIT son verbos típicos. La sentencia continúa con una o más *cláusulas*. Una cláusula puede especificar los datos sobre los que debe actuar la sentencia, o proporcionar más detalles acerca de lo que la sentencia se supone que hace. Todas las cláusulas comienzan también con una palabra clave, tal como WHERE, FROM, INTO y HAVING. Algunas cláusulas son opcionales; otras son necesarias. La estructura y contenido específicos varían de una cláusula a otra. Muchas cláusulas contienen nombres de tablas o columnas; algunas pueden contener palabras clave adicionales, constantes o expresiones.

El estándar SQL ANSI/ISO especifica las palabras clave SQL que se utilizan como verbos y en cláusulas de sentencias. Según el estándar, esas palabras clave no pueden ser utilizadas para designar objetos de la base de datos, tales como tablas, columnas y usuarios. Muchas implementaciones SQL relajan esta restricción, pero generalmente es buena idea evitar las palabras clave al nombrar tablas y columnas. La Tabla 5.2 lista las palabras clave actualmente incluidas en el estándar SQL ANSI/ISO.

A lo largo de todo este libro, las formas aceptables de una sentencia SQL se ilustran mediante un diagrama sintáctico, como el que se muestra en la Figura 5.2. Una sentencia SQL válida o una cláusula se construye «siguiendo la línea» a través del diagrama sintáctico hasta el punto que marca el final del diagrama. Las palabras claves del diagrama sintáctico y de los ejemplos (tales como DELETE y FROM) en la Figura 5.2 se muestran siempre en MAYÚSCULAS, pero casi todas las implementaciones SQL aceptan las palabras clave tanto en mayúsculas como en minúsculas, y con frecuencia es más conveniente declararlas efectivamente en minúsculas.



* No forma parte del estándar SQL ANSI/ISO, pero se encuentra en la mayoría de los productos más populares basados en SQL.

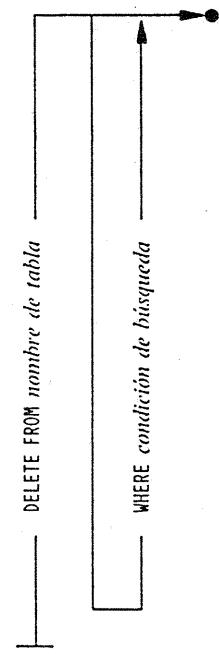


Figura 5.2. Un ejemplo de diagrama sintáctico.

Los ítems variables de una sentencia SQL (tales como el nombre de una tabla y la condición de búsqueda en la Figura 5.2) se muestran en *cursiva minúscula*. Es cuestión del usuario especificar el ítem adecuado cada vez que utilice la sentencia. Las cláusulas y palabras clave optionales, tales como la cláusula WHERE de la Figura 5.2, se indican mediante caminos alternativos dentro del diagrama sintáctico. Cuando se ofrece una elección de palabras clave optionales, la opción por omisión (es decir, el comportamiento de la sentencia si no se especifica palabra clave) está SUBRAYADA

Nombres

Los objetos de una base de datos basada en SQL se identifican asignándoles nombres únicos. Los nombres se utilizan en las sentencias SQL para identificar el objeto de la base de datos sobre la que la sentencia debe actuar. El estándar SQL ANSI/ISO especifica nombres de tabla (que identifican tablas), nombres de columna (que identifican columnas) y nombres de usuario (que identifican usuarios de la base de datos). Muchas implementaciones SQL soportan objetos nombrados adicionalmente, tales como procedimientos almacenados (SQL Server), relaciones clave primaria/clave foránea (DB2) y formularios de entrada de datos (Ingres).

El estándar ANSI/ISO especifica que los nombres SQL deben contener de 1 a 18 caracteres, comenzar con una letra, y que no pueden contener espacios o caracteres de puntuación especiales. En la práctica los nombres soportados por los productos DBMS basados en SQL varían significativamente. DB2, por ejemplo, restringe los nombres de usuario a ocho caracteres, pero permite 18 caracteres en los nombres de tablas y columnas, y muchas otras implementaciones SQL también permiten nombres más largos que los límites ANSI/ISO. Los diferentes productos también se diferencian en los caracteres especiales que permiten en los nombres de tablas. Por portabilidad es mejor mantener los nombres relativamente breves y evitar el uso de caracteres especiales.

Nombres de tabla

Cuando se especifica un nombre de tabla en una sentencia SQL, SQL presupone que se está refiriendo a una de las tablas propias (es decir una tabla ya creada). Con el permiso adecuado, también se puede referir a tablas propiedad de otros usuarios, utilizando un *nombre de tabla cualificado*. Un nombre de tabla cualificado especifica el nombre del propietario de la tabla junto con el nombre de la tabla, separados por un punto (.). Por ejemplo, la tabla CUMPLEAÑOS, propiedad del usuario de nombre SAM, tiene el nombre de tabla cualificado:

Tabla 5.2. Palabras clave de SQL ANSI/ISO.

ADA	CURRENT	FROM	NULL	SELECT
ALL	CURSOR	GO	NUMERIC	SET
AND	DEC	GOTO	OF	SMLLINT
ANY	DECIMAL	GRANT	ON	SOME
AS	DECLARE	GROUP	OPEN	SQL
ASC	DEFAULT	HAVING	OPTION	SQLCODE
AUTHORIZATION	DELETE	IN	OR	SQLERROR
AVG	DESC	INDICATOR	ORDER	SUM
BEGIN	DISTINCT	INSERT	PASCAL	TABLE
BETWEEN	DOUBLE	INT	PLI	TO
BY	END	INTEGER	PRECISION	UNION
C	ESCAPE	INTO	PRIMARY	UNIQUE
CHAR	EXEC	IS	PRIVILEGES	UPDATE
CHARACTER	EXISTS	KEY	PUBLIC	USER
CHECK	FETCH	LANGUAGE	REAL	VALUES
CLOSE	FLOAT	LIKE	REFERENCES	VIEW
COBOL	FOR	MAX	ROLLBACK	WHENEVER
COMMIT	FOREIGN	MIN	SCHEMA	WHERE
CONTINUE	FORTRAN	MODULE	SECTION	WITH
COUNT	FOUND	NOT		WORK
CREATE				

Un nombre de tabla cualificado puede ser utilizado generalmente dentro de una sentencia SQL en cualquier lugar que pueda aparecer un nombre de tabla.

Nombres de columna

Cuando se especifica un nombre de columna en una sentencia SQL, SQL puede determinar normalmente a qué columna se refiere a partir del contexto. Sin embargo, si la sentencia afecta a dos columnas con el mismo nombre correspondientes a dos tablas diferentes, debe utilizarse un *nombre de columna cualificado* para identificar sin ambigüedad la columna designada. Un nombre de columna cualificada especifica tanto el nombre de la tabla que contiene la columna como el nombre de la columna, separados por un punto (.). Por ejemplo, la columna de nombre VENTAS en la tabla REPVENTAS tiene el nombre de columna cualificado REPVENTAS.VENTAS

Si la columna procede de una tabla propiedad de otro usuario, se utiliza un nombre de tabla cualificado en el nombre de columna cualificado. Por ejemplo, la columna DIA_MES en la tabla CUMPLEAÑOS propiedad del usuario SAM se especifica mediante el nombre de columna cualificado completo:

```
SAM.CUMPLEAÑOS.DIA_MES
```

Los nombres de columna cualificados pueden ser utilizados generalmente en una sentencia SQL en cualquier lugar en el que pueda aparecer un nombre de columna simple (no cualificado); las excepciones se indican en las descripciones de las sentencias SQL individuales.

Tipos de datos

El estándar SQL ANSI/ISO especifica varios tipos de datos que pueden ser almacenados en una base de datos basada en SQL y manipulados por el lenguaje SQL. Los tipos de datos especificados por el estándar son sólo un conjunto mínimo, pero casi todos los productos SQL comerciales, los soportan o tienen tipos de datos que son muy similares. Los tipos de datos ANSI/ISO están listados en la Tabla 5.3. Consisten en los siguientes:

- *Cadenas de caracteres de longitud fija.* Las columnas que contienen este tipo de datos almacenan típicamente nombres de personas y empresas, direcciones, descripciones, etc.
- *Enteros.* Las columnas que contienen este tipo de datos almacenan típicamente cuentas, cantidades, edades, etc. Las columnas de valores enteros también se utilizan con frecuencia para contener números identificadores, tales como números de cliente, de empleado y de pedido.
- *Números decimales.* Las columnas con este tipo de datos almacenan números que tienen parte fraccionaria y deben ser calculados exactamente, tales como porcentajes y tasas. También se utilizan frecuentemente para almacenar importes monetarios.
- *Números en coma flotante.* Las columnas con este tipo de datos se utilizan para almacenar números científicos que pueden ser calculados aproximadamente, tales como pesos y distancias. Los números en coma flotante pueden representar mayores rangos de valores que los números decimales, pero pueden producir errores de redondeo en los cálculos.

Tipos de datos extendidos

La mayoría de los productos SQL comerciales ofrecen un conjunto mucho más extenso de tipos de datos que los especificados en el estándar ANSI/ISO. Algunos de los tipos de datos extendidos más populares o importantes son:

- *Cadenas de caracteres de longitud variable.* Casi todos los productos SQL soportan los datos VARCHAR, que permiten que una columna almacene cadenas de caracteres que varían de longitud de una fila a otra, hasta una cierta longitud máxima. El estándar ANSI/ISO solamente especifica cadenas de longitud fija, que son rellenas por la derecha con caracteres en blanco adicionales.
- *Imports monetarios.* Muchos productos SQL soportan un tipo MONEY o CURRENCY, que se almacena generalmente como número decimal o en coma

Tipo de datos	Descripción
CHAR(<i>long</i>)	Cadenas de caracteres de longitud fija.
CHARACTER(<i>long</i>)	
INTEGER	Números enteros.
INT	
SMALLINT	Números enteros pequeños.
NUMERIC(<i>precision</i> , <i>escala</i>)	Números decimales.
DECIMAL(<i>precision</i> , <i>escala</i>)	
DEC(<i>precision</i> , <i>escala</i>)	
FLOAT(<i>precision</i>)	Números en coma flotante.
REAL	Números en coma flotante de baja precisión.
DOUBLE PRECISION	Números en coma flotante de alta precisión.

Tabla 5.3. Tipos de datos SQL ANSI/ISO

Tipo de datos ¹	DB2 y SQL/DS	Oracle ²	Ingres	Rdb/VMS ³
Caracteres de longitud fija	CHARACTER(n)	CHAR(n)	CHAR(n)	CHAR(n)
Caracteres de longitud variable	VARCHAR(n)	VARCHAR(n)	VARCHAR(n) LONG VARCHAR	VARCHAR(n)
Texto largo	LONG VARCHAR	LONG		
Entero	SMLALINT INTEGER	INTEGER1 INTEGER2 INTEGER4	SMALLINT INTEGER QUADWORD(n)	TINYINT SMALLINT INT
Decimal	DECIMAL(p,s)	NUMBER(p,s)	DECIMAL(p,s)	DECIMAL(p,s)
Monetario		MONEY		MONEY(p,s)
Coma flotante	FLOAT(p) REAL DOUBLE PRECISION	FLOAT4 FLOAT8	FLOAT(p) REAL DOUBLE PRECISION	FLOAT FLOAT
Fecha/hora	DATE ⁴	DATE ⁵	DATE	DATETIME DATE TIME TIMESTAMP
Booleano				BIT
Flujo de bytes			RAW LONG RAW	BINARY(n) VARBINARY(n) IMAGE
Otros	GRAPHIC(n) VARGRAPHIC(n)		LONG VARGRAPHIC	SERIAL SYSNAME USER_TYPE_NAME

¹ Muchos de los productos también soportan los tipos ANSI/ISO como sinónimos de sus tipos propios.

² Oracle almacena todos los números utilizando su propio formato interno con 40 dígitos de precisión.

³ Rdb/VMS soporta realmente tipos de datos adicionales que no están disponibles en VAX SQL.

Tabla 5.4. Tipos de datos soportados por productos SQL populares.

		Informix	SQL Server y Sybase	OS/2 EE	dBASE IV	SQLBase ⁶
		CHAR(n)	CHAR(n)	CHAR(n)	CHAR(n)	
						VARCHAR(n)
						LONG VARCHAR
						TEXT
						TINYINT SMALLINT INTEGER NUMBER
						DECIMAL(p,s)
						DECIMAL(p,s)
						DECIMAL(p,s)
						DECIMAL(p,s)
						DECIMAL(p,s)
						DECIMAL(p,s)
						LOGICAL

⁴ El tipo de datos DATE de Oracle es realmente un instante. Oracle dispone de funciones internas que pueden utilizarse para aislar las partes de fecha y hora.

⁵ El tipo de datos DATE de Ingres almacena tanto instantes como duraciones. Ingres soporta aritmética de fecha «con significado» sobre ambos tipos.

⁶ SQLBase almacena todos los números utilizando su propio formato interno con 22 dígitos de precisión.

Tabla 5.4. Tipos de datos soportados por productos SQL populares (continuación).

flotante. El tener un tipo monetario distinto permite al DBMS formatear adecuadamente los importes monetarios cuando son visualizados.

- *Fechas y horas.* Soportar valores para fechas y horas es también habitual en los productos SQL, aunque los detalles varían ampliamente de un producto a otro. Generalmente se soportan variadas combinaciones de fechas, horas, instantes, intervalos de tiempo y aritmética de fecha y hora.

■ *Datos Booleanos.* Algunos productos SQL (incluyendo dBASE IV) soportan los valores lógicos (TRUE o FALSE) como un tipo explícito.

■ *Texto extenso.* Varias bases de datos basadas en SQL soportan columnas que almacenan cadenas de texto extensas (típicamente de hasta 32.000 ó 65.000 caracteres, o entre 10 ó 20 páginas escritas a un solo espacio). El DBMS restringe generalmente el uso de estas columnas en consultas y búsquedas interactivas.

■ *Flyjos de bytes no estructurados.* Oracle y otros varios productos premiten almacenar y recuperar secuencias de bytes de longitud variable sin estructurar. Las columnas que contienen estos datos se utilizan para almacenar imágenes de video comprimidas, código ejecutable y otros tipos de datos sin estructurar.

■ *Caracteres asiáticos.* DB2 soporta cadenas de longitud fija y de longitud variable de caracteres de 16 bits utilizados para representar caracteres Kanji y otros caracteres asiáticos. Sin embargo, la búsqueda y ordenación de estos tipos GRAPHIC y VARGRAPHIC no está permitida.

La Tabla 5.4 (págs. 72-73) resume los tipos de datos soportados por varios productos de base de datos populares basados en SQL.

Diferencias de tipos de datos

Las diferencias entre los tipos de datos ofrecidos por las diversas implementaciones SQL es una de las barreras prácticas a la portabilidad de las aplicaciones basadas en SQL. Los datos fecha/hora proporcionan un excelente ejemplo de estas diferencias. DB2, por ejemplo, soporta tres tipos de datos diferentes fecha/hora:

- DATE, que almacena una fecha tal como junio 30, 1991,
- TIME, que almacena una hora del día tal como 12:30 P.M., y
- TIMESTAMP, que es un instante específico en la historia, con una precisión de hasta el nanosegundo.

Las fechas y horas concretas pueden ser especificadas como constantes cadenas, y se soporta aritmética de fechas. He aquí un ejemplo de una consulta válida utilizando fechas DB2, suponiendo que la columna CONTRATO contiene un dato de tipo DATE:

```
SELECT NOMBRE, CONTRATO
      FROM REVENTAS
     WHERE CONTRATO >= '05/30/1989' + 15 DAYS
```

La OS/2 Extended Edition soporta los tres mismos tipos de datos que DB2, pero no soporta aritmética de fechas, por lo que la consulta anterior debe ser reescrita de este modo:

```
SELECT NOMBRE, CONTRATO
      FROM REVENTAS
     WHERE CONTRATO >= '06/14/1989'
```

SQL Server solamente proporciona un único tipo de dato fecha/hora, llamado DATETIME, que se asemeja estrechamente al tipo de datos TIMESTAMP de DB2. Si CONTRATO contuviera datos de tipo DATETIME, SQL Server podría aceptar la versión OS/2 Extended Edition de la consulta. Como no se especifica una hora concreta en la fecha junio 14, 1989, SQL pone por omisión la hora de medianoche. Por tanto, la consulta de SQL Server significa *realmente*:

```
SELECT NOMBRE, CONTRATO
      FROM REVENTAS
     WHERE CONTRATO >= '06/14/1989 12:00AM'
```

Si la fecha de contrato de un vendedor estuviera almacenada en la base de datos como el mediodía de junio 14, 1989, el vendedor no sería incluido en los resultados de la consulta en SQL Server, pero habría sido incluido en los resultados de OS/2 Extended Edition o DB2 (ya que en ellos sólo se almacenaría la fecha). SQL Server también soporta aritmética de fecha mediante un conjunto de funciones internas. Por tanto la consulta de estilo DB2 también podría ser especificada de este modo:

```
SELECT NOMBRE, CONTRATO
      FROM REVENTAS
     WHERE CONTRATO >= DATEADD(DAY, 15, '05/30/1989')
```

que, naturalmente, es considerablemente diferente a la sintaxis de DB2. Oracle también soporta datos fecha/hora, con un único tipo de datos llamado DATE. Al igual que el tipo de DATETIME de SQL Server, un DATE de Oracle es, de hecho, un instante. También igual que SQL Server, la parte de hora de un valor hasta el nanosegundo.

DATE en Oracle toma por omisión el valor de la medianoche si no se especifica explícitamente una hora. El formato de fecha de omisión en Oracle es diferente de los formatos de DB2 y SQL Server, por lo que la versión Oracle de la consulta pasaría a ser:

```
SELECT NOMBRE, CONTRATO
  FROM REVENTAS
 WHERE CONTRATO >= '14-JUN-89' + 15
```

Oracle también soporta aritmética de fechas limitada, de modo que la consulta de estilo DB2 también puede ser especificada, pero sin la palabra clave DAYS:

```
SELECT NOMBRE, CONTRATO
  FROM REVENTAS
 WHERE CONTRATO >= '30-MAY-89' + 15
```

Como ilustran estos ejemplos (con total detalle), las sutiles diferencias en los tipos de datos entre los variados productos SQL conducen a algunas diferencias significativas en la sintaxis de la sentencia SQL. Incluso pueden hacer que la misma consulta SQL produzca resultados ligeramente diferentes sobre diferentes sistemas de gestión de base de datos. La tan alabada portabilidad de SQL es bastante cierta, pero solamente a un nivel general. Una aplicación puede transferirse de una base de datos SQL a otra. Sin embargo, las sutiles variaciones en las implementaciones SQL significan que los tipos de datos y las sentencias SQL deben casi siempre ser ajustadas de algún modo en el proceso. La portabilidad transparente de las aplicaciones basadas en SQL siguen siendo un objetivo, pero no una realidad.

Constantes

En algunas sentencias SQL, un valor de datos numérico, de caracteres o de fecha debe ser expresado en forma textual. Por ejemplo, en esta sentencia INSERT, que añade un vendedor a la base de datos:

```
INSERT INTO REVENTAS (NUM_EMPL, NOMBRE, CUOTA, CONTRATO, VENTAS)
VALUES (115, 'Dennis Irving', 175000.000, '21-JUN-90', 0.00)
```

el valor para cada columna en la fila recién insertada se especifica en la cláusula VALUES. Los valores de datos constantes también se utilizan en las expresiones, tal como por ejemplo en la sentencia SELECT:

```
SELECT CIUDAD
  FROM OFICINA
 WHERE OBJETIVO > (1.1 * VENTAS) + 10000.00
```

El estándar SQL ANSI/ISO especifica el formato de las constantes numéricas y de caracteres, o *literales*, que representan valores de datos específicos. Estos convenios son seguidos por la mayoría de las implementaciones SQL.

Constantes numéricas

Las constantes enteras y decimales (también denominadas *literales numéricos exactos*) se escriben como números decimales ordinarios dentro de las sentencias SQL, con un signo más o menos opcional delante.

```
21 -375 2000.00 +497500.8778
```

No se debe poner una coma entre los dígitos de una constante numérica, y no todos los dialectos de SQL permiten el signo más inicial, por lo que es mejor evitarlo. Para los datos monetarios, la mayoría de las implementaciones SQL utilizan simplemente constantes enteras o decimales, aunque algunas permiten que la constante sea especificada con un símbolo de moneda:

```
$0.75 $5000.00 $-567.89
```

Las constantes en coma flotante (también llamadas *literales numéricos aproximados*) se especifican utilizando la notación E comúnmente hallada en lenguajes de programación tales como C y FORTRAN. He aquí algunas constantes SQL en coma flotante válidas:

```
1.5E3 -3.14159E1 2.5E-7 0.783926E21
```

La E se lee «por diez elevado a», de modo que la primera constante es «1.5 por diez elevado a tres», o 1500.

Constantes de cadena

El estándar ANSI/ISO especifica que las constantes SQL de caracteres vayan encerradas entre comillas simples ('...'), como en los siguientes ejemplos:

```
'Jones, John J.' 'New York' 'Oeste'
```

Si una comilla simple debe incluirse en el texto constante, ésta se escribe como dos caracteres comilla simple consecutivos. Por tanto este valor constante:

```
'I can't'
```

se convierte en la cadena de siete caracteres "I can't".

Algunas implementaciones SQL, tales como dBASE IV, SQL Server e Informix, aceptan constantes de cadenas encerradas entre dobles comillas ("...").

```
"'Jones, John J'" "New York" "Oeste"
```

Desgraciadamente, las dobles comillas imponen problemas de portabilidad con otros productos SQL, incluyendo algunos problemas de portabilidad únicos con SQL/DS. SQL/DS permite nombres de columnas que contengan blancos y otros caracteres especiales (violando el estándar ANSI/ISO). Cuando estos caracteres aparecen como nombres en una sentencia SQL, deben ir entre comillas dobles. Por ejemplo, si la columna NOMBRE de la tabla REPENTAS fuera en realidad "NOMBRE COMPLETO" en una base de datos SQL/DS, esta sentencia SELECT sería válida.

```
SELECT "NOMBRE COMPLETO", VENTAS, CUOTA
  FROM REPENTAS
 WHERE "NOMBRE COMPLETO" = 'Jones, John J.'
```

Constantes de fecha y hora

En productos SQL que soportan datos fecha/hora, los valores constantes para las fechas, horas e intervalos de tiempo se especifican como constantes de cadenas de caracteres. El formato de estas constantes varía de un DBMS a otro. Incluso se producen adicionales debido a las diferencias en el modo de las fechas y las horas se escriben en diferentes países.

Tanto DB2 como OS/2 Extended Edition soportan varios formatos internacionales diferentes para fechas, horas y constantes de tiempo, tal como se muestra en la Tabla 5.5. La elección del formato se efectúa cuando se instala el DBMS. DB2 (pero no actualmente OS/2 Extended Edition) también soporta duraciones especificadas como constantes «especiales», como en este ejemplo:

```
CONTRATO + 30 DIAS
```

Observe sin embargo, que una duración no puede ser almacenada en la base de datos, ya que DB2 no tiene un tipo de datos DURACION explícito. SQL Server también soporta datos fecha/hora, y acepta una variedad de formatos diferentes para las constantes de fecha y hora. El DBMS acepta automáticamente todas las formas alternativas, y se pueden entrezclar si así se quiere. He aquí algunos ejemplos de constantes de fecha legales en SQL Server:

```
March 15, 1990 Mar 15 1990 3/15/1990 3-15-90 1990 MAR 15
```

Nombre de formato	Formato DATE	Ejemplo de DATE	Formato TIME	Ejemplo de TIME
Americano	mm/dd/aaaa dd.mm.aaaa	5/19/1960 19.5.1960	hh:mm am/pm hh:mm:ss	2:18 PM 14.18.08
Europeo	dd.mm.yyyy	1960-5-19	hh:mm:ss	14:18:08
Japonés	aaaa-mm-dd	1960-5-19	hh:mm:ss	14.18.08
ISO	aaaa-mm-dd	1960-5-19	hh:mm:ss	14.18.08

TIMESTAMP formato: aaaa-mm-dd hh:mm:ss.mmmmmm
TIMESTAMP ejemplo: 1960-05-19-14.18.08.048632

Tabla 5.5. Formatos de fecha y hora en SQL de IBM.

y he aquí algunas constantes de tiempo legales:

```
15:30:25 3:30:25 PM 3:30:25 pm 3 PM
```

Las fechas y horas en Oracle también se escriben como constantes de cadena de caracteres, utilizando este formato:

```
15-MAR-90
```

También se puede utilizar la función interna de Oracle TO_DATE() para convertir constantes de fecha escritas en otros formatos, como en este ejemplo:

```
SELECT NOMBRE, EDAD
  FROM REPENTAS
 WHERE CONTRATO = TO_DATE('JUN 14 1989', 'MON DD YYYY')
```

Constantes simbólicas

Además de las constantes suministradas por el usuario, el lenguaje SQL incluye constantes simbólicas especiales que devuelven valores de datos mantenidos por el propio DBMS. La constante simbólica de OS/2 Extended Edition CURRENT DATE produce el valor de la fecha actual y puede ser utilizada en consultas como las siguientes, que listan los vendedores cuya fecha de contrato está aún en el futuro.

```
SELECT NOMBRE, CONTRATO
  FROM REPENTAS
 WHERE CONTRATO > CURRENT DATE
```

El estándar ANSI/ISO especifica solamente una única constante simbólica (la constante USER descrita en el Capítulo 15), pero la mayoría de los productos SQL proporcionan muchas más. Las constantes simbólicas que se encuentran habitualmente en las implementaciones SQL están listadas en la Tabla 5.6. Generalmente, una constante simbólica puede aparecer en una sentencia SQL en cualquier lugar en el que pudiera aparecer una constante ordinaria del mismo tipo de dato.

Algunos productos SQL, incluyendo SQL Server, proporcionan acceso a valores del sistema mediante funciones internas en lugar de con constantes simbólicas. La versión SQL Server de la consulta anterior es:

```
SELECT NOMBRE, CONTRATO
  FROM REVENTAS
 WHERE CONTRATO > GETDAT EO;
```

Las funciones internas se describen posteriormente en este capítulo.

Constante	Descripción
USER	El nombre de usuario bajo el cual se está accediendo actualmente a la base de datos (DB2, SQL/DS, Oracle, VAX SQL, SQLBase y también especificado en el estándar ANSI/ISO)
CURRENT DATE	La fecha actual (DB2, SQL/DS, OS/2 EE, SQLBase)
CURRENT TIME	La hora actual del día (DB2, SQL/DS, OS/2 EE, SQLBase)
CURRENT TIMESTAMP	La fecha y hora actuales (DB2, SQL/DS, OS/2 EE)
CURRENT TIMEZONE	Una duración temporal que especifica la diferencia entre esta zona horaria y la GMT (el horario de Greenwich) (DB2, SQL/DS, SQLBase)
SYSDATE	La fecha y hora actuales (Oracle, SQL Base)
ROWNUM	El «número ID de fila» interno de una fila (Oracle)
ROWID	El «número ID de fila» interno de una fila (SQLBase)

Tabla 5.6. Constantes simbólicas SQL.

Expresiones

Las expresiones se utilizan en el lenguaje SQL para calcular valores que son de una base de datos y para calcular valores utilizados en la búsqueda de la base de

datos. Por ejemplo, esta consulta calcula las ventas de cada oficina como porcentaje de su objetivo:

```
SELECT CIUDAD, OBJETIVO, VENTAS, (VENTAS/OBJETIVO) * 100
  FROM OFICINAS
```

y esta consulta lista las oficinas cuyas ventas son superiores a \$50.000 por encima del objetivo:

```
SELECT CIUDAD
  FROM OFICINAS
 WHERE VENTAS > OBJETIVO + 50000.00
```

El estándar SQL ANSI/ISO especifica cuatro operaciones aritméticas que pueden ser utilizadas en expresiones: suma ($X + Y$), resta ($X - Y$), multiplicación ($X * Y$) y división (X / Y). También se pueden utilizar paréntesis para formar expresiones más complicadas, como la siguiente:

```
(VENTAS * 1.05) - (OBJETIVO * .95)
```

Estrictamente hablando, los paréntesis no son necesarios en esta consulta, ya que el estándar ANSI/ISO especifica que la multiplicación y la división tienen una precedencia superior a la suma y la resta. Sin embargo, deberían utilizarse siempre los paréntesis para lograr que las expresiones no sean ambiguas, ya que diferentes dialectos de SQL pueden utilizar reglas diferentes. Además los paréntesis incrementan la legibilidad de la sentencia y hacen que las sentencias de SQL programado sean más fáciles de mantener.

El estándar ANSI/ISO también especifica conversión automática de tipos de datos desde números enteros a decimales, y desde números decimales a números en coma flotante, según se precise. Por tanto estos tipos de datos se pueden mezclar en una expresión numérica. Muchas implementaciones SQL soportan otros operadores y permiten operaciones sobre datos de caracteres y de fechas. DB2, por ejemplo, soporta un operador concatenación de cadenas, escrito como dos barras verticales consecutivas (||). Si dos columnas llamadas NOMBRE y APELLIDO contienen los valores «Jim» y «Jackson», entonces esta expresión de DB2:

```
('Mr./Mrs. ' || NOMBRE || ' ' || APELLIDO)
```

produce la cadena «Mr./Mrs Jim Jackson». Como ya se mencionó, DB2 también soporta la suma y resta de datos DATE, TIME y TIMESTAMP, para aquellas ocasiones en que estas operaciones tienen sentido.

Funciones internas

Aunque el estándar ANSI/ISO no las especifica, la mayoría de las implementaciones SQL incluyen una serie de *funciones internas* útiles. Estas utilidades proporcionan con frecuencia facilidades de conversión de tipo de datos. Por ejemplo, las funciones internas de DB2 `MONTH()` y `YEAR()` aceptan un valor `DATE` o `TIMESTAMP` como entrada y devuelven un entero que es el mes o el año del valor. Esta consulta lista el nombre y el mes del contrato de cada vendedor de nuestra base de datos ejemplo:

```
SELECT NOMBRE, MONTH(contrato)
  FROM REPVENTAS
```

y esta otra lista todos los vendedores contratados en 1988:

```
SELECT NOMBRE, MONTH(contrato)
  FROM REPVENTAS
 WHERE YEAR(contrato) = 1988
```

Las funciones internas también suelen utilizarse para reformatear los datos. La función interna de Oracle `TO_CHAR()`, por ejemplo, acepta un tipo de datos `DATE` y una especificación de formato como argumentos, y devuelve una cadena que contiene una versión formateada de la fecha. En los resultados producidos por esta consulta:

```
SELECT NOMBRE, TO_CHAR(contrato, 'DAY MONTH DD, YYYY')
  FROM REPVENTAS
```

las fechas de contrato tienen todas el formato «Miércoles Junio 14, 1989» debido al uso de la función interna.

En general, una función interna puede ser especificada en una expresión SQL en cualquier lugar en que una constante del mismo tipo de datos pueda ser especificada. Las funciones internas soportadas por los dialectos SQL más populares son demasiado numerosas para listarlas aquí. Los dialectos SQL de IBM incluyen alrededor de dos docenas de funciones internas, Oracle soporta un conjunto diferente de alrededor de dos docenas de funciones internas, y SQL Server dispone de varias docenas.

Falta de datos (valores NULL)

Puesto que una base de datos es generalmente un modelo de una situación del mundo real, ciertos datos pueden inevitablemente faltar, ser desconocidos o no

ser aplicables. En la base de datos ejemplo, la columna CUOTA en la tabla REPVENTAS contiene el objetivo de ventas de cada vendedor. Sin embargo el vendedor más reciente aún no tiene asignada una cuota; este dato falta para esa fila de la tabla. Podríamos estar tentados de colocar un cero en la columna para este vendedor, pero ésa no sería un reflejo preciso de la situación. El vendedor no tiene una cuota cero; la cuota simplemente es «desconocida aún».

Análogamente, la columna DIRECTOR en la tabla REPVENTAS contiene el número de empleado de cada director de vendedores. Pero Sam Clark, el Vicepresidente de Ventas, no tiene director en la organización de ventas. Esta columna no es aplicable a Sam. De nuevo, se podría pensar en introducir un cero, o un 9999 en la columna, pero ninguno de estos valores sería realmente el número de empleado del jefe de Sam. Ningún valor de datos es aplicable a esta fila.

SQL soporta explícitamente los datos que faltan, son desconocidos o son inaplicables, a través del concepto de *valor nulo*. Un valor nulo es un *indicador* que dice a SQL (y al usuario) que el dato falta o no es aplicable. Por conveniencia, un dato que falta normalmente se dice que tiene el valor NULL, pero el valor NULL no es un valor de dato real como 0,473,83 o «Sam Clark». En vez de ello, es una señal, o un recordatorio de que el valor de datos falta o es desconocido. La Figura 5.3 muestra el contenido de la tabla REPVENTAS. Observe que los valores de CUOTA y OFICINAREP para la fila de Tom Snyder y el valor DIRECTOR para la fila de Sam Clark en la tabla contienen todos valores NULL.

Tabla REPVENTAS

NUM_EMPL	NOMBRE	EDAD	OFICINAREP	TITULO	CONTRATO	DIR	CUOTA	VENTAS
105	Bill Adams	37	13	Rep Ventas	02-12-1988	104	\$350,000.00	\$367,911.00
109	Mary Jones	31	11	Rep Ventas	10-12-1989	106	\$350,000.00	\$392,725.00
102	Sue Smith	48	21	Rep Ventas	12-10-1986	108	\$350,000.00	\$3474,050.00
106	Sam Clark	52	11	VP Ventas	01-14-1988	NULL	\$275,000.00	\$299,912.00
104	Bob Smith	33	12	Dir Ventas	05/19/1987	106	\$200,000.00	\$142,594.00
101	Dan Roberts	45	12	Rep Ventas	10/20/1988	104	\$300,000.00	\$305,673.00
110	Tom Snyder	41	NULL	Rep Ventas	01/13/1990	101	NULL	\$75,585.00
108	Larry Fitch	62	21	Dir Ventas	10/17/1989	106	\$350,000.00	\$361,865.00
103	Paul Cruz	29	12	Rep Ventas	10/14/1988	108	\$275,000.00	\$286,775.00
107	Nancy Algesti	49	22	Rep Ventas	11/14/1988	108	\$300,000.00	\$186,042.00

Valor desconocido
Valor no aplicable
Valor desconocido

Figura 5.3. Valores NULL en la tabla REPVENTAS.

En muchas situaciones los valores NULL requieren un manejo especial por parte del DBMS. Por ejemplo, si el usuario solicita la suma de la columna CUOTA,

6 *Consultas simples*

¿cómo tendría que manejar el DBMS la falta de datos cuando calcule la suma? La respuesta viene dada por un conjunto de reglas especiales que gobernan el manejo de los valores NULL en las diferentes sentencias y cláusulas SQL. Debido a estas reglas, algunas autoridades importantes en bases de datos opinan que los valores NULL no deberían ser utilizados. Otros, entre los que se incluye el Dr. Codd, han abogado por el uso de múltiples valores NULL, con distintos indicadores para «desconocido» y «no aplicable».

Independientemente de los debates académicos, los valores NULL forman parte del estándar SQL ANSI/ISO, y son soportados en virtualmente todos los productos SQL comerciales. Además juegan un papel importante y práctico en las bases de datos SQL de producción. Las reglas especiales que se aplican a los valores NULL (y los casos en los que los valores NULL son manejados inconsistentemente por varios productos SQL) son señalados en las secciones relevantes de este libro.

Resumen

Este capítulo ha descrito los elementos básicos del lenguaje SQL. La estructura básica de SQL puede ser resumida tal como se expresa seguidamente:

- El lenguaje SQL incluye unas treinta sentencias, cada una formada por un verbo y una o más cláusulas. Cada sentencia efectúa una única función específica.
 - Las bases de datos basadas en SQL pueden almacenar varios tipos de datos entre los que se incluyen texto, enteros, números decimales, números de coma flotante y, generalmente, varios tipos más específicos del vendedor.
 - Las sentencias SQL pueden incluir expresiones que combinen nombres de columnas, constantes, y funciones internas, utilizando operadores aritméticos y otros operadores específicos del vendedor.
 - Las variaciones de los tipos de datos en las constantes y en las funciones internas hacen que la portabilidad de las sentencias SQL sea más difícil de lo que en un principio pudiera parecer.
 - Los valores NULL proporcionan un modo sistemático de manejar los datos que faltan o son implicados dentro del lenguaje SQL.

La sentencia SELECT

En muchos sentidos, las consultas son el corazón del lenguaje SQL. La sentencia SELECT, que se utiliza para expresar consultas SQL, es la más potente y compleja de las sentencias de SQL. A pesar de las muchas opciones permitidas por la sentencia SELECT, es posible comenzar de forma sencilla y luego pasar a elaborar consultas más complejas. Este capítulo discute las consultas SQL más simples; aquellas que recuperan datos de una única tabla en la base de datos.

La sentencia `SELECT` recupera datos de una base de datos y los devuelve en forma de resultados de la consulta. Ya hemos visto muchos ejemplos de la sentencia `SELECT` en el rápido repaso presentado en el Capítulo 2. He aquí varios ejemplos más de consultas que recuperan información referente a las oficinas de ventas:

Lista de las oficinas de ventas con sus objetivos y ventas reales.

SELECT CIUDAD, OBJETIVO, VENTAS
 FROM OFICINAS

CIUDAD	OBJETIVO	VENTAS
Denver	\$300,000.00	\$186,042.00
New York	\$375,000.00	\$892,637.00
Chicago	\$300,000.00	\$735,042.00
Atlanta	\$350,000.00	\$367,911.00
Los Angeles	\$725,000.00	\$835,915.00

Ciudad	Objetivo	Ventas
Denver	\$300,000.00	\$186,042.00
New York	\$375,000.00	\$892,637.00
Chicago	\$300,000.00	\$735,042.00
Atlanta	\$350,000.00	\$367,911.00
Los Angeles	\$725,000.00	\$335,915.00

Lista de las oficinas de ventas en la región. Este con sus objetivos y ventas.

```
SELECT CIUDAD, OBJETIVO, VENTAS
  FROM OFICINAS
 WHERE REGION = 'Oeste'
    ORDER BY CIUDAD
```

Lista de las oficinas de ventas de la región. Este cuyas ventas exceden a sus objetivos, ordenadas en orden alfabético por ciudad.

```
SELECT CIUDAD, OBJETIVO, VENTAS
  FROM OFICINAS
 WHERE REGION = 'Oeste'
   AND VENTAS > OBJETIVO
    ORDER BY CIUDAD
```

CIUDAD	OBJETIVO	VENTAS
Atlanta	\$350,000.00	\$367,911.00
New York	\$575,000.00	\$692,637.00

¿Cuáles son los objetivos y ventas promedio para las oficinas de la región Este?

```
SELECT AVG(OBJETIVO), AVG(VENTAS)
  FROM OFICINAS
 WHERE REGION = 'Este'
    AVG(OBJETIVO) AVG(VENTAS)
```

AVG(OBJETIVO)	AVG(VENTAS)
\$575,000.00	\$598,550.00

Para consultas sencillas, la petición en lenguaje inglés y la sentencia SELECT de SQL son muy similares. Cuando las peticiones se hacen más complejas, deben utilizarse características adicionales de la sentencia SELECT para especificar la consulta con precisión.

La Figura 6.1 muestra el formato completo de la sentencia SELECT, que consta de seis cláusulas. Las cláusulas SELECT y FROM de la sentencia son necesarias. Las cuatro cláusulas restantes son opcionales. Se incluyen en la sentencia SELECT solamente cuando se desean utilizar las funciones que proporcionan. La función de cada cláusula está resumida a continuación:

- La cláusula SELECT lista los datos a recuperar por la sentencia SELECT. Los ítems pueden ser columnas de la base de datos o columnas a calcular por SQL.

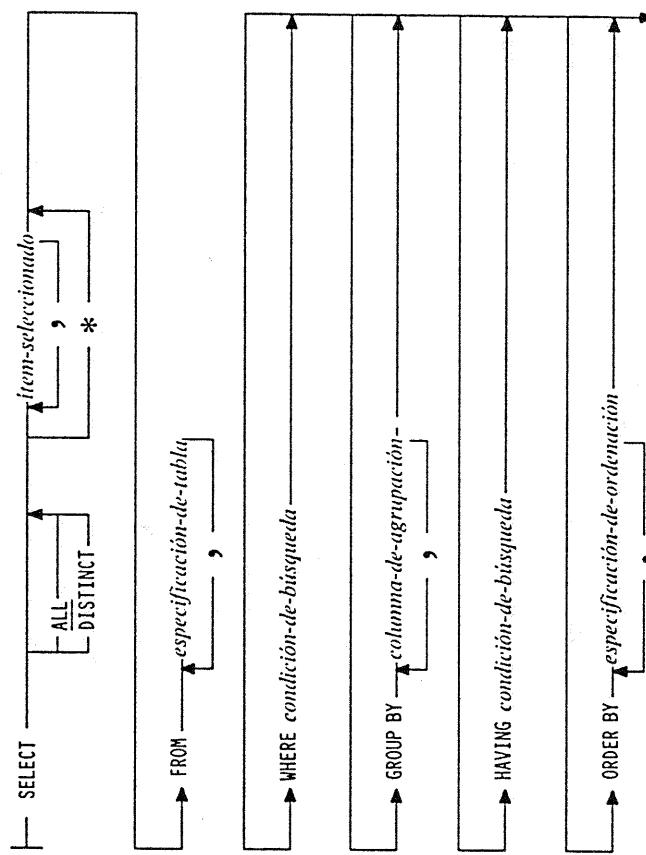


Figura 6.1. Diagrama sintáctico de la sentencia SELECT.

cuando efectúe la consulta. La cláusula SELECT se describe en secciones posteriores de este capítulo.

- La cláusula FROM lista las tablas que contienen los datos a recuperar por la consulta. Las consultas que extraen sus datos de una única tabla se describen en este capítulo. Las consultas más complejas que combinan datos de dos o más tablas se discuten en el Capítulo 7.

- La cláusula WHERE dice a SQL que incluya sólo ciertas filas de datos en los resultados de la consulta. Se utiliza una *condición de búsqueda* para especificar las filas deseadas. Los usos básicos de la cláusula WHERE se describen posteriormente en este capítulo. Aquellos que implican a subconsultas se discuten en el Capítulo 9.

- La cláusula GROUP BY especifica una consulta sumaria. En vez de producir una fila de resultados por cada fila de datos de la base de datos, una consulta sumaria agrupa todas las filas similares y luego produce una fila sumaria de resultados para cada grupo. Las consultas sumarias se describen posteriormente en el Capítulo 8.

■ La cláusula HAVING dice a SQL que incluya sólo ciertos grupos producidos por la cláusula GROUP BY en los resultados de la consulta. Al igual que la cláusula WHERE, utiliza una condición de búsqueda para especificar los grupos deseados. La cláusula HAVING se describe también en el Capítulo 8.

■ La cláusula ORDER BY ordena los resultados de la consulta en base a los datos de una o más columnas. Si se omite, los resultados de la consulta no aparecen ordenados. La cláusula ORDER BY se describe posteriormente en este capítulo.

La cláusula SELECT

La cláusula SELECT que inicia cada sentencia SELECT especifica los ítems de datos a recuperar por la consulta. Los ítems se especifican generalmente mediante una lista de selección, una lista de ítems de selección separados por comas. Cada ítem de selección de la lista genera una única columna de resultados de consulta, en orden de izquierda a derecha. Un ítem de selección puede ser:

- Un nombre de columna, identificando una columna de la tabla designada en la cláusula FROM. Cuando un nombre de columna aparece como ítem de selección, SQL simplemente toma el valor de esa columna de cada fila de la tabla de base de datos y lo coloca en la fila correspondiente de los resultados de la consulta.
- Una constante, especificando que el mismo valor constante va a aparecer en todas las filas de los resultados de la consulta.
- Una expresión SQL, indicando que SQL debe calcular el valor a colocar en los resultados, según el estilo especificado por la expresión.

Cada tipo de ítem de selección se describe posteriormente en este capítulo.

La cláusula FROM

La cláusula FROM consta de la palabra clave FROM, seguida de una lista de especificaciones de tablas separadas por comas. Cada especificación de tabla identifica una tabla que contiene datos a recuperar por la consulta. Estas tablas se denominan *tablas fuente* de la consulta (y de la sentencia SELECT), ya que constituyen la fuente de todos los datos que aparecen en los resultados. Todas las consultas de este capítulo tienen una única tabla fuente, y todas las cláusulas FROM contienen un solo nombre de tabla.

Resultados de consultas

El resultado de una consulta SQL es siempre una tabla de datos, semejante a las tablas de la base de datos. Si usted escribe una sentencia SELECT utilizando SQL interactivo, el DBMS visualizará los resultados de la consulta en forma tabular sobre la pantalla del computador. Si un programa envía una consulta al DBMS utilizando SQL programado, la tabla de resultados de la consulta se devuelve al programa. En cualquier caso, los resultados tienen siempre el mismo formato tabular de fila/columna que las tablas efectivas de la base de datos, tal como se muestra en la Figura 6.2. Generalmente los resultados de la consulta formarán una tabla con varias columnas y varias filas. Por ejemplo, esta consulta produce una tabla de tres columnas y varias filas (ya que pide tres ítems de datos) y diez filas (ya que hay diez vendedores):

Lista los nombres, oficinas y fechas de contrato de todos los vendedores.

```
SELECT NOMBRE, OFICINAREP, CONTRATO
FROM REVENTAS
```

NOMBRE	OFICINAREP	CONTRATO
Bill Adams	13	12-FEB-88
Mary Jones	11	12-OCT-89
Sue Smith	21	10-DIC-86
Sam Clark	11	14-JUN-88
Bob Smith	12	19-MAY-87
Dan Roberts	12	20-OCT-86
Tom Snyder	NULL	13-ENE-90
Larry Fitch	21	12-OCT-89
Paul Cruz	12	01-MAR-87
Nancy Angelli	22	14-NOV-88

En contraste, la consulta siguiente produce una única fila, ya que sólo hay un vendedor que tenga el número de empleado solicitado. Aún cuando esta única fila de resultados tenga un aspecto menos «tabular» que los resultados multifila, SQL sigue considerándolo una tabla de tres columnas y una fila.

¿Cuál es el nombre, cuota y ventas del empleado número 107?

```
SELECT NOMBRE, CUOTA, VENTAS
FROM REVENTAS
WHERE NUMERPL = 107
```

NOMBRE	CUOTA	VENTAS
Nancy Angelli	\$300,000.00	\$186,042.00

```
SQL interactivo
SELECT CITY, REGION
FROM OFFICES
CITY   REGION
====  ====
Denver  Oeste
New York Este
Chicago  Este
Atlanta  Este
Los Angeles Oeste
```

Consulta

CIUDAD	REGION
Denver	Oeste
New York	Este
Chicago	Este
Atlanta	Este
Los Angeles	Oeste

Base de datos

DBMS

Resultados de la consulta

```
Programa
SQL programado
```

Figura 6.2. Estructura tabular de los resultados de una consulta SQL.

En algunos casos los resultados de la consulta pueden ser un único valor, como en el siguiente ejemplo:

«Cuáles son las ventas promedio de nuestros vendedores?

```
SELECT AVG(VENTAS)
FROM REPVENTAS
AVG(VENTAS)
-----
$289,353.20
```

Este resultado sigue siendo una tabla, aunque sea tan pequeña que sólo conste de una fila y una columna. Finalmente, es posible que una consulta produzca cero filas de resultados, como en este ejemplo:

```
Lista el nombre y fecha de contrato de cualquier vendedor cuyas ventas sean
superiores a $500,000.
SELECT NOMBRE, CONTRATO
FROM REPVENTAS
WHERE VENTAS > 500000.00
NOMBRE  CONTRATO
-----
```

Incluso en esta situación, el resultado de la consulta sigue siendo una tabla. Se trata de una tabla vacía con dos columnas y cero filas.

Observe que el soporte SQL para la falta de datos se extiende también a los resultados de la consulta. Si un ítem de datos en la base de datos tiene un valor NULL, el valor NULL aparece en los resultados cuando se recupera el ítem de datos. Por ejemplo, la tabla REPVENTAS contiene valores NULL en las columnas CUOTA y DIRECTOR. La siguiente consulta devuelve estos valores NULL en las columnas segunda y tercera de los resultados:

Lista de los vendedores, sus cuotas y sus directores.

```
SELECT NOMBRE, CUOTA, DIRECTOR
FROM REPVENTAS
NOMBRE      CUOTA    DIRECTOR
-----  -----
Bill Adams  $350,000.00    104
Mary Jones   $300,000.00    106
Sue Smith    $350,000.00    108
Sam Clark    $275,000.00    NULL
Bob Smith    $200,000.00    106
Dan Roberts  $300,000.00    104
Tom Snyder    NULL        101
Larry Fitch   $350,000.00    106
Paul Cruz     $275,000.00    104
Nancy Angelli $300,000.00    108
```

El hecho de que una consulta SQL produzca siempre una tabla de dato es muy importante. Significa que los resultados de la consulta pueden volverse a almacenar en la base de datos como una tabla. Significa que los resultados de dos consultas similares pueden combinarse para formar una tabla mayor de resultados de consulta. Finalmente, significa que los resultados de la consulta pueden ser objetivo ellos mismos de consultas adicionales. La estructura tabular de una base de datos relacional tiene por tanto una relación muy fuerte con las facilidades de consultas relacionales de SQL. Las tablas pueden ser consultadas, y las consultas producen tablas.

Consultas sencillas

Las consultas SQL más sencillas solicitan columnas de datos de una única tabla en la base de datos. Por ejemplo, esta consulta solicita tres columnas de la tabla OFICINAS:

Lista de la población, región y ventas de cada oficina de ventas.

```
SELECT CIUDAD, REGION, VENTAS
FROM OFICINAS
```

CIUDAD	REGION	VENTAS
denver	Oeste	\$186,042.00
New York	Este	\$892,637.00
Chicago	Este	\$735,042.00
Atlanta	Este	\$367,911.00
Los Angeles	Oeste	\$835,915.00

La sentencia SELECT para consultas sencillas como ésta sólo incluye las dos cláusulas imprescindibles. La cláusula SELECT designa a las columnas solicitadas; la cláusula FROM designa a la tabla que las contiene.

Conceptualmente, SQL procesa la consulta recorriendo la tabla nominada en la cláusula FROM, una fila cada vez, como se muestra en la Figura 6.3. Por cada fila, SQL toma los valores de la columnas solicitadas en la lista de selección y produce una única fila de resultados. Los resultados contienen por tanto una fila de datos por cada fila de la tabla.

Columnas calculadas

Además de las columnas cuyos valores provienen directamente de la base de datos, una consulta SQL puede incluir columnas *calculadas* cuyos valores se

calcularán a partir de los valores de los datos almacenados. Para solicitar una columna calculada, se especifica una expresión SQL en la lista de selección. Como discutimos en el Capítulo 5, las expresiones SQL pueden contener sumas, restas, multiplicaciones y divisiones. También se pueden utilizar paréntesis para construir expresiones más complejas. Naturalmente las columnas referenciadas en una expresión aritmética deben tener un tipo numérico. Si usted intenta sumar, restar, multiplicar o dividir columnas que contienen datos de texto, SQL reportará un error.

Esta consulta muestra una columna calculada simple:

Lista la ciudad, la región y el importe por encima o por debajo del objetivo para cada oficina.

```
SELECT CIUDAD, REGION, (VENTAS-OBJETIVO)
FROM OFICINAS
```

CIUDAD	REGION	(VENTAS-OBJETIVO)
Denver	Oeste	-\$113,958.00
New York	Este	\$117,637.00
Chicago	Este	-\$64,958.00
Atlanta	Este	\$17,911.00
Los Angeles	Oeste	\$110,915.00

Para procesar la consulta, SQL examina las oficinas, generando una fila de resultados por cada fila de la tabla OFICINAS, tal como queda reflejado en la Figura 6.4. Las dos primeras columnas de resultados provienen directamente de la tabla OFICINAS. La tercera columna de los resultados se calcula, fila a fila, utilizando los valores de datos de la fila actual de la tabla OFICINAS.

He aquí otros ejemplos de consultas que utilizan columnas calculadas:

Muestra el valor del inventario para cada producto.

```
SELECT ID_FAB, ID_PRODUCTO, DESCRIPCION, (EXISTENCIAS * PRECIO)
FROM PRODUCTOS
```

ID_FAB	ID_PRODUCTO	DESCRIPCION	(EXISTENCIAS * PRECIO)
REI	2A45C	V Stago Trinquete	\$16,590.00
ACI	4100Y	Extractor	\$68,750.00
QSA	XK47	Reductor	\$13,490.00
BIG	4167Z	Placa	\$0.00
IMM	779C	Riostra 2-Tm	\$16,875.00
ACI	41003	Articulo Tipo 3	\$22,149.00
ACI	41004	Articulo Tipo 4	\$16,263.00
BIC	41003	Manivela	\$1,956.00
:	:	:	:

Figura 6.3. Procesamiento de consulta simple (sin cláusula WHERE).

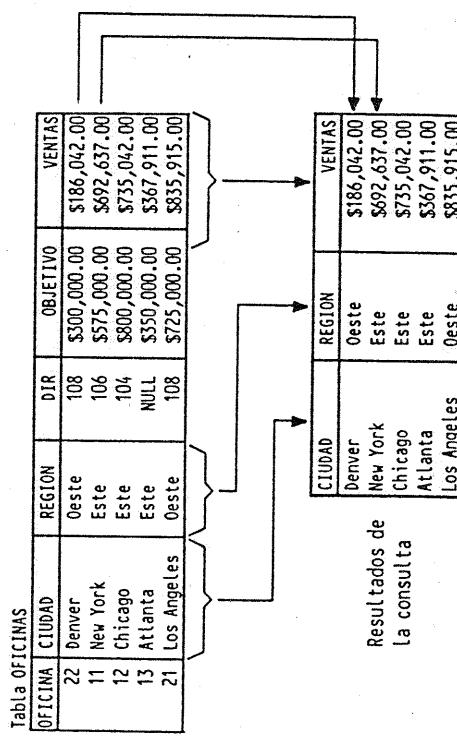


Figura 6.3. Procesamiento de consulta simple (sin cláusula WHERE).

Muestra qué sucederá si se eleva la cuota de cada vendedor un 3 % de sus ventas anuales hasta la fecha correspondiente.

```
SELECT NOMBRE, CUOTA, (CUOTA + (.03 * VENTAS))
FROM REVENTAS
```

NOMBRE	CUOTA	(CUOTA + (.03 * VENTAS))
Bill Adams	\$350,000.00	\$361,037.33
Mary Jones	\$300,000.00	\$311,781.75
Sue Smith	\$350,000.00	\$364,221.50
Sam Clark	\$275,000.00	\$283,997.36
Bob Smith	\$200,000.00	\$204,277.82
Dan Roberts	\$300,000.00	\$309,170.19
Tom Snyder	NULL	NULL
Larry Fitch	\$350,000.00	\$360,855.95
Paul Cruz	\$275,000.00	\$283,603.25
Nancy Angelli	\$300,000.00	\$305,581.26

Como mencionamos en el Capítulo 5, muchos productos SQL disponen de operaciones aritméticas adicionales, operaciones de cadenas de caracteres y funciones internas que pueden ser utilizadas en expresiones SQL. Estas pueden aparecer en expresiones de la lista de selección, como en este ejemplo DB2:

```
Listar el nombre, el mes y el año de contrato para cada vendedor.

SELECT NOMBRE, MONTH(contrato), YEAR(contrato)
FROM REVENTAS
```

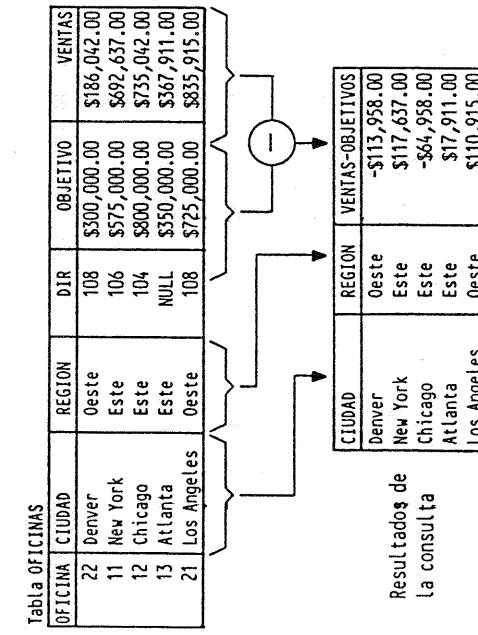


Figura 6.4. Procesamiento de consulta con una columna calculada.

También se pueden utilizar constantes SQL por sí mismas como ítems en una lista de selección. Esto puede ser útil para producir resultados que sean más fáciles de leer e interpretar, como en este ejemplo:

Lista las ventas para cada ciudad.

```
SELECT CIUDAD, 'tiene ventas de ', VENTAS
FROM OFICINAS
```

CIUDAD	TIENE VENTAS DE	VENTAS
Denver	tiene ventas de	\$186,042.00
New York	tiene ventas de	\$692,637.00
Chicago	tiene ventas de	\$735,042.00
Atlanta	tiene ventas de	\$367,911.00
Los Angeles	tiene ventas de	\$835,915.00

Los resultados de la consulta parecen consistir en una «frase» distinta por cada oficina, pero realmente es una tabla de tres columnas. Las columnas primera y tercera contienen valores procedentes de la tabla OFICINAS. La columna segunda siempre contiene la misma cadena de texto de 15 caracteres. Esta distinción es útil cuando los resultados de la consulta se visualizan en una pantalla, pero es crucial en SQL programado, cuando los resultados están siendo recuperados en un programa y utilizados para cálculos.

Selección de todas las columnas (SELECT *)

A veces es conveniente visualizar el contenido de todas las columnas de una tabla. Esto puede ser particularmente útil cuando uno va a utilizar por primera vez una base de datos y desea obtener una rápida compresión de su estructura y de los datos que contiene. Por conveniencia, SQL permite utilizar un asterisco (*) en lugar de la lista de selección como abreviatura de «todas las columnas»:

Muestra todos los datos de la tabla OFICINAS.

```
SELECT *
FROM OFICINAS
```

OFICINA	CIUDAD	REGION	DIR	OBJETIVO	VENTAS
22	Denver	Oeste	108	\$300,000.00	\$186,042.00
11	New York	Este	106	\$375,000.00	\$692,637.00
12	Chicago	Este	104	\$300,000.00	\$735,042.00
13	Atlanta	Este	NULL	\$350,000.00	\$367,911.00
21	Los Angeles	Oeste	108	\$275,000.00	\$835,915.00

Los resultados de la consulta contienen todas las seis columnas de la tabla OFICINAS, en el mismo orden de izquierda a derecha que tienen en la tabla.

El estándar SQL ANSI/ISO especifica que una sentencia SELECT puede tener o bien una selección de todas las columnas o bien una lista de selección, pero no ambas, tal como se muestra en la Figura 6.1. Sin embargo, muchas implementaciones SQL tratan el asterisco (*) como cualquier otro elemento de la lista de selección. Por tanto la consulta:

```
SELECT *, (VENTAS - OBJETIVO)
  FROM OFICINAS
```

es legal en la mayoría de los dialectos SQL comerciales (por ejemplo en DB2, Oracle y SQL Server), pero no está permitida por el estándar ANSI/ISO.

La selección de todas las columnas es muy adecuada cuando se está utilizando el SQL interactivo de forma casual. Debería evitarse en SQL programado, ya que cambios en la estructura de la base de datos pueden hacer que un programa falle. Por ejemplo, supongamos que la tabla OFICINAS fuera suprimida de la base de datos y luego recreada con sus columnas reordenadas y con una nueva séptima columna añadida. SQL automáticamente se cuida de los detalles que implican esos cambios con referencia a la base de datos, pero no puede modificar el programa de aplicación. Si el programa espera que una consulta SELECT * FROM OFICINAS devuelva seis columnas de resultados con ciertos tipos de datos, ciertamente dejará de funcionar cuando las columnas estén reordenadas y se haya añadido una nueva.

Estas dificultades pueden evitarse si se escribe el programa para que solicite las columnas que precise mediante nombre. Por ejemplo, la consulta siguiente produce los mismos resultados que SELECT * FROM OFICINAS. Además es inmune a cambios en la estructura de la base de datos, siempre que las columnas designadas continúen existiendo en la tabla OFICINAS.

```
SELECT OFICINA, CIUDAD, REGION, DIR, OBJETIVO, VENTAS
  FROM OFICINAS
```

Lista los números de empleado de todos los directores de oficinas de ventas.

```
SELECT DIR
  FROM OFICINAS
 WHERE OFICINA = 'DIR'
   ---
```

108
106
105
108

Los resultados tienen cinco filas (uno por cada oficina), pero dos de ellas son duplicados exactos la una de la otra. ¿Por qué? Porque Larry Fitch dirige las oficinas tanto de Los Angeles como de Denver, y su número de empleado (108) aparece en ambas filas de la tabla OFICINAS. Estos resultados no son probablemente lo que uno pretende cuando hace la consulta. Si hubiera cuatro directores diferentes, cabría esperar que sólo aparecieran en los resultados cuatro números de empleado.

Se pueden eliminar las filas duplicadas de los resultados de la consulta insertando la palabra clave DISTINCT en la sentencia SELECT justo antes de la lista de selección. He aquí una versión de la consulta anterior que produce los resultados deseados:

Lista los números de empleado de todos los directores de oficinas de ventas.

```
SELECT DISTINCT DIR
  FROM OFICINAS
 WHERE OFICINA = 'DIR'
   ---
```

104
105
106
108

Conceptualmente, SQL efectúa esta consulta generando primero un conjunto completo de resultados (cinco filas) y eliminando luego las filas que son duplicados exactos de alguna otra para formar los resultados finales. La palabra clave DISTINCT puede ser especificada con independencia de los contenidos de la lista SELECT (con ciertas restricciones para consultas sumarios, como se describirá en el Capítulo 8).

Si se omite la palabra clave DISTINCT, SQL no elimina las filas duplicadas. También se puede especificar la palabra clave ALL para indicar explícitamente que las filas duplicadas sean retenidas, pero es innecesario ya que éste es el comportamiento por omisión.

Filas duplicadas (DISTINCT)

Si una consulta incluye la clave primaria de una tabla en su lista de selección, entonces cada fila de resultados será única (ya que la clave primaria tiene un valor diferente en cada fila). Si no se incluye la clave primaria en los resultados, pueden producirse filas duplicadas. Por ejemplo, supongamos que se hace la siguiente petición:

Selección de fila (cláusula WHERE)

Las consultas SQL que recuperan todas las filas de una tabla son útiles para inspección y elaboración de informes sobre la base de datos, pero para poco más. Generalmente se deseará seleccionar solamente parte de las filas de una tabla, y sólo se incluirán esas filas en los resultados. La cláusula WHERE se emplea para especificar las filas que se desean recuperar. He aquí algunos ejemplos de consultas simples que utilizan la cláusula WHERE:

Muestra las oficinas en donde las ventas exceden al objetivo.

```
SELECT CIUDAD, VENTAS, OBJETIVO
FROM OFICINAS
WHERE VENTAS > OBJETIVO
```

CIUDAD	VENTAS	OBJETIVO
New York	\$622,637.00	\$575,000.00
Atlanta	\$367,911.00	\$350,000.00
Los Angeles	\$335,915.00	\$325,000.00

Muestra el nombre, las ventas y la cuota del empleado número 105.

```
SELECT NOMBRE, VENTAS, CUOTA
FROM REPVENTAS
WHERE NUM_EMPL = 105
```

NOMBRE	VENTAS	CUOTA
Bill Adams	\$367,911.00	\$350,000.00

Muestra los empleados dirigidos por Bob Smith (empleado 104).

```
SELECT NOMBRE, VENTAS
FROM REPVENTAS
WHERE DIRECTOR = 104
```

NOMBRE	VENTAS
Bill Adams	\$367,911.00
Dan Roberts	\$305,673.00
Paul Cruz	\$286,775.00

La cláusula WHERE consta de la palabra clave WHERE seguida de una condición de búsqueda que especifica las filas a recuperar. En la consulta anterior, por ejemplo, la condición de búsqueda es DIRECTOR = 104. La Figura 6.5 muestra cómo funciona la cláusula WHERE. Conceptualmente, SQL recorre cada fila de la tabla REPVENTAS, una a una, y aplica la condición de búsqueda a la fila. Cuando aparece un nombre de columna en la condición de búsqueda (tal como la columna DIRECTOR en este ejemplo), SQL utiliza el valor de la columna en la fila

actual. Por cada fila, la condición de búsqueda puede producir uno de tres resultados:

- Si la condición de búsqueda es TRUE (CIERTA), la fila se incluye en los resultados de la consulta. Por ejemplo, la fila correspondiente a Bill Adams tiene el valor DIRECTOR correcto, y por tanto se incluye.
- Si la condición de búsqueda es FALSE (FALSA), la fila se excluye de los resultados de la consulta. Por ejemplo, la fila correspondiente a Sue Smith tiene un valor DIRECTOR distinto, y por tanto se excluye.
- Si la condición de búsqueda tiene un valor NULL (desconocido), la fila se excluye de los resultados de la consulta. Por ejemplo, la fila correspondiente a Sam Clark tiene un valor NULL en la columna DIRECTOR, y por tanto se excluye.

La Figura 6.6 muestra otro modo de pensar acerca del papel de la condición de búsqueda en la cláusula WHERE. Básicamente, la condición de búsqueda actúa

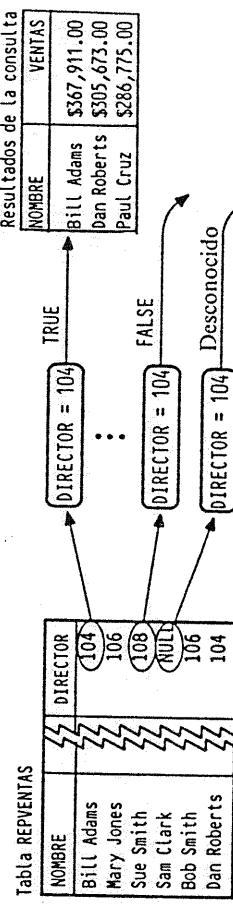


Figura 6.5. Selección de fila con la cláusula WHERE.

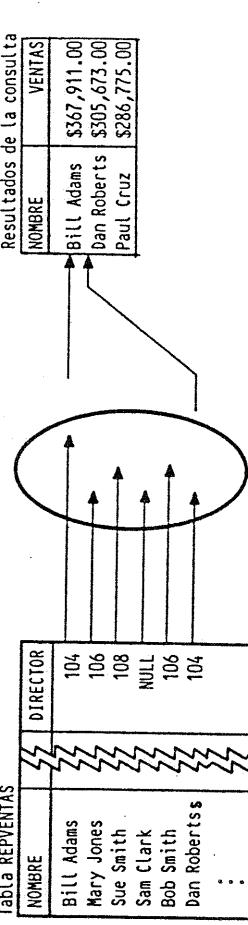


Figura 6.6. La cláusula WHERE como un filtro

como un filtro para las filas de la tabla. Las filas que satisfacen la condición de búsqueda atraviesan el filtro y forman parte de los resultados de la consulta. Las filas que no satisfacen la condición de búsqueda son atrapadas por el filtro y quedan excluidas de los resultados de la consulta.

Condiciones de búsqueda

SQL ofrece un rico conjunto de condiciones de búsqueda que permiten especificar muchos tipos diferentes de consultas eficaz y naturalmente. Aquí resumiremos cinco condiciones básicas de búsqueda (llamadas *predicados* en el estándar ANSI/ISO) y posteriormente las describiremos en otras secciones:

- **Test de comparación.** Compara el valor de una expresión con el valor de otra. Utilice este test para seleccionar las oficinas de la región Este, o los vendedores cuyas ventas superan a sus cuotas.
- **Test de rango.** Examina si el valor de una expresión cae dentro de un rango especificado de valores. Utilice este test para hallar los vendedores cuyas ventas se encuentren entre \$100,000 y \$500,000.
- **Test de pertenencia a conjunto.** Comprueba si el valor de una expresión se corresponde con uno de un conjunto de valores. Utilice este test para seleccionar las oficinas localizadas en New York, Chicago o Los Angeles.
- **Test de correspondencia con patrón.** Comprueba si el valor de una columna que contiene datos de cadena de caracteres se corresponde a un patrón especificado. Utilice este test para seleccionar los clientes cuyos nombres comienzan con la letra «E».
- **Test de valor nulo.** Comprueba si una columna tiene un valor NULL (desconocido). Utilice este test para hallar los vendedores a los que aún no les ha sido asignado un director.

dos expresiones, como se muestra en la Figura 6.7. He aquí algunos ejemplos de tests de comparación típicos:

Halla los vendedores contratados antes de 1988.

```
SELECT NOMBRE
      FROM REPVENTAS
     WHERE CONTRATO < '01-ENE-88'
          NOMBRE
```

Sue Smith
Bob Smith
Dan Roberts
Paul Cruz

Lista las oficinas cuyas ventas están por debajo del 80 % del objetivo.

```
SELECT CIUDAD, VENTAS, OBJETIVO
      FROM OFICINAS
     WHERE VENTAS < (.8 * OBJETIVO)
          CIUDAD   VENTAS   OBJETIVO
Denver    $186,042.00 $300,000.00
```

Lista las oficinas no dirigidas por el empleado número 108.

```
SELECT CIUDAD, DIR
      FROM OFICINAS
     WHERE DIR <> 108
          CIUDAD   DIR
New York  106
Chicago   104
Atlanta   105
```

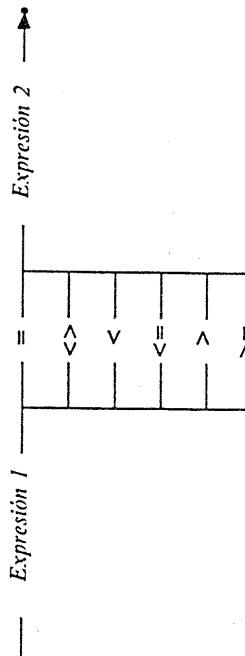


Figura 6.7. Diagrama sintáctico del test de comparación.

La condición de búsqueda más común utilizada en una consulta SQL es el test de comparación. En un test de comparación, SQL calcula y compara los valores de dos expresiones SQL por cada fila de datos. Las expresiones pueden ser tan simples como un nombre de columna o una constante, o pueden ser expresiones aritméticas más complejas. SQL ofrece seis modos diferentes de comparar las

Como se muestra en la Figura 6.7, la comparación de desigualdad se escribe como «A < > B» según la especificación SQL ANSI/ISO. Varias implementaciones SQL utilizan notaciones alternativas, tales como «A != B» (utilizada por SQL Server) y «A $\neg =$ B» (utilizada por DB2 y SQL/DS). En algunos casos, éstas son formas alternativas; en otros, son la única forma aceptable del test de desigualdad.

Cuando SQL compara los valores de dos expresiones en el test de comparación, se pueden producir tres resultados:

- Si la comparación es cierta, el test produce un resultado TRUE.
- Si la comparación es falsa, el test produce un resultado FALSE.
- Si alguna de las dos expresiones produce un valor NULL, la comparación genera un resultado NULL.

Recuperación de una fila. El test de comparación más habitual es el que comprueba si el valor de una columna es igual a cierta constante. Cuando la columna es una clave primaria, el test aisla una sola fila de la tabla, produciendo una sola fila de resultados, como en este ejemplo:

```
Recupera el nombre y el límite de crédito del cliente número 2107.

SELECT EMPRESA, LIMITE_CREDITO
  FROM CLIENTES
 WHERE NUM_CLIE = 2107
EMPRESA.          LIMITE_CREDITO
----- -----
Ace International   $35,000.00
```

Este tipo de consulta es el fundamento de los programas de recuperación de base de datos basadas en formularios. El usuario introduce un número de cliente en el formulario, y el programa utiliza el número para construir y ejecutar una consulta. Luego visualiza los datos recuperados en el formulario.

Observe que la sentencia SQL que recupera un cliente específico por su número, como en este ejemplo, y la que recupera todos los clientes que poseen una cierta característica (tal como aquellos con límites de crédito superiores a \$25,000) tienen ambas exactamente la misma forma. Estos dos tipos de consulta (recuperación por clave primaria y recuperación basada en una búsqueda del dato) serían operaciones muy diferentes en una base de datos no relacional. Esta uniformidad de estrategia hace que SQL sea mucho más sencillo de aprender y utilizar que lenguajes de consulta anteriores.

Consideraciones del valor NULL. El comportamiento de los valores NULL en los tests de comparación puede revelar que algunas nociones «obviamente ciertas» referentes a consultas SQL no son, de hecho, necesariamente ciertas. Por ejemplo, podría parecer que los resultados de estas dos consultas:

Lista vendedores que superan sus cuotas.

```
SELECT NOMBRE
  FROM REVENTAS
 WHERE VENTAS > CUOTA
NOMBRE
-----
Bill Adams
Mary Jones
Sue Smith
Sam Clark
Dan Roberts
Larry Fitch
Paul Cruz
```

Lista los vendedores que están por debajo o en su cuota.

```
SELECT NOMBRE
  FROM REVENTAS
 WHERE VENTAS <= CUOTA
NOMBRE
-----
Bob Smith
Nancy Angelli
```

son dos filas, pero hay diez filas en la tabla REVENTAS. La fila de Tom Snyder tiene un valor NULL en la columna CUOTA, puesto que aún no se ha asignado una cuota. Esta fila no está listada en ninguna de las consultas; «desaparece» en el test de comparación.

Como muestra este ejemplo, es necesario considerar el manejo del valor NULL cuando se especifica una condición de búsqueda. En la lógica trivaluada de SQL, una condición de búsqueda puede producir un resultado TRUE, FALSE o NULL. Sólo las filas en donde la condición de búsqueda genera un resultado TRUE se incluyen en los resultados de la consulta.

Test de rango (BETWEEN)

SQL proporciona una forma diferente de condición de búsqueda con el test de rango (BETWEEN) que se muestra en la Figura 6.8. El test de rango comprueba si un valor de dato se encuentra entre dos valores específicos. Implica el uso de tres expresiones SQL. La primera expresión define el valor a comprobar; las expresiones segunda y tercera definen los extremos superior e inferior del rango a comprobar. Los tipos de datos de las tres expresiones deben ser comparables.

Este ejemplo muestra un test de rango típico:

```
SELECT NUM_PEDIDO, FECHA_PEDIDO, FAB, PRODUCTO, IMPORTE
  FROM PEDIDOS
 WHERE FECHA_PEDIDO BETWEEN '01-OCT-89' Y '31-DIC-89'
-----+-----+-----+-----+-----+
NUM_PEDIDO  FECHA_PEDIDO  FAB  PRODUCTO      IMPORTE
-----+-----+-----+-----+-----+
112961    17-DIC-89  REI  2A44L   $31,500.00
112968    12-OCT-89  ACI  41004   $3,978.00
112965    17-DIC-89  ACI  41004   $3,276.00
112983    27-DIC-89  ACI  41004   $702.00
112979    12-OCT-89  ACI  41002   $15,000.00
112992    04-NOV-89  ACI  41002   $760.00
112975    12-OCT-89  REI  2A44G   $2,100.00
112987    31-DIC-89  ACI  4100Y   $27,500.00
```

El test BETWEEN incluye los puntos extremos del rango, por lo que los pedidos remitidos el 1 de octubre o el 31 de diciembre se incluyen en los resultados de la consulta. He aquí otro ejemplo de test de rango:



Figura 6.8. Diagrama sintáctico del test de rango (BETWEEN).

Halla los pedidos que caen en varios rangos de importe.

```
SELECT NUM_PEDIDO, IMPORTE
  FROM PEDIDOS
 WHERE IMPORTE BETWEEN 20000.00 Y 29999.99
-----+-----+
NUM_PEDIDO  IMPORTE
-----+-----+
113036  $22,500.00
112987  $27,500.00
113042  $22,500.00
```

```
SELECT NUM_PEDIDO, IMPORTE
  FROM PEDIDOS
 WHERE IMPORTE BETWEEN 30000.00 Y 39999.99
-----+-----+
NUM_PEDIDO  IMPORTE
-----+-----+
112961  $31,500.00
113069  $31,350.00
```

```
SELECT NUM_PEDIDO, IMPORTE
  FROM PEDIDOS
 WHERE IMPORTE BETWEEN 40000.00 Y 49999.99
-----+-----+
NUM_PEDIDO  IMPORTE
-----+-----+
113045  $45,000.00
```

La versión negada del test de rango (NOT BETWEEN) comprueba los valores que caen fuera del rango, como en este ejemplo:

Lista los vendedores cuyas ventas no están entre el 80 y el 120 % de su cuota.

```
SELECT NOMBRE, VENTAS, CUOTA
  FROM REPVENTAS
 WHERE VENTAS NOT BETWEEN (.8 * CUOTA) Y (.12 * CUOTA)
-----+-----+-----+
NOMBRE      VENTAS      CUOTA
-----+-----+-----+
Mary Jones   $392,725.00  $300,000.00
Sue Smith   $474,050.00  $350,000.00
Bob Smith   $142,594.00  $200,000.00
Nancy Angelli $186,042.00  $300,000.00
```

La expresión de test especificada en el test BETWEEN puede ser cualquier expresión SQL válida, pero en la práctica generalmente es tan sólo un nombre de columna, como en los ejemplos anteriores.

El estándar ANSI/ISO define reglas relativamente complejas para el manejo de los valores NULL en el test BETWEEN:

- Si la expresión de test produce un valor NULL, o si ambas expresiones definitivas del rango producen valores NULL, el test BETWEEN devuelve un resultado NULL.
- Si la expresión que define el extremo inferior del rango produce un valor NULL, el test BETWEEN devuelve FALSE si el valor de test es superior al límite superior, y NULL en caso contrario.

- Si la expresión que define el extremo superior del rango produce un valor NULL, el test BETWEEN devuelve FALSE si el valor de test es menor que el límite inferior, y NULL en caso contrario.

Antes de confiar en este comportamiento, es buena idea experimentar con el DBMS.

Merece la pena advertir que el test BETWEEN no añade realmente potencia expresiva a SQL, ya que puede ser expresado mediante dos tests de comparación. El test de rango:

A BETWEEN B AND C

es completamente equivalente a:

$(A \geq B) \text{ AND } (A <= C)$

Sin embargo, el test **BETWEEN** es un modo más sencillo de expresar una condición de búsqueda cuando se está pensando en términos de un rango de valores.

Test de pertenencia a conjunto (IN)

Otra condición habitual es el test de pertenencia a conjunto (IN), mostrado en la Figura 6.9. Examina si un valor de dato coincide con uno de una lista de valores objetivo. He aquí varias consultas que utilizan el test de pertenencia a conjunto:

Lista los vendedores que trabajan en New York, Atlanta o Denver.

```
SELECT NOMBRE, CUOTA, VENTAS
  FROM REPVENTAS
 WHERE OFICINAREP IN (11, 13, 22)
    
```

NOMBRE	CUOTA	VENTAS
Bill Adams	\$350,000.00	\$367,911.00
Mary Jones	\$300,000.00	\$392,725.00
Sam Clark	\$275,000.00	\$299,912.00
Nancy Angelli	\$300,000.00	\$188,042.00

Halla todos los pedidos remitidos un jueves en enero de 1990.

```
SELECT NUM_PEDIDO, FECHA_PEDIDO, IMPORTE
  FROM PEDIDOS
 WHERE FECHAPEDIDO IN ('04-ENE-90', '11-ENE-90', '18-ENE-90', '25-ENE-90')
    
```

NUM_PEDIDO	FECHA_PEDIDO	IMPORTE
113012	11-ENE-90	\$3,745.00
113003	25-ENE-90	\$5,625.00

Halla todos los pedidos obtenidos por cuatro vendedores específicos.

```
SELECT NUMORDEN, REP, IMPORTE
  FROM PEDIDOS
 WHERE REP IN (107, 109, 101, 103)
    
```

NUM_PEDIDO	REP	IMPORTE
112968	101	\$3,978.00
113058	109	\$1,480.00
112997	107	\$352.00
113062	107	\$2,430.00
113069	107	\$31,350.00
112975	103	\$2,100.00
113055	101	\$350.00
113003	109	\$5,625.00
113057	103	\$800.00
113042	101	\$22,500.00

Se puede comprobar si el valor del dato no corresponde a ninguno de los valores objetivos utilizando la forma NOT IN del test de pertenencia a conjunto. La expresión de test en un test IN puede ser cualquier expresión SQL, pero generalmente es tan sólo un nombre de columna, como en los ejemplos precedentes. Si la expresión de test produce un valor NULL, el test IN devuelve NULL. Todos los elementos en la lista de valores objetivo deben tener el mismo tipo de datos, y ese tipo debe ser comparable al tipo de dato de la expresión de test. Al igual que el test BETWEEN, el test IN no añade potencia expresiva a SQL, ya que la condición de búsqueda.

$X \text{ IN } (A, B, C)$

es completamente equivalente a:

$(X = A) \text{ OR } (X = B) \text{ OR } (X = C)$

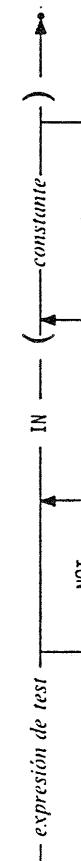
Sin embargo, el test IN ofrece un modo mucho más eficiente de expresar la condición de búsqueda, especialmente si el conjunto contiene más que unos pocos valores. El estándar SQL ANSI/ISO no especifica un límite máximo al número de elementos que pueden aparecer en la lista de valores, y la mayoría de las implementaciones comerciales tampoco establecen un límite superior explícito. Por razones de portabilidad, generalmente es buena idea evitar listas con un solo elemento, tal como la siguiente:

CIUDAD IN ('New York')

y reemplazarla con un test de comparación simple:

CIUDAD = 'New York'

Figura 6.9. Diagrama sintáctico del test de pertenencia a conjunto (IN).



Test de correspondencia con patrón (LIKE)

Se puede utilizar un test de comparación simple para recuperar las filas en donde el contenido de una columna de texto se corresponde con un cierto texto particular. Por ejemplo, esta consulta recupera una fila de la tabla CLIENTES por nombre:

```
Muestra el límite de crédito de Smithson Corp.

SELECT EMPRESA, LIMITE_CREDITO
FROM CLIENTES
WHERE EMPRESA = 'Smithson Corp.'
```

Sin embargo, usted podría olvidar fácilmente si el nombre de la empresa era «Smith», «Smithson» o «Smithsonian». El test de correspondencia con patrón de SQL puede ser utilizado para recuperar los datos en base a una correspondencia parcial del nombre de cliente.

El test de correspondencia con patrón (LIKE), mostrado en la Figura 6.10, comprueba si el valor de dato de una columna se ajusta a un patrón especificado. El patrón es una cadena que puede incluir uno o más caracteres *comodines*. Estos caracteres se interpretan de una manera especial.

Caracteres comodines. El carácter comodín signo de porcentaje (%) se corresponde con cualquier secuencia de cero o más caracteres. He aquí una versión modificada de la consulta anterior que utiliza el signo de porcentaje para correspondencia con patrón:

```
SELECT EMPRESA, LIMITE_CREDITO
FROM CLIENTES
WHERE EMPRESA LIKE 'Smith% Corp.'
```

La palabra clave LIKE dice a SQL que compare la columna NOMBRE con el patrón «Smith% Corp.». Cualquier de los nombres siguientes se ajustarían al patrón:

Smith Corp. Smithson Corp. Smithsonian Corp.

pero estos nombres no se ajustarían:

Smithcorp Smithson Inc.

El carácter comodín subrayado (_) se corresponde con cualquier carácter simple. Si usted está seguro que el nombre de la empresa es o bien «Smithson» o bien «Smithsen», por ejemplo, puede utilizar esta consulta:

```
SELECT EMPRESA, LIMITE_CREDITO
FROM CLIENTES
WHERE EMPRESA LIKE 'Smiths_n Corp.'
```

En este caso, cualquiera de estos nombres cumplirían el patrón:

Smithson Corp. Smithsen Corp. Smithsun Corp.

pero estos nombres no:

Smithsoon Corp. Smithn Corp.

Los caracteres comodines pueden aparecer en cualquier lugar de la cadena patrón, y puede haber varios caracteres comodines dentro de una misma cadena. Esta consulta permite la ortografía «Smithson» o «Smithsen», y también aceptaría «Corp.», «Inc.», o cualquier otra terminación del nombre de la empresa.

```
SELECT EMPRESA, LIMITE_CREDITO
FROM CLIENTES
WHERE EMPRESA LIKE 'Smiths_n %'
```

Se pueden localizar cadenas que no se ajusten a un patrón utilizando el formato NOT LIKE del test de correspondencia de patrones. El test LIKE debe aplicarse a una columna con un tipo de datos cadena. Si el valor del dato en la columna es NULL, el test LIKE devolverá un resultado NULL.

Si usted ha utilizado computadores personales, probablemente habrá visto casos de correspondencia con patrones de texto anteriormente. Por ejemplo, en el lenguaje de órdenes de MS-DOS o de OS/2, se puede teclear la orden:

DIR D*



para listar todos los archivos cuyos nombres comienzan con la letra «D». En MS-DOS, se utiliza el asterisco (*) en lugar del signo de porcentaje (%) de SQL, y el signo de interrogación (?) en lugar del subrayado (_), en SQL, pero las capacidades de correspondencia con patrones son las mismas.

Figura 6.10. Diagrama sintáctico del test de correspondencia con patrón (LIKE).

Caracteres escape*. Uno de los problemas de la correspondencia con patrones en cadenas es cómo hacer corresponder los propios caracteres comodines como caracteres literales. Para comprobar la presencia de un carácter tanto por ciento en una columna de datos de texto, por ejemplo, no se puede simplemente incluir el signo del tanto por ciento en el patrón, ya que SQL lo trataría como un comodín. Con la mayoría de los productos comerciales SQL, no se pueden hacer corresponder literalmente los dos caracteres comodines. Esto generalmente no plantea serios problemas, ya que los caracteres comodines no aparecen frecuentemente en nombres, números de productos y otros datos de texto que generalmente se almacenan en una base de datos.

El estándar SQL ANSI/ISO especifica una manera de comparar literalmente caracteres comodines, utilizando un *carácter escape* especial. Cuando el carácter escape aparece en el patrón, el carácter que le sigue inmediatamente se trata como un carácter literal en lugar de como un carácter comodín. (El segundo carácter se dice que ha sido *escapado*.) El carácter escapado puede ser uno de los dos caracteres comodines, o el propio carácter de escape, que ha tomado ahora un significado especial dentro del patrón.

El carácter de escape se especifica como una cadena constante de un carácter escape aparece en la cláusula ESCAPE de la condición de búsqueda, como queda reflejado en la Figura 6.10. He aquí un ejemplo utilizando un signo dólar (\$) como carácter de escape:

Halla los productos cuyo id comience con las cuatro letras «A%BC».

```
SELECT NUM_PEDIDO, PRODUCTO
  FROM PEDIDOS
 WHERE PRODUCTO LIKE 'A$%BC%' ESCAPE '$'
```

El primer signo de porcentaje en el patrón, que sigue a un carácter escape, es tratado como un signo literal; el segundo funciona como un comodín.

El uso de los caracteres de escape es muy habitual en aplicaciones de correspondencia con patrones, que es la razón por la que el estándar ANSI/ISO lo ha especificado. Sin embargo, no formaba parte de las implementaciones SQL iniciales, y no ha sido ampliamente adoptado. DB2, OS/2 Extended Edition, Oracle y SQL Server no los soportan hoy día. Para asegurar la portabilidad, la cláusula ESCAPE debería ser evitada.

Test de valor nulo (IS NULL)

Los valores NULL crean una lógica trivaluada para las condiciones de búsqueda en SQL. Para una fila determinada, el resultado de una condición de búsqueda puede ser TRUE o FALSE, o puede ser NULL debido a que una de las columnas utilizadas en la evaluación de la condición de búsqueda contenga un valor NULL.



Figura 6.11. Diagrama sintáctico del test de valor nulo (IS NULL).

A veces es útil comprobar explícitamente los valores NULL en una condición de búsqueda y manejarlos directamente. SQL proporciona un test especial de valor nulo (IS NULL), mostrado en la Figura 6.11, para tratar esta tarea.

Esta consulta utiliza el test de valor nulo para hallar el vendedor en la base de datos ejemplo al que aún no le ha sido asignada una oficina:

Halla el vendedor que aún no tiene asignada una oficina.

```
SELECT NOMBRE
  FROM REVENTAS
 WHERE OFICINAREP IS NULL
```

NOMBRE
Tom Snyder

La forma negada del test de valor nulo (IS NOT NULL) encuentra las filas que no contienen un valor NULL:

Lista los vendedores a los que se les ha asignado una oficina.

```
SELECT NOMBRE
  FROM REVENTAS
 WHERE OFICINAREP IS NOT NULL
```

NOMBRE
Bill Adams
Mary Jones
Sue Smith
Sam Clark
Bob Smith
Dan Roberts
Larry Fitch
Paul Cruz
Nancy Angeli

A diferencia de las condiciones de búsqueda descritas anteriormente, el test de valor nulo no puede producir un resultado NULL. Será siempre TRUE o FALSE.

Puede parecer extraño que no se pueda comprobar un valor NULL utilizando una condición de búsqueda de comparación simple, tal como ésta:

```
SELECT NOMBRE
      FROM REVENTAS
     WHERE OFICINA REP = NULL
```

La palabra clave NULL no puede ser utilizada aquí ya que no es realmente un valor; es tan sólo una señal de que el valor es desconocido. Incluso si el test de comparación:

```
OFICINA REP = NULL
```

fueran las reglas para manejar los valores NULL en comparaciones harían que se comportara de forma diferente a lo que cabría esperar. Cuando SQL encontrara una fila en donde la columna DIR fuera NULL, la condición de búsqueda detectaría:

```
NULL = NULL
```

¿El resultado es TRUE o FALSE? Puesto que los valores de ambos lados del signo igual son desconocidos, SQL no puede decirlo, por lo que las reglas de la lógica SQL dicen que la propia condición de búsqueda debe producir un resultado NULL. Ya que la condición de búsqueda no produce un resultado cierto, la fila queda excluida de los resultados de la consulta —precisamente lo contrario de lo que se desearia que sucediera—. Como resultado de la manera en que SQL maneja los NULL en las comparaciones, se debe utilizar explícitamente el test de valor nulo para comprobar los valores NULL.

Condiciones de búsqueda compuestas (AND, OR y NOT)

Las condiciones de búsqueda simples, descritas en las secciones precedentes devuelven un valor TRUE, FALSE o NULL cuando se aplican a una fila de datos.

Finalmente, se puede utilizar la palabra clave NOT para seleccionar filas en donde la condición de búsqueda es falsa:

Halla todos los vendedores que están por debajo de la cuota, pero cuyas ventas no son inferiores a \$150,000.

```
SELECT NOMBRE, CUOTA, VENTAS
      FROM REVENTAS
     WHERE VENTAS < CUOTA
          AND NOT VENTAS < 150000.00
```

NOMBRE	CUOTA	VENTAS
Bob Smith	\$200,000.00	\$142,594.00
Nancy Angelli	\$300,000.00	\$186,042.00

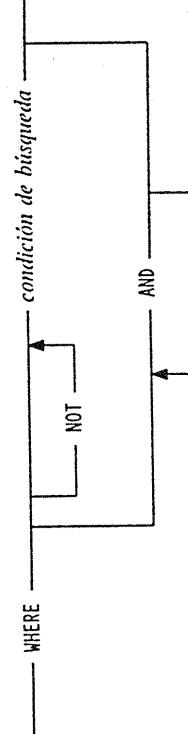


Figura 6.12. Diagrama sintáctico de la cláusula WHERE.

Utilizando las palabras clave AND, OR y NOT y los paréntesis para agrupar los criterios de búsqueda, se pueden construir criterios de búsqueda muy complejos, tal como el que se presenta en esta consulta:

Halla todos los rendidores que: a) trabajan en Denver, New York o Chicago; b) no tienen director y fueron contratados a partir de junio de 1988, o c) están por encima de la cuota, pero tienen ventas de \$600,000 o menos.

```
SELECT NOMBRE
  FROM REVENTAS
 WHERE (OFICINAREP IN ('22, 11, 12'))
   OR (DIRECTOR IS NULL AND CONTRATO >= '01-JUN-88')
   OR (VENTAS > CUOTA AND NOT VENTAS > 600000.00)
```

El porqué usted podría querer ver esta lista particular de nombres es un misterio, pero el ejemplo ilustra una consulta razonablemente compleja.

Como con las condiciones de búsqueda simples, los valores NULL influencian el resultado de las condiciones de búsqueda compuestas, y éstos son sútiles. En particular, el resultado de (NULL OR TRUE), es TRUE, no NULL como cabría esperar. Las Tablas 6.1, 6.2 y 6.3 especifican las tablas de verdad para AND, OR y NOT, respectivamente, y muestran la influencia de los valores NULL.

Cuando se combinan más de dos condiciones de búsqueda con AND, OR y NOT, el estándar ANSI/ISO especifica que NOT tiene la precedencia más alta, seguido de AND y por último OR. Para asegurar la portabilidad, es siempre buena idea utilizar paréntesis y suprimir cualquier posible ambigüedad.

Al igual que las filas de una tabla en la base de datos las filas de los resultados de una consulta no están dispuestas en ningún orden particular. Se puede pedir a SQL que ordene los resultados de una consulta incluyendo la cláusula ORDER BY en la sentencia SELECT. La cláusula ORDER BY, mostrada en la Figura 6.13, consta de las palabras claves ORDER BY, seguidas de una lista de especificaciones de ordenación separadas por comas. Por ejemplo, los resultados de esta consulta están ordenados en dos columnas, REGION y CIUDAD.

Muestra las ventas de cada oficina, ordenadas en orden alfabético por región, y dentro de cada región por ciudad.

```
SELECT CIUDAD, REGION, VENTAS
  FROM OFICINAS
 ORDER BY REGION, CIUDAD
```

CIUDAD	REGION	VENTAS
Atlanta	Este	\$367,911.00
Chicago	Este	\$735,042.00
New York	Este	\$692,637.00
Denver	Oeste	\$186,042.00
Los Angeles	Oeste	\$835,915.00

Figura 6.1. Tabla de verdad de AND.

AND	TRUE	FALSE	NULL
TRUE	TRUE	NULL	NULL
FALSE	FALSE	NULL	NULL
NULL	NULL	NULL	NULL

Figura 6.2. Tabla de verdad de OR.

OR	TRUE	FALSE	NULL
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	NULL
NULL	TRUE	NULL	NULL

Figura 6.13. Diagrama sintáctico de la cláusula ORDER BY.
(cláusula ORDER BY)

Figura 6.3. Tabla de verdad de NOT.

NOT	TRUE	FALSE	NULL
	FALSE	TRUE	NULL

La primera especificación de ordenación (REGION) es la clave de ordenación *mayor*, las que le sigan (CIUDAD, en este caso) son progresivamente claves de ordenación *menores*, utilizadas para «desempatar» cuando dos filas de resultados tienen los mismos valores para las claves mayores. Utilizando la cláusula ORDER BY se puede solicitar la ordenación en secuencia ascendente o descendente, y se puede ordenar con respecto a cualquier elemento en la lista de selección de la consulta.

Por omisión, SQL ordena los datos en secuencia ascendente. Para solicitar ordenación en secuencia descendente, se incluye la palabra clave DESC en la especificación de ordenación, como en este ejemplo:

Lista las oficinas, clasificadas en orden descendente de ventas, de modo que las oficinas con mayores ventas aparezcan en primer lugar.

```
SELECT CIUDAD, REGION, VENTAS
  FROM OFICINAS
 ORDER BY VENTAS, DESC
    CIUDAD   REGION      VENTAS
-----  -----
Los Angeles  Oeste     $835,915.00
Chicago      Este      $735,042.00
New York     Este      $692,637.00
Atlanta      Este      $367,911.00
Denver       Oeste     $186,042.00
```

Como se indica en la Figura 6.13, también se puede utilizar la palabra clave ASC para especificar orden ascendente, pero puesto que ésta es la secuencia de ordenación por omisión, la palabra clave se suele omitir.

Si la columna de resultados de la consulta utilizada para ordenación es una columna calculada, no tiene nombre de columna que se pueda emplear en una especificación de ordenación. En este caso, debe especificarse un número de columna en lugar de un nombre, como en este ejemplo:

Lista las oficinas, clasificadas en orden descendente de rendimiento de ventas, de modo que las oficinas con mejor rendimiento aparezcan primero.

```
SELECT CIUDAD, REGION, (VENTAS - OBJETIVO)
  FROM OFICINAS
 ORDER BY 3 DESC
    CIUDAD   REGION      (VENTAS-OBJETIVO)
-----  -----
New York  Este        $117,637.00
Los Angeles  Oeste    $110,915.00
Atlanta   Este        $17,911.00
Chicago   Este        -$66,938.00
Denver    Oeste     -$113,938.00
```

Estos resultados están ordenados por la tercera columna, que es la diferencia calculada entre VENTAS y OBJETIVO para cada oficina. Combinando números de columna, nombres de columna, ordenaciones ascendentes y ordenaciones descendentes, se pueden especificar ordenaciones bastante complejas de los resultados de una consulta, como en el siguiente ejemplo final:

Lista las oficinas, clasificadas en orden alfabético por región, y dentro de cada región en orden descendente de rendimiento de ventas.

```
SELECT CIUDAD, REGION, (VENTAS - OBJETIVO)
  FROM OFICINAS
 ORDER BY REGION ASC, 3 DESC
    CIUDAD   REGION      (VENTAS-OBJETIVO)
-----  -----
New York  Este        $117,637.00
Atlanta   Este        $17,911.00
Chicago   Este        -$66,938.00
Los Angeles  Oeste    $110,915.00
Denver    Oeste     -$113,938.00
```

Reglas para procesamiento de consultas de tabla única

Las consultas de tabla única son generalmente sencillas, y generalmente es fácil entender el significado de una consulta tan sólo leyendo la sentencia SELECT. Cuando las consultas pasan a ser más complejas, sin embargo, es importante tener una «definición» más precisa de los resultados de la consulta que se producirán mediante una determinada sentencia SELECT. La Figura 6.14 describe el procedimiento para generar los resultados de una consulta SQL que incluye las cláusulas descritas en este capítulo.

Como muestra la Figura 6.14, los resultados producidos por una sentencia SELECT se especifican aplicando cada una de sus cláusulas, una a una. La cláusula FROM se aplica en primer lugar (seleccionando la tabla que contiene los datos a recuperar). A continuación se aplica la cláusula WHERE (seleccionando filas específicas de la tabla). Posteriormente se aplica la cláusula SELECT (que genera las columnas específicas de resultados y que elimina las filas duplicadas, si así se solicita). Finalmente, se aplica la cláusula ORDER BY para ordenar los resultados de la consulta.

Las reglas para procesamiento de consultas SQL que se muestran en la Figura 6.14 serían ampliadas varias veces en los próximos tres capítulos para incluir las restantes cláusulas de la sentencia SELECT.

■ En la práctica, la mayoría de las sentencias SELECT simples no producen filas duplicadas, por lo que el comportamiento por omisión es la no eliminación de duplicados.

■ En la práctica, la mayoría de las operaciones UNION producirían filas duplicadas no deseadas, por lo que la operación por omisión es la eliminación de duplicados.

La eliminación de filas duplicadas en los resultados de la consulta es un proceso que consume mucho tiempo, especialmente si los resultados contienen un gran número de filas. Si usted sabe, basándose en las consultas individuales implicadas, que la operación UNION no puede producir filas duplicadas, debería utilizar específicamente la operación UNION ALL, ya que la consulta se ejecutará mucho más rápidamente.

Uniones y ordenación*

La cláusula ORDER BY no puede aparecer en ninguna de las dos sentencias SELECT combinadas por una operación UNION. No tendría mucho sentido ordenar los dos conjuntos de resultados de ninguna manera, ya que éstos se dirigen directamente a la operación UNION y nunca son visibles al usuario. Sin embargo, el conjunto *combinado* de los resultados de la consulta producidos por la operación UNION pueden ser ordenados especificando una cláusula ORDER BY después de la segunda sentencia SELECT. Ya que las columnas producidas por la operación UNION no tienen nombres, la cláusula ORDER BY debe especificar las columnas por número. He aquí la misma consulta de productos que se mostró en la Figura 6.15, con los resultados ordenados por fabricante y número de producto:

Lista todos los productos en donde el precio del producto supera a \$2.000 o en donde más de \$30.000 del producto han sido ordenados en un solo pedido, clasificados por fabricante y número de producto.

```
SELECT ID_FAB, ID_PRODUCTO
  FROM PRODUCTOS
 WHERE PRECIO > 2000.00
UNION
SELECT DISTINCT FAB, PRODUCTO
  FROM PEDIDOS
 WHERE IMPORTE > 30000.00
 ORDER BY 1, 2
ACI 4100Y
ACI 4100Z
IMM 775C
REI 2A44L
REI 2A44R
```

Uniones múltiples*

La operación UNION puede ser utilizada repetidamente para combinar tres o más conjuntos de resultados, tal como se muestra en la Figura 6.16. La unión de la Tabla B y la Tabla C en la figura produce una única tabla combinada. Esta tabla se combina luego con la Tabla A en otra operación UNION. La consulta de la figura se escribe de este modo:

```
SELECT *
  FROM A
UNION (SELECT *
        FROM B
      UNION (SELECT *
             FROM C))
```

Bill
Mary
George
Fred
Sue
Julia
Harry

Los paréntesis que aparecen en la consulta indican que UNION debería ser realizada en primer lugar. De hecho, si todas las UNIONES de la sentencia eliminan

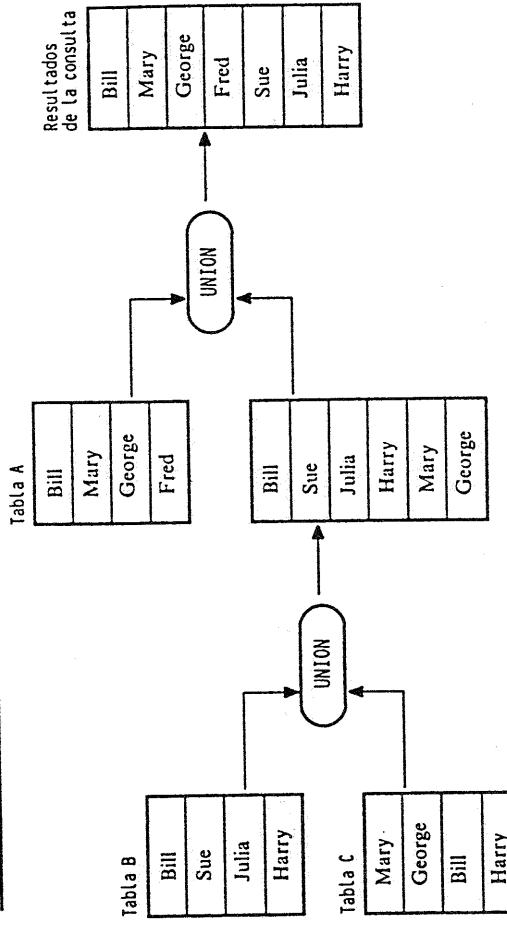


Figura 6.16. Operaciones UNION anidadas.

filas duplicadas, o si todas ellas retienen filas duplicadas, el orden en que efectúan no tiene importancia. Estas tres expresiones son completamente equivalentes:

`A UNION (B UNION C)`

`(A UNION B) UNION C`

`(A UNION C) UNION B`

y producen siete filas de consulta. Análogamente, las tres expresiones siguientes son completamente equivalentes y producen doce filas de resultados, debido a los duplicados que retienen:

`A UNION ALL (B UNION ALL C)`

`(A UNION ALL B) UNION ALL C`

`(A UNION ALL C) UNION ALL B`

Sin embargo, si las uniones implican una mezcla de UNION y UNION ALL, el orden de la evaluación sí importa. Si esta expresión:

`A UNION ALL B UNION C`

se interpreta como:

`A UNION ALL (B UNION C)`

entonces se producen diez filas de resultados (seis de la UNION interna, más cuatro filas de la Tabla A). Sin embargo, si se interpreta como:

`(A UNION ALL B) UNION C`

entonces sólo se producen cuatro filas, ya que la UNION externa elimina todas las filas duplicadas. Por esta razón, es siempre buena idea utilizar paréntesis en UNIONES de tres o más tablas para especificar el orden de evaluación pretendido.

Resumen

Este capítulo es el primero de cuatro capítulos referentes a consultas SQL. En él se han descrito las siguientes características de consulta:

- La sentencia SELECT se utiliza para expresar una consulta SQL. Toda sentencia

SELECT produce una tabla de resultados que contiene una o más columnas y cero o más filas.

- La cláusula FROM especifica la(s) tabla(s) que contiene(n) los datos a recuperar por una consulta.
- La cláusula SELECT especifica la(s) columna(s) de datos a incluir en los resultados de la consulta, que pueden ser columnas de datos de la base de datos o columnas calculadas.
- La cláusula WHERE selecciona las filas a incluir en los resultados aplicando una condición de búsqueda a las filas de la base de datos.
- Una condición de búsqueda puede seleccionar filas mediante comparación de valores, mediante comparación de un valor con un rango o un grupo de valores, por correspondencia con un patrón de cadena o por comprobación de valores NULL.
- Las condiciones de búsqueda simples pueden combinarse mediante AND, OR y NOT para formar condiciones de búsqueda más complejas.
- La cláusula ORDER BY especifica que los resultados de la consulta deben ser ordenados en sentido ascendente o descendente, basándose en los valores de una o más columnas.
- La operación UNION puede ser utilizada dentro de una sentencia SELECT para combinar dos o más conjuntos de resultados y formar un único conjunto.

7 Consultas multitable (composiciones)

Muchas consultas útiles solicitan datos procedentes de dos o más tablas en la base de datos. Por ejemplo, estas peticiones de datos para la base de datos ejemplo extraen los datos de dos, tres o cuatro tablas:

- Lista los vendedores y las oficinas en donde trabajan (tablas REPVENTAS y OFICINAS).
- Lista los pedidos remitidos la última semana, mostrando el importe del pedido, el nombre del cliente que lo ordenó y el nombre del producto solicitado (tablas PEDIDOS, CLIENTES y REPVENTAS).
- Muestra todas las órdenes aceptadas por vendedores en la región Este, mostrando la descripción del producto y el vendedor (tablas PEDIDOS, REPVENTAS, OFICINAS y PRODUCTOS).

SQL permite recuperar datos que responden a estas peticiones mediante consultas multitable que *componen* (*join*) datos procedentes de dos o más tablas. Estas consultas y la facilidad de composición de SQL se describen en este capítulo.

Ejemplo de consulta de dos tablas

Para mejor comprender las facilidades que SQL proporciona para consultas multitable lo mejor es comenzar con una petición simple que combina datos de dos tablas diferentes:

«Lista todos los pedidos, mostrando el número de pedido y su importe, y el nombre y el balance contable del cliente que lo solicitó».

- Los cuatro datos específicos solicitados están evidentemente almacenados en dos tablas diferentes, tal como se muestra en la Figura 7.1:
- La tabla PEDIDOS contiene el número de pedido y el importe de cada pedido, pero no tiene los nombres de cliente ni los límites de crédito.
 - La tabla CLIENTES contiene los nombres de cliente y sus balances pero le falta la información referente a los pedidos.

Tabla PEDIDOS

NUM_PEDIDO	FECHA_PEDIDO	CLIE	REP	CANT	IMPORTE
112961	12/17/1989	2117	106	7	\$31,500.00
113012	01/11/1990	2111	105	35	\$3,755.00
112959	01/03/1990	2101	106	6	\$1,458.00
...					

Relación clave primaria/
clave foránea

Tabla CLIENTES

NUM_CLIE	EMPRESA	REP_CLIE	LIMITE_CREDITO
2118	Holm & Landis	109	\$55,000.00
2111	J.P. Sinclair	106	\$35,000.00
2122	Three-Way Lines	105	\$30,000.00
...			

Lista cada pedido, mostrando el número e importe del pedido, y el nombre y límite de crédito del cliente que lo solicitó.

1. Comenzar escribiendo los nombres de las cuatro columnas para los resultados de la consulta. Luego pasar a la tabla PEDIDOS y comenzar con el primer pedido.
2. Recorrer la fila para hallar el número de pedido (112961) y el importe (\$31,500,00) y copiar ambos valores en la primera fila de resultados de la consulta.
3. Recorrer la fila para hallar el número de cliente que remitió el pedido (2117), y pasar a la tabla CLIENTES para hallar el número de cliente 2117 buscándolo en la columna NUM_CLIE.
4. Recorrer la fila de la tabla CLIENTES para hallar el nombre del cliente ('J. P. Sinclair') y su límite de crédito (\$35,000,00), y copiarlos a la tabla de resultados.
5. Ya hemos generado una fila de resultados de la consulta. Regresamos a la tabla PEDIDOS y continuamos con la fila siguiente. El proceso se repite, comenzando por el Paso 2, hasta que se agotan los pedidos.

Naturalmente ésta no es la única manera de generar los resultados de la consulta, pero independientemente de cómo se haga, dos cosas serán ciertas:

- Cada fila de resultados de la consulta extraerá sus datos de un *par* específico de filas, una de la tabla PEDIDOS y otra de la tabla CLIENTES.
- El par de filas en cuestión se determinará haciendo coincidir los contenidos de las columnas correspondientes de las tablas.

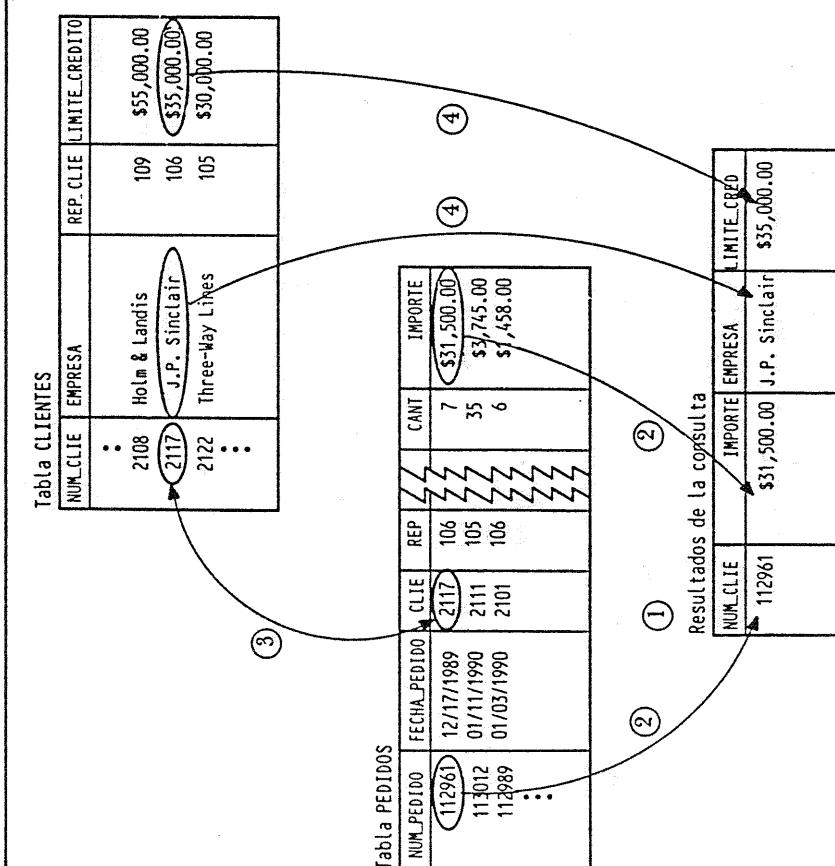
Composiciones simples (equicomposiciones) (join)

El proceso de formar parejas de filas haciendo coincidir los contenidos de las columnas relacionadas se denomina *componer (joining)* las tablas. La tabla resultante (que contiene datos de las dos tablas originales) se denomina una *composición* entre las dos tablas. (Una composición basada en una coincidencia exacta entre dos columnas se denomina más precisamente una *equicomposición*. Las composiciones también pueden basarse en otros tipos de comparaciones de columnas, como se describirá posteriormente en este capítulo.)

Las composiciones son el fundamento del procesamiento de consultas multitable en SQL. Todos los datos de una base de datos relacional están almacenados en sus columnas con valores explícitos, de modo que todas las relaciones posibles entre tablas pueden formarse comparando los contenidos de las columnas.

Figura 7.1. Una petición que afecta a dos tablas.

Existe, sin embargo, un enlace entre estas dos tablas. En cada fila de la tabla PEDIDOS, la columna CLIE contiene el número del cliente que ordenó el pedido, el cual se corresponde con el valor en la columna NUM_CLIE de una de las filas de la tabla CLIENTES. Evidentemente, la sentencia SELECT que maneja la petición debe utilizar de algún modo este enlace entre tablas para generar sus resultados. Antes de examinar la sentencia SELECT correspondiente a la consulta, es instructivo imaginar cómo podríamos manejar manualmente la petición, utilizando lápiz y papel. La Figura 7.2 muestra lo que probablemente habría que hacer:



NUM_PEDIDO	IMPORTE	EMPRESA	LIMIT_CREDITO
112989	\$1,438.00	Jones Mfg.	\$65,000.00
112988	\$3,978.00	First Corp.	\$65,000.00
112963	\$3,276.00	Acme Mfg.	\$50,000.00
112987	\$27,500.00	Acme Mfg.	\$50,000.00
112983	\$702.00	Acme Mfg.	\$50,000.00
113027	\$4,104.00	Fred Lewis Corp.	\$65,000.00
112993	\$1,896.00	Fred Lewis Corp.	\$65,000.00
113085	\$2,130.00	Ace International	\$35,000.00
113036	\$22,500.00	Ace International	\$35,000.00
113034	\$632.00	Ace International	\$35,000.00
113058	\$1,480.00	Holm & Landis	\$55,000.00
113055	\$1,150.00	Holm & Landis	\$55,000.00
113003	\$5,625.00	Holm & Landis	\$55,000.00
...			

Esta tiene el mismo aspecto que las consultas del capítulo anterior, con dos características novedosas. Primero, la cláusula FROM lista dos tablas en lugar de una sola. En segundo lugar, la condición de búsqueda:

CLIE = NUM_CLIE

compara columnas de dos tablas diferentes. A estas dos columnas las llamamos las *columnas de emparejamiento* para las dos tablas. Como todas las condiciones de búsqueda, ésta restringe las filas que aparecen en los resultados de la consulta. Puesto que ésta es una consulta de dos tablas, la condición de búsqueda restringe las *pares* de filas que generan los resultados. De hecho, la condición de búsqueda especifica las mismas columnas de emparejamiento utilizadas en el proceso de consulta de lápiz y papel. Realmente capture el espíritu de la correspondencia manual de las columnas muy bien, diciendo:

«Genera resultados sólo para los pares de filas en los que el número de clientes (CLIE) en la tabla PEDIDOS coincide con el número de cliente (NUM_CLIE) en la tabla CLIENTES.»

Observe que la sentencia SELECT no dice nada acerca de *cómo* SQL debería ejecutar la consulta. No hay mención de «comenzar con los pedidos» o «comenzar con los clientes». En lugar de ello, la consulta dice a SQL *qué* resultados deberían aparecer, y deja a SQL que decida cómo generarlos.

Las composiciones proporcionan por tanto una potente facilidad para *ejercitarse* las relaciones de datos dentro de una base de datos. Puesto que SQL maneja las consultas multitable mediante comparación de columnas, no debería ser una sorpresa que la sentencia SELECT para una consulta multitable deba contener una condición de búsqueda que especifique la comparación de columnas. He aquí la sentencia SELECT correspondiente a la consulta que fue efectuada manualmente en la Figura 7.2:

Lista todos los pedidos mostrando su número, importe, número de cliente y el límite de crédito del cliente.

```
SELECT NUM_PEDIDO, IMPORTE, EMPRESA, LIMIT_CREDITO
  FROM PEDIDOS, CLIENTES
 WHERE CLIE = NUM_CLIE
```

Consultas padre/hijo

Las consultas multitable más comunes implican a dos tablas que tienen una relación natural padre/hijo. La consulta referente a pedidos y clientes de la sección precedente es un ejemplo de tal tipo de consulta. Cada pedido (hijo) tiene un cliente asociado (padre), y cada cliente (padre) puede tener muchos pedidos asociados (hijos). Los pares de filas que generan los resultados de la consulta son combinaciones de fila padre/hijo.

Podemos recordar del Capítulo 4 que las claves foráneas y las claves primarias crean relaciones padre/hijo en una base de datos SQL. La tabla que contiene la clave foránea es el hijo en la relación; la tabla con la clave primaria es el padre. Para ejercitarse la relación padre/hijo en una consulta debe especificarse una condición de búsqueda que compare la clave foránea y la clave primaria. He aquí otro ejemplo de una consulta que ejerce una relación padre/hijo, mostrada en la Figura 7.3.

Lista cada uno de los vendedores y la ciudad y región en donde trabaja.

```
SELECT NOMBRE, CIUDAD, REGION
  FROM REPVENTAS, OFICINAS
 WHERE OFICINAREP = OFICINA
```

NOMBRE	CIUDAD	REGION
Mary Jones	New York	Este
Sam Clark	New York	Este
Bob Smith	Chicago	Este
Paul Cruz	Chicago	Este
Dan Roberts	Chicago	Este
Bill Adams	Atlanta	Este
Sue Smith	Los Angeles	Oeste
Larry Fitch	Los Angeles	Oeste
Nancy Angelli	Denver	Oeste

La tabla REPVENTAS (hijo) contiene OFICINAREP, una clave foránea para la tabla OFICINAS (padre). Esta relación se utiliza para hallar la fila OFICINA correcta para cada vendedor, de modo que pueda incluirse la ciudad y región correctas en los resultados de la consulta.

He aquí otra consulta que referencia las mismas dos tablas, pero con los papeles de padre e hijo invertidos, tal como se muestra en la Figura 7.4:

Lista las oficinas y los nombres y títulos de sus directores.

```
SELECT CIUDAD, NOMBRE, TITULO
  FROM OFICINAS, REPVENTAS
 WHERE DIR = NUM_EMPL
```

Tabla OFICINAS					
OFIC	CIUDAD	REGION	DIR	OBJETIVO	VENTAS
22	Denver	Oeste	108	\$300,000.00	\$186,000.00
11	New York	Este	106	\$575,000.00	\$692,637.00
12	Chicago	Este	104	\$800,000.00	\$755,042.00
13	Atlanta	Este	NULL	\$350,000.00	\$367,911.00
21	Los Angeles	Oeste	108	\$725,000.00	\$835,915.00

Tabla REPVENTAS					
NUM_EMPL	NOMBRE	ED	OFICINAREP	TITULO	CLUIDAD REGION
105	Bill Adams	37	13	Rep Ventas	
109	Mary Jones	31	11	Rep Ventas	
102	Sue Smith	48	21	Rep Ventas	
106	Sam Clark	52	11	VP Ventas	
104	Bob Smith	33	12	Dir Ventas	
101	Dan Roberts	45	12	Rep Ventas	
110	Tom Snyder	41	NULL	Rep Ventas	
108	Larry Fitch	62	21	Dir Ventas	
103	Paul Cruz	29	12	Rep Ventas	
107	Nancy Angelli	49	22	Rep Ventas	

Figura 7.3. Una consulta padre/hijo con OFICINAS y REPVENTAS.

NOMBRE	CIUDAD	REGION
Bob Smith	Chicago	Dir Ventas
Bill Adams	Atlanta	Rep Ventas
Sam Clark	New York	VP Ventas
Larry Fitch	Denver	Dir Ventas
Larry Fitch	Los Angeles	Dir Ventas

La tabla OFICINAS (hijo) contiene DIR, una clave foránea para la tabla REPVENTAS (padre). Esta relación se utiliza para hallar la fila REPVENTAS correcta para cada vendedor, de modo que puedan incluirse el nombre y el título correctos del director en los resultados de la consulta.

SQL no requiere que las columnas de emparejamiento sean incluidas en los resultados de una consulta multitable. Con frecuencia pueden ser omitidas en la práctica como en los dos ejemplos precedentes. Esto es debido a que las claves primarias y las claves foráneas suelen ser números de identificación (tal como los números de oficina y números de empleado en los ejemplos), que los humanos encuentran difíciles de recordar, mientras que los nombres asociados (ciudades, regiones, nombres, títulos) son más fáciles de entender. Es bastante habitual que los números de identificación sean utilizados en la cláusula WHERE para componer dos tablas, y que se especifiquen nombres más descriptivos en la cláusula SELECT para generar las columnas de los resultados de la consulta.

segundo test selecciona adicionalmente sólo aquellos pares de filas en donde la oficina está por encima del objetivo.

Múltiples columnas de emparejamiento

La tabla PEDIDOS y la tabla PRODUCTOS de la base de datos ejemplo están relacionadas por un par de claves foránea/primaria. Las columnas FAB Y PRODUCTO de la tabla PEDIDOS forman juntas una clave foránea para la tabla PRODUCTOS, que se emparejan con las columnas ID_FAB E ID_PRODUCTO, respectivamente. Para combinar las tablas basándose en esta relación padre/hijo, deben especificarse *ambos* pares de columnas de emparejamiento, tal como se muestra en este ejemplo:

Tabla OFICINAS			
OFICINA	CIUDAD	REGION	DIR
22	Denver	Oeste	103
11	New York	Este	106
12	Chicago	Este	104
13	Atlanta	Este	NULL
21	Los Angeles	Oeste	108

Tabla REPVENTAS			
NUM_EMPL	NOMBRE	EDAD	OFICINA REP
105	Bill Adams	37	13
109	Mary Jones	31	11
102	Sue Smith	48	21
106	Sam Clark	52	11
104	Bob Smith	33	12
101	Dan Roberts	45	12
110	Tom Snyder	41	NULL
103	Larry Fitch	62	21
105	Paul Cruz	29	12
107	Nancy Angelini	49	22

Figura 7.4. Una consulta padre/hijo diferente con OFICINAS y REPVENTAS.

Resultados de la consulta			
CIUDAD	NOMBRE	TÍTULO	
Denver	Bob Smith	Dir Ventas	
New York	Mary Jones	Rep Ventas	
Chicago	Sam Clark	Rep Ventas	
Atlanta	Tom Snyder	Rep Ventas	
Los Angeles	Larry Fitch	Rep Ventas	

Figura 7.4. Una consulta padre/hijo diferente con OFICINAS y REPVENTAS.

Composiciones con criterios de selección de fila

La condición de búsqueda que especifica las columnas de emparejamiento en una consulta multitable puede combinarse con otras condiciones de búsqueda para restringir aún más los contenidos de los resultados. Supongamos que se desea volver a correr la consulta anterior, mostrando únicamente las oficinas con mayores objetivos de ventas:

Lista las oficinas con un objetivo superior a \$600.000.

```
SELECT CIUDAD, NOMBRE, TITULO
FROM OFICINAS, REPVENTAS
WHERE DIR = NUM_EMPL
AND OBJETIVO < 600000.0
CIUDAD      NOMBRE      TITULO
Chicago    Bob Smith   Dir Ventas
Los Angeles Larry Fitch Dir Ventas
```

Con la condición de búsqueda adicional, las filas que aparecen en los resultados están aún más restringidas. El primer test (DIR = NUM_EMPL) selecciona solamente pares de filas OFICINAS y REPVENTAS que tienen la adecuada relación padre/hijo; el

condición de búsqueda en la consulta dice a SQL que los pares de filas relacionados en las tablas PEDIDOS Y PRODUCTOS son aquéllas en las que ambos pares de columnas coincidentes contienen los mismos valores. Las composiciones multicolumna son menos habituales que las composiciones monocolumna, y se encuentran generalmente en consultas que afectan a claves foráneas compuestas tal como las del ejemplo.

Consultas de tres o más tablas

SQL puede combinar datos de tres o más tablas utilizando las mismas técnicas básicas utilizadas para las consultas de dos tablas. He aquí un sencillo ejemplo de una composición con tres tablas:

Lista los pedidos superiores a \$25.000, incluyendo el nombre del vendedor que tomó el pedido y el nombre del cliente que lo solicitó.

```
SELECT NUM_PEDIDO, IMPORTE, EMPRESA, NOMBRE
FROM PEDIDOS, CLIENTES, REPVENTAS
WHERE CLIE = NUM_CLIE
AND REP = NUM_EMPL
AND IMPORTE > 25000.00
```

NUM_PEDIDO	IMPORTE	EMPRESA	NOMBRE
112987	\$27,500.00	Acme Mfg.	Bill Adams
113069	\$31,350.00	Chen Associates	Nancy Angelli
113045	\$45,000.00	Zetacorp	Larry Fitch
112961	\$31,500.00	J.P. Sinclair	Sam Clark

Esta consulta utiliza dos claves foráneas en la tabla PEDIDOS, tal como se muestra en la Figura 7.5. La columna CLIE es una clave foránea para la tabla CLIENTES, que enlaza cada pedido con el cliente que lo remitió. La columna REP es una clave foránea para la tabla REPVENTAS, que liga cada pedido con el vendedor que lo aceptó. Hablando informalmente, la consulta liga cada pedido con su cliente y vendedor asociados.

He aquí otra consulta de tres tablas que utiliza una disposición diferente de las relaciones padre/hijo:

Lista los pedidos superiores a \$25.000, mostrando el nombre del cliente que remitió el pedido y el nombre del vendedor asignado a ese cliente.

```
SELECT NUM_PEDIDO, IMPORTE, EMPRESA, NOMBRE
FROM PEDIDOS, CLIENTES, REPVENTAS
WHERE CLIE = NUM_CLIE
AND REP_CLIE = NUM_EMPL
AND IMPORTE > 25000.00
```

NUM_PEDIDO	IMPORTE	EMPRESA	NOMBRE
112987	\$27,500.00	Acme Mfg.	Bill Adams
113069	\$31,500.00	Chen Associates	Paul Cruz
113045	\$45,000.00	Zetacorp	Larry Fitch
112961	\$31,500.00	J.P. Sinclair	Sam Clark

La Figura 7.6 muestra las relaciones ejercitadas por esta consulta. La primera relación utiliza de nuevo la columna CLIE de la tabla PEDIDO de la tabla PEDIDOS como clave foránea para la tabla CLIENTES. La segunda utiliza la columna REP_CLIE de la tabla CLIENTES como clave foránea para la tabla REPVENTAS. Hablando informalmente, esta consulta liga cada pedido con su cliente y cada cliente con su vendedor.

No es frecuente encontrar consultas de tres o incluso cuatro tablas en aplicaciones SQL de producción. Incluso dentro de los confines de la sencilla base de datos ejemplo que contiene cinco tablas, no es difícil hallar una consulta de cuatro tablas que tenga sentido:

Lista los pedidos superiores a \$25.000, mostrando el nombre del cliente que lo ordenó, el vendedor asociado al cliente y la oficina en donde el vendedor trabaja.

```
SELECT NUM_PEDIDO, IMPORTE, EMPRESA, NOMBRE
FROM PEDIDOS, CLIENTES, REPVENTAS, OFICINAS
WHERE CLIE = NUM_CLIE
AND REP_CLIE = NUM_EMPL
AND OFICINA.REP = OFICINA_CLIE
AND IMPORTE > 25000.00
```

Resultados de la consulta			
NUM_PEDIDO	IMPORTE	EMPRESA	NOMBRE
112987	\$27,500.00	Acme Mfg.	Bill Adams
113069	\$31,350.00	Chen Associates	Paul Cruz
113045	\$45,000.00	Zetacorp	Larry Fitch
112961	\$31,500.00	J.P. Sinclair	Sam Clark

Figura 7.5. Una composición con tres tablas.

Atlanta
Chicago
Los Angeles
New York

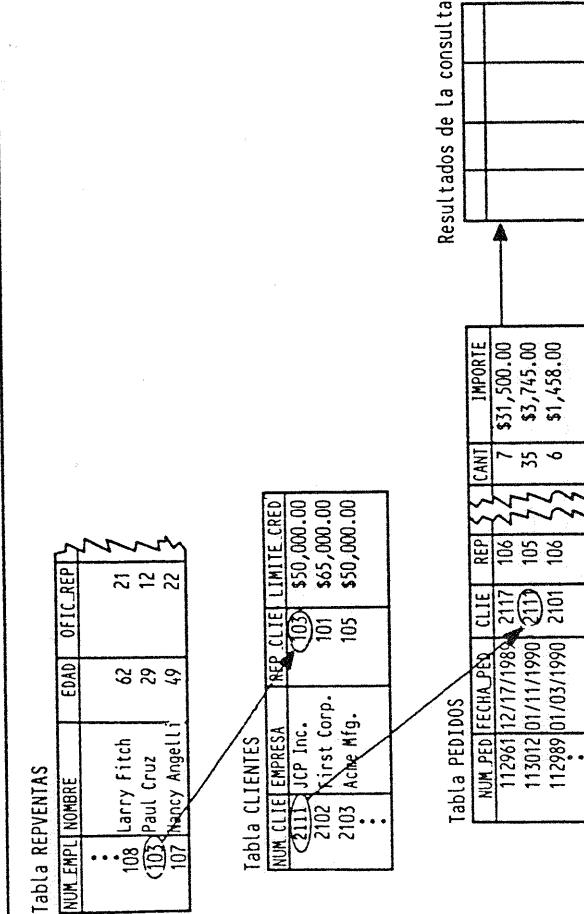


Figura 7.6. Una composición de tres tablas con relaciones padre-hijo en cascada.

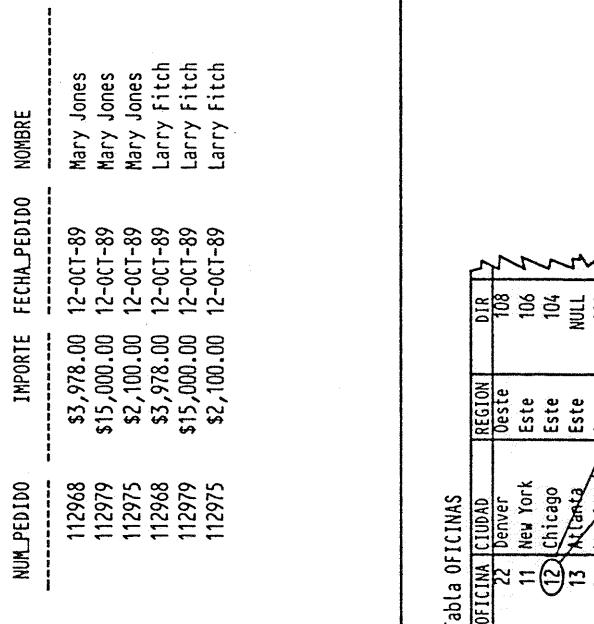


Figura 7.7. Una composición con cuatro tablas.

La Figura 7.7 muestra las relaciones padre/hijo de esta consulta. Lógicamente, amplia la secuencia de composición del ejemplo anterior en un paso más, ligando un pedido a su cliente, el cliente a su vendedor y el vendedor a su oficina.

Otras equicomposiciones

La inmensa mayoría de las consultas multitable se basan en relaciones padre/hijo, pero SQL no exige que las columnas de emparejamiento estén relacionadas como clave primaria y clave foránea. Cualquier par de columnas de dos tablas pueden servir como columnas de emparejamiento, siempre que tengan tipos de datos comparables. He aquí un ejemplo de una columna que utiliza un par de fechas como columnas de emparejamiento:

Halla todos los pedidos recibidos en los días en que un nuevo vendedor fue contratado.

```
SELECT NUM_PEDIDO, IMPORTE, FECHA_PEDIDO, NOMBRE
  FROM PEDIDOS, REPVENTAS
 WHERE FECHA_PEDIDO = CONTRATO
```

Figura 7.7. Una composición con cuatro tablas.

Los resultantes de esta consulta provienen de los pares de filas de las tablas PEDIDOS y REPVENTAS en donde el valor en la columna FECHAPEDIDO coincide con el valor en la columna CONTRATO para el vendedor, como se ve en la Figura 7.8. Ninguna de estas columnas es una clave primaria o una clave foránea, y la relación entre los pares de filas es ciertamente una relación extraña —la única cosa que los pedidos y vendedores correspondientes tienen en común es que resultan tener las mismas fechas—. Sin embargo, SQL compone las tablas felizmente.

Las columnas de emparejamiento como las de este ejemplo generan una relación de muchos a muchos entre las dos tablas. Puede haber muchos pedidos que comparten una única fecha de contrato del vendedor, y más de un vendedor puede haber sido contratado en la fecha de pedido de un determinado pedido. Por ejemplo, observe que tres pedidos diferentes (112968, 112975 y 112979) fueron recibidos el 12 de octubre de 1989, y dos vendedores diferentes (Larry Fitch y Mary Jones) fueron contratados el mismo día. Los tres pedidos y los dos vendedores producen seis filas de resultados de la consulta.

- Las composiciones que asocian claves primarias a claves foráneas siempre crean relaciones padre/hijo de uno a muchos.
- Otras composiciones también pueden generar relaciones de uno a muchos, si la columna de emparejamiento en al menos una de las tablas tiene valores únicos para todas las filas de la tabla.
- En general, las composiciones sobre las columnas de emparejamiento arbitrarias generan relaciones de muchos a muchos.

Observe que estas tres situaciones diferentes no tienen nada que ver con cómo se escribe la sentencia SELECT que expresa la composición. Los tres tipos de composiciones se escriben de la misma manera —incluyen un test de comparación para los pares de columnas de emparejamiento en la cláusula WHERE—. No obstante, es útil pensar acerca de las composiciones de esta manera para entender cómo transformar una petición en lenguaje ordinario a una sentencia SELECT correcta.

Composiciones basadas en desigualdad

El término «composición» (*join*) se aplica a cualquier consulta que combina datos de dos tablas mediante comparación de los valores en una pareja de columnas de las tablas. Aunque las composiciones basadas en la igualdad entre columnas correspondientes (equicomposiciones) son con mucho las composiciones más habituales, SQL también permite componer tablas basándose en otros operadores de comparación. He aquí un ejemplo en donde se utiliza un test de comparación mayor que (>) como base para una composición.

Lista todas las combinaciones de vendedores y oficinas en donde la cuota del vendedor es superior al objetivo de la oficina.

```
SELECT NOMBRE, CUOTA, CIUDAD, OBJETIVO
  FROM REPVENTAS, OFICINAS
 WHERE CUOTA > OBJETIVO
```

NOMBRE	CUOTA	CIUDAD	OBJETIVO
Bill Adams	\$350,000.00	Denver	\$300,000.00
Sue Smith	\$350,000.00	Denver	\$300,000.00
Larry Fitch	\$350,000.00	Denver	\$300,000.00

Como en todas las consultas de dos tablas, cada fila de resultados proviene de un par de filas, en este caso de las tablas REPVENTAS y OFICINAS. La condición de búsqueda:

CUOTAS > OBJETIVO

Figura 7.8. Una composición que no utiliza claves primarias ni foráneas.

Esta relación de muchos a muchos es diferente de la relación de uno a muchos creada por columnas de emparejamiento clave primaria/clave foránea. La situación puede resumirse del siguiente modo:

Tabla PEDIDOS		
NUM_PEDIDO	FECHAPEDIDO	CLI
...		
113051	02/10/1990	2118
112968	07/12/1989	2102
113056	01/30/1990	2107
...		

Tabla REPVENTAS	
NUM_EMPL_NOMBRE	CONTRATO
103 Bill Adams	02/12/1988
109 Mary Jones	07/12/1989
102 Sue Smith	12/10/1986
106 Sam Clark	06/14/1988
104 Bob Smith	05/19/1987
101 Dan Roberts	10/20/1986
110 Tom Snyder	01/13/1990
108 Larry Fitch	02/12/1989
103 Paul Cruz	03/01/1987
107 Nancy Angelli	11/14/1989
...	

selecciona pares de filas en donde la columna CIUDAD de la fila REPVENTAS excede a la columna OBJETIVO de la fila OFICINAS. Observe que los pares de filas REPVENTAS y OFICINAS están relacionadas *únicamente* de este modo; no se requiere específicamente que la fila REPVENTAS represente a alguien que trabaja en la oficina representada por la fila OFICINAS. Evidentemente, el ejemplo es un poco extremado, e ilustra por qué las composiciones basadas en desigualdades no son muy comunes. Sin embargo, pueden ser útiles en aplicaciones de soporte de decisiones y en otras aplicaciones que exploran interrelaciones más complejas en la base de datos.

Consideraciones SQL para consultas multitable

Las consultas multitable descritas hasta ahora no han requerido ninguna característica especial del lenguaje o la sintaxis SQL aparte de las descritas para las consultas monotabla. Sin embargo, algunas consultas multitable no pueden ser expresadas sin las características adicionales del lenguaje SQL descritas en las secciones siguientes. Específicamente:

- Los *nombres de columna cualificados* son necesarios a veces en consultas multitable para eliminar referencias de columna ambiguas.
- Las *selecciones de todas las columnas* (SELECT *) tienen un significado especial para las consultas multitable.
- Las *autocomposiciones* pueden ser utilizadas para crear una consulta multitable que relaciona una tabla consigo misma.
- Los *alias de tablas* pueden ser utilizados en la cláusula FROM para simplificar nombres de columna cualificados y permitir referencias de columna no ambiguas en autocomposiciones.

Nombres de columna cualificados

La base de datos ejemplo incluye varias instancias en donde dos tablas contienen columnas con el mismo nombre. La tabla OFICINAS y la tabla REPVENTAS, por ejemplo, tienen ambas una columna de nombre VENTAS. La columna de la tabla OFICINAS contiene las ventas anuales hasta la fecha para cada oficina; la de la tabla REPVENTAS contiene las ventas anuales hasta la fecha de cada vendedor. Normalmente, no hay confusión entre ambas columnas, ya que la cláusula FROM determina cuál de ellas es la adecuada en una consulta determinada, como en estos ejemplos:

Muestra las ciudades en donde las ventas superan al objetivo.

```
SELECT CIUDAD, VENTAS
      FROM OFICINAS
     WHERE VENTAS > OBJETIVO
```

Muestra todos los vendedores con ventas superiores a \$350.000.

```
SELECT NOMBRE, VENTAS
      FROM REPVENTAS
     WHERE VENTAS > 350000.00
```

Sin embargo, he aquí una consulta en donde los nombres duplicados provocan un problema:

Muestra el nombre, las ventas y la oficina de cada vendedor.

```
SELECT NOMBRE, VENTAS, CIUDAD
      FROM REPVENTAS, OFICINAS
     WHERE OFICINA.REP = OFICINA
```

Error: Nombre de columna ambiguo "VENTAS"

Aunque la descripción textual de la consulta implica que se desea la columna VENTAS de la tabla REPVENTAS, la consulta SQL es ambigua. Para eliminar la ambigüedad, debe utilizarse un nombre de columna cualificado para identificar la columna. Recuérdese del Capítulo 5 que un nombre de columna cualificado especifica el nombre de una columna y la tabla que contiene a la columna. Los nombres cualificados de las dos columnas VENTAS en la base de datos ejemplo son:

```
OFICINAS.VENTAS y REPVENTAS.VENTAS
```

Un nombre de columna cualificado puede ser utilizado en una sentencia SELECT en cualquier lugar en donde se permite un nombre de columna. La tabla especificada en el nombre de columna cualificado, debe, naturalmente, corresponder a una de las tablas especificadas en la lista FROM. He aquí una versión corregida de la consulta anterior que utiliza un nombre de columna cualificado:

Muestra el nombre, las ventas y la oficina de cada vendedor.

```
SELECT NOMBRE, REPVENTAS.VENTAS, CIUDAD
      FROM REPVENTAS, OFICINAS
     WHERE OFICINA.REP = OFICINA
```

NOMBRE	REPVENTAS.VENTAS	CIUDAD
Mary Jones	\$392,725.00	New York
Sam Clark	\$299,912.00	New York
Bob Smith	\$142,594.00	Chicago
Paul Cruz	\$286,775.00	Chicago
Dan Roberts	\$305,673.00	Chicago
Bill Adams	\$367,911.00	Atlanta
Sue Smith	\$474,050.00	Los Angeles
Larry Fitch	\$361,865.00	Los Angeles
Nancy Angelli	\$186,042.00	Denver

Utilizar nombres de columna cualificados en una consulta multitable es siempre una buena medida. La desventaja, naturalmente, es que hacen que el texto de la consulta sea mayor. Cuando se utiliza SQL interactivo, puede ser deseable intentar primeramente una consulta con nombres de columna sin cualificar y dejar que SQL encuentre las columnas ambiguas. Si SQL informa de un error, se puede volver a editar la consulta para cualificar las columnas ambiguas.

Selecciones de todas las columnas

Como se discutió en el Capítulo 6, «SELECT *» puede ser utilizado para seleccionar todas las columnas de la tabla designada en la cláusula FROM. En una consulta multitable, el asterisco selecciona todas las columnas de todas las tablas listadas en la cláusula FROM. La siguiente consulta, por ejemplo, produciría 15 columnas de resultados —las 9 columnas de la tabla REPVENTAS seguidas de las 6 columnas de la tabla OFICINAS:

Informa sobre todo los vendedores y todas las oficinas en las que trabajan.

```
SELECT *
FROM REPVENTAS, OFICINAS
WHERE OFICINA.REP = OFICINA
```

Obviamente, la forma SELECT * de una consulta resulta ser mucho menos práctica cuando hay dos, tres o más tablas en la cláusula FROM.

Muchos dialectos SQL tratan el asterisco como un tipo especial de nombre de columna comodín que se expande a un lista de columnas. En estos dialectos, el asterisco puede ser cualificado con un nombre de tabla, al igual que una referencia de columna cualificada. En la siguiente consulta, el ítem de selección REPVENTAS.* se expande a una lista que contiene únicamente las columnas halladas en la tabla REPVENTAS:

Informa acerca de todos los vendedores y los lugares en los que trabajan.

```
SELECT REPVENTAS.* , CIUDAD, REGION
      FROM REPVENTAS, OFICINAS
     WHERE OFICINA.REP = OFICINA
```

La consulta produciría once columnas de resultados —las nueve columnas de la tabla REPVENTAS, seguidas de las dos columnas explicitamente solicitadas de la tabla OFICINAS—. Aunque este tipo de selección «todas las columnas cualificadas» se soporta en muchos productos de DBMS basados en SQL, no está permitido por el estándar ANSI/ISO.

Autocomposiciones

Algunas consultas multitable afectan a una relación que una tabla tiene consigo misma. Por ejemplo, supongamos que se desea listar los nombres de todos los vendedores y sus directores. Cada vendedor aparece como una fila en la tabla REPVENTAS, y la columna DIRECTOR contiene el número de empleado del director del vendedor. Parecería que la columna DIRECTOR debería ser una clave foránea para la tabla que contiene datos referentes a los directores. En efecto así es —es una clave foránea para la propia tabla REPVENTAS.

Si se tratara de expresar esta consulta como cualquier otra consulta de dos tablas implicando una coincidencia clave foránea/clave primaria, aparecería tal como ésta:

```
SELECT NOMBRE, NOMBRE
      FROM REPVENTAS, REPVENTAS
     WHERE DIRECTOR = NUM_EMPL
```

Esta sentencia SELECT es ilegal debido a la referencia duplicada a la tabla REPVENTAS en la cláusula FROM. También podría intentarse eliminar la segunda referencia a la tabla REPVENTAS:

```
SELECT NOMBRE, NOMBRE
      FROM REPVENTAS
     WHERE DIRECTOR = NUM_EMPL
```

Esta consulta es legal, pero no hará lo que se desea que haga. Es una consulta monotabla, por lo que SQL recorre la tabla REPVENTAS fila a fila, aplicando la condición de búsqueda:

```
DIRECTOR = NUM_EMPL
```

Las filas que satisfacen esta condición son aquéllas en donde las dos columnas tienen el mismo valor —es decir, las filas en donde un vendedor es su propio director—. No existen tales filas, por lo que la consulta no produciría resultados.

Para comprender cómo resuelve SQL este problema, imaginemos que hubiera *dos copias idénticas* de la tabla REPVENTAS, una llamada EMPS, que contuviera los empleados, y otra llamada DIRS, que contuviera los directores, tal como se muestra en la Figura 7.9. La columna DIRECTOR de la tabla EMPS sería entonces una clave foránea para la tabla DIRS, y la siguiente consulta funcionaría:

Lista los nombres de los vendedores y sus directores.

```
SELECT EMPS.NOMBRE, DIRS.NOMBRE
FROM EMPS, DIRS
WHERE EMPS.DIRECTOR = DIRS.NUM_EMPL
```

Tabla DIRS (copia de REPVENTAS)

NUM_EMPL	NOMBRE	EDAD	OFICINA REP
105	Bill Adams	37	13
109	Mary Jones	31	11
102	Sue Smith	48	21
106	Sam Clark	52	11
104	Bob Smith	33	12
101	Dan Roberts	45	12
100	Tom Snyder	41	NULL
103	Larry Fitch	62	21
102	Paul Cruz	29	12
107	Nancy Angelli	49	22

Tabla EMPS (copia de REPVENTAS)

NUM_EMPL	NOMBRE	EDAD	OFICINA REP	TITULO	CONTRATO	DIRECTOR
105	Bill Adams	37	13	Rep Ventas	01/12/1988	105
109	Mary Jones	31	11	Rep Ventas	10/12/1989	106
102	Sue Smith	48	21	Rep Ventas	12/10/1986	108
106	Sam Clark	52	11	VP Ventas	06/14/1988	NULL
104	Bob Smith	33	12	Dir Ventas	05/19/1987	106
101	Dan Roberts	45	12	Rep Ventas	10/20/1986	107
110	Tom Snyder	41	NULL	Rep Ventas	01/13/1990	101
108	Larry Fitch	62	21	Dir Ventas	10/12/1989	106
103	Paul Cruz	29	12	Rep Ventas	03/01/1987	107
107	Nancy Angelli	49	22	Rep Ventas	11/14/1988	108

Figura 7.9. Una autocomposición de la tabla REPVENTAS.

Puesto que las columnas de las dos tablas tienen nombres idénticos, todas las referencias de columnas están cualificadas. En caso contrario, esto aparecería como una consulta ordinaria de dos tablas.

SQL utiliza exactamente esta estrategia de «tabla duplicada imaginaria» para componer una tabla consigo misma. En lugar de duplicar realmente el contenido de la tabla, SQL simplemente permite referirse a ella mediante un nombre diferente, llamado un *alias de tabla*. He aquí la misma consulta, escrita utilizando los alias EMPS y DIRS para la tabla REPVENTAS:

Lista los nombres de los vendedores y sus directores.

```
SELECT EMPS.NOMBRE, DIRS.NOMBRE
FROM REPVENTAS EMPS, REPVENTAS DIRS
WHERE EMPS.DIRECTOR = DIRS.NUM_EMPL
```

La cláusula FROM asigna un alias a cada «copia» de la tabla REPVENTAS especificando el nombre del alias inmediatamente después del nombre real de la tabla. Como muestra el ejemplo, cuando una cláusula FROM contiene un alias, el alias debe ser utilizado para identificar la tabla en referencias de columna cualificadas. Naturalmente sólo es realmente necesario utilizar un alias para una de las dos ocurrencias de tabla de esta consulta. Podríamos haber escrito más fácilmente:

```
SELECT REPVENTAS.NOMBRE, DIRS.NOMBRE
FROM REPVENTAS, REPVENTAS DIRS
WHERE REPVENTAS.DIRECTOR = DIRS.NUM_EMPL
```

Aquí el alias DIRS ha sido asignado a una «copia» de la tabla, mientras que el propio nombre de la tabla se utiliza para otra copia.

He aquí algunos ejemplos adicionales de autocomposiciones:

```
Lista los vendedores con una cuota superior a la de su director.

SELECT REPVENTAS.NOMBRE, REPVENTAS.CUOTA, DIRS.CUOTA
FROM REPVENTAS, REPVENTAS DIRS
WHERE REPVENTAS.DIRECTOR = DIRS.NUM_EMPL
AND REPVENTAS.CUOTA > DIRS.CUOTA
```

REPVENTAS.NOMBRE	REPVENTAS.CUOTA	DIRS.CUOTA
Bill Adams	\$350,000.00	\$200,000.00
Dan Roberts	\$300,000.00	\$200,000.00
Paul Cruz	\$275,000.00	\$200,000.00
Mary Jones	\$300,000.00	\$275,000.00
Larry Fitch	\$350,000.00	\$275,000.00

Lista los vendedores que trabajan en diferentes oficinas que sus directores, mostrando el nombre y la oficina en donde trabaja cada uno.

```

SELECT EMPS.NOMBRE, EMP_OFICINA.CIUDAD, DIRS.NOMBRE, DIR_OFICINA.CIUDAD
  FROM REPVENTAS EMPs, REPVENTAS DIRS
    OFICINAS EMP_OFICINA, OFICINAS DIR_OFICINA
 WHERE EMPS.OFICINA.REP = EMP_OFICINA.OFICINA
   AND DIRS.OFICINA.REP = DIR_OFICINA.OFICINA
   AND EMPs.DIRECTOR = DIRS.NUM_EMPL
   AND EMPs.OFICINA.REP <> DIRS.OFICINA.REP
EMP.S.NOMBRE      EMP_OFICINA.CIUDAD  DIRS.NOMBRE      DIR_OFICINA.CIUDAD
-----  -----  -----  -----
Bob Smith        Chicago          Sam Clark       New York
Bill Adams       Atlanta         Bob Smith      Chicago
Larry Fitch      Los Angeles    Sam Clark       New York
                  Denver          Larry Fitch   Los Angeles
Nancy Angelli

```

Alias de tablas

Como se ha descrito en la sección anterior, los alias de tablas se requieren en consultas que afectan autocomposiciones. Sin embargo, se puede utilizar un alias en cualquier consulta. Por ejemplo, si una consulta se refiere a la tabla de otro usuario, o si el nombre de una tabla es muy largo, puede ser tedioso de escribir el nombre de la tabla como cualificador de una columna. Esta consulta, que referencia la tabla CUMPLEAÑOS propiedad del usuario SAM:

Lista los nombres, cuotas y cumpleaños de los vendedores.

```

SELECT REPVENTAS.NOMBRE, CUOTA, SAM.CUMPLEAÑOS.FECHA
  FROM REPVENTAS, CUMPLEAÑOS
 WHERE REPVENTAS.NOMBRE = SAM.CUMPLEAÑOS.NOMBRE
      WHERE S.NOMBRE = B.NOMBRE

```

resulta más fácil de leer y escribir cuando se utilizan los alias S y B para las dos tablas:

Lista los nombres, cuotas y cumpleaños de los vendedores.

```

SELECT S.NOMBRE, S.CUOTA, B.FECHA
  FROM REPVENTAS S, SAM.CUMPLEAÑOS B
 WHERE S.NOMBRE = B.NOMBRE

```

La Figura 7.10 muestra el formato de la cláusula FROM para una sentencia SELECT multitable, completada con alias. La cláusula tiene dos funciones importantes:

- La cláusula FROM identifica todas las tablas que contribuyen con datos a los resultados de la consulta. Las tablas referenciadas en la sentencia SELECT deben provenir de una de las tablas designadas en la cláusula FROM. (Hay una excepción que afecta a las referencias externas contenidas en una subconsulta, como se describirá en el Capítulo 8.)
- La cláusula FROM determina la *marca* que se utiliza para identificar la tabla en referencias de columna cualificadas dentro de la sentencia SELECT. Si se especifica un alias, éste pasa a ser la marca de la tabla; en caso contrario se utiliza como marca el nombre de la tabla, tal como aparece en la cláusula FROM.

La única exigencia para las marcas de tablas en la cláusula FROM es que todas las marcas de una cláusula FROM determinada deben ser distintas unas de otras.

Rendimiento de consultas multitable

Al crecer el número de tablas en una consulta, el esfuerzo requerido para llevarla a cabo se incrementa rápidamente. El lenguaje SQL no pone límites al número de tablas compuestas en una consulta. Algunos productos SQL limitan el número de tablas, siendo bastante común un límite alrededor de ocho tablas. El alto coste de procesamiento de las consultas que componen muchas tablas impone un límite práctico aún más bajo en muchas aplicaciones.

En aplicaciones de procesamiento de transacciones en línea (OLTP), es habitual que una consulta afecte solamente a una o dos tablas. En estas aplicaciones, el tiempo de respuesta es crítico —el usuario introduce típicamente uno o dos datos y necesita una respuesta de la base de datos en un segundo o dos—. He aquí algunas consultas OLTP típicas para la base de datos ejemplo:

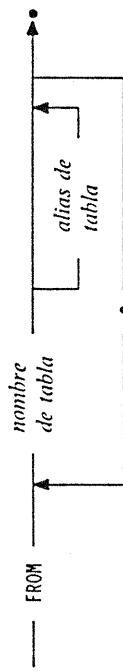


Figura 7.10. Diagrama sintáctico de la cláusula FROM.

las filas de una tabla no se empareja en este proceso, la composición puede producir resultados inesperados, como ilustran estas consultas:

Lista los vendedores y las oficinas en que trabajan.

```
SELECT NOMBRE, OFICINA REP
  FROM REVENTAS
 WHERE OFICINA REP = OFICINA
```

NOMBRE	OFICINA REP	NOMBRE	CIUDAD
Bill Adams	13	Tom Snyder	NULL
Mary Jones	11	Mary Jones	New York
Sue Smith	21	Sam Clark	New York
Sam Clark	11	Bob Smith	Chicago
Bob Smith	12	Paul Cruz	Chicago
Dan Roberts	12	Dan Roberts	Chicago
Tom Snyder	NULL	Bill Adams	Atlanta
Larry Fitch	21	Sue Smith	Los Angeles
Paul Cruz	12	Larry Fitch	Los Angeles
Nancy Angelli	22	Nancy Angelli	Denver

Lista los vendedores y las ciudades en que trabajan.

```
SELECT NOMBRE, CIUDAD
  FROM REVENTAS, OFICINAS
 WHERE OFICINA REP = OFICINA
```

NOMBRE	CIUDAD
Mary Jones	New York
Sam Clark	New York
Bob Smith	Chicago
Paul Cruz	Chicago
Dan Roberts	Chicago
Bill Adams	Atlanta
Sue Smith	Los Angeles
Larry Fitch	Los Angeles
Nancy Angelli	Denver

Basándonos en la versión en lenguaje ordinario de la petición, cabría esperar que la segunda consulta produjera resultados como éstos:

Lista los vendedores y las ciudades en que trabajan.

```
SELECT NOMBRE, CIUDAD
  FROM REVENTAS, OFICINAS
 WHERE OFICINA REP *= OFICINA
```

NOMBRE	CIUDAD		
Bill Adams	13	Tom Snyder	NULL
Mary Jones	11	Mary Jones	New York
Sue Smith	21	Sam Clark	New York
Sam Clark	11	Bob Smith	Chicago
Bob Smith	12	Paul Cruz	Chicago
Dan Roberts	12	Dan Roberts	Chicago
Tom Snyder	NULL	Bill Adams	Atlanta
Larry Fitch	21	Sue Smith	Los Angeles
Paul Cruz	12	Larry Fitch	Los Angeles
Nancy Angelli	22	Nancy Angelli	Denver

Estos resultados se generan utilizando un tipo diferente de operación de composición, llamada *composición externa* (indicada por la notación «*» en la cláusula WHERE). La composición externa es una extensión de la composición estándar descrita con anterioridad en este capítulo, la cual a veces se denomina *composición interna*. El estándar ANSI/ISO actual especifica únicamente la composición interna; no incluye la composición externa. Los productos SQL de IBM también soportan únicamente la composición interna. Sin embargo, la composición externa es un componente bien entendido y útil del modelo de base de datos relacional, y ha sido implementado en muchos productos SQL, incluyendo SQL Server, Oracle y SQLBase. La composición externa es también el modo más natural de expresar un cierto tipo de petición de consulta, tal como se muestra en el resto de esta sección.

Para entender la composición externa, es útil prescindir de la base de datos ejemplo y considerar las dos sencillas tablas de la Figura 7.13. La tabla CHICAS lista cinco chicas y las ciudades en donde viven; la tabla CHICOS lista cinco chicos y las ciudades en donde viven. Para encontrar las parejas chica/chica que viven en la misma ciudad, se podría utilizar esta consulta, que forma la composición interna de las dos tablas:

Lista los chicas y chicos que viven en la misma ciudad.

```
SELECT *
  FROM CHICAS, CHICOS
 WHERE CHICAS.CIUDAD = CHICOS.CIUDAD
```

Aparentemente, sería de esperar que estas dos consultas produjeran el mismo número de filas, pero la primera lista diez vendedores, y la segunda sólo nueve. ¿Por qué? Porque Tom Snyder está actualmente sin asignar y tiene un valor NULL en la columna OFICINAS.REP (que es la columna de emparejamiento para la composición). Este valor NULL no se corresponde con ninguno de los números de oficina en la tabla OFICINAS, por lo que la oficina de Tom en la tabla REVENTAS está sin emparejar. Como resultado, «desaparece» en la composición. La composición SQL estándar tiene por tanto el potencial de perder información si las tablas que se componen contienen filas sin emparejar.

ción externa, y la composición externa se muestra en la Figura 7.13. He aquí la sentencia SQL que produce la composición externa:

Lista las chicas y chicos de la misma ciudad, incluyendo las chicas y chicos desparejados

```
SELECT *
  FROM CHICAS, CHICOS
 WHERE CHICAS.CIUDAD *=* CHICOS.CIUDAD
```

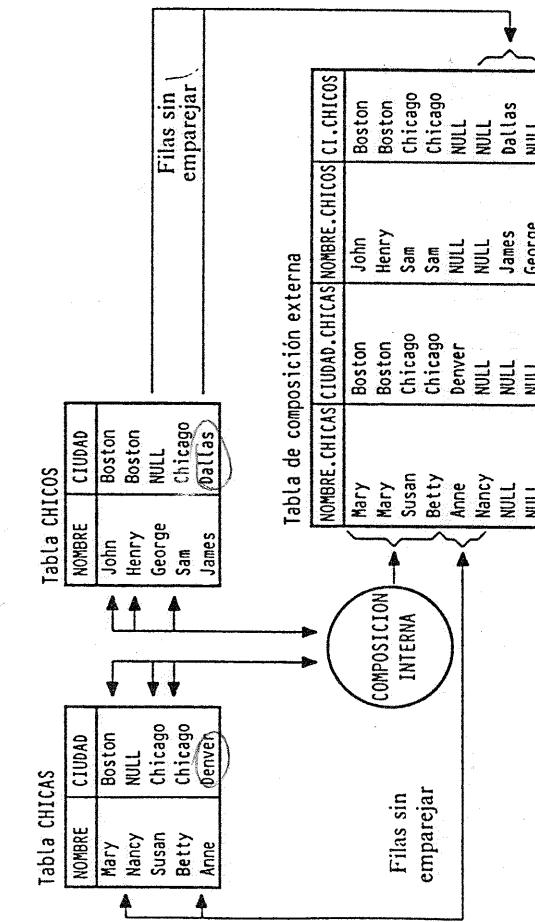


Figura 7.13. Anatomía de una composición externa.

La composición interna produce cuatro filas de resultados de la consulta. Observe que dos de las chicas (Anne y Nancy) y dos de los chicos (James y George) no están representados en los resultados. Estas filas no pueden emparejarse con ninguna fila de la otra tabla, por tanto faltan en los resultados de la composición interna. Dos de las filas sin emparejar (Anne y James) tienen valores válidos en sus columnas CIUDAD, pero no coinciden con ninguna de las ciudades de la tabla opuesta. Las otras dos filas sin emparejar (Nancy y George) tienen valores NULL en sus columnas CIUDAD, y por las reglas del manejo de NULL en SQL, el valor NULL no coincide con ningún otro valor (ni siquiera con otro valor NULL).

Supongamos que se desea listar los pares chico/chica que comparten las mismas ciudades, incluyendo los chicos y chicas desparejados en la lista. La composición externa de las tablas CHICAS y CHICOS produce exactamente este resultado. La Figura 7.14 muestra el procedimiento para construir la composición externa de las dos tablas.

La composición externa de las dos tablas contiene ocho filas. Cuatro de las filas son idénticas a las de la composición interna entre las dos tablas. Las otras dos filas, para Anne y Nancy, provienen de filas no coincidentes de la tabla CHICAS. Estas filas han sido NULL-ampliadas emparejándolas con una fila imaginaria de todo NULL en la tabla CHICOS, y añadidas a los resultados de la consulta. Las dos últimas filas, para James y George provienen de las filas no coincidentes de la tabla CHICOS. Estas filas también han sido NULL-ampliadas emparejándolas a una fila imaginaria de todo NULL en la tabla CHICAS, y añadidas a los resultados de la consulta.

Como muestra este ejemplo, la composición externa es una composición «preservadora de información». Cada fila de la tabla CHICOS está representada en los resultados (alguna más de una vez). Análogamente, cada fila de la tabla

1. Comenzar con la composición interna de las dos tablas, utilizando columnas de emparejamiento del modo normal.
2. Por cada fila de la primera tabla que no haya correspondido a ninguna fila de la segunda tabla, añadir una fila a los resultados, utilizando los valores de las columnas de la primera tabla, y suponiendo un valor NULL para todas las columnas de la segunda tabla.
3. Por cada fila de la segunda tabla que no haya correspondido a ninguna fila de la primera tabla, añadir una fila a los resultados, utilizando los valores de las columnas de la segunda tabla, y suponiendo un valor NULL para todas las columnas de la primera tabla.
4. La tabla resultante es la composición externa de las dos tablas.

Figura 7.14. Definición de una composición externa.

CHICAS está representada en los resultados de la consulta (de nuevo, alguna más de una vez).

Composiciones externas izquierda y derecha*

Técnicamente, la composición externa producida por la consulta anterior se denomina *composición externa completa* de las dos tablas. Ambas tablas son tratadas simétricamente en la composición externa completa. Hay otras dos composiciones externas bien definidas que no tratan a las dos tablas simétricamente.

La *composición externa izquierda* entre dos tablas se produce siguiendo el Paso 1 y el Paso 2 de la Figura 7.14, pero omitiendo el Paso 3. La composición externa izquierda por tanto incluye copias NULL-ampliadas de las filas no emparejadas de la primera (izquierda) tabla, pero no incluye las filas no emparejadas de la segunda (derecha) tabla. He aquí la composición externa izquierda entre las tablas CHICAS y CHICOS:

Lista las chicas y chicos de la misma ciudad y las chicas desparejadas.

```
SELECT *
  FROM CHICAS, CHICOS
 WHERE CHICAS.CIUDAD = CHICOS.CIUDAD
      -----
```

CHICAS.NOMBRE	CHICAS.CIUDAD	CHICOS.NOMBRE	CHICOS.CIUDAD
Mary	Boston	John	Boston
Mary	Boston	Henry	Boston
Susan	Chicago	Sam	Chicago
Betty	Chicago	Sam	Chicago
NULL	NULL	James	Dallas
NULL	NULL	George	NULL

La consulta produce seis filas de resultados, mostrando los pares chico/chica emparejados y las chicas desparejadas. Los chicos desparejados faltan en los resultados.

Análogamente, la *composición externa derecha* entre dos tablas se produce siguiendo el Paso 1 y el Paso 3 de la Figura 7.14, pero omitiendo el Paso 2. La composición externa derecha incluye por tanto copias NULL-ampliadas de las filas no emparejadas de la segunda (derecha) tabla, pero no incluye las filas no emparejadas de la primera (izquierda) tabla. He aquí una composición externa derecha entre las tablas CHICAS y CHICOS:

Lista las chicas y chicos de la misma ciudad y las chicas desparejadas.

```
SELECT *
  FROM CHICAS, CHICOS
 WHERE CHICAS.CIUDAD =* CHICOS.CIUDAD
      -----
```

CHICAS.NOMBRE	CHICAS.CIUDAD	CHICOS.NOMBRE	CHICOS.CIUDAD
Mary	Boston	John	Boston
Mary	Boston	Henry	Boston
Susan	Chicago	Sam	Chicago
Betty	Chicago	Sam	Chicago
NULL	NULL	James	Dallas
NULL	NULL	George	NULL

Esta consulta también produce seis filas de resultados, mostrando los pares chico/chica coincidentes y los chicos no coincidentes. Esta vez las chicas desparejadas faltan de los resultados.

En la práctica, las composiciones externa izquierda y derecha son más útiles que la composición externa total, especialmente en composiciones que afectan a emparejamientos clave foránea/clave primaria. En tal composición, la columna de clave foránea puede contener valores NULL, produciendo filas no emparejadas de la tabla hijo (la tabla que contiene la clave foránea). Una composición externa asimétrica incluiría estas filas hijo no emparejadas en los resultados de la consulta, sin incluir además las filas padre no emparejadas.

Notación de composición externa*

Ya que la composición externa no forma parte del estándar SQL ANSI/ISO y no está implementada en los productos SQL IBM, los vendedores DBMS que soportan la composición externa han utilizado variadas notaciones en sus dialectos SQL. La notación «* =*» utilizada en los ejemplos anteriores de esta sección es la utilizada por SQL Server. Esta notación indica una composición externa añadiendo un asterisco (*) al test de comparación de la cláusula WHERE que define la condición de composición. Para indicar la composición externa completa entre dos tablas, TBL1 y TBL2, sobre las columnas COL1 y COL2, se coloca un asterisco (*) delante y detrás del operador de composición externa completo WHERE COL1 * =* COL2.

Para indicar una composición externa izquierda, sólo se especifica el primer asterisco, dando un test de comparación como éste:

```
WHERE COL1 * =* COL2
```

Para indicar una composición externa derecha, sólo se especifica el segundo asterisco, dando un test de comparación como éste:

```
WHERE COL1 * =* COL2
```

Una composición externa puede utilizarse con cualquiera de los operadores de comparación utilizando la misma notación. Por ejemplo, una composición externa izquierda utilizando una comparación mayor o igual que (\geq) produciría un test de comparación como éste:

```
WHERE COL1 * $\geq$  COL2
```

Tanto Oracle como SQLBase soportan la operación de composición externa, pero utilizan una notación diferente. Esta notación indica la composición externa en la cláusula WHERE incluyendo un signo más entre paréntesis a continuación de la columna cuya tabla va a tener añadida la fila NULL imaginaria. La composición externa izquierda produce una condición de búsqueda tal como ésta:

```
WHERE COL1 = COL2 (+)
```

y la composición externa derecha produce una condición de búsqueda tal como ésta:

```
WHERE COL1 (+) = COL2
```

Observe que el signo más aparece en el lado opuesto de la comparación con respecto al lugar donde aparece el asterisco en la notación SQL Server. Ni Oracle ni SQLBase soportan una composición externa completa.

Aunque ambas notaciones de composición externa son relativamente convenientes, también hay algo frustrante. Las reglas para el procesamiento de consultas SQL de la Figura 7.12 comienzan construyendo el producto de las dos tablas, y eliminando luego las filas que no satisfacen la condición de búsqueda de la cláusula WHERE. Pero la tabla producto no incluye las filas NULL-ampliadas generadas por la composición externa. ¿Cómo se introducen en los resultados de la consulta? La respuesta es que la cláusula WHERE debe ser consultada cuando se forme el producto, para ver si deben incluirse las filas NULL-ampliadas. Además, una composición entre dos tablas puede implicar a más de un par de columnas de emparejamiento, y no está claro cómo debería utilizarse la notación cuando hay dos o tres pares de columnas de emparejamiento.

Otro problema con la notación de composición externa se presentan cuando se amplia a tres o más tablas. Conceptualmente, es fácil extender la notación de una composición externa a tres tablas:

```
TBL1 COMP-EXTERNA TBL2 COMP-EXTERNA TBL3
```

Pero el resultado depende del orden en que se efectúen las operaciones de composición externa. Los resultados de:

```
(TBL1 COMP-EXTERNA TBL2) COMP-EXTERNA TBL3
```

serán en general diferentes de los resultados de:

```
TBL1 COMP-EXTERNA (TBL2 COMP-EXTERNA TBL3)
```

Utilizando tanto las notaciones de SQL Server como las de Oracle o SQLBase, es imposible especificar el orden de evaluación de las composiciones externas. Debido a esto, los resultados producidos por la composición externa de tres o más tablas no están muy bien definidos.

Las propuestas para el estándar SQL2 ANSI/ISO incluyen soporte para la operación de composición externa. Para eliminar los problemas de notación que acabamos de discutir, la propuesta SQL2 especifica la composición externa en la cláusula FROM, designando explícitamente las tablas a combinar y el orden de composición. Si esta propuesta será o no aceptada como implementación estándar de la composición externa está aún por ver.

Resumen

Este capítulo ha descrito cómo gestiona SQL las consultas que combinan los datos de dos o más tablas:

- En una consulta multitableta, las tablas que contienen los datos son designadas en la cláusula FROM.
- Cada fila de resultados es una combinación de datos procedentes de una única fila en cada una de las tablas, y es la *única* fila que extrae sus datos de esa combinación particular.
- Las consultas multitableta más habituales utilizan las relaciones padre/hijo creadas por las claves primarias y claves foráneas.
- En general, las composiciones pueden construirse comparando *cualquier* par(es) de columnas de las dos tablas compuestas, utilizando un test de desigualdad o cualquier otro test de comparación.
- Una composición puede ser considerada como el producto de dos tablas del cual se han suprimido algunas de las filas.
- Una tabla puede comprenderse consigo misma; las autocomposiciones requieren el uso de alias.
- Las composiciones externas amplían la composición estándar (interna) reteniendo las filas no emparejadas de las tablas compuestas en los resultados de la consulta.

8 Consultas sumarias

Muchas peticiones de información no requieren el nivel de detalle proporcionado por las consultas SQL descritas en los últimos dos capítulos. Por ejemplo, cada una de las siguientes peticiones solicita un único valor o un pequeño número de valores que sumarizan los contenidos de la base de datos:

- ¿Cuál es la cuota total para todos los vendedores?
- Cuáles son las cuotas asignadas mínima y máxima?
- ¿Cuántos vendedores han superado su cuota?
- ¿Cuál es el tamaño del pedido medio?
- ¿Cuál es el tamaño del pedido medio para cada oficina de ventas?
- ¿Cuántos vendedores hay asignados a cada oficina de ventas?

SQL soporta estas peticiones de datos sumarios mediante funciones de columna y mediante las cláusulas GROUP BY y HAVING de la sentencia SELECT, que se describen en este capítulo.

Funciones de columna

SQL permite summarizar datos de la base de datos mediante un conjunto de *funciones de columna*. Una función de columna SQL acepta una columna entera de datos como argumentos y produce un único dato que sumariza la columna. Por ejemplo, la función de columna AVG() acepta una columna de datos y calcula

su promedio. He aquí una consulta que utiliza la función de columna `AVG()` para calcular el valor promedio de dos columnas de la tabla REPVENTAS:

```
¿Cuál es la cuota promedio y las ventas promedio de los vendedores?
SELECT AVG(CUOTA), AVG(VENTAS)
FROM REPVENTAS
      AVG(CUOTA)    AVG(VENTAS)
-----  -----
$300,000.00   $289,353.20
```

La Figura 8.1 muestra gráficamente cómo se producen los resultados de la consulta. La primera función de columna de la consulta toma los valores de la columna CUOTA y calcula su promedio. La segunda promedia los valores de la columna VENTAS. La consulta produce una única fila de resultados que sumarizan los datos de la tabla REPVENTAS.

SQL ofrece seis funciones de columna diferentes, tal como se muestra en la Figura 8.2. Las funciones de columna ofrecen diferentes tipos de datos sumarios:

- `SUM()` calcula el total de una columna.
- `AVG()` calcula el valor promedio de una columna.
- `MIN()` encuentra el valor más pequeño en una columna.
- `MAX()` encuentra el valor mayor en una columna.
- `COUNT()` cuenta el número de valores en una columna.
- `COUNT(*)` cuenta las filas de resultados de la consulta.

El argumento de una función columna puede ser un solo nombre de columna, como en el ejemplo anterior, o puede ser una expresión SQL, como se muestra en la página siguiente:

¿Cuál es el rendimiento de cuota promedio de los vendedores?

```
SELECT AVG(100 * (VENTAS / CUOTA))
FROM REPVENTAS
      AVG(100*(VENTAS/CUOTA))
-----  -----
102.60
```

Para procesar esta consulta, SQL construye una columna temporal que contiene el valor de la expresión $(100 * (VENTAS / CUOTA))$ por cada fila de la tabla REPVENTAS, y luego calcula los promedios de la columna temporal.

Tabla REPVENTAS	
NUM_EMPL	NOMBRE
105	Bill Adams
109	Mary Jones
102	Sue Smith
106	Sam Clark
104	Bob Smith
101	Dan Roberts
110	Ton Snyder
108	Larry Fitch
103	Paul Cruz
107	Nancy Angelli
105	Bill Adams

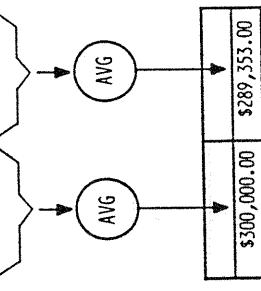


Figura 8.1. Una consulta sumaria en operación.

Cálculo del total de una columna (SUM)

La función columna `SUM()` calcula la suma de una columna de valores de datos. Los datos de la columna deben tener un tipo numérico (entero, decimal, coma flotante o monetario). El resultado de la función `SUM()` tiene el mismo tipo de dato básico que los datos de la columna, pero el resultado puede tener una precisión superior. Por ejemplo, si se aplica una función `SUM()` a una columna de enteros de 16 bits, se puede producir un entero de 32 bits como resultado.

He aquí algunos ejemplos que utilizan la función de columna `SUM()`:

¿Cuáles son las cuotas y ventas totales para todos los vendedores?

```
SELEC SUM(CUOTA), SUM(VENTAS)
FROM REPVENTAS
      SUM(CUOTA)    SUM(VENTAS)
-----  -----
$2,700,000.00   $2,893,352.00
```

```

SELECT COUNT(NUM_PEDIDO)
FROM PEDIDOS
WHERE IMPORTE > 25000.00
COUNT (NUM_PEDIDO)
-----
```

4

De hecho, es extraño pensar en la consulta como «contar cuántos importes de pedido» o «contar cuántos números de pedido», es mucho más fácil imaginar que se pide «contar cuántos pedidos». Por esta razón, SQL soporta una función de columna especial COUNT(*) que cuenta filas en lugar de valores de datos. He aquí la misma consulta, reescrita una vez más para utilizar la función COUNT(*):

```

SELECT COUNT (*)
FROM PEDIDOS
WHERE IMPORTE > 25000.00
COUNT (*)
```

4

Si usted piensa en la función COUNT(*) como en una función «cuenta filas», la consulta resulta más fácil de leer. En la práctica, se utiliza casi siempre la función COUNT(*) en lugar de la función COUNT() para contar filas.

Funciones de columna en la lista de selección

Las consultas simples con una función de columna en la lista de selección son bastante fáciles de entender. Sin embargo, cuando la lista de selección incluye varias funciones de columna, o cuando el argumento para una función de columna es una expresión compleja, la consulta puede ser más fácil de leer y entender. La Figura 8.3 muestra las reglas para el procesamiento de consultas SQL a partir de la Figura 7.12, ampliada una vez más para describir cómo se gestionan las funciones de columna. Como antes, las reglas se pretende que proporcionen una definición precisa de lo que significa la consulta, y no una descripción de cómo el DBMS se encarga realmente de producir los resultados de la consulta.

Uno de los mejores modos de imaginar las consultas sumarias y las funciones de columna es pensar en el procesamiento de la consulta dividido en dos pasos. Primero, imaginamos cómo funcionaría la consulta *sin* las funciones de columna, produciendo muchas filas de resultados de consulta detallados. Luego imaginamos a SQL aplicando las funciones de columna a los resultados de consulta detallados, produciendo una sola fila sumaria. Por ejemplo, consideremos la siguiente consulta compleja:

Halla el importe medio de pedidos, el importe total de pedidos, el importe medio de pedido como porcentaje del límite de crédito del cliente y el importe medio de pedido como porcentaje de la cuota de vendedor.

```

SELECT AVG(IMPORTE), SUM(IMPORTE), (100 * AVG(IMPORTE/LIMITE_CREDITO)),
       (100 * AVG(IMPORTE/QUOTA))
  FROM PEDIDOS, CLIENTES, REPVENTAS
 WHERE CLIE = NUM_CLIE
   AND REP = NUM_EMPL
```

2.51

24.45

\$8,256.37

\$247,691.00

(100 * AVG(IMPORTE/LIMITE_CREDITO))

(100 * AVG(IMPORTE/QUOTA))

24.45

Sin las funciones de columna aparecería de este modo:

```

SELECT IMPORTE, IMPORTE, IMPORTE/LIMITE_CREDITO,
       IMPORTE/QUOTA
  FROM PEDIDOS, CLIENTES, REPVENTAS
 WHERE CLIE = NUM_CLIE AND
   AND REP = NUM_EMPL
```

Para generar los resultados de consulta de una sentencia SELECT:

1. Si la sentencia es una UNION de sentencias SELECT, aplicar los Pasos 2 hasta el 5 a cada una de las sentencias para generar sus resultados individuales.
2. Formar el producto de las tablas indicadas en la cláusula FROM. Si la cláusula FROM designa una sola tabla, el producto es esa tabla.
3. Si hay una cláusula WHERE, aplicar su condición de búsqueda a cada fila de la tabla producto, reteniendo aquellas filas para las cuales la condición de búsqueda es TRUE (y descartando aquéllas para las cuales es FALSE o NULL).
4. Para cada fila restante, calcular el valor de cada elemento en la lista de selección para producir una única fila de resultados. Para una referencia de columna simple, utilizar el valor de la columna en la fila (o grupo de filas) actual. Para una función de columna, utilizar como argumento el conjunto entero de filas.
5. Si se especifica SELECT DISTINCT, eliminar las filas duplicadas de los resultados que se hubieran producido.
6. Si la sentencia es una UNION de sentencias SELECT, mezclar los resultados de consulta para las sentencias individuales en una única tabla de resultados. Eliminar las filas duplicadas a menos que se haya especificado UNION ALL.
7. Si hay una cláusula ORDER BY, ordenar los resultados de la consulta según se haya especificado.

Las filas generadas por este procedimiento forman los resultados de la consulta.

Figura 8.3. Reglas de procesamiento de consultas SQL (con funciones de columna).

y produciría una fila de resultados de columna detallados para cada pedido. Las funciones de columna utilizan las columnas de esta tabla de resultados detallados para generar una tabla de una sola fila con los resultados sumarios.

Una función de columna puede aparecer en la lista de selección en cualquier lugar en el que puede aparecer un nombre de columna. Puede, por ejemplo, formar parte de una expresión que suma o resta los valores de dos funciones de columna. Sin embargo, el argumento de una función de columna no puede contener a otra función de columna, ya que la expresión resultante no tendría sentido. Esta regla se resume a veces así: «es ilegal anidar funciones de columna».

También es ilegal mezclar funciones de columna y nombres de columna ordinarios en una lista de selección, de nuevo porque la consulta resultante no tendría sentido. Por ejemplo, consideremos esta consulta:

```
SELECT NOMBRE, SUM(VENTAS)
  FROM SALESREPS;
```

El primer ítem de selección pide a SQL que genere una tabla de diez filas con resultados —una fila por cada vendedor—. El segundo elemento de selección pide a SQL que genere una columna de una fila con resultados sumarios que contengan el total de la columna de VENTAS. Los dos elementos SELECT se contradicen el uno al otro, produciendo un error. Por esta razón, o bien todas las referencias de columna en la lista de selección deben aparecer dentro del argumento de una función de columna (produciendo una consulta sumaria), o bien la lista de selección no debe contener ninguna función de columna (produciendo una consulta detallada). Realmente, la regla es ligeramente más compleja cuando se consideran consultas agrupadas y subconsultas. Los refinamientos necesarios se describirán más tarde en este capítulo.

Valores NULL y funciones de columna

Las funciones de columna SUM(), AVG(), MIN() y COUNT() aceptan cada una de ellas una columna de valores de datos como argumento y producen un único valor como resultado. ¿Qué sucede si uno o más de los valores de la columna es un valor NULL? El estándar SQL ANSI/ISO especifica que los valores NULL de la columna sean *ignorados* por las funciones de columna.

Esta consulta muestra cómo la función de columna COUNT() ignora los valores NULL de una columna:

```
SELECT COUNT(*), COUNT(VENTAS), COUNT(CUOTA)
  FROM REPVENTAS;
```

COUNT(*)	COUNT(VENTAS)	COUNT(CUOTA)
10	10	9

La tabla REPVENTAS contiene diez filas, por lo que COUNT(*) devuelve una cuenta de diez. La columna VENTAS contiene diez valores no NULL, por lo que la función COUNT(VENTAS) también devuelve una cuenta de diez. La columna CUOTA es NULL para el vendedor más reciente. La función COUNT(CUOTA) ignora este valor NULL y devuelve una cuenta de nueve. Debido a estas anomalías, la función COUNT(*) es utilizada casi siempre en lugar de la función COUNT(), a menos que específicamente se desee excluir del total los valores NULL de una columna particular. La ignorancia de los valores NULL tiene poco impacto en las funciones de columna MIN() y MAX(). Sin embargo, también puede producir sutiles problemas para las funciones de columna SUM() y AVG(), como se ilustra en esta consulta:

```
SELECT SUM(VENTAS), SUM(CUOTA),
       (SUM(VENTAS) - SUM(CUOTA)), SUM(VENTAS-CUOTA)
  FROM REPVENTAS;
```

SUM(VENTAS)	SUM(CUOTA)	(SUM(VENTAS)-SUM(CUOTA))	SUM(VENTAS-CUOTA)
\$2,893,532.00	\$2,700,000.00	\$193,532.00	\$117,547.00

Sería de esperar que las dos expresiones:

```
(SUM(VENTAS) - SUM(CUOTA)) y SUM(VENTAS-CUOTA)
```

en la lista de selección produjeran resultados idénticos, pero el ejemplo muestra que no es así. El vendedor con un valor NULL en la columna CUOTA es de nuevo la razón. La expresión:

```
SUM(VENTAS)
```

totaliza las ventas para todos los diez vendedores, mientras que la expresión:

```
SUM(CUOTA)
```

totaliza solamente los nueve valores de cuota no NULL. La expresión:

```
SUM(VENTAS) - SUM(CUOTA)
```

calcula la diferencia de estos dos importes. Sin embargo, la función de columna:

```
SUM(VENTAS-CUOTA)
```

tiene un valor de argumento no NULL para sólo nueve de los diez vendedores. En la fila con un valor de cuota NULL, la resta produce un NULL, que es ignorado por la función SUM(). Por tanto, las ventas del vendedor sin cuota, que están incluidas en el cálculo previo, se excluyen de este cálculo.

¿Cuál es la respuesta correcta? Ambas lo son. La primera expresión calcula

exactamente lo que dice: «la suma de VENTAS, menos la suma de CUOTA». La expresión segunda también calcula exactamente lo que dice: «la suma de (VENTAS - CUOTA)». Cuando se producen los valores NULL, sin embargo, los cálculos no son exactamente iguales.

El estándar ANSI/ISO especifica estas reglas precisas para manejar los valores NULL en funciones de columna:

- Si alguno de los valores de datos de la columna es NULL, se ignora para el propósito de calcular el valor de la función de columna.
 - Si todos los datos de una columna son NULL, las funciones de columna SUM(), AVG(), MIN() y MAX() devuelven un valor NULL; la función COUNT() devuelve un valor de cero.
 - Si no hay datos en la columna (es decir, la columna está vacía), las funciones de columna SUM(), AVG(), MIN() y MAX() devuelven un valor de cero.
 - La función COUNT(*) cuenta filas, y no depende de la presencia o ausencia de valores NULL en la columna. Si no hay filas, devuelve un valor de cero.
- Aunque el estándar es muy claro en este punto, los productos SQL comerciales pueden producir resultados diferentes al especificado en el estándar, especialmente si todos los valores de datos de una columna son NULL o cuando la función de columna se aplica a una tabla vacía. Antes de presumir el comportamiento especificado por el estándar, debería comprobarse cada DBMS particular.

Eliminación de filas duplicadas (DISTINCT)

Recuerde del Capítulo 6 que se puede especificar la palabra clave DISTINCT al comienzo de la lista de selección para eliminar las filas duplicadas de los resultados de la consulta. También se puede pedir a SQL que elimine valores duplicados de una columna antes de aplicarle una función de columna. Para eliminar valores duplicados, la palabra clave DISTINCT se incluye delante del argumento de la función de columna, inmediatamente después del paréntesis abierto.

He aquí dos consultas que ilustran la eliminación de filas duplicadas para funciones de columna:

¿Cuántos títulos diferentes tienen los vendedores?

```
SELECT COUNT(DISTINCT TITULO)
FROM REVENTAS
COUNT(DISTINCT TITULO)
```

3

¿Cuántas oficinas de ventas tienen vendedores que superan a sus cuotas?

```
SELECT COUNT(DISTINCT OFICINA.REP)
FROM REVENTAS
WHERE VENTAS > CUOTA
COUNT(DISTINCT OFICINA.REP)
```

4

El estándar ANSI/ISO especifica que cuando se utiliza la palabra clave DISTINCT, el argumento de la función columna debe ser un nombre de columna único; no puede ser una expresión. El estándar permite la palabra clave DISTINCT para las funciones de columna SUM() y AVG(). El estándar no permite el uso de la palabra clave DISTINCT con las funciones de columna MIN() y MAX(), ya que no tiene impacto sobre sus resultados, pero muchas implementaciones SQL lo permiten igualmente. El estándar también *requiere* la palabra clave DISTINCT para la función de columna COUNT(), pero muchas implementaciones SQL permiten el uso de la función COUNT() sin ella. DISTINCT no puede ser especificado con la función COUNT(*), ya que ésta no trata con una columna de valores de datos en absoluto —simplemente cuenta filas.

Además, la palabra clave DISTINCT sólo puede ser especificada una vez en una consulta. Si aparece en el argumento de una función de columna, no puede aparecer en ninguna otra. Si se especifica delante de la lista de selección, no puede aparecer en ninguna función de columna. La única excepción es que DISTINCT puede ser especificada una segunda vez dentro de una subconsulta (contenida dentro de la consulta). Las subconsultas se describen en el Capítulo 9.

Consultas agrupadas (cláusula GROUP BY)

Las consultas sumarias descriptas hasta ahora son como los totales al final de un informe. Condensan todos los datos detallados del informe en una única fila sumaria de datos. Al igual que los subtotales son útiles en informes impresos, con frecuencia es conveniente sumarizar los resultados de la consulta a un nivel «subtotal». La cláusula GROUP BY de la sentencia SELECT proporciona esta capacidad.

La función de la cláusula GROUP BY se entiende más fácilmente mediante ejemplos. Consideremos estas dos consultas:

¿Cuál es el tamaño medio de pedido?

```
SELECT AVG(IMPORTE)
FROM PEDIDOS
AVG(IMPORTE)
```

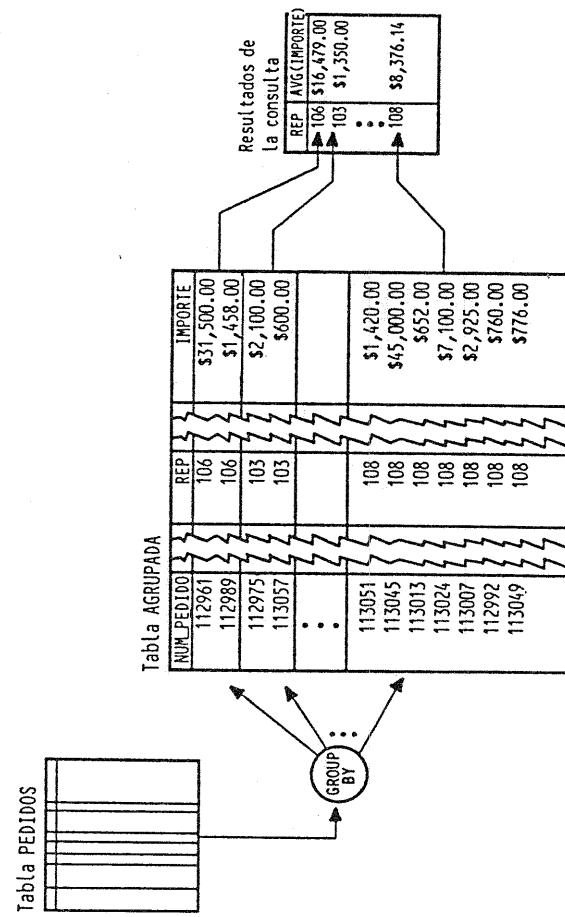
\$8,256.37

5

¿Cuál es el tamaño medio de pedido para cada vendedor?

```
SELECT REP, AVG(IMPORTE)
FROM PEDIDOS
GROUP BY REP
REP  AVG(IMPORTE)
--- -----
101   $8,876.00
102   $5,694.00
103   $1,350.00
105   $7,865.40
106   $16,479.00
107   $11,477.33
108   $8,376.14
109   $3,552.50
110   $11,566.00
```

La primera consulta es una consulta sumaria simple como la de los ejemplos anteriores de este capítulo. La segunda consulta produce varias filas sumarias —una fila por cada grupo, sumarizando los pedidos aceptados por un solo vendedor—. La Figura 8.4 muestra cómo funciona la segunda consulta. Conceptualmente, SQL lleva a cabo la consulta del modo siguiente:



- SQL divide los pedidos en grupos de pedidos, un grupo por cada vendedor. Dentro de cada grupo, todos los pedidos tienen el mismo valor en la columna REP.
- Por cada grupo, SQL calcula el valor medio de la columna IMPORTE para todas las filas del grupo, y genera una única fila sumario de resultados. La fila contiene el valor de la columna REP del grupo y el tamaño de pedido medio calculado.

Una consulta que incluya la cláusula GROUP BY se denomina *consulta agrupada*, ya que agrupa los datos de las tablas fuente y produce una única fila sumaria por cada grupo de filas. Las columnas indicadas en la cláusula GROUP BY se denominan *columnas de agrupación* de la consulta, ya que ellas son las que determinan cómo se dividen las filas en grupo. He aquí algunos ejemplos adicionales de consultas agrupadas:

¿Cuál es el rango de cuotas asignadas en cada oficina?

```
SELECT OFICINA.REP, MIN(CUOTA), MAX(CUOTA)
FROM REPVENTAS
GROUP BY OFICINA.REP
```

¿Cuántos vendedores están asignados a cada oficina?

```
SELECT OFICINA.REP, COUNT(*)
FROM REPVENTAS
GROUP BY OFICINA.REP
OFICINA.REP  COUNT(*)
----- -----
NULL        1
11          2
12          3
13          1
21          2
22          1
```

¿Cuántos clientes diferentes son atendidos por cada vendedor?

```
SELECT COUNT(DISTINCT NUM_CLIE), 'clientes por rep. de ventas', REP_CLIE
FROM CLIENTES
GROUP BY REP_CLIE
```

Figura 8.4. Una consulta agrupada en operación.

COUNT(DISTINCT NUM_CLIE)	CLIENTES POR REP. DE VENTAS	REP_CLIE
3	clientes por rep. de ventas	101
4	clientes por rep. de ventas	102
3	clientes por rep. de ventas	103
1	clientes por rep. de ventas	104
2	clientes por rep. de ventas	105
2	clientes por rep. de ventas	106

Hay una conexión íntima entre las funciones de columna SQL y la cláusula GROUP BY. Recuerde que las funciones de columna toman una columna de valores de datos y producen un resultado único. Cuando la cláusula GROUP BY está presente, informa a SQL que debe dividir los resultados detallados en grupos y aplicar la función de columna separadamente a cada grupo, produciendo un único resultado por cada grupo. La Figura 8.5 muestra las reglas para procesamiento de consultas SQL, ampliadas una vez más para incluir las consultas agrupadas.

Múltiples columnas de agrupación

SQL puede agrupar resultados de consulta en base a contenidos de dos o más columnas. Por ejemplo, supongamos que se desea agrupar los pedidos por vendedor y por cliente. Esta consulta agrupa los datos basándose en ambos criterios:

Calcula los pedidos totales por cada cliente y por cada vendedor.

```
SELECT REP, CUST, SUM(IMPORTE)
  FROM PEDIDOS
 GROUP BY REP, CUST
      REP   CLIE  SUM(IMPORTE)
      ---  ----  -----
      101  2102  $3,978.00
      101  2108  $150.00
      101  2113  $22,500.00
      102  2106  $4,026.00
      102  2114  $15,000.00
      102  2120  $3,750.00
      103  2111  $2,750.00
      105  2103  $35,582.00
      105  2111  $3,745.00
```

Incluso con múltiples columnas de agrupación, SQL sólo proporciona un único nivel de agrupación. La consulta produce una fila sumaria separada por

Para generar los resultados de consulta de una sentencia SELECT:

1. Si la sentencia es una UNION de sentencias SELECT, aplicar los Pasos 2 hasta el 6 a cada una de las sentencias para generar sus resultados individuales.
2. Formar el producto de las tablas indicadas en la cláusula FROM. Si la cláusula FROM designa una sola tabla, el producto es esa tabla.
3. Si hay una cláusula WHERE, aplicar su condición de búsqueda a cada fila de la tabla producto, reteniendo aquellas filas para las cuales la condición de búsqueda es TRUE (y descartando aquellas para las cuales es FALSE o NULL).
4. Si hay una cláusula GROUP BY, disponer las filas restantes de la tabla producto en grupos de filas, de modo que las filas de cada grupo tengan valores idénticos en todas las columnas de agrupación.
5. Para cada fila (o grupo de filas) restante, calcular el valor de cada elemento en la lista de selección para producir una única fila de resultados. Para una referencia de columna simple, utilizar el valor de la columna en la fila (o grupo de filas) actual. Para una función de columna, utilizar como argumento el grupo de filas actual si se especificó GROUP BY; en caso contrario, utilizar el conjunto entero de filas.
6. Si se especifica SELECT DISTINCT, eliminar las filas duplicadas de los resultados que se hubieran producido.
7. Si la sentencia es una UNION de sentencias SELECT, mezclar los resultados de consulta para las sentencias individuales en una única tabla de resultados. Eliminar las filas duplicadas a menos que se haya especificado UNION ALL.
8. Si hay una cláusula ORDER BY, ordenar los resultados de la consulta según se haya especificado.

Las filas generadas por este procedimiento forman los resultados de la consulta.

Figura 8.5. Reglas de procesamiento de consultas SQL (con GROUP BY).

cada pareja vendedor/cliente. Es imposible crear grupos y subgrupos con dos niveles de subtotales en SQL. Lo mejor que se puede hacer es ordenar los datos de modo que las filas de resultados aparezcan en el orden adecuado. En muchas implementaciones SQL, la cláusula GROUP BY tendrá automáticamente el efecto lateral de ordenar los datos, pero usted puede obviar esta ordenación con una cláusula ORDER BY, como se muestra aquí:

```
SELECT CLIE, REP, SUM(IMPORTE)
  FROM PEDIDOS
 GROUP BY CLIE, REP
 ORDER BY CLIE, REP
```

Calcula los pedidos totales para cada cliente de cada vendedor, ordenados por cliente y dentro de cada cliente por vendedor.

CLIE	REP	SUM(IMPORTE)	sum
2101	106	\$1,458.00	\$26,628.00
2102	101	\$3,978.00	
2103	105	\$35,582.00	
2106	102	\$4,026.00	
2107	110	\$23,132.00	
2108	101	\$150.00	
2108	109	\$7,105.00	
2109	107	\$31,350.00	
2111	103	\$2,700.00	
2111	105	\$3,745.00	
:			

Observe que también es imposible obtener resultados a la vez detallados y sumarios en una consulta simple. Para obtener resultados detallados con subtotales, o para obtener subtotales multivivel, es necesario escribir un programa de aplicación utilizando SQL programado y calcular los subtotales dentro de la lógica del programa. SQL Server superó esta limitación del SQL estándar añadiendo una cláusula COMPUTE al final de la sentencia SELECT. La cláusula COMPUTE calcula subtotales y sub-subtotales como se muestra en este ejemplo:

Calcula los pedidos totales para cada cliente de cada vendedor, ordenados por vendedor, y dentro de cada vendedor por cliente.

REP	CLIE	IMPORTE	sum
102	2114	\$15,000.00	\$4,026.00
102	2106	\$2,130.00	
102	2106	\$1,896.00	
102	2120	\$3,750.00	
102	2102	\$3,978.00	
101	2102	\$3,978.00	\$3,978.00
REP	CLIE	IMPORTE	sum
101	2108	\$150.00	\$22,776.00
101	2113	\$22,500.00	\$5,694.00
:			

La consulta produce una fila de resultados detallados por cada fila de la tabla PEDIDOS, ordenados por CLIE dentro de REP. Además, calcula la suma de los pedidos por cada pareja cliente/vendedor (un subtotal de bajo nivel) y calcula la suma de los pedidos y el tamaño medio de pedido por cada vendedor (un subtotal de alto nivel). Los resultados de la consulta tienen por tanto una mezcla de filas de detalle y de filas de sumario, que incluyen tanto subtotales como subtotales.

La cláusula COMPUTE no es estándar en absoluto, y de hecho es propia del dialecto Transact-SQL utilizado por SQL Server. Además, viola los principios básicos de las consultas relacionales, ya que los resultados de la sentencia SELECT no son una tabla, sino una extraña combinación de diferentes tipos de filas. No obstante, como muestra el ejemplo, puede ser muy útil.

Restricciones en consultas agrupadas

Las consultas agrupadas están sujetas a algunas limitaciones bastante estrictas. Las columnas de agrupación deben ser columnas efectivas de las tablas designadas en la cláusula FROM de la consulta. No se pueden agrupar las filas basándose en el valor de una expresión calculada.

También hay restricciones sobre los elementos que pueden aparecer en la lista de selección de una consulta agrupada. Todos los elementos de la lista de selección deben tener un único valor por cada grupo. Básicamente, esto significa que un elemento de selección en una consulta agrupada puede ser:

- una constante,
- una función de columna, que produce un único valor summarizando las filas del grupo,
- una columna de agrupación, que por definición tiene el mismo valor en todas las filas del grupo, o
- una expresión que afecte a combinaciones de los anteriores.

En la práctica, una consulta agrupada incluirá siempre una columna de agrupación y una función de columna en su lista de selección. Si no aparece una función de columna, la consulta puede expresarse más sencillamente utilizando SELECT DISTINCT, sin GROUP BY. Y al contrario, si no se incluye una columna de agrupación en los resultados de la consulta, ¡no será posible decir qué fila de los resultados de la consulta proviene de qué grupo!

Otra limitación de las consultas agrupadas es que SQL ignora información referente a claves primarias y claves foráneas cuando analiza la validez de una consulta agrupada. Consideremos esta consulta:

```
Calcula los pedidos totales para cada vendedor.
SELECT NUM_EMPL, NOMBRE, SUM (IMPORTE)
FROM PEDIDOS, REPVENTAS
WHERE REP = NUM_EMPL
GROUP BY NUM_EMPL
```

Error: "NOMBRE" no es una expresión GROUP BY

Dada la naturaleza de los datos, la consulta tiene perfecto sentido, ya que la agrupación sobre el número de empleado del vendedor es en efecto igual que la agrupación sobre el nombre del vendedor. Más precisamente, NUM_EMPL, la columna de agrupación, es la clave primaria de la tabla REPVENTAS, por lo que la columna NOMBRE debe tener un valor único por cada grupo. No obstante, SQL informa de un error, ya que la columna NOMBRE no está explícitamente especifica-

da como columna de agrupación. Para corregir el problema, simplemente se incluye la columna NOMBRE como segunda (redundante) columna de agrupación:

Calcula los pedidos totales por cada vendedor.

```
SELECT NUM_EMPL, NOMBRE, SUM(IMPORTE)
FROM PEDIDOS, REPVENTAS
```

WHERE REP = NUM_EMPL

GROUP BY NUM_EMPL, NOMBRE

NUM_EMPL	NOMBRE	SUM(IMPORTE)
101	Dan Roberts	\$26,628.00
102	Sue Smith	\$22,776.00
103	Paul Cruz	\$2,700.00
105	Bill Adams	\$39,327.00
106	Sam Clark	\$32,958.00
107	Nancy Angelotti	\$34,432.00
108	Larry Fitch	\$58,633.00
109	Mary Jones	\$7,105.00
110	Tom Snyder	\$23,132.00

Naturalmente, si el número de empleado del vendedor no se necesita en los resultados de la consulta, se puede eliminar completamente de la lista de selección, obteniendo entonces:

Calcula los pedidos totales por cada vendedor.

```
SELECT NOMBRE, SUM(IMPORTE)
FROM PEDIDOS, REPVENTAS
```

WHERE REP = NUM_EMPL

GROUP BY NAME

NOMBRE	SUM(IMPORTE)
Bill Adams	\$39,327.00
Dan Roberts	\$26,628.00
Larry Fitch	\$2,700.00
Mary Jones	\$7,105.00
Nancy Angelotti	\$34,432.00
Paul Cruz	\$32,958.00
Sam Clark	\$22,776.00
Sue Smith	\$23,132.00

Valores NULL en columnas de agrupación

Un valor NULL presenta un problema especial cuando aparece en una columna de agrupación. Si el valor de la columna es desconocido, (en qué grupo debería

Aunque este comportamiento de NULL en las columnas de agrupación está claramente especificado en el estándar ANSI/ISO, no está implementado en todos los dialectos SQL. Es buena idea construir una pequeña tabla de test y comprobar el comportamiento de cada producto DBMS particular antes de confiar en un comportamiento específico.

NOMBRE	PELO	OJOS
Cindy	Castaño	Azules
Louise	NULL	Azules
Harry	NULL	Azules
Samantha	NULL	NULL
Joanne	NULL	NULL
George	Castaño	NULL
Mary	Castaño	NULL
Paula	Castaño	NULL
Kevin	Castaño	Negros
Joel	Rubio	Azules
Susan	Rubio	Azules
Marie	Rubio	Azules

Figura 8.6. La tabla PERSONAS.

colocarse la fila? En la cláusula WHERE, cuando se comparan dos valores NULL diferentes, el resultado es NULL (no TRUE), es decir, los dos valores NULL no se consideran iguales. Aplicando el mismo convenio a la cláusula GROUP BY se forzaría a SQL a colocar cada fila con una columna de agrupación NULL en un grupo aparte.

En la práctica, esta regla se demuestra que es demasiado onerosa. En vez de ello, el estándar SQL ANSI/ISO considera que dos valores NULL son iguales a efectos de la cláusula GROUP BY. Si dos filas tienen NULL en las mismas columnas de agrupación y valores idénticos en las columnas de agrupación no NULL, se agrupan dentro del mismo grupo de filas. La pequeña tabla ejemplo mostrada en la Figura 8.6 ilustra el manejo ANSI/ISO de los valores NULL con la cláusula GROUP BY, como se muestra en esta consulta:

```
SELECT PELO, OJOS, COUNT(*)
  FROM PERSONAS
 GROUP BY PELO, OJOS
      PELO      OJOS      COUNT(*)
```

Castaño	Azules	1
NULL	Azules	2
NULL	NULL	2
Castaño	NULL	3
Castaño	Negros	2
Castaño	Negros	2

La Figura 8.7 muestra gráficamente cómo efectúa SQL la consulta. La cláusula GROUP BY dispone primero de los pedidos en grupos por vendedor. La cláusula HAVING elimina entonces los grupos en donde el total de los pedidos no excede a \$30,000. Finalmente, la cláusula SELECT calcula el tamaño de pedido medio para cada uno de los grupos restantes y genera los resultados de la consulta.

Las condiciones de búsqueda que se pueden especificar en la cláusula HAVING son las mismas utilizadas en la cláusula WHERE, que se describió en el Capítulo 6 y se describirá en el Capítulo 9. He aquí otro ejemplo del uso de la condición de búsqueda de grupos:

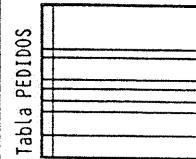


Tabla PEDIDOS

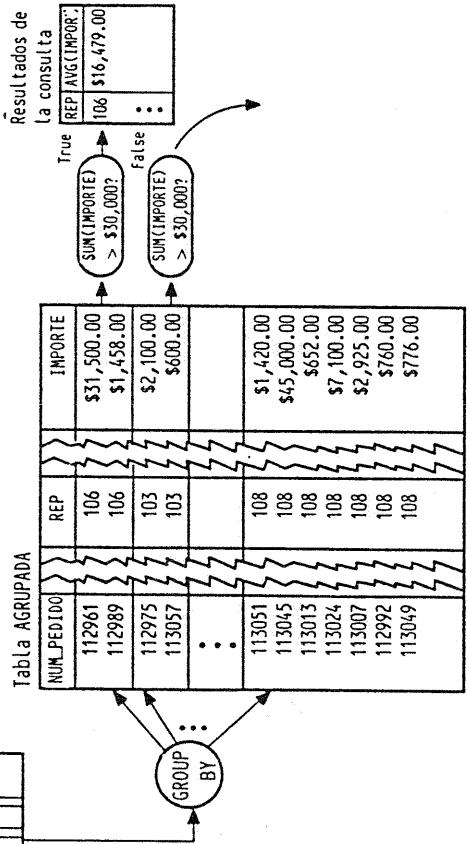


Figura 8.7. Una condición de búsqueda de grupos en operación.

Por cada oficina con dos o más personas, calcular la cuota total y las ventas totales para todos los vendedores que trabajan en la oficina.

```
SELECT CIUDAD, SUM(CUOTA), SUM(REPVENTAS.VENTAS)
FROM OFICINAS, REPVENTAS
WHERE OFICINA = OFICINA.REP
GROUP BY CIUDAD
HAVING COUNT(*) >= 2
```

CIUDAD	SUM(CUOTA)	SUM(REPVENTAS.VENTAS)
Chicago	\$775,000.00	\$735,042.00
Los Angeles	\$700,000.00	\$835,915.00
New York	\$575,000.00	\$692,637.00

La Figura 8.8 muestra las reglas para el procesamiento de consultas SQL, ampliadas una vez más para que se incluyan las condiciones de búsqueda de grupos. Siguiendo los pasos de la figura, SQL maneja esta consulta del modo siguiente:

1. Componer las tablas OFICINAS y REPVENTAS para hallar la ciudad en donde trabaja cada vendedor.

Para generar los resultados de consulta de una sentencia SELECT:

1. Si la sentencia es una UNION de sentencias SELECT, aplicar los Pasos 2 hasta el 7 a cada una de las sentencias para generar sus resultados de consulta individuales.
2. Formar el producto de las tablas indicadas en la cláusula FROM. Si la cláusula FROM designa una única tabla, el producto es esa tabla.
3. Si hay una cláusula WHERE, aplicar su condición de búsqueda a cada fila de la tabla producto, reteniendo aquellas filas para las cuales la condición de búsqueda es TRUE (y descartando aquéllas para las cuales es FALSE o NULL).
4. Si hay una cláusula GROUP BY, disponer las filas restantes de la tabla producto en grupos de filas, de modo que las filas de cada grupo tengan valores idénticos en todas las columnas de agrupación.
5. Si hay una cláusula HAVING, aplicar su condición de búsqueda a cada grupo de filas, reteniendo aquellos grupos para los cuales la condición de búsqueda es TRUE (y descartando aquéllos para los cuales es FALSE o NULL).
6. Para cada fila (o grupo de filas) restante, calcular el valor de cada elemento en la lista de selección para producir una única fila de resultados. Para una referencia de columna simple, utilizar el valor de la columna en la fila (o grupo de filas) actual. Para una función de columna, utilizar como argumento el grupo de filas actual si se especificó GROUP BY; en caso contrario, utilizar el conjunto entero de filas.
7. Si se especifica SELECT DISTINCT, eliminar las filas duplicadas de los resultados que se hubieran producido.

8. Si la sentencia es una UNION de sentencias SELECT, mezclar los resultados de consulta para las sentencias individuales en una única tabla de resultados. Eliminar las filas duplicadas a menos que se haya especificado UNION ALL.
9. Si hay una cláusula ORDER BY, ordenar los resultados de la consulta según se haya especificado.

Las filas generadas por este procedimiento forman los resultados de la consulta.

Figura 8.8. Reglas de procesamiento de consultas SQL (con HAVING).

2. Agrupa las filas resultantes por oficina.
3. Elimina los grupos con dos o menos filas —estas representan oficinas que no satisfacen el criterio de la cláusula HAVING.
4. Calcula la cuota total y las ventas totales para cada grupo.

He aquí un ejemplo más, que utiliza todas las cláusulas de la sentencia SELECT:

Muestra el precio, la existencia y la cantidad total de los pedidos de cada producto para los cuales la cantidad total pedida es superior al 75 %.

```
SELECT DESCRIPCION, PRECIO, EXISTENCIAS, SUM(CANT)
FROM PRODUCTOS, PEDIDOS
WHERE FAB = ID_FAB
AND PRODUCTO = ID_PRODUCTO
GROUP BY ID_FAB, ID_PRODUCTO, DESCRIPCION, PRECIO, EXISTENCIAS
HAVING SUM(CANT) > (.75 * EXISTENCIAS)
ORDER BY EXISTENCIAS DESC
```

DESCRIPCION	PRECIO	EXISTENCIAS	SUM(CANT)
Reductor	\$355.00	38	32
Ajustador	\$25.00	37	30
Bancada motor	\$243.00	15	16
Bisagra dcha.	\$4,500.00	12	15
Riostra 1-1m	\$1,425.00	5	22

Para procesar esta consulta, SQL efectúa conceptualmente los siguientes pasos:

- Si la sentencia es una UNION de sentencias SELECT, aplicar los Pasos 2 hasta el 7 a cada una de las sentencias para generar sus resultados de consulta individuales.
- Agrupa las filas resultantes por fabricante e id de producto.
- Elimina los grupos en donde la cantidad pedida (el total de la columna CANT para todos los pedidos del grupo) es más de la mitad de las existencias.
- Calcula la cantidad total pedida para cada grupo.
- Genera una fila sumaria de resultados por cada grupo.
- Ordena los resultados para que los productos con el mayor valor de existencias aparezcan en primer lugar.

Como se describió anteriormente, DESCRIPCION, PRECIO y EXISTENCIAS deben ser especificadas como columnas de agrupación en esta consulta únicamente porque aparecen en la lista de selección. Realmente no contribuyen en nada al proceso de agrupación, ya que ID_FAB e ID_PRODUCTO especifican completamente una única fila en la tabla PRODUCTOS, haciendo automáticamente que las otras tres columnas tengan un único valor por grupo.

Restricciones en condiciones de búsqueda de grupos

La cláusula HAVING se utiliza para incluir o excluir grupos de filas de los resultados de la consulta, por lo que la condición de búsqueda que especifica debe ser

aplicable al grupo en su totalidad en lugar de a filas individuales. Esto significa que un elemento que aparezca dentro de la condición de búsqueda en una cláusula HAVING puede ser:

- una constante,
- una función de columna, que produzca un único valor que sumarice las filas del grupo,
- una columna de agrupación, que por definición tiene el mismo valor para todas las filas del grupo,
- una expresión que afecte a combinaciones de las anteriores.

En la práctica, la condición de búsqueda de la cláusula HAVING incluirá siempre al menos una función de columna. Si no lo hiciera, la condición de búsqueda podría expresarse con la cláusula WHERE y aplicarse a filas individuales. El modo más fácil de imaginar si una condición de búsqueda pertenece a la cláusula WHERE o la cláusula HAVING es recordar cómo se aplican ambas cláusulas:

- La cláusula WHERE se aplica a *filas individuales*, por lo que las expresiones que contiene debe ser calculables para filas individuales.
- La cláusula HAVING se aplica a *grupos de filas*, por lo que las expresiones que contengan deben ser calculables para un grupo de filas.

Valores NULL y condiciones de búsqueda de grupos

Al igual que la condición de búsqueda de la cláusula WHERE, la condición de búsqueda de la cláusula HAVING puede producir uno de los tres resultados:

- Si la condición de búsqueda es TRUE, se retiene el grupo de filas, y contribuye con una fila sumaria a los resultados de la consulta.
- Si la condición de búsqueda es FALSE, el grupo de filas se descarta, y no contribuye con una fila sumaria a los resultados de la consulta.
- Si la condición de búsqueda es NULL, el grupo de filas se descarta, y no contribuye con una fila sumaria a los resultados de la consulta.

Las anomalías que pueden producirse con valores NULL en la condición de búsqueda son las mismas que para la cláusula WHERE, y han sido descriptas en el Capítulo 6.

HAVING sin GROUP BY

La cláusula HAVING se utiliza casi siempre juntamente con la cláusula GROUP BY, pero la sintaxis de la sentencia SELECT no lo precisa. Si una cláusula HAVING aparece sin una cláusula GROUP BY, SQL considera el conjunto entero de resultados detallados como un único grupo. En otras palabras, las funciones de columna de la cláusula HAVING se aplican a un solo y único grupo para determinar si el grupo está incluido o excluido de los resultados, y ese grupo está formado por todas las filas. El uso de una cláusula HAVING sin una cláusula correspondiente GROUP BY casi nunca se ve en la práctica.

Resumen

Este capítulo ha descrito las consultas sumarias que summarizan los datos de la base de datos:

- Las consultas sumarias utilizan funciones de columna SQL para condensar una columna de valores en un único valor que sumarice la columna.
- Las funciones de columna pueden calcular el promedio, la suma, el valor mínimo y el valor máximo de una columna, contar el número de valores de datos de una columna o contar el número de filas de los resultados de la consulta.
- Una consulta sumaria sin una cláusula GROUP BY genera una única fila de resultados, summarizando todas las filas de una tabla o de un conjunto compuesto de tablas.
- Una consulta sumaria con una cláusula GROUP BY genera múltiples filas de resultados, cada una summarizando las filas de un grupo particular.
- La cláusula HAVING actúa como cláusula WHERE para grupos, seleccionando los grupos de filas que contribuyen a los resultados de consulta sumarios.

9 Subconsultas

Resumen

Este capítulo ha descrito las consultas sumarias que summarizan los datos de la base de datos:

La característica de subconsulta de SQL permite utilizar los resultados de una consulta como parte de otra. La capacidad de utilizar una consulta dentro de otra fue la razón original para la palabra «estructurada» en el nombre Lenguaje de Consultas Estructuradas (Structured Query Language-SQL). La característica de composición es menos conocida que la característica de composición de SQL, pero juega un papel importante por tres razones:

- Una sentencia SQL con una subconsulta es frecuentemente el modo más natural de expresar una consulta, ya que se asemeja más estrechamente a la descripción de la consulta en lenguaje natural.
- Las subconsultas hacen más fácil la escritura de sentencia SELECT, ya que permiten «descomponer una consulta en partes» (la consulta y sus subconsultas) y luego «recomponer las partes».
- Hay algunas consultas que no pueden ser expresadas en el lenguaje SQL sin utilizar una subconsulta.

Este capítulo describe las subconsultas y cómo se utilizan en las cláusulas WHERE y HAVING de una sentencia SQL.

Utilización de subconsultas

Una *subconsulta* es una consulta que aparece dentro de la cláusula WHERE o HAVING de otra sentencia SQL. Las subconsultas proporcionan un modo eficaz y

natural de manejar las peticiones de consultas que se expresan en términos de los resultados de otras consultas. He aquí un ejemplo de tal tipo de petición.

«Lista las oficinas en donde el objetivo de ventas de la oficina excede a la suma de las cuotas de los vendedores individuales.»

La petición solicita una lista de oficinas de la tabla OFICINAS, en donde el valor de la columna OBJETIVO satisface cierta condición. Parece razonable que la sentencia SELECT que expresa la consulta debería ser semejante a ésta:

```
SELECT CIUDAD
      FROM OFICINAS
     WHERE OBJETIVO > ???
```

El valor «??» necesita ser sustituido, y debería ser igual a «la suma de las cuotas de los vendedores asignados a la oficina de cuestión». ¿Cómo se puede especificar ese valor en la consulta? Del Capítulo 8 sabemos que la suma de las cuotas para una oficina específica (digamos, la oficina número 21) puede ser obtenida con esta consulta:

```
SELECT SUM(CUOTA)
      FROM REPENTAS
     WHERE OFICINA.REP = 21
```

Però ¿cómo se pueden poner los resultados de esta consulta en la consulta primera sustituyendo a los signos de interrogación? Parecería razonable comenzar con la primera consulta y reemplazar los «??» con la segunda consulta, del modo siguiente:

```
SELECT CIUDAD
      FROM OFICINAS
     WHERE OBJETIVO > (SELECT SUM(CUOTA)
                           FROM REPENTAS
                          WHERE OFICINA.REP = OFICINA)
```

De hecho, ésta es una consulta SQL correctamente construida. Por cada oficina, la «consulta interna» (la subconsulta) calcula la suma de las cuotas para los vendedores que trabajan en esa oficina. La «consulta externa» (la *consulta principal*) compara el objetivo de la oficina con el total calculado y decide si añadir la oficina a los resultados de la consulta principal. Trabajando conjuntamente, la consulta principal y la subconsulta expresan la petición original y recuperan los datos solicitados de la base de datos.

Las subconsultas SQL aparecen siempre como parte de la cláusula WHERE o la cláusula HAVING. En la cláusula WHERE, ayudan a seleccionar las filas individuales que aparecen en los resultados de la consulta. En la cláusula HAVING, ayudan a seleccionar los grupos de filas que aparecen en los resultados de la consulta.

¿Qué es una subconsulta?

La Figura 9.1 muestra el formato de una subconsulta SQL. La subconsulta está siempre encerrada entre paréntesis, pero por otra parte tiene el formato familiar de una sentencia SELECT, con una cláusula FROM y cláusulas optionales WHERE, GROUP BY y HAVING. El formato de estas cláusulas en una subconsulta es idéntico al que tienen en una sentencia SELECT, y efectúan sus funciones normales cuando se utilizan dentro de una subconsulta. Sin embargo, hay unas cuantas diferencias entre una subconsulta y una sentencia SELECT real:

- Una subconsulta debe producir una única columna de datos como resultados. Esto significa que una subconsulta siempre tiene un único elemento de selección en su cláusula SELECT.
- La cláusula ORDER BY no puede ser especificada en una subconsulta. Los resultados de la subconsulta se utilizan internamente por parte de la consulta principal y nunca son visibles al usuario, por lo que tiene poco sentido ordenarlos de ningún modo.
- Una subconsulta no puede ser la UNION de varias sentencias SELECT diferentes; sólo se permite una única SELECT.

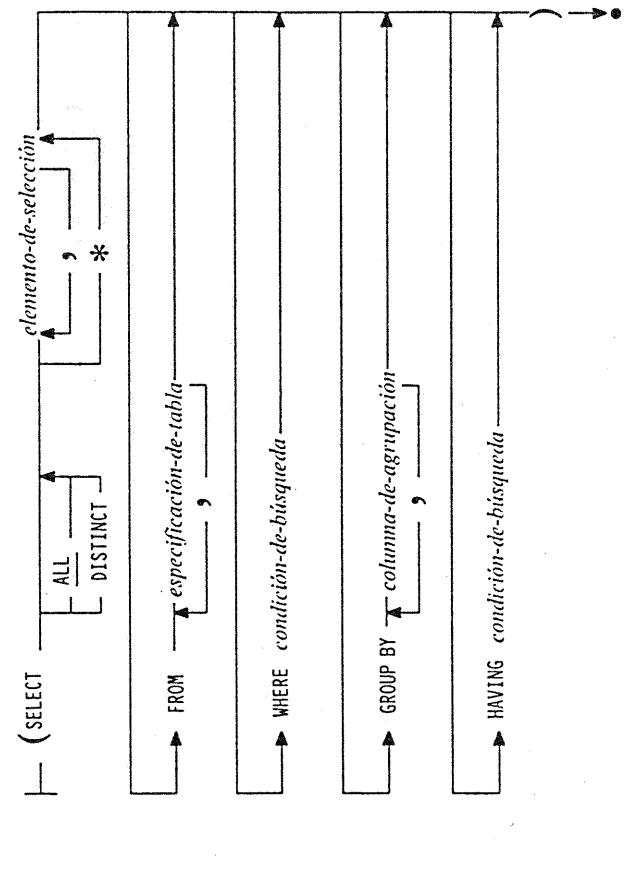


Figura 9.1. Diagrama sintáctico de subconsultas.

■ Los nombres de columna que aparecen en una subconsulta pueden referirse a columnas de tablas en la consulta principal. Estas *referencias externas* se describen con detalle posteriormente en este capítulo.

Subconsultas en la cláusula WHERE

Las subconsultas suelen ser utilizadas principalmente en la cláusula WHERE de una sentencia SQL. Cuando aparece una subconsulta en la cláusula WHERE, ésta funciona como parte del proceso de selección de filas. Consideremos una vez más la consulta de la sección anterior:

Lista las oficinas en donde el objetivo de ventas de la oficina excede a la suma de las cuotas de los vendedores individuales.

```
SELECT CIUDAD
  FROM OFICINAS
 WHERE OBJETIVO > (SELECT SUM(CUOTA)
                      FROM REPVENTAS
                     WHERE OFICINA.REP = OFICINA)
```

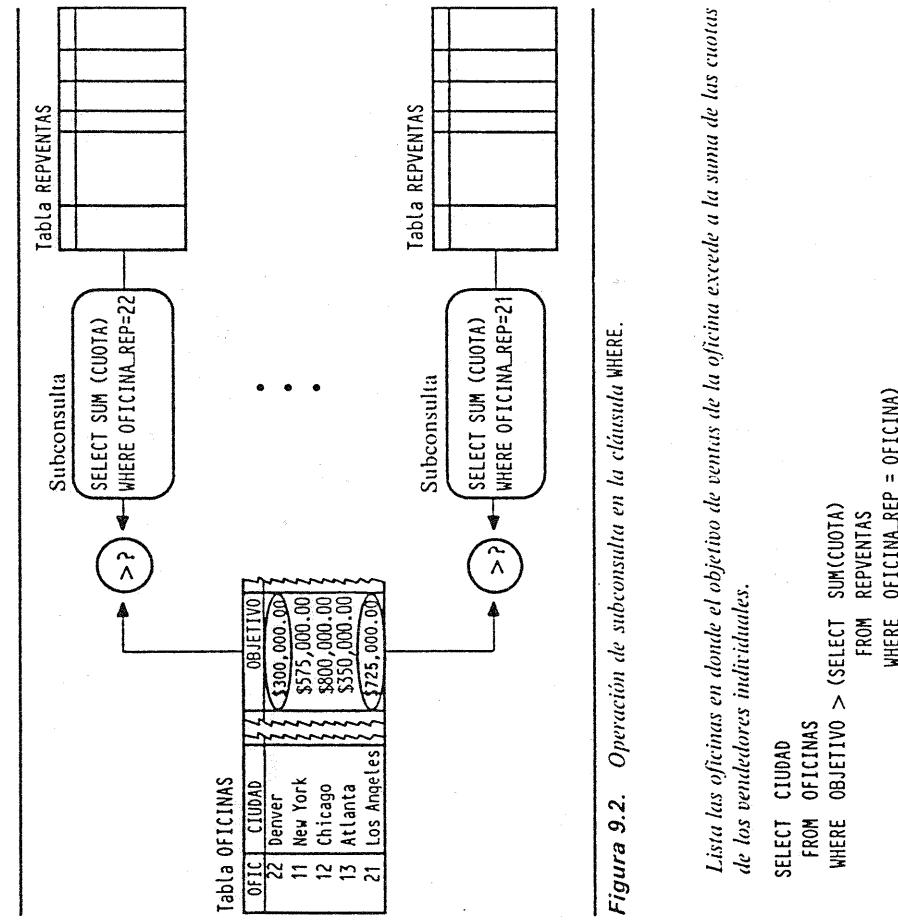
CIUDAD

Chicago
Los Angeles

La Figura 9.2 muestra conceptualmente cómo lleva a cabo SQL una consulta principal que extrae sus datos de la tabla OFICINAS, y la cláusula WHERE selecciona qué oficinas serán incluidas en los resultados de la consulta. SQL recorre las filas de la tabla OFICINAS una a una, aplicándoles el test establecido en la cláusula WHERE. La cláusula WHERE compara el valor de la columna OBJETIVO de la fila actual con el valor producido por la subconsulta. Para examinar el valor OBJETIVO, SQL lleva a cabo la subconsulta, determinando la suma de las cuotas para los vendedores en la oficina «actual». La subconsulta produce un número, y la cláusula WHERE compara el número con el valor OBJETIVO, seleccionando o rechazando la oficina actual en base a la comparación. Como muestra la figura, SQL efectúa la subconsulta repetidamente, una vez por cada fila examinada en la cláusula WHERE de la consulta principal.

Referencias externas

Dentro del cuerpo de una subconsulta, con frecuencia es necesario referirse al valor de una columna en la fila «actual» de la consulta principal. Consideremos una vez más la consulta de la sección anterior:



Como muestra el ejemplo anterior, el valor de la columna en una referencia externa se toma de la fila que actualmente está siendo examinada por la consulta principal.

Condiciones de búsqueda en subconsultas

Una subconsulta forma parte siempre de una condición de búsqueda en la cláusula WHERE o HAVING. El Capítulo 6 describió las condiciones de búsqueda simples que pueden ser utilizadas en estas cláusulas. Además, SQL ofrece estas *condiciones de búsqueda en subconsultas*:

- **Test de comparación subconsulta.** Compara el valor de una expresión con un valor único por una subconsulta. Este test se asemeja al test de comparación simple.

■ **Test de pertenencia a conjunto subconsulta.** Comprueba si el valor de una expresión coincide con uno del conjunto de valores producido por una subconsulta. Este test se asemeja al test de pertenencia a conjunto simple.

■ **Test de existencia.** Examina si una subconsulta produce alguna fila de resultados.

■ **Test de comparación cuantificada.** Compara el valor de una expresión con cada uno del conjunto de valores producido por una subconsulta.

NOMBRE
Bill Adams
Sue Smith
Larry Fitch

La subconsulta del ejemplo recupera el objetivo de ventas de la oficina de Atlanta. El valor se utiliza entonces para seleccionar los vendedores cuyas cuotas son superiores o iguales al objetivo.

El test de comparación subconsulta ofrece los mismos seis operadores de comparación ($=$, $<$, $>$, \leq , \geq) disponibles con el test de comparación simple. La subconsulta especificada en este test debe producir una *única fila* de resultados. Si la subconsulta produce múltiples filas, la comparación no tiene sentido, y SQL informa de una condición de error. Si la subconsulta no produce filas o produce un valor NULL, el test de comparación devuelve NULL.

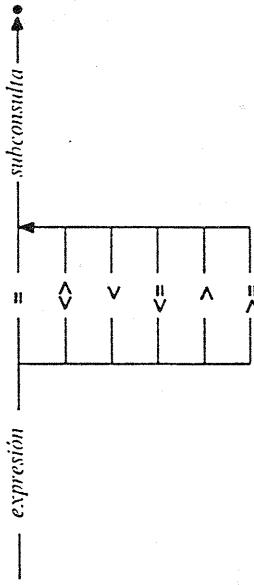


Figura 9.3. Diagrama sintáctico del test de comparación subconsulta.

He aquí algunos ejemplos adicionales del test de comparación subconsulta:

Lista todos los clientes atendidos por Bill Adams.

```

SELECT EMPRESA
      FROM CLIENTES
     WHERE REP_CLIE = (SELECT NUM_EMPL
                           FROM REPENTAS
                          WHERE NOMBRE = 'Bill Adams')

EMPRESA
-----+
Acme Mfg.
Three-Way Lines
  
```

Test de comparación subconsulta

$(=, <, >, \leq, \geq, \neq)$

El test de comparación subconsulta es una forma modificada del test de comparación simple, tal como se muestra en la Figura 9.3. Compara el valor de una expresión con el valor producido por una subconsulta, y devuelve un resultado TRUE si la comparación es cierta. Este test se utiliza para comparar un valor de la fila que está siendo examinada con un valor único producido por una subconsulta, como en este ejemplo:

Lista los vendedores cuyas cuotas son iguales o superiores al objetivo de la oficina de ventas de Atlanta.

```

SELECT NOMBRE
      FROM REPENTAS
     WHERE CUOTA >= (SELECT OBJETIVO
                           FROM OFICINAS
                          WHERE CIUDAD = 'Atlanta')

NOMBRE
-----+
  
```

Lista todos los productos del fabricante ACI para los cuales las existencias superan a las existencias del producto AC1-41004.

```
SELECT DESCRIPCION, EXISTENCIAS
FROM PRODUCTOS
WHERE ID_FAB = 'ACI'
AND EXISTENCIAS > (SELECT EXISTENCIAS
                     FROM PRODUCTOS
                     WHERE ID_FAB = 'ACI'
                     AND ID_PRODUCTO = '41004')

DESCRIPCION      EXISTENCIAS
-----  -----
Artículo Tipo 3    207
Artículo Tipo 1    277
Artículo Tipo 2    167
```

Observe que el test de comparación subconsulta permite una subconsulta únicamente en el lado derecho del operador de comparación. Esta comparación:

$A < (\text{subconsulta})$

está permitida, pero esta comparación:

$(\text{subconsulta}) > A$

no está permitida. Esto no limita la potencia del test de comparación, ya que el operador de una comparación de desigualdad siempre puede ser «volteado» para que la subconsulta se coloque en el lado derecho de la desigualdad. Sin embargo, esto significa que a veces se debe «dar la vuelta» a la lógica de una petición en lenguaje natural para obtener un formato de la petición que corresponda a una sentencia legal en SQL.

Test de pertenencia a conjunto (IN)

El test de pertenencia a conjunto subconsulta (IN) es una forma modificada del test de pertenencia a conjunto simple, como se muestra en la Figura 9.4. Compara un único valor de datos con una columna de valores producida por una subconsulta y devuelve un resultado TRUE si el valor coincide con uno de los valores de la columna. Este test se utiliza cuando se necesita comparar un valor de la fila que está siendo examinada con un *conjunto* de valores producidos por una subconsulta, como se muestra en estos ejemplos:

Lista los vendedores que trabajan en oficinas que superan su objetivo.

```
SELECT NOMBRE
      FROM REVENTAS
     WHERE OFICINA REP IN (SELECT OFICINA
                           FROM OFICINAS
                          WHERE VENTAS > OBJETIVO)
```

Lista los vendedores que trabajan en oficinas dirigidas por Larry Fitch (empleado 108).

```
SELECT NOMBRE
      FROM REVENTAS
     WHERE OFICINA REP NOT IN (SELECT OFICINA
                               FROM OFICINAS
                              WHERE DIR = 108)
```

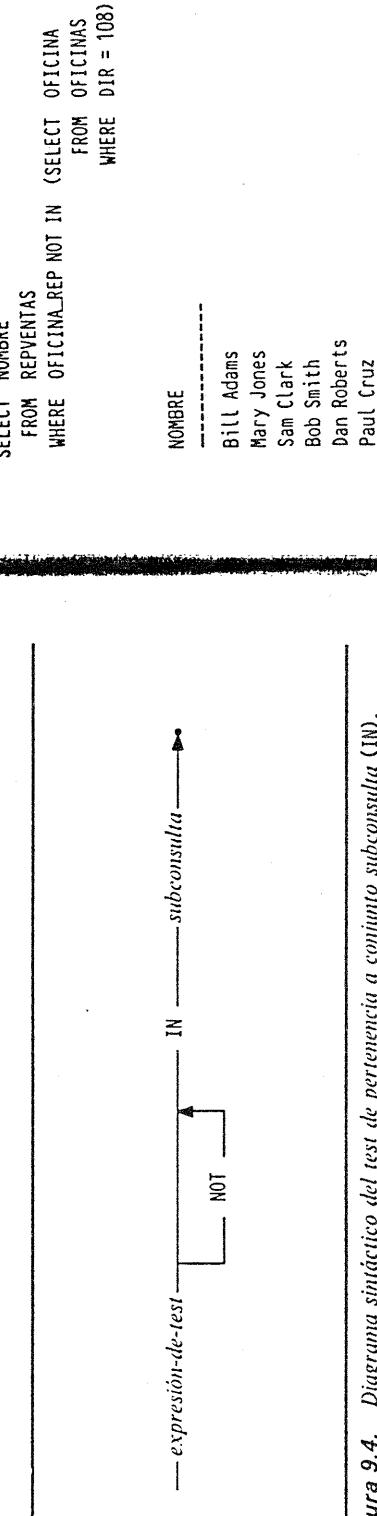


Figura 9.4. Diagrama sintáctico del test de pertenencia a conjunto subconsulta (IN).

Lista todos los clientes que han recibido pedidos de ACI Widgets (fabricante ACI, números de productos que comienzan con «4100») entre enero y junio de 1990.

```
SELECT EMPRESA
      FROM CLIENTES
     WHERE NUM_CLIE IN (SELECT DISTINCT CLIE
                           FROM PEDIDOS
                          WHERE FAB = 'ACI'
                            AND PRODUCTO LIKE '4100 %'
                            AND FECHA_PEDIDO BETWEEN '01-ENE-90'
                                         AND '30-JUN-90')
EMPRESA
-----
Acme Mfg.
Ace International
Holm & Landis
JCP Inc.
```

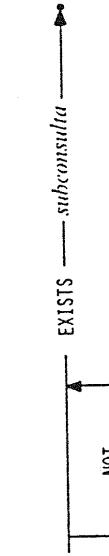
En cada uno de estos ejemplos, la subconsulta produce una columna de valores, y la cláusula WHERE de la consulta principal comprueba si un valor de una fila de la consulta principal coincide con uno de los valores de la columna. El formato de la subconsulta en el test IN funciona por tanto exactamente igual al del test IN simple, excepto que el conjunto de valores es producido por una subconsulta en lugar de ser explícitamente listado en la sentencia.

Test de existencia (EXISTS)

El test de existencia (EXISTS) comprueba si una subconsulta produce alguna fila de resultados, tal como se muestra en la Figura 9.5. No hay test de comparación simple que se asemeje al test de existencia; solamente se utiliza con subconsultas.

He aquí un ejemplo de una petición que puede ser expresada sencillamente utilizando un test de existencia:

«Lista los productos para los cuales se ha recibido un pedido de \$25.000 o más.»



La petición podría ser fácilmente expresada también de esta forma:

«Lista los productos para los cuales existe al menos un pedido en la tabla PEDIDOS a) que se refiere al producto en cuestión y b) que tiene un importe de al menos \$25.000.»

La sentencia SELECT utilizada para recuperar la lista solicitada de productos se asemeja estrechamente a la petición así expresada:

```
SELECT DISTINCT DESCRIPCION
      FROM PRODUCTOS
     WHERE EXISTS (SELECT NUM_PEDIDO
                     FROM PEDIDOS
                    WHERE PRODUCTO = ID_PRODUCTO
                      AND FAB = ID_FAB
                      AND IMPORTE >= 25000.00)
DESCRIPCION
-----
500-lb Brace
Left Hinge
Right Hinge
Widget Remover
```

Conceptualmente, SQL procesa esta consulta recorriendo la tabla PRODUCTOS y efectuando la subconsulta para cada producto. La subconsulta produce una columna que contiene los números de pedidos de aquellos pedidos del producto «actual» que superan los \$25.000. Si hay alguno de tales pedidos (es decir, si la columna no está vacía), el test EXISTS es TRUE. Si la subconsulta no produce filas, el test EXISTS es FALSE. El test EXISTS no puede producir un valor NULL.

Se puede inventar la lógica del test EXISTS utilizando la forma NOT EXISTS. En este caso, el test es TRUE si la subconsulta no produce filas, y FALSE en caso contrario.

Observe que la condición de búsqueda EXISTS no *utiliza* realmente los resultados de la subconsulta. Simplemente comprueba si la subconsulta produce algún resultado. Por esta razón, SQL relaja la regla de que «las subconsultas deben devolver una única columna de datos», y permite utilizar la forma SELECT * en la subconsulta de un test EXISTS. La subconsulta podría por tanto haber sido escrita:

Lista los productos para los cuales se ha recibido un pedido de \$25.000, o más.

```
SELECT DESCRIPCION
      FROM PRODUCTOS
     WHERE EXISTS (SELECT *
                     FROM PEDIDOS
                    WHERE PRODUCTO = ID_PRODUCTO
                      AND FAB = ID_FAB
                      AND IMPORTE >= 25000.00)
```

Figura 9.5. Diagrama sintáctico del test de existencia (EXISTS).

En la práctica, la subconsulta en un test EXISTS se escribe *siempre* utilizando la notación SELECT *.

He aquí algunos ejemplos adicionales de consultas que utilizan EXISTS:

“*Lista los clientes asignados a Sue Smith que no han remitido un pedido superior a \$3.000.*

```
SELECT EMPRESA
  FROM CLIENTES
 WHERE REP_CLIE = (SELECT NUM_EMPL
                      FROM REPIENTES
                     WHERE NOMBRE = 'Sue Smith')
```

AND NOT EXISTS (SELECT *
 FROM PEDIDOS
 WHERE CLIE = NUM_CLIE
 AND IMPORTE > 3000.00)

EMPRESA
Carter & Sons
Fred Lewis Corp.

Listar las oficinas en donde haya un vendedor cuya cuota represente más del 55 % del objetivo de la oficina.

```
SELECT CIUDAD
  FROM OFICINAS
 WHERE EXISTS (SELECT *
                  FROM REPIENTES
                 WHERE OFICINA.REP = OFICINA
                   AND CUOTA > (.55 * OBJETIVO))
```

CIUDAD
Denver
Atlanta

Observe que en cada uno de estos ejemplos la subconsulta incluye una referencia externa a una columna de la tabla en la consulta principal. En la práctica, la subconsulta de un test EXISTS siempre contiene una referencia externa que «enlaza» la subconsulta a la fila que actualmente está siendo examinada por la consulta principal.

Test cuantificados (ANY y ALL) *

La versión subconsulta del test IN comprueba si un valor de dato es igual a algún valor en una columna de los resultados de la subconsulta. SQL proporciona dos test cuantificados, ANY y ALL, que extienden esta noción a otros operadores de comparación, tales como mayor que (>) y menor que (<). Ambos tests

comparan un valor de dato con la columna de valores producidos por una subconsulta, como se muestra en la Figura 9.6.

El test ANY *. El test ANY se utiliza conjuntamente con uno de los seis operadores de comparación SQL (=, <, >, <=, >=) para comparar un único valor de test con una columna de valores producidos por una subconsulta. Para efectuar el test, SQL utiliza el operador de comparación especificado para comparar el valor de test con *cada* valor de datos en la columna, uno cada vez. Si *alguna* de las comparaciones individuales producen un resultado TRUE, el test ANY devuelve un resultado TRUE.

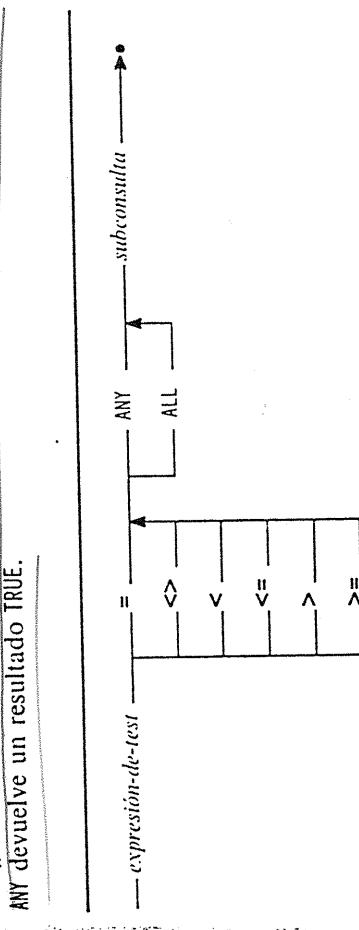


Figura 9.6. Diagrama sintáctico de los tests de comparación cuantificada (ANY y ALL).

He aquí un ejemplo de una petición que puede ser manejada con el test ANY:

Lista los vendedores que han aceptado un pedido que represente más del 10 % de su cuota.

```
SELECT NOMBRE
  FROM REPIENTES
 WHERE (.1 * CUOTA) < ANY (SELECT IMPORTE
                           FROM PEDIDOS
                          WHERE REP = NUM_EMPL)
```

NOMBRE
Sam Clark
Larry Fitch
Nancy Angelli

Conceptualmente, la consulta principal examina cada fila de la tabla REPIENTES, una a una. La subconsulta encuentra todos los pedidos aceptados por el vendedor «actual» y devuelve una columna que contiene el importe de esos pedidos. La cláusula WHERE de la consulta principal calcula entonces el diez por ciento de la cuota del vendedor actual y la utiliza como valor del test, comparando con los resultados de la subconsulta.

rándolo con todos los importes de pedidos producidos por la subconsulta. Si hay *algún* importe que excede al valor de test calculado, el test «*< ANY*» devuelve TRUE y el vendedor queda incluido en los resultados de la consulta. La palabra clave SOME es una alternativa para ANY especificada por el estándar SQL ANSI/ISO. Qualquier de las dos palabras clave puede ser utilizada generalmente, pero algunos productos DBMS no soportan SOME.

El test ANY puede ser a veces difícil de entender ya que afecta a un conjunto entero de comparaciones, no sólo a una. Sirve de ayuda leer el test de una manera ligeramente diferente a la que aparece en la sentencia. Si aparece este test ANY:

```
WHERE X < ANY (SELECT Y ...)
```

en vez de leer el test de este modo:

«donde X es menor que cualquier Y seleccionado...»

pruebe a leerlo de este otro modo:

«en donde, para *algún* Y, X es menor que Y»

Cuando se utiliza este truco, la consulta anterior pasa a ser:

«Selecciona los vendedores donde, para *algún* pedido aceptado por el vendedor, el diez por ciento de la cuota del vendedor es menor que el importe del pedido.»

Si la subconsulta en un test ANY no produce filas de resultados, o si los resultados de la consulta incluyen valores NULL, la operación del test ANY puede variar de un DBMS a otro. El estándar SQL ANSI/ISO especifica estas reglas detalladas para describir los resultados del test ANY cuando el valor de test se compara con la columna de resultados de la subconsulta:

■ Si la subconsulta produce una columna vacía de resultados, el test ANY devuelve FALSE —no hay ningún valor producido por la subconsulta para el cual el test de comparación resulte cierto.

■ Si el test de comparación es TRUE para *al menos uno* de los valores de datos en la columna, entonces la condición de búsqueda ANY devuelve TRUE —ciertamente existe algún valor producido por la subconsulta para el cual el test de comparación es cierto.

■ Si el test de comparación es FALSE para *todos* los valores de datos en la columna, entonces la condición de búsqueda ANY devuelve FALSE. En este caso,

se puede establecer sin lugar a dudas que no hay ningún valor producido por la subconsulta para el cual el test de comparación sea cierto.

- Si el test de comparación no es TRUE para ningún valor de datos en la columna, pero es NULL para uno o más de los valores, entonces la condición de búsqueda ANY devuelve NULL. En esta situación, no se puede concluir si existe un valor producido por la subconsulta para el cual el test de comparación sea cierto; puede haberlo o no, dependiendo de los valores «correctos» de los datos NULL (desconocidos).

El operador de comparación ANY puede ser engañoso al utilizarlo en la práctica, especialmente junto con el operador de comparación desigualdad ($< >$). He aquí un ejemplo que expone el problema:

«Lista los nombres y edades de todas las personas en el equipo de ventas que no dirigen una oficina.»

Es tentador expresar esta consulta del modo siguiente:

```
SELECT NOMBRE, EDAD
  FROM REPVENTAS
 WHERE NUM_EMPL < > ANY (SELECT DIR
   FROM OFICINAS)
```

La subconsulta:

```
SELECT DIR
  FROM OFICINAS
```

produce obviamente los números de empleado de los directores, y por tanto la consulta *parece* estar diciendo:

«Halla los vendedores que no son directores de ninguna oficina.»

Pero esto *no es* lo que la consulta dice. Lo que dice en realidad es:

«Halla cada uno de los vendedores que, *para alguna oficina*, no es el director de esa oficina.»

Naturalmente para cualquier vendedor, es posible hallar *alguna* oficina en donde ese vendedor no es el director. Los resultados de la consulta incluirían *a todos* los vendedores, y por tanto fallaría en responder a la cuestión que le fue propuesta. La consulta correcta es:

```
SELECT NOMBRE, EDAD
  FROM REPVENTAS
 WHERE NOT (NUM_EMPL = ANY (SELECT DIR
   FROM OFICINAS))
```

NOMBRE	EDAD
Mary Jones	31
Sue Smith	48
Dan Roberts	45
Tom Snyder	41
Paul Cruz	29
Nancy Angelli	49

Siempre se puede transformar una consulta con un test ANY en una consulta con un test EXISTS trasladando la comparación *al interior* de la condición de búsqueda de la subconsulta. Generalmente ésta es una buena idea, ya que elimina errores como el que acabamos de describir. He aquí una forma alternativa de la consulta, utilizando el test EXISTS:

NOMBRE	EDAD
Mary Jones	31
Sue Smith	48
Dan Roberts	45
Tom Snyder	41
Paul Cruz	29
Nancy Angelli	49

El test ALL *. Al igual que el test ANY, el test ALL se utiliza conjuntamente con uno de los seis operadores de comparación SQL ($=$, $<$, $>$, \leq , \geq) para comparar un único valor de test con una columna de valores de datos producidos por una subconsulta. Para efectuar el test, SQL utiliza el operador de comparación especificado para comparar el valor de test con todos y *cada* uno de los valores de datos de la columna. Si *todos* las comparaciones individuales producen un resultado TRUE, el test ALL devuelve un resultado TRUE. He aquí un ejemplo de una petición que puede ser manejada con el test ALL:

Lista las oficinas y sus objetivos en donde todos los vendedores tienen ventas que superan al 50 % del objetivo de la oficina.

```
SELECT CIUDAD, OBJETIVO
  FROM OFICINAS
 WHERE (.50 * OBJETIVO) < ALL(SELECT VENTAS
                                FROM REVENTAS
                               WHERE OFICINA.REP = OFICINA)
```

CIUDAD	OBJETIVO
Denver	\$300,000.00
New York	\$575,000.00
Atlanta	\$350,000.00

Conceptualmente, la consulta principal examina cada fila de la tabla OFFICES, una a una. La subconsulta encuentra todos los vendedores que trabajan en la oficina «actual» y devuelve una columna que contiene las ventas correspondientes a cada vendedor. La cláusula WHERE de la consulta principal calcula entonces el 50 % del objetivo de la oficina y lo utiliza como valor de ventas producidos por la subconsulta. Si *todos* los valores de ventas exceden al valor de test calculado, el test «< ALL» devuelve TRUE y la oficina se incluye en los resultados de la consulta. Si no, la oficina no se incluye en los resultados de la consulta.

Al igual que el test ANY, el test ALL puede ser difícil de entender ya que afecta a un conjunto entero de comparaciones, no sólo a una. De nuevo, sirve de ayuda leer el test de una manera ligeramente diferente a la que aparece en la sentencia. Si aparece este test ALL:

WHERE X < ALL (SELECT Y ...)

en vez de leerlo de este modo:

«donde X es menor que todo Y seleccionado...»

pruebe a leer el test de este otro modo:

«donde, para todo Y, X es menor que Y»

Cuando se utiliza este truco, la consulta anterior pasa a ser:

«Selecciona las oficinas en donde, para *todos* los vendedores que trabajan en la oficina, el 50 % del objetivo de la oficina es menor que las ventas del vendedor.»

Si la subconsulta en un test ALL no produce filas de resultados, o si los resultados incluyen valores NULL, la operación del test ALL puede variar de un DBMS a otro. El estándar SQL ANSI/ISO especifica estas reglas detalladas para describir los resultados del test ALL cuando el valor de test se compara con la columna de resultados de la subconsulta:

- Si la subconsulta produce una columna vacía de resultados, el test ALL devuelve TRUE. El test de comparación es cierto para todos los valores producidos por la subconsulta; lo que pasa es que no hay ningún valor.

■ Si el test de comparación es TRUE para todos y cada uno de los valores de datos en la columna, entonces la condición de búsqueda ALL devuelve TRUE. Una vez más, el test de comparación es cierto para todos los valores producidos por la subconsulta.

■ Si el test de comparación es FALSE para algún valor de dato en la columna, entonces la condición de búsqueda ALL devuelve FALSE. En este caso, no se puede concluir que el test de comparación sea cierto para todos y cada uno de los valores de datos producidos por la consulta.

■ Si el test de comparación no es FALSE para ningún valor de datos en la columna, pero es NULL para uno o más de los valores, entonces la condición de búsqueda ALL devuelve NULL. En esta situación, no se puede concluir si existe un valor producido por la subconsulta para el cual el test de comparación sea cierto; puede haberlo o no, dependiendo de los valores «correctos» de los datos NULL (desconocidos).

Los errores sutiles que pueden ocurrir con el test ANY cuando se combina con el operador de comparación desigualdad ($<$ $>$) también ocurren con el test ALL. Como con el test ANY, el test ALL puede siempre convertirse a un test EXISTS equivalente mediante el traslado de la comparación al interior de la subconsulta.

Subconsultas y composiciones

Usted puede haber advertido mientras leía este capítulo que muchas de las consultas que han sido escritas utilizando subconsultas podrían haber sido formuladas como consultas multitable o composiciones. Este es con frecuencia el caso, y SQL permite escribir la consulta de cualquiera de ambos modos. Este ejemplo ilustra el punto:

Lista los nombres y edades de los vendedores que trabajan en oficinas de la región Oeste.

```
SELECT NOMBRE, EDAD
  FROM REPVENTAS
 WHERE OFICINAREP IN (SELECT OFICINA
                        FROM OFICINAS
                      WHERE REGION = 'Oeste')
```

NOMBRE	EDAD
Sue Smith	48
Larry Fitch	62
Nancy Angelli	49

Esta forma de consulta sigue un estrecho paralelo con la petición formulada. La subconsulta genera una lista de oficinas en la región Oeste, y la consulta principal determina los vendedores que trabajan en una de las oficinas de la lista. He aquí una forma alternativa de la consulta, utilizando una composición de dos tablas:

Lista los nombres y edades de los vendedores que trabajan en oficinas de la región Oeste.

```
SELECT NOMBRE, EDAD
  FROM REPVENTAS, OFICINAS
 WHERE OFICINAREP = OFICINA
   AND REGION = 'Oeste'
```

NOMBRE	EDAD
Sue Smith	48
Larry Fitch	62
Nancy Angelli	49

Esta forma de la consulta compone la tabla REPVENTAS con la tabla OFICINAS para determinar la región en donde cada vendedor trabaja, y luego elimina aquellos que no trabajan en la región Oeste.

Cualquiera de las dos consultas encontrará los vendedores correctos, y ninguna de las dos es «correcta» o «equivocada». Mucha gente encontrará la primera forma (con la subconsulta) más natural, ya que la petición en lenguaje natural no pide ninguna información respecto a oficinas, y porque parece un poco extraño componer las tablas REPVENTAS y OFICINAS para responder a la petición. Naturalmente si la petición se modifica para pedir información de la tabla OFICINAS:

«Lista los nombres y edades de los vendedores que trabajan en oficinas en la región Oeste y las ciudades en donde ellos trabajan.»

la forma subconsulta ya no funcionará, y deberá ser utilizada la consulta de dos tablas. Y al contrario, hay muchas consultas con subconsultas que no *pueden* ser traducidas a una composición equivalente. He aquí un sencillo ejemplo:

Lista los nombres y edades de los vendedores que tienen cuotas por encima del promedio.

```
SELECT NOMBRE, EDAD
  FROM REPVENTAS
 WHERE CUOTA > (SELECT AVG(CUOTA)
                  FROM REPVENTAS)
```

NOMBRE	EDAD
Bill Adams	37
Sue Smith	48
Larry Fitch	62

En este caso, la consulta interna es una consulta sumaria y la consulta externa no lo es, por lo que no hay modo de que las dos consultas puedan combinarse en una única composición.

Subconsultas anidadas

Todas las consultas descritas hasta ahora en este capítulo han sido consultas «de dos niveles», afectando a una consulta principal y una subconsulta. Del mismo modo que se puede utilizar una subconsulta «dentro de» una consulta principal, se puede utilizar una subconsulta dentro de otra subconsulta. He aquí un ejemplo de una petición que se representa naturalmente como una consulta de tres niveles, con una consulta principal, una subconsulta y una subsubconsulta:

Lista los clientes cuyos vendedores están asignados a oficinas en la región de ventas Este.

```
SELECT EMPRESA
      FROM CLIENTES
     WHERE REP_CLIE IN (SELECT NUM_EMPL
                           FROM REPVENTAS
                          WHERE OFICINA REP IN (SELECT OFICINA
                                                 FROM OFICINAS
                                                WHERE REGION = 'Oeste'))
```

EMPRESA

```
First Corp.
Smithson Corp.
AAA Investments
JCP Inc.
Chen Associates
QMA Assoc.
Ian & Schmidt
Acme Mfg.
```

En este ejemplo, la subconsulta más interna:

```
SELECT OFICINA
      FROM OFICINAS
     WHERE REGION = 'Este'
```

produce una columna que contiene los números de oficina de las oficinas en la región Este. La subconsulta siguiente:

```
SELECT NUM_EMPL
      FROM REPVENTAS
     WHERE OFICINA REP IN (subconsulta)
```

produce una columna que contiene los números de empleado de los vendedores que trabajan en una de las oficinas seleccionadas. Finalmente, la consulta más externa:

```
SELECT EMPRESA
      FROM CLIENTES
     WHERE REP_CLIE IN (subconsulta)
```

Todas las clientes cuyos vendedores tienen uno de los números de empleado seleccionados.

La misma técnica utilizada en esta consulta de tres niveles puede utilizarse para construir consultas con cuatro o más niveles. El estándar SQL ANSI/ISO no especifica un número máximo de niveles de anidación, pero en la práctica una consulta consume mucho más tiempo cuando se incrementa el número de niveles. La consulta también resulta más difícil de leer, comprender y mantener cuando contiene más de uno o dos niveles de subconsultas. Muchas implementaciones SQL restringen el número de niveles a un número relativamente pequeño.

Subconsultas correlacionadas*

Conceptualmente, SQL efectúa una subconsulta una y otra vez —una vez por cada fila de la consulta principal—. Para muchas subconsultas, sin embargo, la subconsulta produce los *mismos* resultados para cada fila o grupo de filas. He aquí un ejemplo:

Lista las oficinas de ventas cuyas ventas están por debajo del objetivo medio.

```
SELECT CIUDAD
      FROM OFICINAS
     WHERE VENTAS < (SELECT AVG(OBJETIVO)
                       FROM OFICINAS)
```

```
CIUDAD
-----
Denver
Atlanta
```

En esta consulta sería tonto efectuar la subconsulta cinco veces (una vez por cada oficina). El objetivo medio no cambia con cada oficina; es totalmente independiente de la oficina que actualmente esté siendo examinada. Como resultado, SQL puede manejar la consulta efectuando primero la subconsulta,

determinando el objetivo medio (\$550.000), y convirtiendo luego la consulta principal en:

```
SELECT CIUDAD
      FROM OFICINAS
     WHERE VENTAS < 550000.00
```

Las implementaciones comerciales de SQL utilizan este atajo cada vez que es posible para reducir la cantidad de procesamiento requerido por una subconsulta. Sin embargo, el atajo no puede ser utilizado si la subconsulta contiene una referencia externa, como en este ejemplo:

Lista todas las oficinas cuyos objetivos exceden a la suma de las cuotas de los vendedores que trabajan en ellas:

```
SELECT CIUDAD
      FROM OFICINAS
     WHERE OBJETIVO > (SELECT SUM(CUOTA)
                           FROM REPVENTAS
                          WHERE OFICINAREP = OFICINA)

      CIUDAD
      -----
Chicago
Los Angeles
```

Por cada fila de la tabla OFICINAS que se examina mediante la cláusula WHERE de la consulta principal, la columna OFICINA (que aparece en la subconsulta como una referencia externa) tiene un valor diferente. Por tanto SQL no tiene elección sino que debe llevar a cabo esta subconsulta cinco veces —una vez por cada fila de la tabla OFICINAS—. Una subconsulta que contiene una referencia externa se denomina *subconsulta correlacionada*, ya que sus resultados están correlacionados con cada fila individual de la consulta principal. Por la misma razón, una referencia externa se denomina a veces *referencia correlacionada*.

Una subconsulta puede contener una referencia externa a una tabla en la cláusula FROM de cualquier consulta que contenga la subconsulta, no importa lo profundamente que las subconsultas estén anidadas. Un nombre de columna en una subconsulta de cuarto nivel, por ejemplo, puede referirse a una de las tablas indicadas en la cláusula FROM de la consulta principal, o a una tabla de cualquiera de las subconsultas que contiene la subconsulta en la cual el nombre de columna aparece. Con independencia del nivel de anidación, una referencia externa siempre toma el valor de la columna en la fila «actual» de la tabla que está siendo examinada.

Puesto que una subconsulta puede contener referencias externas, hay incluso más posibilidad de que aparezcan nombres de columnas ambiguas en una subconsulta que en una consulta principal. Cuando un nombre de columna sin

cualificar aparece dentro de una subconsulta, SQL debe determinar si se refiere a una tabla en la cláusula FROM propia de la subconsulta, o en una cláusula FROM de una consulta que contiene a la subconsulta. Para minimizar la posibilidad de confusión, SQL siempre interpreta una referencia a columna en una subconsulta utilizando la cláusula FROM más cercana posible. Para ilustrar este punto, he aquí un ejemplo que utiliza la misma tabla en la consulta y en la subconsulta:

Lista los vendedores que tienen más de 40 años y que dirigen a un vendedor por encima de la cuota.

```
SELECT NOMBRE
      FROM REPVENTAS
     WHERE EDAD > 40
       AND NUMEMPL IN (SELECT DIRECTOR
                           FROM REPVENTAS
                          WHERE VENTAS > CUOTA)

      NOMBRE
      -----
Sam Clark
Larry Fitch
```

Las columnas DIRECTOR, CUOTA y VENTAS en la subconsulta son referencias a la tabla REPVENTAS en la cláusula FROM propia de la subconsulta; SQL no las interpreta como referencias externas, y la subconsulta no es una subconsulta correlacionada. Como discutimos anteriormente, en este caso SQL puede efectuar la subconsulta primero, encontrando los vendedores que están por encima de la cuota y generando una lista de los números de empleado de su directores. SQL puede entonces volver la atención a la consulta principal, seleccionando los directores cuyo número de empleado aparezcan en la lista general.

Si se desea utilizar una referencia externa dentro de una subconsulta como la del ejemplo anterior, debe utilizarse un alias de tabla para forzar la referencia externa. Esta petición, que añade una condición más de cualificación que la anterior, aparece así:

Lista los directores mayores de 40 años y que dirigen a un vendedor cuyas ventas superan a la cuota y que no trabaja en la misma oficina de ventas que el director.

```
SELECT NOMBRE
      FROM REPVENTAS DIRS
     WHERE EDAD > 40
       AND DIRS.NUMEMPL IN (SELECT DIRECTOR
                           FROM REPVENTAS EMPS
                          WHERE EMPS.CUOTA > EMPS.VENTAS
                            AND EMPS.OFICINAREP <> DIRS.OFICINAREP)

      NOMBRE
      -----
Sam Clark
Larry Fitch
```

La copia de la tabla REPVENTAS utilizada en la consulta principal tiene ahora la marca DIRS, y la copia en la subconsulta tiene la marca EMPS. La subconsulta contiene una condición de búsqueda adicional, requiriendo que el número de oficina del empleado no coincida con el del director. El nombre de columna cualificado DIRS.OFICINA en la subconsulta es una referencia externa, y esta subconsulta es una subconsulta correlacionada.

Subconsultas en la cláusula HAVING *

Aunque las subconsultas suelen encontrarse sobre todo en la cláusula WHERE, también pueden utilizarse en la cláusula HAVING de una consulta. Cuando una subconsulta aparece en la cláusula HAVING, funciona como parte de la selección de grupo de filas efectuada por la cláusula HAVING. Consideremos esta consulta con una subconsulta:

Lista los vendedores cuyo tamaño de pedido medio para productos fabricados por ACI es superior al tamaño de pedido medio global.

```
SELECT NOMBRE, AVG(IMPORTE)
FROM REPVENTAS, PEDIDOS
WHERE NUM_EMPL = REP
AND FAB = 'ACI'
GROUP BY NOMBRE
HAVING AVG(IMPORTE) > (SELECT AVG(IMPORTE)
                           FROM PEDIDOS
                           WHERE NOMBRE = REP)
```

NOMBRE	AVG(IMPORTE)
Sue Smith	\$15,000.00
Tom Snyder	\$22,500.00

La Figura 9.7 muestra conceptualmente cómo funciona esta consulta. La subconsulta calcula el «tamaño de pedido medio global». Se trata de una sencilla subconsulta que no contiene referencias externas, por lo que SQL puede calcular el promedio una vez y utilizarlo luego repetidamente en la cláusula HAVING. La consulta general recorre la tabla PEDIDOS, hallando todos los pedidos de productos ACI, y los agrupa por vendedor. La cláusula HAVING comprueba entonces cada grupo de filas para ver si el tamaño de pedido medio de ese grupo es superior al promedio de todos los pedidos, calculado con antelación. Si es así, el grupo de filas es retenido; si no, el grupo de filas es descartado. Finalmente, la cláusula SELECT produce una fila sumaria por cada grupo, mostrando el nombre del vendedor y el tamaño de pedido medio para cada uno.

También se puede utilizar una subconsulta correlacionada en la cláusula HAVING. Sin embargo, puesto que la subconsulta se evalúa una vez por cada

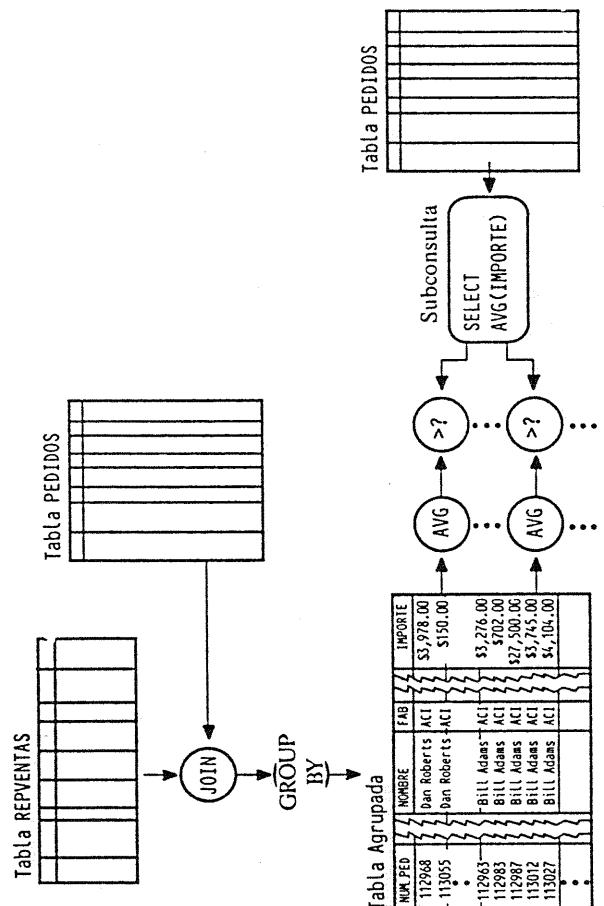


Figura 9.7. Operación subconsulta en la cláusula HAVING.

grupo de filas, todas las referencias externas en la subconsulta correlacionada deben tener un solo valor por cada grupo de filas. Efectivamente, esto significa que la referencia externa debe ser o bien una referencia a una columna de agrupación de la consulta externa o bien estar contenida dentro de una función de columna. En el último caso, el valor de la función de columna para el grupo de filas que está siendo examinado se calcula como parte del procesamiento de la subconsulta.

Si la petición anterior se modifica ligeramente, la subconsulta en la cláusula HAVING se convierte en una subconsulta correlacionada:

Lista los vendedores cuyo tamaño de pedido medio para productos fabricados por ACI es menor tan grande como el tamaño de pedido medio global de ese vendedor.

```
SELECT NOMBRE, AVG(IMPORTE)
FROM REPVENTAS, PEDIDOS
WHERE NUM_EMPL = REP
  AND FAB = 'ACI'
  GROUP BY NOMBRE
  HAVING AVG(IMPORTE) >= (SELECT AVG(IMPORTE)
                                FROM PEDIDOS
                                WHERE REP = NUM_EMPL)
```

NOMBRE	Avg(IMPORTE)
Bill Adams	\$7,865.40
Sue Smith	\$15,000.00
Tom Snyder	\$22,500.00

En este nuevo ejemplo, la subconsulta debe producir «el tamaño de pedido medio global» para los vendedores cuyo grupo de filas está actualmente siendo examinado por la cláusula HAVING. La subconsulta selecciona pedidos para ese vendedor particular, utilizando la referencia externa NUM_EMPL. La referencia externa es legal, ya que NUM_EMPL tiene el mismo valor en todas las filas de un grupo producido por la consulta principal.

Resumen

Este capítulo ha descrito las subconsultas, las cuales permiten utilizar los resultados de una consulta para ayudar a definir otra consulta:

- Una subconsulta es una «consulta dentro de una consulta». Las subconsultas aparecen dentro de una de las condiciones de búsqueda subconsulta en la cláusula WHERE o HAVING.
- Cuando aparece una subconsulta en la cláusula WHERE, los resultados de la subconsulta se utilizan para seleccionar las filas individuales que contribuyen a los datos de los resultados de la consulta.
- Cuando una subconsulta aparece en la cláusula HAVING, los resultados de la subconsulta se utilizan para seleccionar los grupos de filas que contribuyen con datos a los resultados de la consulta.
- Las subconsultas pueden animarse dentro de otras subconsultas.
- La forma subconsulta del test de comparación utiliza uno de los operadores de comparación simple para comparar un valor de test con el valor único devuelto por una subconsulta.
- La forma subconsulta del test de pertenencia a conjunto (IN) compara el valor de test con el conjunto de valores devuelto por una subconsulta.
- El test de existencia (EXISTS) comprueba si una subconsulta devuelve algún valor.
- Los test cuantificados (ANY y ALL) utilizan uno de los operadores de comparación simple para comparar un valor de test con todos los valores devueltos por una subconsulta, comprobando si la comparación resulta cierta para alguno o todos los valores.

- Una subconsulta puede incluir una *referencia externa* a una tabla en cualquiera de las consultas que la contienen, enlazando la subconsulta con la fila «actual» de esa consulta.

Consultas SQL. Resumen final

Aquí concluye la discusión de las consultas SQL y la sentencia SELECT que se inició en el Capítulo 6. Como hemos descrito en los últimos cuatro capítulos, las

Para generar los resultados de una sentencia SELECT:

1. Si la sentencia es una UNION de sentencias SELECT, aplicar los Pasos 2 hasta el 7 a cada una de las sentencias para generar sus resultados de consulta individuales.
2. Formar el producto de las tablas indicadas en la cláusula FROM. Si la cláusula FROM designa una única tabla, el producto es esa tabla.
3. Si hay una cláusula WHERE, aplicar su condición de búsqueda es TRUE (y producto, reteniendo aquellas filas para las cuales la condición de búsqueda es TRUE y descartando aquéllas para las cuales es FALSE o NULL). Si la cláusula WHERE contiene una subconsulta, la subconsulta se efectúa por cada fila conforme es examinada.
4. Si hay una cláusula GROUP BY, disponer las filas restantes de la tabla producto en grupos de filas, de modo que las filas de cada grupo tengan valores idénticos en todas las columnas de agrupación.
5. Si hay una cláusula HAVING, aplicar su condición de búsqueda es TRUE (y reteniendo aquellos grupos para los cuales la condición de búsqueda es TRUE y descartando aquéllas para las cuales es FALSE o NULL). Si la cláusula HAVING contiene una subconsulta, la subconsulta se efectúa para cada grupo de filas conforme es examinada.
6. Para cada fila (o grupo de filas) restante, calcular el valor de cada elemento en la lista de selección para producir una única fila de resultados. Para una referencia de columna simple, utilizar el valor de la columna en la fila (o grupo de filas) actual. Para una función de columna, utilizar como argumento el grupo de filas actual si se especificó GROUP BY, en caso contrario, utilizar el conjunto entero de filas.
7. Si se especifica SELECT DISTINCT, eliminar las filas duplicadas de los resultados que se hubieran producido.
8. Si la sentencia es una UNION de sentencias SELECT, mezclar los resultados de consulta para las sentencias individuales en una única tabla de resultados. Eliminar las filas duplicadas a menos que se haya especificado UNION ALL.
9. Si hay una cláusula ORDER BY, ordenar los resultados de la consulta según se haya especificado.

Las filas generadas por este procedimiento forman los resultados de la consulta.

Figura 9.8. Reglas de procesamiento de consultas SQL (versión final).

cláusulas de las sentencia SELECT proporcionan un conjunto potente y flexible de capacidades para recuperar datos de la base de datos. Cada cláusula juega un papel específico en la recuperación de datos:

- La cláusula FROM especifica las tablas fuente que contribuyen con datos a los resultados de la consulta. Todos los hombres de columna en el cuerpo de la sentencia SELECT deben identificarse sin ambigüedad a una columna de una de estas tablas, o debe ser una referencia externa a una columna de una tabla fuente en una consulta externa.
- La cláusula WHERE, si está presente, selecciona combinaciones individuales de filas procedentes de las tablas fuente que participan en los resultados de la consulta. Las subconsultas en la cláusula WHERE se evalúan para cada fila individual.

- La cláusula GROUP BY, si está presente, agrupa las filas individuales seleccionadas por la cláusula WHERE en grupos de filas.
- La cláusula HAVING, si está presente, selecciona grupos de filas que participan en los resultados de la consulta. Las subconsultas en la cláusula HAVING se evalúan para cada grupo de filas.
- La cláusula SELECT determina qué valores de datos aparecen realmente como columnas en los resultados finales.
- La palabra clave DISTINCT, si está presente, elimina filas duplicadas de los resultados de la consulta.

- El operador UNION, si está presente, mezcla los resultados producidos por las sentencias SELECT individuales en un único conjunto de resultados de la consulta.
- La cláusula ORDER BY, si está presente, ordena los resultados finales basándose en una o más columnas.

La Figura 9.8 muestra la versión final de las reglas para el procesamiento de consultas SQL, ampliadas para incluir subconsultas. Proporciona una definición completa de los resultados producidos por una sentencia SELECT.

Actualización de datos

Tercera parte

SQL no es únicamente un lenguaje de consultas, es un lenguaje completo para recuperar y modificar datos en una base de datos. Los próximos tres capítulos se centran en las actualizaciones de bases de datos. El Capítulo 10 describe las sentencias SQL que añaden datos a una base de datos, suprimen datos de una base de datos y modifican datos existentes en la base de datos. El Capítulo 11 describe cómo mantiene SQL la integridad de los datos almacenados cuando éstos se modifican. El Capítulo 12 describe las características de procesamiento de transacciones en SQL que soportan actualizaciones concurrentes por parte de muchos usuarios diferentes.

10 Actualizaciones de bases de datos

SQL es un lenguaje completo de manipulación de datos que se utiliza no solamente para consultas, sino también para modificar y actualizar los datos de la base de datos. Comparadas con la complejidad de la sentencia SELECT, que soporta las consultas en SQL, las sentencias de SQL que modifican los contenidos de la base de datos son extremadamente sencillas. Sin embargo, las actualizaciones de una base de datos presentan algunos retos para un DBMS a los presentados por las consultas de la base de datos. El DBMS debe proteger la integridad de los datos almacenados durante los cambios, asegurándose que sólo se introduzcan datos válidos y que la base de datos permanezca autoconsistente, incluso en caso de fallos del sistema. El DBMS debe coordinar también las actualizaciones simultáneas por parte de múltiples usuarios, asegurándose que los usuarios y sus modificaciones no interfieran unos con otros.

Este capítulo describe las tres sentencias SQL que se emplean para modificar los contenidos de una base de datos:

- INSERT, que añade nuevas filas de datos a una tabla,
- DELETE, que elimina filas de datos de una tabla, y
- UPDATE, que modifica datos existentes en la base de datos.

En el Capítulo 11 se describen las facilidades SQL para mantener la integridad de datos. El Capítulo 12 trata del soporte SQL para concurrencia multiusuario.

Adición de datos a la base de datos

Tipicamente, una nueva fila de datos se añade a una base de datos relacional cuando una nueva entidad representada por la fila «aparece en el mundo exterior». Por ejemplo, en la base de datos ejemplo:

- Cuando se contrata un nuevo vendedor, debe añadirse una nueva fila a la tabla REPVENTAS para almacenar los datos de ese vendedor.
- Cuando un vendedor firma con un nuevo cliente, debe añadirse una nueva fila a la tabla CLIENTES, que represente al nuevo cliente.
- Cuando un cliente remite un pedido, debe añadirse una nueva fila a la tabla PEDIDOS que contenga los datos del pedido.

En cada caso, la nueva fila se añade para mantener la base de datos como un modelo preciso del mundo real. La unidad de datos más pequeña que puede añadirse a una base de datos relacional es una fila. En general, un DBMS basado en SQL proporciona tres maneras de añadir nuevas filas de datos a una base de datos:

- Una sentencia INSERT de una fila añade una única nueva fila de datos a una tabla. Se utiliza habitualmente en aplicaciones diarias, por ejemplo, en programas de entrada de datos.
- Una sentencia INSERT *multifila* extrae filas de datos de otra parte de la base y las añade a una tabla. Se utiliza habitualmente en procesamiento de fin de mes o de fin de año cuando filas «contiguas» de una tabla se trasladan a una tabla inactiva.
- Una utilidad de *carga masiva* añade datos a una tabla desde un fichero externo a la base de datos. Se utiliza habitualmente para cargar inicialmente la base de datos o para incorporar datos transferidos desde otro sistema informático o recolectados desde muchas localizaciones.

Supongamos que se acaba de contratar un nuevo vendedor, Henry Jacobsen, con los siguientes datos personales:

Nombre:	Henry Jacobsen
Edad:	36
Número de empleado:	111
Título:	Director de ventas
Oficina:	Atlanta (número de oficina 13)
Fecha de contrato:	25 de julio, 1990
Cuota:	Aún no asignada
Ventas anuales hasta la fecha:	\$0.00

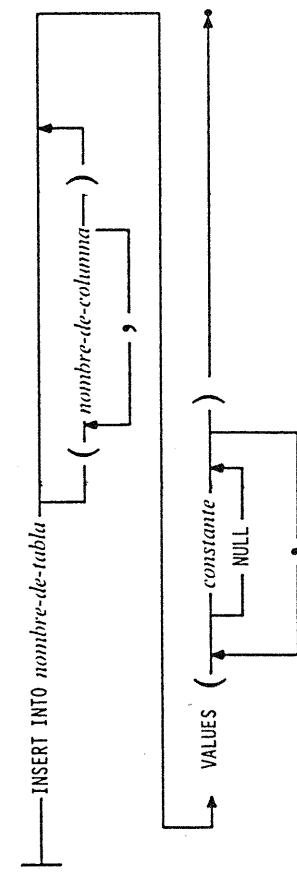
He aquí la sentencia INSERT que añade al Sr. Jacobsen a la base de datos ejemplo:

Incluir a Henry Jacobsen como nuevo vendedor.

```
INSERT INTO
  REPVENTAS (NOMBRE, EDAD, NUM_EMPL, VENTAS, TITULO, CONTRATO, OFICINA, REP)
VALUES ('Henry Jacobsen', 36, 111, 0.00, 'Dir Ventas', '25-jul-90', 13)
```

1 fila insertada.

La Figura 10.2 ilustra gráficamente cómo realiza SQL este sentencia INSERT. Conceptualmente, la sentencia INSERT construye una fila de datos que se corresponde con la estructura en columnas de la tabla, la rellena con los datos de la cláusula VALUES y luego añade la nueva fila a la tabla. Las filas de una tabla no están ordenadas, por lo que no existe la noción de insertar la fila «al comienzo» o «al final» o «entre dos filas» de la tabla. Después de ejecutar la sentencia INSERT, la nueva fila simplemente forma parte de la tabla. Una consulta



La sentencia INSERT de una fila

La sentencia INSERT de una fila, mostrada en la Figura 10.1, añade una nueva fila a una tabla. La cláusula INTO especifica la tabla que recibe la nueva fila (la tabla *destino*), y la cláusula VALUE especifica los valores de datos que la nueva fila contendrá. La lista de columnas indica qué valor va a qué columna de la nueva fila.

Figura 10.1. Diagrama sintáctico de la sentencia INSERT de una fila.

Como muestra este ejemplo, la sentencia INSERT puede ser larga si hay muchas columnas de datos, pero su formato sigue siendo muy sencillo. La segunda sentencia INSERT utiliza la constante de sistema CURRENT_DATE en la cláusula VALUES, haciendo que se inserte la fecha actual como fecha de pedido. DB2, OS/2 Extended Edition y varios otros productos populares SQL soportan esta constante de sistema. Otros productos DBMS disponen de otras constantes de sistema o funciones internas para obtener la fecha y hora actuales.

Se puede utilizar la sentencia INSERT con SQL interactivo para añadir filas a una tabla que crece muy raramente, como por ejemplo la tabla OFICINAS. En la práctica, sin embargo, los datos referentes a un nuevo cliente, un nuevo pedido o un nuevo vendedor son casi siempre añadidos a una base de datos mediante un programa de entrada orientado a formularios. Cuando la entrada de datos está completa, el programa de aplicación inserta la nueva fila de datos utilizando SQL interactivo o programado, sin embargo, la sentencia INSERT es la misma.

El nombre de tabla especificado en la sentencia INSERT es normalmente un nombre de tabla no cualificado, que especifica una tabla propia. Para insertar datos en una tabla propiedad de otro usuario, se puede especificar un nombre de tabla cualificado. Naturalmente es necesario además tener permiso para insertar datos en la tabla porque en caso contrario la sentencia INSERT fallará. El esquema de seguridad de SQL y los permisos se describen en el Capítulo 15. El propósito de la lista de columnas en la sentencia INSERT es hacer corresponder los valores de datos en la cláusula VALUES con las columnas que van a recibirlas. La lista de valores y la lista de columnas deben contener el mismo número de elementos, y el tipo de dato de cada valor debe ser compatible con el tipo de dato de la columna correspondiente, o en caso contrario se producirá un error. El estándar ANSI/ISO exige nombres de columna sin cualificar en la lista de columnas, pero muchas implementaciones permiten nombres cualificados. Naturalmente no puede existir ambigüedad alguna en los nombres de columna, ya que deben referenciar todas a columnas de la tabla destino.

Inserción de valores NULL. Cuando SQL inserta una nueva fila de datos en una tabla, automáticamente asigna un valor NULL a cualquier columna cuyo nombre falté de la lista de columnas de la sentencia INSERT. En esta sentencia INSERT, que añade el Sr. Jacobsen a la tabla REPVENTAS, las columnas CUOTA y DIRECTOR están omitidas:

```
INSERT INTO
  REPVENTAS
  VALUES ('Henry Jacobsen', 36, 111, 0.00, 'Dir Ventas', '25-jul-90', 13)
```

Como resultado, la fila recién añadida tiene un valor NULL en las columnas CUOTA y DIRECTOR, tal como se muestra en la Figura 10.2. Se puede hacer más explícita la asignación del valor NULL incluyendo las columnas en la lista y especificando

Sentencia SQL

```
INSERT INTO
  REPVENTAS (NOMBRE, EDAD, NUM_EMPL,...)
  VALUES ('Henry Jacobsen', 36, 111, ...)
```

Nueva fila

NUM_EMPL	NOMBRE	EDAD	REP_OFICINA	TITULO	CONTRATO	DIRECTOR	CUOTA	VENTAS
105	Bill Adams	37	13	Rep Ventas	12-FEB-88	104	\$350,000.00	\$367,911.00
109	Mary Jones	31	11	Rep Ventas	12-OCT-89	106	\$300,000.00	\$392,725.00
102	Sue Smith	48	21	Rep Ventas	10-BLC-86	108	\$350,000.00	\$474,050.00
106	Sam Clark	52	11	VP Ventas	14-JUN-88	NULL	\$275,000.00	\$299,912.00
104	Bob Smith	33	12	Dir Ventas	19-MAY-87	106	\$200,000.00	\$142,594.00
101	Dan Roberts	45	12	Rep Ventas	20-OCT-86	104	\$300,000.00	\$305,673.00
110	Tom Snyder	41	NULL	Rep Ventas	13-ENE-90	101	NULL	\$75,985.00
108	Larry Fitch	62	21	Dir Ventas	12-OCT-89	106	\$350,000.00	\$361,865.00
103	Paul Cruz	29	12	Rep Ventas	01-MAR-87	104	\$275,000.00	\$286,775.00
107	Nancy Angelli	49	22	Rep Ventas	14-NOV-88	108	\$300,000.00	\$186,042.00

Figura 10.2. Inserción de una sola fila.

subsiguiente de la tabla REPVENTAS incluirá la nueva fila, pero puede aparecer en cualquier punto entre las filas de los resultados de la consulta.

Supongamos que el Sr. Jacobsen recibe ahora su primer pedido, de InterCorp, un nuevo cliente que tiene asignado el número de cliente 2126. El pedido es para 20 Widgets ACI-41004, por un precio total de \$2,340, y le ha sido asignado el número de pedido 113069. He aquí las sentencias INSERT que añaden el nuevo cliente y el pedido a la base de datos:

Insera un nuevo cliente y un nuevo pedido para el Sr. Jacobsen.

```
INSERT INTO
  CLIENTES (EMPRESA, NUM_CLIE, LIMITE_CREDITO, REP_CLIE)
  VALUES ('InterCorp', 2126, 15000.00, 111)
```

1 fila insertada.

```
INSERT INTO
  PEDIDOS (IMPORTE, FAB, PRODUCTO, CANT, FECHA_PEDIDO, NUM_PEDIDO, CLIE, REP)
  VALUES (2340.00, 'ACI', '41004', 20, CURRENT_DATE, 113069, 2126, 111)
```

1 fila insertada.

la palabra clave NULL en los valores. Esta sentencia INSERT tiene exactamente el mismo efecto que la anterior:

```
INSERT INTO REPVENTAS (NOMBRE, EDAD, NUM_EMPL, VENTAS, CUOTA, TITULO,
DIRECTOR, CONTRATO, OFICINA,REP)
VALUES ('Henry Jacobsen', 36, 111, 0.00, NULL, 'Dir Ventas',
NULL, '25-jul-90', 13)
```

Inserción de todas las columnas. Por conveniencia, SQL permite omitir la lista de columnas de la sentencia INSERT. Cuando se omite la lista de columnas, SQL genera automáticamente una lista formada por todas las columnas de la tabla, en secuencia de izquierda a derecha. Esta es la misma secuencia de columnas generadas por SQL cuando se utiliza una consulta SELECT *. Utilizando esta abreviatura, la sentencia INSERT anterior podría reescribirse de forma equivalente como:

```
INSERT INTO REPVENTAS
VALUES (111, 'Henry Jacobsen', 36, 13, 'Dir Ventas',
'25-jul-90', NULL, NULL, 0.00)
```

Cuando se omite la lista de columnas, la palabra clave NULL *debe* ser utilizada en la lista de valores para asignar explícitamente valores NULL a las columnas, tal como se muestra en el ejemplo. Además, la secuencia de valores de datos debe corresponder *exactamente* con la secuencia de columnas de la tabla.

La omisión de la lista de columnas es conveniente en SQL interactivo, ya que reduce la longitud de la sentencia INSERT que se debe escribir. Para SQL programado, la lista de columnas debería ser específica siempre, ya que esto hace el programa más fácil de leer y entender.

La sentencia INSERT *multifila*

La segunda forma de la sentencia INSERT, mostrada en la Figura 10.3, añade múltiples filas de datos a su tabla destino. En esta forma la sentencia INSERT, los valores de datos para las nuevas filas no son especificados explícitamente dentro del texto de la sentencia. En su lugar, la fuente de las nuevas filas es una consulta de base de datos, especificada en la sentencia.

La adición de filas cuyos valores provienen de la propia base de datos puede parecer extraña al principio, pero es muy útil en algunas situaciones especiales. Por ejemplo, supongamos que se desea copiar el número de pedido, la fecha y el importe de todos los pedidos remitidos con anterioridad al 1 de enero de 1990, a partir de la tabla PEDIDOS en otra tabla, llamada ANTIPEDIDOS. La sentencia INSERT multifila proporciona un modo eficiente y compacto de copiar los datos:

Copia pedidos antiguos en la tabla ANTIPEDIDOS.

```
INSERT INTO ANTIPEDIDOS (NUM_PEDIDO, FECHA_PEDIDO, IMPORTE)
SELECT NUM_PEDIDO, FECHA_PEDIDO, IMPORTE
FROM PEDIDOS
WHERE FECHA_PEDIDO < '01-ENE-90'
9 filas insertadas
```

La sentencia INSERT parece complicada, pero realmente es muy simple. La sentencia identifica la tabla que va a recibir las nuevas filas (ANTIPEDIDOS) y las columnas que reciben los datos, lo mismo que la sentencia INSERT de una sola fila. El resto de la sentencia es una consulta que recupera datos de la tabla PEDIDOS. La Figura 10.4 ilustra gráficamente la operación de esta sentencia INSERT. Conceptualmente, SQL efectúa primero la consulta sobre la tabla PEDIDOS, y luego inserta los resultados, fila a fila, en la tabla ANTIPEDIDOS.

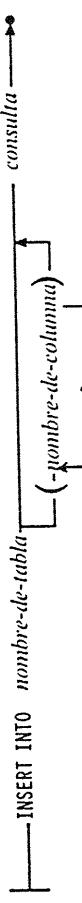


Figura 10.3. Diagrama sintáctico de la sentencia INSERT multifila.

He aquí otra situación en donde se podría utilizar la sentencia INSERT multifila. Supongamos que se desea analizar los patrones de compra de un cliente examinando qué clientes y qué vendedores son responsables de los grandes pedidos —aquellos que superan los \$15.000—. Las consultas que se realizarían combinarián datos de las tablas CLIENTES, REPVENTAS y PEDIDOS. Estas consultas de tres tablas se efectuarían bastante rápidamente en nuestra pequeña base de datos ejemplo, pero en una base de datos corporativa real con muchos miles de filas, llevaría mucho tiempo. En lugar de ejecutar muchas consultas de tres tablas largas, se podría crear una nueva tabla llamada GRANPEDIDOS para que contuviera los datos requeridos, definida del modo siguiente:

Columna	Descripción
IMPORTE	Importe del pedido (procedente de PEDIDOS)
EMPRESA	Nombre del cliente (procedente de CLIENTES)
NOMBRE	Nombre del vendedor (procedente de REPVENTAS)
REND	Importe superior o inferior a la cuota (calculado de REPVENTAS)
FAB	Id del fabricante (procedente de PEDIDOS)
PRODUCTO	Id de producto (procedente de PEDIDOS)
CANT	Cantidad pedido (procedente de PEDIDOS)

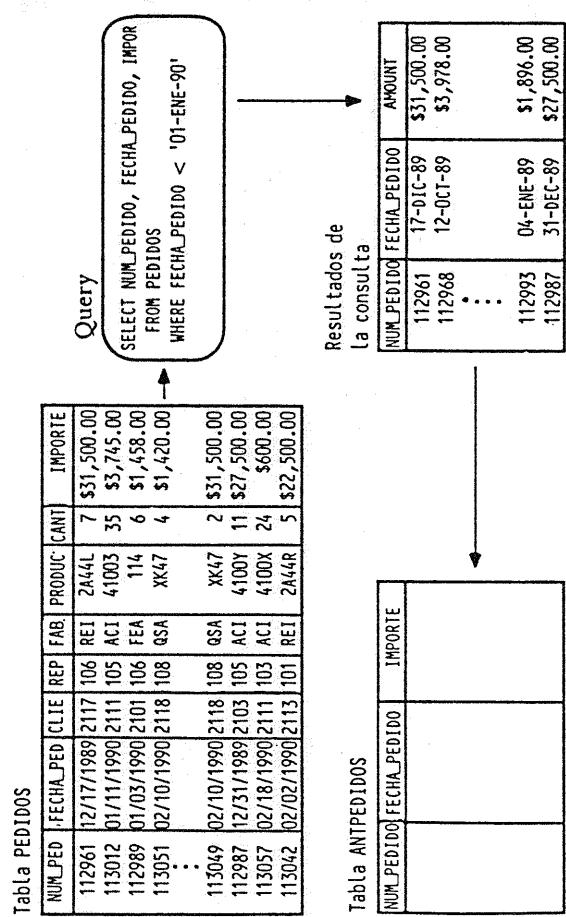


Figura 10.4. Inserción de múltiples filas.

Una vez que se ha creado la tabla GRANPEDIOS, esta sentencia INSERT puede ser utilizada para rellenarla:

Carga datos en la tabla GRANPEDIOS para análisis.

```
INSERT INTO GRANPEDIOS (IMPORTE, EMPRESA, NOMBRE, REND, PRODUCTO, DIR, CANT)
SELECT IMPORTE, EMPRESA, NOMBRE, (VENTAS - CUOTA), PRODUCTO, DIR, CANT
FROM PEDIDOS, CLIENTES, REPVENTAS
WHERE CLIE = NUM_CLIE
AND REP_NUM_EMPL
AND IMPORTE > 15000.00
```

6 filas insertadas.

En una base de datos de grandes dimensiones, esta sentencia INSERT puede tardar un buen rato en ejecutarse, ya que implica una consulta de tres tablas. Cuando la sentencia se complete, los datos de la tabla GRANPEDIOS contendrán información duplicada de otras tablas. Además, la tabla GRANPEDIOS no se mantendrá automáticamente actualizada cuando se añadan nuevos pedidos a la base de datos, por lo que sus datos pueden quedar rápidamente obsoletos. Cada uno de estos factores parece una desventaja. Sin embargo, las consultas de análisis subsiguientes que utilizan la tabla GRANPEDIOS pueden expresarse de

forma muy sencilla —pasan a ser consultas de una sola tabla—. Además, cada una de esas consultas se ejecutarán mucho más rápido que si fuera una composición de tres tablas. Consecuentemente, ésta es probablemente una buena estrategia para llevar a cabo el análisis, especialmente si las tres tablas originales son grandes.

El estándar SQL ANSI/ISO especifica varias restricciones sobre la consulta que aparece dentro de la sentencia INSERT multilínea:

- La consulta no puede contener una cláusula ORDER BY. No es útil ordenar los resultados de la consulta de ningún modo, ya que van a ser insertados en una tabla que es, como todas las tablas, desordenada.
- La consulta no puede ser la UNION de varias sentencias SELECT diferentes. Sólo puede especificarse una única sentencia SELECT.
- La tabla destino de la sentencia INSERT no puede aparecer en la cláusula FROM de la consulta o de ninguna subconsulta que ésta contenga. Esto prohíbe insertar parte de una tabla en sí misma.

Utilidades de carga masiva

Los datos a insertar en una base de datos son con frecuencia extraídos de otro sistema informático o recolectados de otros lugares y almacenados en un archivo secuencial. Para cargar los datos en una tabla, se podría escribir un programa con un bucle que leyera cada registro del archivo y utilizará la sentencia INSERT de una fila para añadir la fila a la tabla. Sin embargo, el recargo de hacer INSERT de forma repetidamente sentencias INSERT de una fila puede ser bastante alto. Si insertar una sola fila tarda medio segundo para una carga de sistema típica, éste es un rendimiento probablemente aceptable para un programa interactivo. Pero este rendimiento rápidamente pasa a ser inaceptable cuando se aplica a la tarea de cargar 50.000 filas de datos de una vez. En este caso, la carga de los datos requeriría más de seis horas.

Por esta razón, todos los productos DBMS comerciales incluyen una capacidad de carga masiva que carga los datos desde un archivo a una tabla a alta velocidad. El estándar SQL ANSI/ISO no considera esta función, y suele ser suministrada como un programa de utilidad autónomo en lugar de formar parte del lenguaje SQL. La utilidad que cada vendedor suministra dispone de un conjunto ligeramente diferente de características, funciones y órdenes.

Supresión de datos de la base de datos

Tipicamente, una fila de datos se suprime de una base de datos cuando la entidad representada por la fila «desaparece del mundo exterior». Por ejemplo, en la base de datos ejemplo:

- Cuando un cliente cancela un pedido, la correspondiente fila de la tabla PEDIDOS debe ser suprimida.
- Cuando un vendedor abandona la empresa, la fila correspondiente de la tabla REPVENTAS debe ser eliminada.
- Cuando una oficina de ventas se cierra, la fila correspondiente de la tabla OFICINAS debe ser cancelada. Si los vendedores de la oficina son despedidos, sus filas también deberían ser suprimidas de la tabla REPVENTAS. Si son reasignados, sus columnas OFICINA_REP deben ser actualizadas.

En cada caso, la fila se suprime para mantener la base de datos como un modelo preciso del mundo real. La unidad más pequeña de datos que puede ser suprimida de una base de datos relacional es una única fila.

La sentencia DELETE

La sentencia DELETE, mostrada en la Figura 10.5, elimina filas seleccionadas de datos de una única tabla. La cláusula FROM especifica la tabla destino que contiene las filas. La cláusula WHERE especifica qué filas de la tabla van a ser suprimidas.

Supongamos que Henry Jacobsen, el nuevo vendedor anteriormente contratado en este capítulo, acaba de decidir abandonar la empresa. He aquí la sentencia DELETE que elimina su fila de la tabla REPVENTAS.

Elimina a Henry Jacobsen de la base de datos.

```
DELETE FROM REPVENTAS
WHERE NOMBRE = 'Henry Jacobsen'
```

1 fila suprimida.

La cláusula WHERE de este ejemplo identifica una sola fila de la tabla REPVENTAS, que SQL elimina de la tabla. La cláusula WHERE tiene un aspecto familiar —es exactamente la misma cláusula WHERE que se especificaría en una sentencia SELECT para *recuperar la misma fila de la tabla*. Las condiciones de búsqueda que pueden especificarse en la cláusula WHERE de la sentencia DELETE son las mismas disponibles en la cláusula WHERE de la sentencia SELECT, descrita en los Capítulos 6 y 9.

Recuerde que las condiciones de búsqueda en la cláusula WHERE de una sentencia SELECT pueden especificar una sola fila o un conjunto entero de filas, dependiendo de la condición de búsqueda específica. Lo mismo es aplicable a la cláusula WHERE en una sentencia DELETE. Supongamos, por ejemplo, que el cliente del Sr. Jacobsen, InterCorp (número de cliente 2126), ha llamado para cancelar todos sus pedidos. He aquí la sentencia de supresión que elimina los pedidos de la tabla PEDIDOS:

Elimina todos los pedidos de InterCorp (número de cliente 2126).

```
DELETE FROM PEDIDOS
WHERE CLIE = 2126
```

2 filas suprimidas

En este caso, la cláusula WHERE selecciona varias filas de la tabla PEDIDOS, y SQL elimina todas las filas seleccionadas de la tabla. Conceptualmente, SQL aplica la cláusula WHERE a cada una de las filas de la tabla PEDIDOS, suprimiendo aquéllas para las cuales la condición de búsqueda produce un resultado TRUE y reteniendo aquéllas para las cuales la condición de búsqueda produce un resultado FALSE o NULL. Puesto que este tipo de sentencia DELETE busca a través de la tabla las filas a suprimir, a veces se le denomina sentencia DELETE *buscada*. Este término se utiliza para contrastarlo con otra forma de la sentencia DELETE, llamada sentencia DELETE *posicionada*, que siempre suprime una única fila. La sentencia DELETE posicionada se aplica únicamente a SQL programado, y se describe en el Capítulo 17.

He aquí algunos ejemplos adicionales de sentencias DELETE buscadas:

Suprime todos los pedidos remitidos antes del 15 de noviembre de 1989.

```
DELETE FROM PEDIDOS
WHERE FECHA_PEDIDO < '15-NOV-89'
```

5 filas suprimidas.

Suprime todas las filas correspondientes a los clientes atendidos por Bill Adams, Mary Jones o Dan Roberts (números de empleado 105, 109 y 101).

```
DELETE FROM CLIENTES
WHERE REP_OLE IN (105, 109, 101)
```

7 filas suprimidas.



Figura 10.5. Diagrama sintáctico de la sentencia DELETE.

Suprime todos los vendedores contratados antes de julio de 1988 a los que aún no se les ha asignado una nota.

```
DELETE FROM REPVENTAS
WHERE CONTRATO < '30-JUN-88'
AND CUOTA IS NULL
```

0 filas suprimidas.

Supresión de todas las filas

La cláusula WHERE en una sentencia DELETE es opcional, pero casi siempre está presente. Si se omite la cláusula WHERE de una sentencia DELETE, se suprimen *todas* las filas de la tabla destino, como en este ejemplo:

Suprime todos los pedidos.

```
DELETE FROM PEDIDOS
```

30 filas suprimidas.

Aunque esta sentencia DELETE produce una tabla vacía, no borra la tabla PEDIDOS de la base de datos. La definición de la tabla PEDIDOS y sus columnas siguen estando almacenadas en la base de datos. La tabla aún existe, y nuevas filas pueden ser insertadas en la tabla PEDIDOS con la sentencia INSERT. Para cancelar la definición de la tabla de la base de datos, debe utilizarse la sentencia DROP TABLE (descripción en el Capítulo 13).

Debido al daño potencial que puede producir una sentencia DELETE tal como ésta, es importante especificar siempre una condición de búsqueda y tener cuidado de que se seleccionan realmente las filas que se desean. Cuando se utiliza SQL interactivo, es buena idea utilizar primero la cláusula WHERE en una sentencia SELECT para visualizar las filas seleccionadas, asegurarse de que son las que realmente se desea suprimir, y sólo entonces utilizar la cláusula WHERE en una sentencia DELETE.

DELETE con subconsulta *

Las sentencias DELETE con condiciones de búsqueda simples, como las de los ejemplos anteriores, seleccionan las filas a suprimir basándose únicamente en los propios contenidos de las filas. A veces, la selección de las filas debe efectuarse en base a datos contenidos en otras tablas. Por ejemplo, supongamos que se desea suprimir todos los pedidos aceptados por Sue Smith. Sin saber su número de empleado, no se pueden determinar los pedidos consultando únicamente la

tabla PEDIDOS. Para hallar los pedidos, podría utilizarse una consulta de dos tablas:

Halla los pedidos aceptados por Sue Smith.

```
SELECT NUM_PEDIDO, IMPORTE
FROM PEDIDOS, REPVENTAS
WHERE REP = NUM_EMPL
AND NOMBRE = 'Sue Smith'
```

NUM_PEDIDO	IMPORTE
112979	\$15,000.00
113065	\$2,130.00
112993	\$1,896.00
113048	\$3,750.00

Pero no se puede utilizar una composición en una sentencia DELETE. La sentencia DELETE paralela es ilegal:

```
DELETE FROM PEDIDOS, REPVENTAS
WHERE REP = NUM_EMPL
AND NOMBRE = 'Sue Smith'
```

Error: Más de una tabla especificada en la cláusula FROM.

El modo de manejar la petición es con una de las condiciones de búsqueda subconsulta. He aquí una forma válida de la sentencia DELETE que realiza la petición:

Suprime los pedidos aceptados por Sue Smith.

```
DELETE FROM PEDIDOS
WHERE REP = (SELECT NUM_EMPL
FROM REPVENTAS
WHERE NOMBRE = 'Sue Smith')
```

4 filas suprimidas.

La consulta halla el número de empleado de Sue Smith, y la cláusula WHERE selecciona entonces los pedidos con un valor correspondiente. Como muestra este ejemplo, las subconsultas pueden jugar un papel importante en la sentencia DELETE, ya que permiten suprimir filas en base a información contenida en otras tablas. He aquí otros dos ejemplos de sentencias DELETE que utilizan condiciones de búsqueda subconsulta:

Suprime los clientes atendidos por vendedores cuyas ventas son inferiores al 80 por 100 de su cuota.

```
DELETE FROM CLIENTES
WHERE REP_CLIE IN (SELECT NUM_EMPL
                     FROM REVENTAS
                   WHERE VENTAS < (.8 * CUOTA))
```

2 filas suprimidas.

Suprime los vendedores cuyo total de pedidos actual es menor que el 2 por 100 de sus cuotas.

```
DELETE FROM REVENTAS
WHERE (.02 * CUOTA) < (SELECT SUM(IMPORTE)
                           FROM PEDIDOS
                         WHERE REP = NUM_EMPL)
```

1 fila suprimida.

Las subconsultas en la cláusula WHERE pueden anidarse al igual que sucedía en la cláusula WHERE de la sentencia SELECT. También pueden contener referencias externas a la tabla destino de la sentencia DELETE. A este respecto, la cláusula FROM de la sentencia DELETE funciona igual que la cláusula FROM de la sentencia SELECT. He aquí un ejemplo de una petición de supresión que requiere una subconsulta con una referencia externa:

Suprime los clientes que no han realizado pedidos desde el 10 de noviembre de 1989.

```
DELETE FROM CLIENTES
WHERE NOT EXISTS (SELECT *
                     FROM PEDIDOS
                   WHERE CLIE = NUM_CLIE
                     AND FECHA_PEDIDO < '10-Nov-89')
```

16 filas suprimidas.

Conceptualmente, esta sentencia DELETE opera recorriendo la tabla CLIENTES, fila a fila, y comprobando la condición de búsqueda. Para cada cliente, la subconsulta selecciona los pedidos remitidos por ese cliente antes de la fecha de corte. La referencia a la columna NUM_CLI en la subconsulta es una referencia externa al número de cliente en la fila de la tabla CLIENTES que actualmente está siendo comprobada por la sentencia DELETE. La subconsulta de este ejemplo es una subconsulta correlacionada, tal como se describió en el Capítulo 9.

Las referencias externas aparecerán con frecuencia en las subconsultas de una sentencia DELETE, ya que implementan la «composición» entre la(s) tabla(s) de la subconsulta y la tabla destino de la sentencia DELETE. La única restricción al uso de subconsultas en una sentencia DELETE es que la tabla destino no puede aparecer en la cláusula FROM de una subconsulta ni en ninguna de sus subconsultas a ningún nivel de anidación. Esto impide que las subconsultas referencien la tabla destino (algunas de cuyas filas pueden ya haber sido suprimidas), excepto para referencias externas a la fila actualmente en comprobación por la condición de búsqueda de la sentencia DELETE.

Modificación de datos en la base de datos

Típicamente, los valores de los datos almacenados en una base de datos se modifican cuando se producen cambios correspondientes en el mundo exterior. Por ejemplo, en la base de datos ejemplo:

- Cuando un cliente llama para modificar la cantidad de un pedido, la columna CANT de la fila apropiada en la tabla PEDIDOS debe ser modificada.
- Cuando un director se traslada de una oficina a otra, la columna DIR en la tabla OFICINAS y la columna OFICINA REP en la tabla REVENTAS deben ser modificadas para que reflejen la nueva asignación.
- Cuando las cuotas de ventas se elevan un 5 por 100 en la oficina de ventas de New York, la columna CUOTA de las filas apropiadas en la tabla REVENTAS debe ser modificada.

En cada caso, los valores de los datos en la base de datos se actualizan para mantener la base de datos como un modelo preciso del mundo real. La unidad más pequeña de datos que puede modificarse en una base de datos es una única columna de una única fila.

La sentencia UPDATE

La sentencia UPDATE, mostrada en la Figura 10.6, modifica los valores de una o más columnas en las filas seleccionadas de una tabla única. La tabla destino a

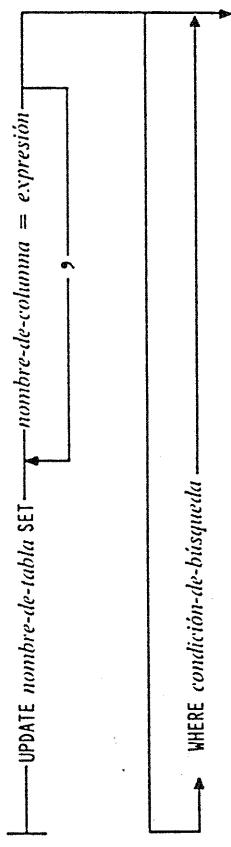


Figura 10.6. Diagrama sintáctico de la sentencia UPDATE.

actualizar se indica en la sentencia, y es necesario disponer de permiso para actualizar la tabla además de cada una de las columnas individuales que serán modificadas. La cláusula WHERE selecciona las filas de la tabla a modificar. La cláusula SET especifica qué columnas se van a actualizar y calcula los nuevos valores.

He aquí una sentencia UPDATE sencilla que modifica el límite de crédito y el vendedor asignado a un cliente:

Eleva el límite de crédito de la empresa Acme Manufacturing a \$60.000 y la reasigna a Mary Jones (número de empleado 109).

```
UPDATE CLIENTES
SET LIMITE_CREDITO = 60000.00, REP_CLIE = 109
WHERE EMPRESA = 'Acme Mfg.'
```

1 fila actualizada.

En este ejemplo, la cláusula WHERE identifica una sola fila de la tabla CLIENTES, y la cláusula SET asigna nuevos valores a dos de las columnas de esta fila. La cláusula WHERE es exactamente la misma que se utilizaría en una sentencia DELETE o SELECT para identificar la fila. De hecho, las condiciones de búsqueda que pueden aparecer en la cláusula WHERE de una sentencia UPDATE son exactamente las mismas que las disponibles en las sentencias SELECT y DELETE.

Al igual que la sentencia DELETE, la sentencia UPDATE puede actualizar varias filas de una vez con la condición de búsqueda adecuada, como en este ejemplo:

Transfiere todos los vendedores de la oficina de Chicago (número 12) a la oficina de New York (número 11), y rebaja sus cuotas un 10 por 100.

```
UPDATE REPVENTAS
SET OFICINA.REP = 11, CUOTA = .9 * CUOTA
WHERE OFICINA.REP = 12
```

3 filas actualizadas.

En este caso, la cláusula WHERE selecciona varias filas de la tabla REPVENTAS, y el valor de las columnas OFICINA.REP y CUOTA se modifica en todas ellas. Conceptualmente, SQL procesa la sentencia UPDATE al recorrer la tabla REPVENTAS fila a fila, actualizando aquellas filas para las cuales la condición de búsqueda produce un resultado TRUE y omitiendo aquéllas para las cuales la condición de búsqueda produce un resultado FALSE o NULL. Puesto que realiza búsquedas en la tabla, esta forma de la sentencia UPDATE se denomina a veces sentencia UPDATE buscada. Este término la distingue de una forma diferente de sentencia UPDATE, llamada sentencia UPDATE posicionada, que siempre actualiza una única fila. La sentencia UPDATE posicionada se aplica únicamente a SQL programado, y se describe en el Capítulo 17.

He aquí algunos ejemplos adicionales de sentencias UPDATE buscadas:

Reasigna todos los clientes atendidos por los empleados número 105, 106 ó 107 al empleado número 102.

```
UPDATE CLIENTES
SET REP_CLIE = 102
WHERE REP_CLIE IN (105, 106, 107)
```

5 filas actualizadas.

Asigna una cuota de \$100.000 a todos aquellos vendedores que actualmente no tienen cuota.

```
UPDATE REPVENTAS
SET CUOTA = 100000.00
WHERE CUOTA IS NULL
```

1 fila actualizada.

La cláusula SET en la sentencia UPDATE es una lista de asignaciones separadas por comas. Cada asignación identifica una columna destino a actualizar y especifica cómo calcular el nuevo valor para la columna destino. Cada columna destino debería aparecer solamente una vez en la lista; no debería haber dos asignaciones para la misma columna destino. La especificación ANSI/ISO exige nombres sin cualificar para las columnas destino, pero algunas implementaciones SQL permiten la implementación de nombres de columna cualificados. No puede existir ambigüedad alguna en los nombres de columna, ya que deben referirse a columnas de la tabla destino.

La expresión en cada asignación puede ser cualquier expresión SQL que genere un valor del tipo de dato apropiado para la columna destino. La expresión debe ser calculable basada en los valores de la fila actualmente en actualización en la tabla destino. No pueden incluirse funciones de columna ni subconsultas.

Si una expresión en la lista de asignación referencia a una de las consultas de la tabla destino, el valor utilizado para calcular la expresión es el valor de esa columna en la fila actual *antes* de que se aplique ninguna actualización. Lo mismo es aplicable a las referencias de columna que se producen en la cláusula WHERE. Por ejemplo, consideremos esta sentencia UPDATE (ciertamente artificiosa):

```
UPDATE OFICINAS
SET CUOTA = 400000.00, VENTAS = CUOTA
WHERE CUOTA < 400000.00
```

Antes de la actualización, Bill Adams tenía un valor de CUOTA de \$350.000 y un valor de VENTAS de \$367.911. Después de la actualización, su fila tiene un valor de VENTAS de \$350.000, y *no* de \$400.000. El orden de las asignaciones en la cláusula

SET es por tanto inmaterial; las asignaciones pueden especificarse en cualquier orden.

Actualización de todas las filas

La cláusula WHERE en la sentencia UPDATE es opcional. Si se omite la cláusula WHERE, entonces se actualizan *todas* las filas de la tabla destino, como en este ejemplo:

Elevar todas las cuotas un 5 por 100.

```
UPDATE REPVENTAS
SET CUOTA = 1.05 * CUOTA
10 filas actualizadas.
```

A diferencia de la sentencia DELETE, en donde la cláusula WHERE casi nunca se omite, la sentencia UPDATE sin una cláusula WHERE realiza una función útil. Básicamente efectúa una actualización masiva de toda la tabla, como demuestra el ejemplo anterior.

UPDATE con subconsulta *

Al igual que con la sentencia DELETE, las subconsultas pueden jugar un papel importante en la sentencia UPDATE ya que permiten seleccionar filas a actualizar en base a información contenida en otras tablas. He aquí varios ejemplos de sentencias UPDATE que utilizan subconsultas:

Elevar en \$5.000 el límite de crédito de cualquier cliente que haya remitido una orden de más de \$25.000.

```
UPDATE CLIENTES
SET LIMITE_CREDITO = LIMITE_CREDITO + 5000.00
WHERE NUM_CLIE IN (SELECT DISTINCT CLIE
FROM PEDIDOS
WHERE IMPORTE > 25000.00)
4 filas actualizadas.
```

Algunos todos los clientes atendidos por vendedores cuyas ventas son menores al 80 por 100 de sus cuotas.

```
UPDATE CLIENTES
SET REP_CLIE = 105
WHERE REP_CLIE IN (SELECT NUMENPL
FROM REPVENTAS
WHERE VENTAS < (1.8 * CUOTA))
2 filas actualizadas.
```

Hace que todos los vendedores que atienden a más de tres clientes estén bajo las órdenes de Sam Clark (empleado número 106).

```
UPDATE REPVENTAS
SET DIRECTOR = 106
WHERE 3 < (SELECT IMPORTE(*)
FROM CLIENTES
WHERE REP_CLIE = NUMENPL)
```

1 fila actualizada.

Como en la sentencia SELECT, las subconsultas en la cláusula WHERE de la sentencia UPDATE pueden anidarse a cualquier nivel, y pueden contener referencias externas a la tabla destino de la sentencia UPDATE. La columna NUM_ENPL en la subconsulta del ejemplo anterior es una referencia externa; se refiere a la columna NUM_EMPL en la fila de la tabla REPVENTAS actualmente comprobada por la sentencia UPDATE. La subconsulta de este ejemplo es una subconsulta corregida, como se describió en el Capítulo 9.

Las referencias externas suelen aparecer en subconsultas de una sentencia UPDATE, ya que implementan la «composición» entre la(s) tabla(s) en la subconsulta y la tabla destino de la sentencia UPDATE. La misma restricción que se aplicaba a la sentencia DELETE es aplicable aquí: la tabla destino no puede aparecer en la cláusula FROM de ninguna subconsulta a ningún nivel de anidación. Esto impide que las subconsultas referencien la tabla destino (algunas de cuyas filas pueden ya haber sido actualizadas). Cualesquier referencias a la tabla destino en las subconsultas son por tanto referencias externas a la fila de la tabla destino que actualmente está siendo comprobada por la cláusula WHERE de la sentencia UPDATE.

Resumen

Este capítulo ha descrito las sentencias SQL que se emplean para modificar los contenidos de una base de datos:

- La sentencia INSERT de una fila añade una fila de datos a una tabla. Los valores para la nueva fila se especifican en la sentencia como constantes.
- La sentencia INSERT multifila añade cero o más filas a una tabla. Los valores para las nuevas filas provienen de una consulta, especificada como parte de la sentencia INSERT.
- La sentencia DELETE suprime cero o más filas de datos de una tabla. Las filas a suprimir son especificadas mediante una condición de búsqueda.

- La sentencia UPDATE modifica los valores de una o más columnas en cero o más filas de una tabla. Las filas a actualizar son especificadas mediante una condición de búsqueda. Las columnas a actualizar, y las expresiones que calculan sus nuevos valores, se especifican en la sentencia UPDATE.

- A diferencia de la sentencia SELECT, que puede operar sobre múltiples tablas, las sentencias INSERT, DELETE y UPDATE funcionan solamente sobre una única tabla cada vez.

- La condición de búsqueda utilizada en las sentencias DELETE y UPDATE tiene la misma forma que la condición de búsqueda de la sentencia SELECT.

11 *Integridad de datos*

- El término *integridad de datos* se refiere a la corrección y completitud de los datos en una base de datos. Cuando los contenidos de una base de datos se modifican con sentencias INSERT, DELETE o UPDATE, la integridad de los datos almacenados puede perderse de muchas maneras diferentes. Por ejemplo:
- Pueden añadirse datos no válidos a la base de datos, tales como un pedido que especifica un producto no existente.
 - Pueden modificarse datos existentes tomando un valor incorrecto, como por ejemplo si se reasigna un vendedor a una oficina no existente.
 - Los cambios a la base de datos pueden perderse debido a un error del sistema o a un fallo en el suministro de potencia.
 - Los cambios pueden ser aplicados parcialmente, como por ejemplo si se añade un pedido de un producto sin ajustar la cantidad disponible para vender.
- Una de las funciones importantes de un DBMS relacional es preservar la integridad de sus datos almacenados en la mayor medida posible. Este capítulo describe las características del lenguaje SQL que ayudan al DBMS en esta tarea.

¿Qué es la integridad de datos?

Para preservar la consistencia y corrección de los datos almacenados, un DBMS relacional impone típicamente una o más *restricciones de integridad de datos*. Estas restricciones restringen los valores que pueden ser insertados en la base de

datos o creados mediante una actualización de la base de datos. Varios tipos diferentes de restricciones de integridad de datos suelen encontrarse en las bases de datos relacionales, incluyendo:

■ **Datos requeridos.** Algunas columnas en una base de datos deben contener un valor de dato válido en cada fila; no se permite que contengan valores NULL o que falte. En la base de datos ejemplo, cada pedido debe tener un cliente asociado que permita el pedido. Por tanto la columna CLIE en la tabla de PEDIDOS es una *columna requerida*. El DBMS tendría que impedir la existencia de valores NULL en esta columna.

■ **Chequeo de validez.** Cada columna de una base de datos tiene un *dominio*, un conjunto de valores que son legales para esa columna. La base de datos ejemplo utiliza números de pedidos que comienzan en 100001, por lo que el dominio de la columna NUM_PEDIDO es el de los enteros positivos superiores a 100.000. Analogamente, los números de empleado en la columna NUM_EMPL deben estar dentro del rango numérico entre 101 y 999. El DBMS puede ser preparado para impedir otros valores de datos en estas columnas.

■ **Integridad de entidad.** La clave primaria de una tabla debe contener un valor único en cada fila, diferente de los valores de todas las filas restantes. Por ejemplo, cada fila de la tabla PRODUCTOS tiene un conjunto único de valores en sus columnas ID_FAB e ID_PRODUCTO, que identifica únicamente el producto representado por esa fila. Los valores duplicados son ilegales, ya que no permitirían a la base de datos distinguir un producto de otro. El DBMS puede ser preparado para forzar esta restricción de valores únicos.

■ **Integridad referencial.** Una clave foránea en una base de datos relacional enlaza cada fila de la tabla hijo que contiene la clave foránea con la fila de la tabla padre que contiene el valor de clave primaria correspondiente. En la base de datos ejemplo, el valor en la columna REP_OFICINA de cada fila en REVENTAS enlaza el vendedor representado por esa fila a la oficina en donde trabaja. La columna REP_OFICINA *debe* contener un valor válido de la columna OFICINA de la tabla OFICINAS o en caso contrario el vendedor tendría asignada una oficina inválida. El DBMS puede ser preparado para forzar esta restricción de clave foránea/clave primaria.

■ **Reglas comerciales.** Las actualizaciones de una base de datos pueden estar restringidas por reglas comerciales que gobernan las transacciones en el mundo real que están representadas por las actualizaciones. Por ejemplo, la empresa que utiliza la base de datos ejemplo puede tener una regla comercial que prohíba aceptar un pedido para el cual exista un producto inadecuado en el inventario. El DBMS puede ser preparado para comprobar cada nueva fila añadida a la tabla PEDIDOS asegurándose que el valor en la columna CANT no viole esta regla comercial.

■ **Consistencia.** Muchas transacciones del mundo real producen múltiples actualizaciones a una base de datos. Por ejemplo, la aceptación de un pedido de un cliente puede implicar añadir una fila a la tabla PEDIDOS, incrementar la columna VENTAS en la tabla REVENTAS para la persona que aceptó el pedido e incrementar la columna VENTAS en la tabla OFICINAS correspondiente a la oficina a la cual el vendedor está asignado. Una sentencia INSERT y dos UPDATEs deben *todos* tener lugar para que la base de datos continúe en un estado correcto y consistente. El DBMS puede ser preparado para forzar este tipo de regla de consistencia o para soportar aplicaciones que implementen tales reglas.

El estándar SQL ANSI/ISO especifica algunas de las restricciones de integridad de datos más simples. Por ejemplo, la restricción de datos requeridos es soportada por el estándar ANSI/ISO e implementada de manera uniforme en casi todos los productos SQL comerciales. Las restricciones más complejas, tales como las restricciones de reglas comerciales, no están especificadas por el estándar ANSI/ISO, y existe una amplia variación en las técnicas y las sintaxis SQL utilizadas para soportarlas. Las características SQL que soportan las cinco primeras restricciones de integridad se describen en este capítulo. El mecanismo de transacción SQL, que soporta la restricción de consistencia, se describe en el Capítulo 12.

Datos requeridos

La restricción de integridad de datos más simple requiere que una columna tenga un valor no NULL. El estándar ANSI/ISO y la mayoría de los productos SQL comerciales soportan esta restricción permitiendo la declaración de que una columna es NOT NULL cuando la tabla que contiene la columna se crea por primera vez. La restricción NOT NULL se especifica como parte de la sentencia CREATE TABLE, descrita en el Capítulo 13.

Cuando una columna se declara NOT NULL, el DBMS fuerza la restricción asegurando lo siguiente:

- La sentencia INSERT que añade una nueva fila a la tabla debe especificar un valor de datos no NULL para la columna. Un intento de insertar una fila que contenga un valor NULL (explícita o implícitamente) da lugar a un error.
- La sentencia UPDATE que actualiza la columna debe asignarle un valor de dato no NULL. Igualmente, un intento de actualizar la columna con un valor NULL da lugar a un error.

Una desventaja de la restricción NOT NULL es que generalmente debe ser especificada cuando la tabla se crea por primera vez. Tipicamente, no se puede volver atrás una tabla previamente creada y desaprobar los valores NULL para una columna. Generalmente esta desventaja no es seria, puesto que es obvio cuando la tabla se crea por primera vez qué columnas deberían tener permitidos valores NULL y cuáles no.

La incapacidad de añadir una restricción NOT NULL a una tabla existente es el resultado del método que la mayoría de los productos DBMS utilizan para implementar los valores NULL internamente. Generalmente un DBMS reserva un byte adicional por cada fila almacenada de datos para cada columna que permita valores NULL. El byte adicional sirve como «indicador nulo» para la columna y contiene un cierto valor especificado que indica valor NULL. Cuando una columna se define como NOT NULL, el byte indicador no está presente, ahorrando espacio de almacenamiento en disco. La adición y supresión dinámica de restricciones NOT NULL requeriría por tanto reconfiguración «sobre la marcha» de las filas almacenadas en el disco, lo cual no es práctico en una base de datos de grandes dimensiones.

Comprobación de validez

El estándar SQL ANSI/ISO proporciona soporte limitado para restringir los valores legales que pueden aparecer en una columna. Cuando se crea una tabla, cada columna de la tabla tiene asignado un tipo de datos, y el DBMS asegura que únicamente datos del tipo especificado sean introducidos en la columna. Por ejemplo, la columna NUM_EMPL en la tabla REPVENTAS está definida como INTEGER, y el DBMS produciría un error si una sentencia INSERT o UPDATE intenta almacenar una cadena de caracteres o un número decimal en la columna.

Sin embargo, el estándar ANSI/ISO y la mayoría de los productos SQL comerciales no proporcionan una manera de restringir una columna a ciertos valores de datos específicos. El DBMS insertará felizmente una fila REPVENTAS con un número de empleado 12.345, aun cuando los números de empleado de la base de datos ejemplo tienen tres dígitos por convenio. Una fecha de contrato 25 de diciembre también sería aceptada, aun cuando la empresa está cerrada el día de Navidad.

Algunas implementaciones SQL comerciales proporcionan características extendidas para comprobar valores de datos legales. En DB2, por ejemplo, cada tabla de la base de datos puede tener asignado un *procedimiento de validación correspondiente*, un programa escrito por usuario para checar la validez de los valores de datos. DB2 invoca el procedimiento de validación cada vez que una sentencia SQL intenta cambiar o insertar una fila de la tabla, y da al procedimiento de validación los valores de columna «propuestos» para la fila. El

procedimiento de validación comprueba los datos e indica mediante un valor de retorno si los datos son aceptables. El procedimiento de validación es un programa convencional (escrito en ensamblador S/370/ o PL/I, por ejemplo), por lo que puede efectuar cualquier comprobación de valores que se requiera, incluyendo chequeos de rango y chequeos de consistencia interna dentro de la fila. Sin embargo, el procedimiento de validación *no puede* acceder a la base de datos, por lo que no puede ser utilizado para comprobar valores únicos o relaciones clave foránea/clave primaria.

SQL Server también proporciona una capacidad de validación de datos al permitir crear una *regla* que determine qué datos pueden ser introducidos en una columna particular. SQL Server comprueba la regla cada vez que se intenta una sentencia INSERT o UPDATE para la tabla que contiene la columna. A diferencia de los procedimientos de validación de DB2, las reglas de SQL Server están escritas en el dialecto Transact-SQL utilizado por SQL Server. Por ejemplo, he aquí una sentencia Transact-SQL que establece una regla para la columna CUOTA en la tabla REPVENTAS:

```
CREATE RULE LIMITE_CUOTA
AS ((@)VALUE BETWEEN 0.00 AND 500000.00)
```

Esta regla impide insertar o actualizar una cuota con un valor negativo o un valor superior a \$500.000. Como se muestra en el ejemplo, SQL Server permite asignar un nombre a la regla («LIMITE_CUOTA» en este ejemplo). Al igual que los procedimientos de validación de DB2, sin embargo, las reglas SQL Server no pueden referenciar columnas u otros objetos de la base de datos.

Varios productos DBMS proporcionan comprobación de validez como parte de sus entradas de datos o paquetes formularios en lugar de soportarla dentro del lenguaje SQL. Por ejemplo, el paquete de entrada de datos puede permitir especificar un rango de valores legales que pueden ser introducidos en un campo de un formulario de entrada de datos. Algunos productos permiten especificar que un valor de dato introducido tiene que ser chequeado con los valores de la base de datos, permitiendo cheques de validez que no pueden ser implementados con el planteamiento de DB2 o de SQL Server.

Integridad de entidad

La clave primaria de una tabla debe tener un valor único para cada fila de la tabla o si no la base de datos perderá su integridad como modelo del mundo exterior. Por ejemplo, si dos filas de la tabla REPVENTAS tienen ambas el valor 106 en la columna NUM_EMPL, sería imposible decir qué fila representa realmente a la entidad del mundo real asociada con ese valor clave —Bill Adams, cuyo número

vendedor pueda ser reasignado a una oficina diferente, esa oficina debe ya existir en la tabla OFICINAS. De nuevo, esta restricción tiene sentido en la base de datos ejemplo.

El cuarto problema (UPDATE de la clave primaria en la tabla padre) también se maneja por prohibición. Antes que la clave primaria pueda ser modificada, el DBMS efectúa comprobaciones para asegurarse que no haya filas hijo que tengan valores de clave foránea correspondientes. Si hay tales filas hijo, la sentencia UPDATE se rechaza junto con un mensaje de error. En la Figura 11.1 esto significa que un número de oficina no puede ser modificado en la tabla OFICINAS a menos que no haya vendedores en la tabla REPVENTAS actualmente asignados a esa oficina. En la práctica, esta restricción no suele provocar problemas, ya que los valores de clave primaria casi nunca se modifican.

El tercer problema (DELETE de una fila padre) es más complejo. Por ejemplo, supongamos que se cierra la oficina de Los Angeles y que se desea suprimir la fila correspondiente de la tabla OFICINAS en la Figura 11.1. ¿Qué le sucedería a las dos filas hijo en la tabla REPVENTAS que representan a los vendedores asignados a la oficina de Los Angeles? Dependiendo de la situación, se podría desechar:

- impedir que la oficina fuera suprimida hasta que los vendedores fueran reasignados,
 - suprimir automáticamente los dos vendedores de la tabla REPVENTAS también,
 - poner la columna OFICINA.REP de los dos vendedores a NULL, indicando que la asignación de oficina es desconocida.
- El estándar ANSI/ISO solamente proporciona la primera opción. En efecto, maneja este problema de integridad referencial exactamente del mismo modo que hizo con los otros tres —mediante prohibición—. Sin embargo, DB2 proporciona las tres opciones a través de su concepto de *reglas de supresión*.

ra 11.1, esta regla puede resumirse como «No se puede suprimir una oficina si tiene vendedores asignados».

■ La regla de supresión CASCADE dice al DBMS que cuando una fila padre sea suprimida, todas sus filas hijo también deberían ser suprimidas automáticamente de la tabla hijo. Las supresiones de la tabla padre «se propagan» por tanto a la tabla hijo. Para la Figura 11.1 esta regla puede resumirse como «La supresión de una oficina suprime automáticamente todos los vendedores asignados a esa oficina».

■ La regla de supresión SET NULL dice al DBMS que cuando una fila padre sea suprimida, los valores de clave foránea en todas las filas hijo deben automáticamente pasarse a NULL. Las supresiones de la tabla padre provocan por tanto una actualización de «pasó a NULL» en las columnas seleccionadas de la tabla hijo. Para las tablas de la Figura 11.1 esta regla puede resumirse como «Si una oficina se suprime, indíquese que la asignación de oficina actual de sus vendedores es desconocida».

Si no se especifica una regla de supresión, la regla RESTRICT es el valor por omisión, ya que tiene el menor potencial de destrucción accidental de datos. Cada una de las reglas de supresión es adecuada a situaciones diferentes. Generalmente, el comportamiento del mundo real modelado por la base de datos indicará qué regla de supresión es la apropiada. En la base de datos ejemplo, la tabla PEDIDOS contiene tres relaciones clave foránea/clave primaria, tal como se muestra en la Figura 11.2. Estas tres relaciones enlazan cada pedido a, (a) el producto que fue ordenado, (b) el cliente que solicitó el pedido y (c) el vendedor que aceptó el pedido. Una regla de supresión diferente parece apropiada a cada una de estas relaciones:

- La relación entre un pedido y el producto ordenado debería utilizar probablemente la regla RESTRICT. No debería ser posible eliminar pedidos actuales para ese producto.
- La relación entre un pedido y el cliente que lo ordenó debería probablemente utilizar la regla CASCADE. Probablemente sólo se suprimirá una fila de cliente de la base de datos si el cliente está inactivo o finiquita su relación con la empresa. En este caso, al suprimir el cliente, también deberían suprimirse los pedidos actuales correspondientes a ese cliente.
- La relación entre un pedido y el vendedor que lo aceptó debería probablemente utilizar la regla SET NULL. Si el vendedor abandona la empresa, los pedidos aceptados por ese vendedor pasarán a ser responsabilidad de un «vendedor desconocido» hasta que sean reasignados.

Reglas de supresión

Por cada relación padre/hijo creada mediante una clave foránea en una base de datos DB2, se puede especificar una regla de supresión asociada. La regla de supresión dice a DB2 qué debe hacer cuando un usuario trate de suprimir una fila de la tabla padre. Se pueden especificar una de tres reglas de supresión posibles:

- La regla de supresión RESTRICT impide suprimir una fila de la tabla padre si la fila tiene algún hijo. Una sentencia DELETE que intente suprimir una fila padre tal como esa es rechazada junto con un mensaje de error. Las supresiones de la tabla padre están por tanto restringidas a filas sin hijos. Aplicada a la Figura

Desgraciadamente, la primera sentencia INSERT (para Ben Adams) fallará. ¿Por qué? Porque la nueva fila se refiere a la oficina número 14, la cual aún no existe en la base de datos. Naturalmente el invertir el orden de las sentencias INSERT no arregla nada:

```
INSERT INTO OFICINAS (OFICINA, CIUDAD, REGION, DIR, OBJETIVO, VENTAS)
VALUES (14, 'Detroit', 'Eastern', 115, 0.00, 0.00)

INSERT INTO REPVENTAS (NUM_EMPL, NOMBRE, OFICINA.REP, CONTRATO, VENTAS)
VALUES (115, 'Ben Adams', 14, '01-ABR-90', 0.00)
```

La primera sentencia INSERT (para Detroit esta vez) seguirá fallando, ya que la nueva fila se refiere al empleado número 115 como director de la oficina, y Ben Adams aún no existe en la base de datos. Para impedir este «interbloqueo de inserción» al menos una de las claves foráneas en un ciclo referencial debe permitir valores NULL. En la definición efectiva de la base de datos ejemplo, la columna DIR no permite NULL, pero la columna OFICINA.REP sí lo permite. La inserción de dos filas puede entonces llevarse a cabo con dos INSERTS y un UPDATE, como se muestra aquí:

```
INSERT INTO REPVENTAS (NUM_EMPL, NOMBRE, OFICINA.REP, CONTRATO, VENTAS)
VALUES (115, 'Ben Adams', NULL, '01-ABR-90', 0.00)

INSERT INTO OFICINAS (OFICINA, CIUDAD, REGION, DIR, TITULO, VENTAS)
VALUES (14, 'Detroit', 'Este', 115, 0.00, 0.00)

UPDATE REPVENTAS
SET OFICINA.REP = 14
WHERE NUM_EMPL = 115
```

Como muestra el ejemplo, hay veces en que sería conveniente que la restricción de integridad referencial no fuera comprobada hasta después de que se ejecutaran una serie de actualizaciones interrelacionales. Desgraciadamente, este tipo de «comprobación diferida» compleja no la proporciona SQL.

Los ciclos referenciales también restringen las reglas de supresión que pueden especificarse para las relaciones que forman el ciclo. Consideremos las tres tablas del ciclo referencial mostrado en la Figura 11.6. La tabla PERROS muestra tres perros y los chicos que ellos quieren, la tabla CHICAS muestra tres chicas y los perros que ellas quieren y la tabla CHICOS muestra cuatro chicos y las chicas que ellos quieren, formando un ciclo referencial. Todas las relaciones del ciclo especifican la regla de supresión RESTRICT. Observe que la fila de George es la *única* fila que se puede suprimir de las tres tablas. Cualquier otra fila es padre de alguna relación, y por tanto está protegida frente a supresiones por la regla RESTRICT. Debido a esta anomalía, no se podría especificar la regla RESTRICT para todas las relaciones en un ciclo referencial.

La regla CASCADE presenta un problema análogo, como se muestra en la Figura 11.7. Esta figura contiene exactamente los mismos datos que la Figura 11.6, pero las tres reglas de supresión se han cambiado a CASCADE. Supongamos que se intenta suprimir a Bob de la tabla CHICOS. Las reglas de supresión fuerzan al DBMS a suprimir Rover (que quiere a Bob) de la tabla PERROS, lo cual fuerza a suprimir a Betty (que quiere a Rover) de la tabla CHICAS, lo cual fuerza a suprimir a Sam (que quiere a Betty), y así sucesivamente hasta que todas las filas de las tres tablas han sido suprimidas. Para estas tres pequeñas tablas esto podría ser práctico, pero para una base de datos de producción con miles de filas rápidamente

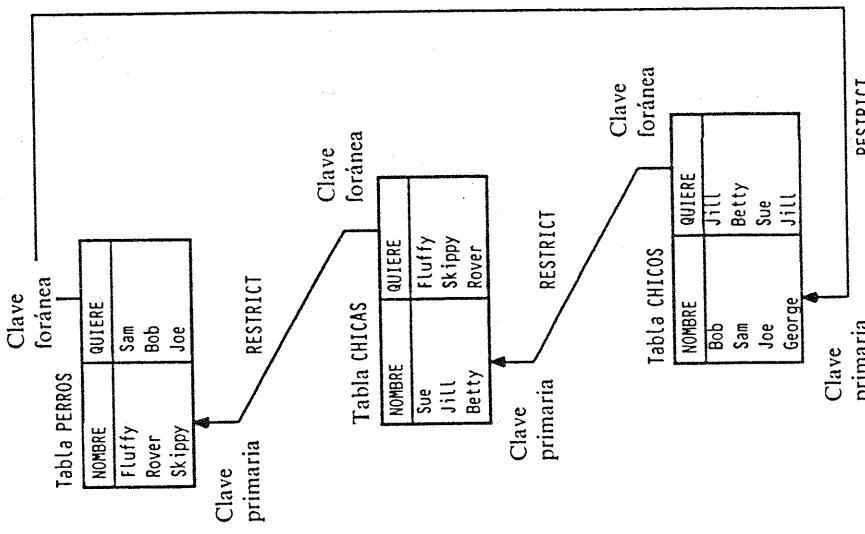


Figura 11.6. Un ciclo con todas reglas RESTRICT.

mente sería imposible llevar la cuenta de las supresiones en cascada y retener la integridad de la base de datos. Por esta razón DB2 fuerza una regla que impide ciclos referenciales de dos o más tablas donde todas las reglas de supresión son CASCADE. Al menos una relación de ciclo *debe* tener una regla de supresión RESTRICT o SET NULL para romper el ciclo de supresiones en cascada.

Claves foráneas y valores NULL*

A diferencia de las claves primarias, las claves foráneas de una base de datos relacional se permite que contengan valor NULL. En la base de datos ejemplo, la

clave foránea OFICINA.REP, en la tabla REPENTAS, permite valores NULL. De hecho, esta columna contiene realmente un valor NULL en la fila de Tom Snyder, ya que Tom aún no ha sido asignado a una oficina. Pero el valor NULL presenta una cuestión interesante referente a la restricción de integridad referencial creada por la relación clave primaria/clave foránea. ¿Corresponde el valor NULL a uno de los valores de clave primaria o no? La respuesta es «quizás» —depende del valor «real» del dato que falta o es desconocido.

Tanto DB2 como el ANSI/ISO suponen automáticamente que una clave foránea que contiene un valor NULL satisface la restricción de integridad referencial. En otras palabras, dan a la fila «el beneficio de la duda» y le permiten que forme parte de la tabla hijo, aun cuando su valor de clave foránea no coincida con ninguna fila en la tabla padre. Lo que es más interesante, la restricción de integridad referencial se supone que se satisface si *cualquier parte* de la clave foránea tiene un valor NULL. Esto puede producir un comportamiento inesperado y poco intuitivo en claves foráneas compuestas, tal como la que enlaza la tabla PEDIDOS con la tabla PRODUCTOS.

Supongamos por un momento que la tabla PEDIDOS en la base de datos ejemplo permite valores NULL para la columna PRODUCTO, y que la relación PRODUCTOS/PEDIDOS tiene una regla de supresión SET NULL. (Esta no es la estructura real de la base de datos ejemplo, por razones ilustradas en esta discusión.) Un pedido para un producto con un id de fabricante (FAB) «ABC» y un id de producto NULL (PRODUCTO) puede ser insertado con éxito en la tabla PEDIDOS, debido al valor NULL en la columna PRODUCTO. DB2 y el estándar ANSI/ISO suponen que la fila satisface la restricción de integridad referencial para PEDIDOS y PRODUCTOS, aun cuando ningún producto en la tabla PRODUCTOS tiene un id de fabricante «ABC».

La regla de supresión SET NULL puede producir un efecto análogo. La supresión de una fila de la tabla PRODUCTOS hará que el valor de clave foránea en todas sus filas hijo en la tabla PEDIDOS pase a ser el valor NULL. Realmente, sólo aquellas columnas de la clave foránea que acepten valores NULL se pondrán a NULL. Si hubiera una sola fila en la tabla PRODUCTOS para el fabricante «DEF», suprimir esa fila haría que sus filas hijo en la tabla PEDIDOS tuvieran la columna PRODUCTO fijada a NULL, pero la columna FAB continuaría teniendo el valor «DEF». Como resultado, las filas tendrían un valor FAB que no coincidiría con ninguna fila en la tabla PRODUCTOS.

Para evitar crear esta situación, debería tenerse cuidado con los valores NULL en claves foráneas compuestas. Una aplicación que introduce o actualiza datos en la tabla que contiene la clave foránea debería generalmente forzar una regla de «*o todo* NULL o no NULL» para las columnas de la clave foránea. Las claves foráneas que son parcialmente NULL y parcialmente no NULL pueden crear problemas fácilmente.

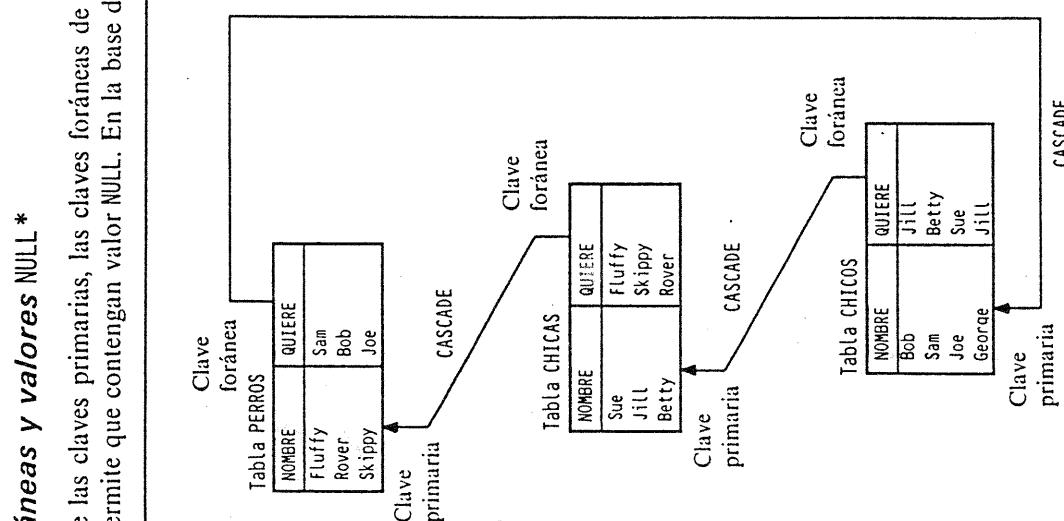


Figura 11.7. Un ciclo ilegal con reglas todas CASCADE.

Reglas comerciales

- Muchas de las cuestiones de integridad de datos en el mundo real tienen que ver con las reglas y procedimientos de una organización. Por ejemplo, la empresa que está modelada en la base de datos podría tener reglas como éstas:
 - No se permite que ningún cliente ordene pedidos que pudieran exceder el límite de crédito del cliente.
 - El vicepresidente de ventas debe ser notificado cada vez que a un cliente se le asigne un límite de crédito superior a \$50.000.
 - Los pedidos sólo pueden permanecer en los registros durante seis meses; los pedidos más antiguos de seis meses deben ser cancelados o reintroducidos.

Además, con frecuencia existen «reglas de contabilidad» que deben ser seguidas para mantener la integridad de totales, cuentas y otras cantidades almacenadas en una base de datos. Para la base de datos ejemplo, estas reglas probablemente tienen sentido:

- Cada vez que se admite un nuevo pedido, la columna VENTAS del vendedor que aceptó el pedido y de la oficina en donde ese vendedor trabaja deberían ser incrementadas en el importe del pedido. La supresión de un pedido o el cambio de su importe también deberían provocar que las columnas VENTAS se reajustaran.

■ Cada vez que se acepta una nueva orden, la columna EXISTENCIAS del producto pedido debería disminuir en la cantidad del producto solicitado. La supresión de un pedido, la modificación de la cantidad o la modificación del producto solicitado también deberían provocar los correspondientes ajustes en la columna EXISTENCIAS.

Estas reglas caen fuera del ámbito del lenguaje SQL tal como está definido en el estándar ANSI/ISO e implementado por la mayoría de los productos DBMS basados en SQL hoy día. El DBMS toma la responsabilidad de almacenar y organizar los datos y asegurar su integridad básica, pero forzar las reglas comerciales es responsabilidad de los programas de aplicación que acceden a la base de datos.

Dejar la tarea de forzar las reglas comerciales a los programas de aplicación que acceden a las bases de datos tiene varias desventajas:

- **Duplicación de esfuerzo.** Si seis programas diferentes realizan varias actualizaciones de la tabla PEDIDOS, cada uno de ellos debe incluir código que fuerce las reglas relativas a las actualizaciones de PEDIDOS

■ **Falta de consistencia.** Si varios programas escritos por diferentes programadores manejan las actualizaciones de una tabla, probablemente forzarán reglas en cierto modo diferentes.

■ **Problemas de mantenimiento.** Si las reglas comerciales cambian, los programadores deben identificar todos y cada uno de los programas que fuerzan las reglas, localizar el código y modificarlo correctamente.

■ **Complejidad.** Con frecuencia hay muchas reglas que recordar. Incluso en la pequeña base de datos ejemplo un programa que gestione cambios de pedidos debe preocuparse de forzar límites de créditos, ajuste de totales de ventas para vendedores y oficinas y ajuste de existencias. Un programa que gestione actualizaciones sencillas puede convertirse en complejo muy rápidamente.

La exigencia de que los programas de aplicación fuerzen las reglas comerciales no es propia únicamente de SQL. Los programas de aplicación han tenido esa responsabilidad desde los primitivos días de programas COBOL y sistemas de archivos. Sin embargo, ha habido una tendencia continua a lo largo de estos años para poner más «entendimiento» sobre los datos y más responsabilidad respecto a su integridad en la propia base de datos. En 1986, El DBMS Sybase introdujo el concepto de *disparador* como un paso adelante a la inclusión de reglas comerciales en una base de datos relacional.

¿Qué es un disparador?

El concepto de disparador, tal como lo define Sybase, es relativamente sencillo. Para cualquier evento que provoca un cambio en el contenido de una tabla, un usuario puede especificar una acción asociada que el DBMS debería efectuar. Los tres eventos que pueden disparar una acción son inventos de INSERT, DELETE o UPDATE filas de la tabla. La acción disparada por un evento se especifica mediante una secuencia de sentencias SQL, escritas en el dialecto Transact-SQL de Sybase.

Para entender cómo funciona un disparador, examinemos un ejemplo concreto. Cuando se añade un nuevo pedido a la tabla PEDIDOS, estos dos cambios también podrían tener lugar en la base de datos:

- La columna VENTAS del vendedor que aceptó la orden debería incrementarse en el importe del pedido.
- El valor de EXISTENCIAS para el producto ordenado debería disminuir en la cantidad solicitada.

Esta sentencia de Transact-SQL define un disparador, de nombre NUEVOPEDIDO, que hace que estas actualizaciones sucedan automáticamente:

```
CREATE TRIGGER NUEVOPEDIDO
ON PEDIDOS
FOR INSERT
AS UPDATE 'REPVENTAS'
SET VENTAS = VENTAS + INSERTED.IMPORTE
FROM REPVENTAS, INSERTED
WHERE REPVENTAS.NUM_EMPL = INSERTED.REP

UPDATE PRODUCTOS
SET EXISTENCIAS = EXISTENCIAS - INSERTED.CANT
FROM PRODUCTOS, INSERTED
WHERE PRODUCTOS.ID_DIR = INSERTED.FAB
AND PRODUCTOS.ID_PRODUCTO = INSERTED.PRODUCTO
```

La primera parte de la definición del disparador dice s Sybase que el disparador va a ser invocado cada vez que se intente una sentencia INSERT sobre la tabla PEDIDOS. El resto de la definición (después de la palabra clave AS) define la acción del disparador. En este caso, la acción es una secuencia de dos sentencias UPDATE, una para la tabla REPVENTAS y otra para la tabla PRODUCTOS. La fila que está siendo insertada es referida utilizando el nombre de pseudotabla INSERTED dentro de las sentencias UPDATE. Como muestra el ejemplo, Transact-SQL extiende el lenguaje SQL sustancialmente para soportar disparadores. Otras extensiones no mostradas aquí incluyen tests IF/THEN/ELSE, iteraciones, llamadas de procedimiento, e incluso sentencias PRINT que visualizan mensajes de usuario.

Disparadores e integridad referencial

Los disparadores proporcionan un modo alternativo de implementar las restricciones de integridad referencial proporcionadas por claves foráneas y claves primarias. De hecho, los defensores de la característica disparadora señalan que el mecanismo disparador es más flexible que la integridad referencial proporcionada por DB2 y el estándar ANSI/ISO. Por ejemplo, he aquí un disparador que fuerza la integridad referencial para la relación OFICINAS/REPVENTAS, y que también muestra un mensaje cuando una actualización intentada falla:

```
CREATE TRIGGER ACTUALREP
ON REPVENTAS
FOR INSERT, UPDATE
AS IF ((SELECT IMPORTE (*)
        FROM OFICINAS, INSERTED
       WHERE OFICINAS.OFICINA = INSERTED.OFICINA.REP) = 0)
      PRINT ("Especificado un número de oficina invalido")
ROLLBACK TRANSACTION
END
```

Los disparadores también pueden ser utilizados para proporcionar formas ampliadas de integridad referencial no ofrecidas por DB2 ni por el estándar SQL ANSI/ISO. Por ejemplo, DB2 proporciona supresiones propagadas mediante la regla de supresión CASCADE, pero no soporta «actualizaciones propagadas» si se modifica un valor de clave primaria. Esta limitación no es aplicable a los disparadores de Sybase. El siguiente disparador propaga cualquier actualización de la columna OFICINA en la tabla OFICINAS a la columna OFICINA.REP de la tabla REPVENTAS:

```
CREATE TRIGGER CAMBIA_OFICINA_REP
ON OFICINAS
FOR UPDATE
AS IF UPDATE (OFICINA)
BEGIN
  UPDATE REPVENTAS
    SET REPVENTAS.OFICINA.REP = INSERTED.OFICINA
  FROM REPVENTAS, INSERTED,
       WHERE REPVENTAS.OFICINA.REP = DELETED.OFICINA
END
```

Las referencias DELETED.OFICINA e INSERTED.OFICINA en el disparador se definen, respectivamente, a los valores de la columna OFICINA antes y después de la sentencia UPDATE.

El futuro de los disparadores

Una discusión completa de los disparadores de Sybase está fuera del alcance de este libro, pero incluso este sencillo ejemplo muestra la potencia del mecanismo disparador. La principal ventaja de los disparadores es que las reglas comerciales pueden almacenarse en las bases de datos y ser forzadas consistentemente con cada actualización de la base de datos. Esto puede reducir sustancialmente la complejidad de los programas de aplicación que acceden a la base de datos. Los disparadores tienen también algunas desventajas, incluyendo estas dos:

■ *Complejidad de la base de datos.* Cuando las reglas se trasladan al interior de la base de datos, preparar la base de datos pasa a ser una tarea más compleja. Los usuarios que razonablemente podían esperar crear pequeñas aplicaciones propias con SQL, encontrarán que la lógica de programación de los disparadores hace la tarea mucho más difícil.

■ *Reglas ocultas.* Con las reglas ocultas dentro de la base de datos, programas que parecen efectuar sencillas actualizaciones de la base de datos pueden, de hecho, generar una cantidad enorme de actividad en la base de datos. El programador ya no controla totalmente lo que sucede en la base de datos.

12 Procesamiento de transacciones

Los disparadores son una de las características más ampliamente alabadas y publicadas de Sybase. Con el apoyo de Microsoft tras SQL Server (el cual se deriva del DBMS Sybase), otros varios vendedores independientes de DBMS han proclamado públicamente planes para añadir disparadores a las versiones futuras de sus productos SQL. IBM ha guardado silencio con respecto a este tema, y parece poco probable que DB2, SQL/DS u OS/2 Extended Edition dispongan de disparadores en un futuro cercano. Ninguno de los principales DBMS comerciales aparte de Sybase y SQL Server dispone de disparadores hoy día.

Los disparadores están, naturalmente, completamente ausentes del estándar SQL ANSI/ISO actual. De hecho, las extensiones procedimentales de SQL incluidas en Transact-SQL y que se requieren para hacer disparadores flexibles representan un planteamiento muy diferente de SQL programado con respecto al «SQL incorporado» descrito por el estándar. La importancia futura de los disparadores es por tanto un poco nebulosa, y solamente el tiempo dirá si se convierten en parte importante del entorno SQL.

Resumen
El lenguaje SQL dispone de una serie de características que ayudan a proteger la integridad de los datos almacenados en una base de datos relacional:

- Pueden especificarse columnas requeridas cuando se crea una tabla, y el DBMS impedirá los valores NULL en estas columnas.
- La validación de datos se limita al chequeo del tipo de dato en SQL estándar, pero muchos productos DBMS ofrecen otras características de validación de datos.
- Las restricciones de integridad de entidad aseguran que la clave primaria identifique únicamente a cada entidad representada en la base de datos.
- Las restricciones de integridad referencial aseguran que las relaciones entre entidades en la base de datos se preserven durante las actualizaciones.
- DB2 proporciona soporte de integridad referencial, incluyendo reglas de supresión que dicen cómo manejar la supresión de filas que son referenciadas mediante otras filas. Otros vendedores de DBMS han proporcionado soporte de integridad referencial o están a punto de hacerlo.
- Las reglas comerciales pueden ser forzadas por el DBMS mediante el mecanismo disparador popularizado por Sybase y SQL Server. Los disparadores permiten al DBMS tomar acciones complejas en respuesta a eventos tales como sentencias INSERT, DELETE o UPDATE intentadas.

Las actualizaciones de la base de datos son generalmente originadas por eventos del mundo real tales como la recepción de un nuevo pedido de un cliente. De hecho, la recepción de un nuevo pedido generaría no sólo una, sino una serie de cuatro actualizaciones a la base de datos ejemplo:

- Añadir el nuevo pedido a la tabla PEDIDOS.
 - Actualizar el total de ventas para el vendedor que aceptó el pedido.
 - Actualizar el total de ventas para la oficina del vendedor.
 - Actualizar el total de existencias para el producto ordenado.
- Para dejar la base de datos en un estado autoconsistente, las cuatro actualizaciones deben producirse como una unidad. Si un fallo del sistema u otro error crea una situación en donde algunas de las actualizaciones son procesadas y otras no, la integridad de la base de datos se perderá. Análogamente, si otro usuario calcula totales o porcentajes en medio de la secuencia de actualizaciones, los cálculos serán incorrectos. La secuencia de actualizaciones deben ser una proposición «todo o nada» en la base de datos. SQL proporciona precisamente esta capacidad a través de sus características de procesamiento de transacciones, que se describen en este capítulo.

¿Qué es una transacción?

Una transacción es una secuencia de una o más sentencias SQL que juntas forman una unidad de trabajo. Las sentencias SQL que forman la transacción

suelen estar estrechamente relacionadas y efectuar acciones interdependientes. Cada sentencia de una transacción efectúa una parte de una tarea, pero todas ellas son necesarias para completar la tarea. La agrupación de las sentencias en una sola transacción indica al DBMS que la secuencia de sentencias entera debe ser ejecutada atómicamente; *todas* las sentencias deben completarse para que la base de datos esté en un estado consistente.

He aquí algunos ejemplos de transacciones típicas para la base de datos ejemplo, junto con la secuencia de sentencias SQL que forman cada transacción:

■ *Añadir un pedido.* Para aceptar el pedido de un cliente, el programa de entrada de pedido debería: *a)* consultar la tabla PRODUCTOS para asegurar que el producto está en stock; *b)* insertar el pedido en la tabla PEDIDOS; *c)* actualizar la tabla PRODUCTOS, restando la cantidad solicitada de las existencias del producto; *d)* actualizar la tabla REPVENTAS, sumando el importe del pedido a las ventas totales del vendedor que lo aceptó, y *e)* actualizar la tabla OFICINAS, añadiendo el importe del pedido a las ventas totales de la oficina en donde el vendedor trabaja.

■ *Cancelar un pedido.* Para cancelar el pedido de un cliente, el programa debería: *a)* suprimir el pedido de la tabla PEDIDOS; *b)* actualizar la tabla PRODUCTOS, ajustando el total de existencias del producto; *c)* actualizar la tabla REPVENTAS, restando el importe del pedido del total de ventas del vendedor, y *d)* actualizar la tabla OFICINAS, restando el importe del pedido de las ventas totales de la oficina.

■ *Reasignar un cliente.* Cuando se reasigna un cliente de un vendedor a otro, el programa debería: *a)* actualizar la tabla CLIENTES para reflejar el cambio; *b)* actualizar la tabla PEDIDOS para mostrar el nuevo vendedor para todos los pedidos remitidos por el cliente; *c)* actualizar la tabla REPVENTAS, reduciendo la cuota del vendedor que pierde el cliente, y *d)* actualizar la tabla OFICINAS, elevando la cuota del vendedor que gana el cliente.

En cada uno de estos casos se requiere una secuencia de cuatro o cinco acciones, cada una formada por una sola sentencia SQL, para manejar la única transacción «lógica».

El concepto de transacción es crítico para los programas que actualizan una base de datos ya que asegura la integridad de la base de datos. Un DBMS basado en SQL efectúa este compromiso referente a las sentencias de una transacción:

Las sentencias de una transacción se ejecutarán como una unidad atómica de trabajo en la base de datos. O *todas* las sentencias son ejecutadas con éxito, o *ninguna* de las sentencias es ejecutada.

El DBMS es responsable de mantener este compromiso incluso si el programa de aplicación aborta o se produce un fallo hardware a mitad de la transacción, tal como se muestra en la Figura 12.1. En cada caso, el DBMS debe asegurarse que cuando se complete una recuperación del fallo, la base de datos nunca refleje una «transacción parcial».

COMMIT Y ROLLBACK

SQL soporta las transacciones de base de datos mediante dos sentencias de procesamiento de transacciones SQL, mostradas en la Figura 12.2:

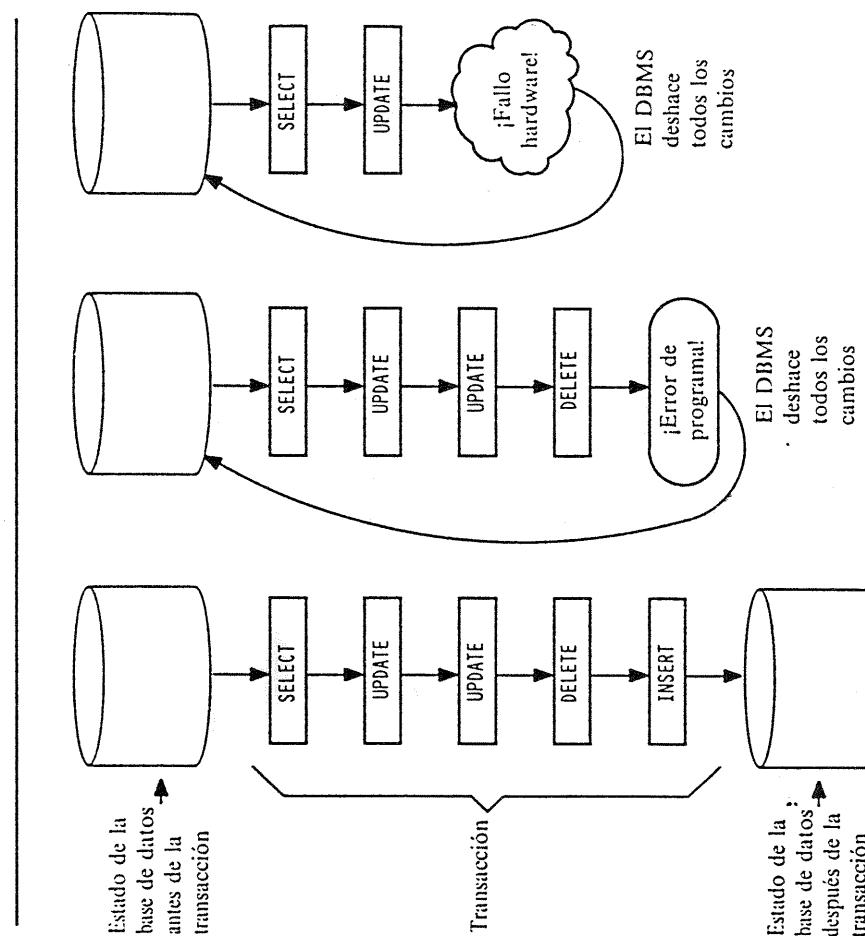


Figura 12.1. El concepto de transacción en SQL.

■ La sentencia COMMIT señala el final correcto de una transacción. Informa al DBMS que la transacción está ahora completa; todas las sentencias que forman la transacción han sido ejecutadas, y la base de datos es autoconsistente.

■ La sentencia ROLLBACK señala el final sin éxito de una transacción. Informa al DBMS que el usuario no desea completar la transacción; en vez de ello, el DBMS debe *deshacer* los cambios efectuados a la base de datos durante la transacción. En efecto, el DBMS restaura la base de datos a su estado antes de que la transacción comienza.

Las sentencias COMMIT y ROLLBACK son sentencias SQL ejecutadas, lo mismo que SELECT, INSERT y UPDATE. He aquí un ejemplo de una transacción de actualización correcta que modifica la cantidad e importe de un pedido y ajusta los totales para el producto, el vendedor y la oficina asociada con el pedido. Una modificación como ésta sería típicamente manejada por un programa de «cambio de pedido» basado en formularios, que utilizaría SQL programado para ejecutar las sentencias mostradas:

Cambiala la cantidad del pedido numero 113051 de a 10, lo cual elevará su importe de \$1.458 a \$3.550. El pedido es de Reductores QSA-YK47, y fue atendido por Larry Fitch (empleado numero 108) que trabaja en Los Angeles (oficina numero 2).

```
UPDATE PEDIDOS  
SET CANT = 10, IMPORTE = 3550.00
```

```

    UPDATE REPVENTAS
      SET VENTAS = VENTAS - 1458.00 + 3550.00
    WHERE NUMEMPL = 108
  
```

```
UPDATE OFICINAS  
SET VENTAS = VENTAS - 1458.00 + 3550.00  
WHERE OFICINA = 21
```

```

UPDATE PRODUCTOS
SET EXISTENCIAS = EXISTENCIAS + 4 - 10
WHERE ID_FAB = 'QSA'
      AND ID_PRODUCTO = 'KX47';

```

confirma el cambio una última vez con el cliente...

He aquí la misma transacción, pero esta vez suponiendo que el usuario comete un error al introducir el número del producto. Para corregir el error, la

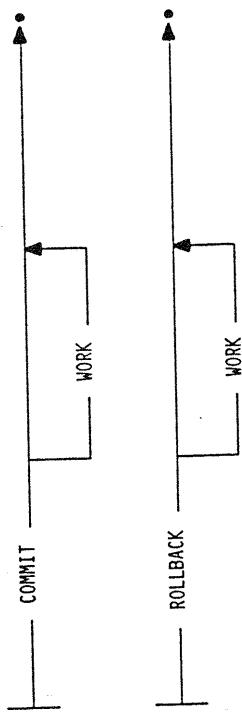


Figura 12.2. Diagramas sintácticos de las sentencias COMMIT y ROLLBACK.

transacción vuelve atrás, para que el número pueda ser reintroducido correctamente:

Cambiala la cantidad del pedido número 113051 de a 10, lo cual eleva su importe de \$1,428 a \$3,550. El pedido es de Reductores QAS-XK47, y fue atendido por Larry Fitch (empleado número 108) que trabaja en Los Angeles (oficina número 21).

```

UPDATE PEDIDOS
    SET CANT = 10, IMPORTE = 3550.00
  WHERE NR_PEDIDO = 113051

UPDATE REVENTAS
    SET VENTAS = VENTAS - 1458.00 + 3550.00
  WHERE NUMEMPL = 108

UPDATE OFICINAS
    SET VENTAS = VENTAS - 1458.00 + 3550.00
  WHERE OFICINA = 21

UPDATE PRODUCTOS
    SET EXISTENCIAS = EXISTENCIAS + 4 - 10
  WHERE ID_FAB = 'QAS'
    AND ID_PRODUCTO = 'XK47'

...junto el fabricante es «QSA», y no «QAS»

```

El modelo de transacción ANSI/ISO

El estándar SQL ANSI/ISO define un *modelo de transacción* SQL y los papeles de la sentencias COMMIT y ROLLBACK. La mayoría, aunque no todos, los productos

SQL comerciales utilizan este modelo de transacción, que está basado en DB2. El estándar específico que una transacción SQL comienza *automáticamente* con la primera sentencia SQL ejecutada por un usuario o un programa. La transacción continúa con las sentencias SQL subsiguientes hasta que finaliza de uno de cuatro modos:

- Una sentencia COMMIT finaliza la transacción con éxito, haciendo que los cambios a la base de datos sean permanentes. Una nueva transacción comienza inmediatamente después de la sentencia COMMIT.
- Una sentencia ROLLBACK aborta la transacción, deshaciendo las modificaciones que haya efectuado a la base de datos. Una nueva transacción comienza inmediatamente después de la sentencia ROLLBACK.
- La terminación de un programa con éxito (para SQL programado) también finaliza la transacción correctamente, igual que si se hubiera ejecutado una sentencia COMMIT. Puesto que el programa está finalizado, no hay ninguna nueva transacción que comenzar.
- La terminación anormal del programa (para SQL programado) también aborta la transacción, del mismo modo que si se hubiera ejecutado una sentencia ROLLBACK. Puesto que el programa está finalizado, no hay ninguna nueva transacción que comenzar.

La Figura 12.3 muestra algunas transacciones típicas que ilustran estas cuatro condiciones. Observe que el usuario o programa está *siempre* en una transacción bajo el modelo de transacción ANSI/ISO. No se requiere ninguna acción explícita para comenzar una transacción; comienza automáticamente con la primera sentencia SQL o inmediatamente después que la transacción precedente acaba. Recuerde que el estándar SQL ANSI/ISO especifica un lenguaje SQL *programado* para uso en programas de aplicación. Las transacciones juegan un papel importante en SQL programado, ya que incluso un sencillo programa de aplicación necesita efectuar con frecuencia una secuencia de dos o tres sentencias SQL para completar su tarea. Puesto que los usuarios pueden cambiar de intención o pueden producirse otras condiciones (tal como quedarce sin existencias de un producto que un usuario desea pedir), un programa de aplicación debe ser capaz de realizar parte de una transacción y luego elegir entre abortar o continuar. Las sentencias COMMIT y ROLLBACK proporcionan precisamente esta capacidad.

Las sentencias COMMIT y ROLLBACK también pueden ser utilizadas en SQL interactivo, pero en la práctica raramente se ven en este contexto. SQL interactivo es utilizado generalmente para consultas de la base de datos; las actualizaciones son menos comunes, y las actualizaciones multisentencia casi nunca se efectúan escribiendo las sentencias en una facilidad SQL interactiva. Como

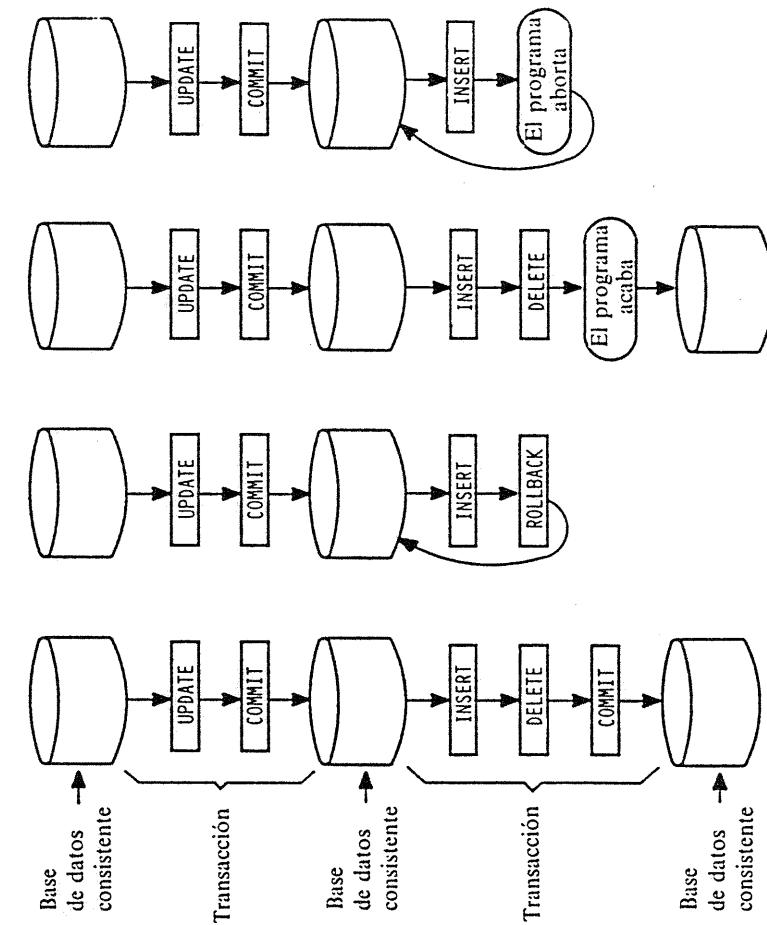


Figura 12.3. Transacciones cumplimentadas COMMIT y vueltas atrás ROLLBACK.

resultado, las transacciones son típicamente una preocupación menor en SQL interactivo. De hecho, muchos productos SQL interactivo trabajan por omisión en modo «autocomplimentación», en el cual una sentencia COMMIT se ejecuta automáticamente después de cada sentencia SQL escrita por el usuario. Esto hace efectivamente que cada sentencia SQL interactiva sea su propia transacción.

Otros modelos de transacciones

Unos cuantos productos SQL comerciales se apartan del modelo de transacción ANSI/ISO y DB2 para proporcionar capacidades adicionales de proceso de transacciones a sus usuarios. El DBMS Sybase, que está diseñado para aplicaciones de procesamiento de transacciones en línea, es un ejemplo. SQL Server,

que es una derivación del producto Sybase, también utiliza el modelo de tracción Sybase.

El dialecto Transact-SQL utilizado por Sybase incluye cuatro sentencias de procesamiento de transacciones:

- La sentencia BEGIN TRANSACTION señala el comienzo de una transacción. A diferencia del modelo de transacción ANSI/ISO, que implícitamente inicia una nueva transacción cuando la anterior finaliza, Sybase requiere una sentencia explícita para iniciar una transacción.
- La sentencia COMMIT TRANSACTION señala el final con éxito de una transacción. Como en el modelo ANSI/ISO, todos los cambios efectuados a la base de datos durante la transacción pasan a ser permanentes. Sin embargo, no se inicia automáticamente una nueva transacción.
- La sentencia SAVE TRANSACTION establece un *punto de guarda* a mitad de una transacción. Sybase guarda el estado de la base de datos en un punto actual de la transacción y le asigna al estado guardado un *nombre de punto de guarda*, especificado en la sentencia.
- La sentencia ROLLBACK TRANSACTION tiene dos papeles. Si se designa un punto de guarda en la sentencia ROLLBACK, Sybase deshace los cambios de la base de datos efectuados desde el punto de guarda, dando marcha atrás efectivamente a la transacción hasta el punto en donde la sentencia SAVE TRANSACTION fue ejecutada. Si no hay ningún punto de guarda designado, la sentencia ROLLBACK deshace todos los cambios efectuados desde la sentencia BEGIN TRANSACTION.

El mecanismo de punto de guarda de Sybase es especialmente útil en transacciones complejas que contienen muchas sentencias, tal como se muestra en la Figura 12.4. El programa de aplicación de la figura guarda periódicamente su estado conforme progresó la transacción, estableciendo dos puntos de guarda nombrados. Si más tarde se presentan problemas durante la transacción, el programa de aplicación no tiene que abortar la transacción completa. En vez de ello, puede dar marcha atrás a la transacción hasta cualquier de sus puntos de guarda y proseguir adelante desde allí. Todas las sentencias ejecutadas antes del punto de guarda permanecen en efecto; las ejecutadas desde el punto de guarda son deshechas por la operación de marcha atrás.

Observe que la *transacción completa* sigue siendo la unidad lógica de trabajo para Sybase, como lo es para el modelo ANSI/ISO. Si se produce un fallo del sistema o del hardware en mitad de una transacción, por ejemplo, la transacción entera es suprimida de la base de datos. Por tanto, los puntos de guarda son una conveniencia para el programa de aplicación, pero no un cambio fundamental respecto al modelo de transacción ANSI/ISO.

El uso explícito de una sentencia BEGIN TRANSACTION es, sin embargo, una desviación significativa del modelo ANSI/ISO. Las sentencias SQL que se ejecu-

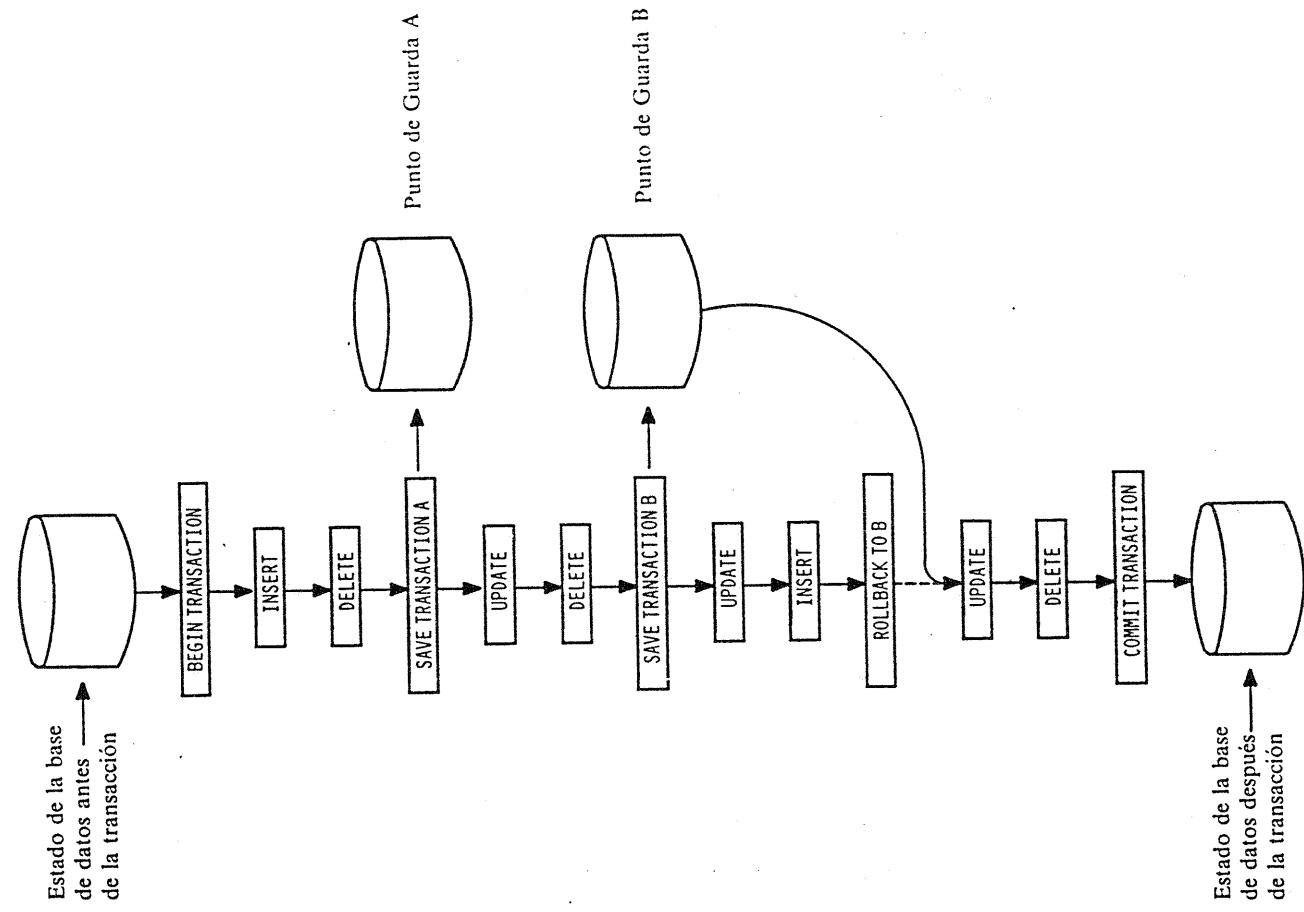


Figura 12.4. El modelo de transacción de Sybase.

tan «fuera de una transacción» (es decir, las sentencias que no aparecen entre un par de sentencias BEGIN/COMMIT o BEGIN/ROLLBACK) son efectivamente manejadas en modo «autocomplimentación». Cada sentencia se cumple cuando se ejecuta; no hay modo de deshacer la sentencia una vez que se ha ejecutado con éxito.

Sybase prohíbe específicamente la ocurrencia de ciertas sentencias dentro de una transacción, incluyendo las que alteran la estructura de una base de datos (tales como CREATE TABLE, ALTER TABLE y DROP TABLE discutidas en el Capítulo 13), las que alteran la seguridad de la base de datos (GRANT y REVOKE, discutidas en el Capítulo 15) y las que crean tablas temporales. Estas sentencias deben ser ejecutadas fuera de una transacción más fácil de implementar, ya que asegura que la estructura de la base de datos no puede cambiar durante una transacción. En contraste, la estructura de una base de datos puede ser alterada significativamente durante una transacción de estilo ANSI/ISO (las tablas pueden ser descartadas, creadas y rellenas, por ejemplo), y el DBMS debe ser capaz de deshacer todas las alteraciones si el usuario decide posteriormente dar marcha atrás en la transacción. En la práctica, las prohibiciones de Sybase no afectan a la utilidad del DBMS. Puesto que estas prohibiciones probablemente contribuyen a acelerar el rendimiento de la transacción, la mayoría de los usuarios aceptan con agrado este compromiso.

Transacciones: tras el telón *

El compromiso de «todo o nada» que un DBMS hace para las sentencias de una transacción parece algo casi mágico a un nuevo usuario SQL. ¿Cómo puede el DBMS deshacer los cambios efectuados a una base de datos, especialmente si ocurre un fallo del sistema a mitad de una transacción? Las técnicas actuales utilizadas por los productos DBMS varían, pero casi todos ellos se basan en un *registro de transacción*, (*transaction log*) tal como se muestra en la Figura 12.5.

He aquí cómo funciona el registro de transacción, de forma simplificada. Cuando un usuario ejecuta una sentencia SQL que modifica la base de datos, el DBMS escribe automáticamente una anotación en el registro de transacción mostrando dos copias de cada fila afectada por la sentencia. Una copia muestra la fila *antes* del cambio y la otra copia muestra la fila *después* del cambio. Solo después que se escribe el registro modifica el DBMS realmente la fila en el disco. Si el usuario ejecuta posteriormente una sentencia COMMIT, el fin de transacción se anota en el registro de transacción. Si el usuario ejecuta una sentencia ROLLBACK, el DBMS examina el registro para encontrar las imágenes «de antes» de las filas que han sido modificadas desde que comenzó la transacción. Utilizando estas imágenes, el DBMS restaura las filas a su estado anterior, deshaciendo efectivamente todas las modificaciones a la base de datos efectuadas durante la transacción.

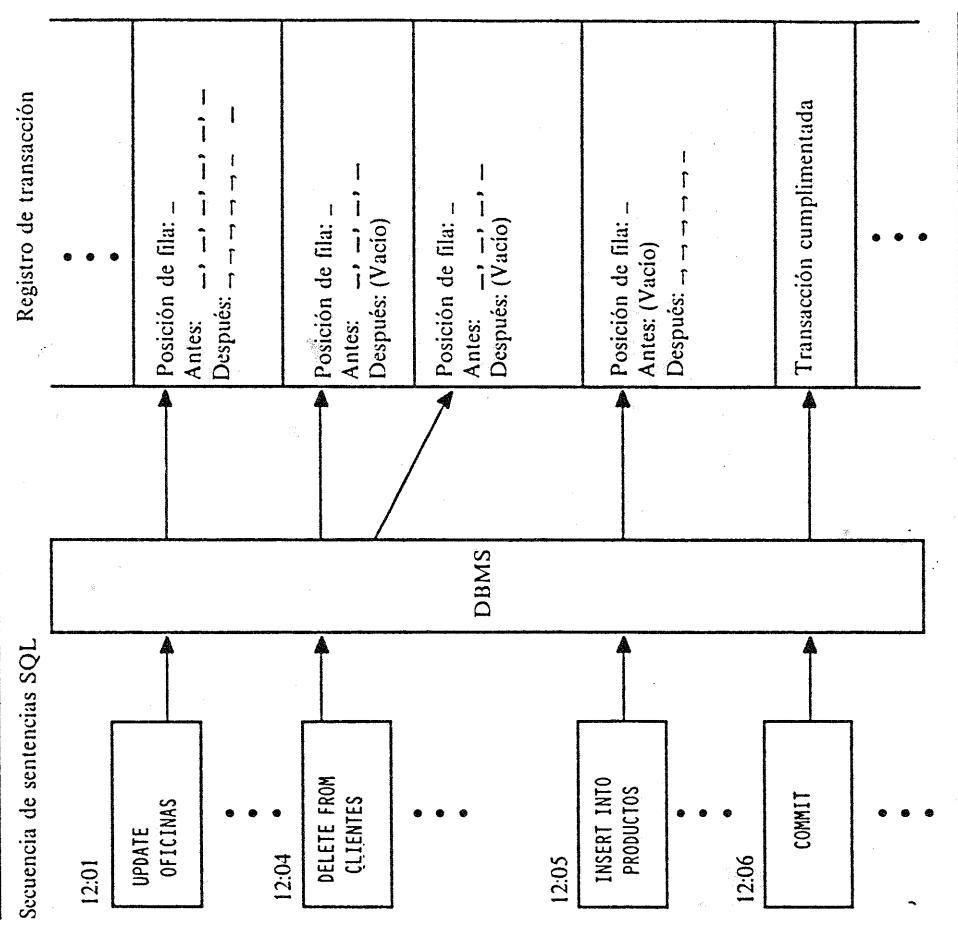


Figura 12.5. El registro de transacción.

Si se produce un fallo del sistema, el operador típicamente recupera la base de datos ejecutando una utilidad de recuperación especial suministrada con el DBMS. La utilidad de recuperación examina el final del registro de transacción, buscando las transacciones que no fueron cumplimentadas antes del fallo. La utilidad vuelve atrás cada una de estas transacciones incompletas y sólo las transacciones cumplimentadas se reflejan en la base de datos; las transacciones en proceso en el momento del fallo han sido deshechas.

El uso de un registro de transacción impone obviamente un recargo a las actualizaciones de la base de datos. Algunos productos DBMS sobre computadores personales permiten desactivar el registro de transacción para incrementar

el rendimiento del DBMS. Aunque esto puede ser aceptable en aplicaciones personales, el registro de transacción es esencial en bases de datos de producción para proporcionar recuperación de la base de datos y la capacidad ROLLBACK. En un DBMS multiusuario moderno, el registro de transacción se almacena generalmente en una unidad de disco rápida, diferente a la que almacena la base de datos, para minimizar la contención por acceso al disco.

Transacciones y procesamiento multiusuario

Cuando dos o más usuarios acceden concurrentemente a una base de datos, el procesamiento de transacciones toma una nueva dimensión. Ahora el DBMS no solamente debe recuperarse adecuadamente de los fallos o errores del sistema, también debe asegurarse que las acciones de los usuarios no interfieran unas con otras. Idealmente, cada usuario debería ser capaz de acceder a la base de datos como si tuviera acceso exclusivo a ella, sin preocuparse de las acciones del resto de los usuarios. El modelo de transacción SQL permite a un DBMS basado en SQL aislar a los usuarios unos de otros de este modo.

La mejor manera de entender cómo maneja SQL las transacciones concurrentes es examinar los problemas que resultan si las transacciones no se manejan adecuadamente. Aunque pueden aparecer en muchos modos diferentes, hay tres problemas fundamentales que pueden ocurrir. Las tres secciones siguientes muestran un sencillo ejemplo de cada problema.

El problema de la actualización perdida

La Figura 12.6 muestra una sencilla aplicación en donde dos usuarios aceptan pedidos de los clientes por teléfono. El programa de entrada de pedidos comprueba en el archivo PRODUCTO la existencia de un inventario adecuado antes de aceptar el pedido de 100 artículos ACI-41004 por parte de su cliente. Al mismo tiempo, Mary comienza a introducir el pedido de su cliente de 125 artículos ACI-41004. Cada programa de entrada de pedido efectúa una consulta sobre el archivo PRODUCTOS, y cada uno de ellos encuentra que hay 139 artículos en el almacén —más que suficiente para cubrir la solicitud del cliente—. Joe pide a su cliente que confirme el pedido, y su copia del programa de entrada de pedidos actualiza el archivo PRODUCTOS para que muestre $(139 - 100) = 39$ artículos restantes para vender e inserta un nuevo pedido de 100 artículos en la tabla PEDIDOS. Unos segundos más tarde, Mary pide a su cliente que confirme el pedido. Su copia del programa de entrada de pedidos actualiza el archivo PRODUCTOS para que muestre $(139 - 125) = 14$ artículos que quedan en almacen e inserta un nuevo pedido de 125 artículos en la tabla PEDIDOS.

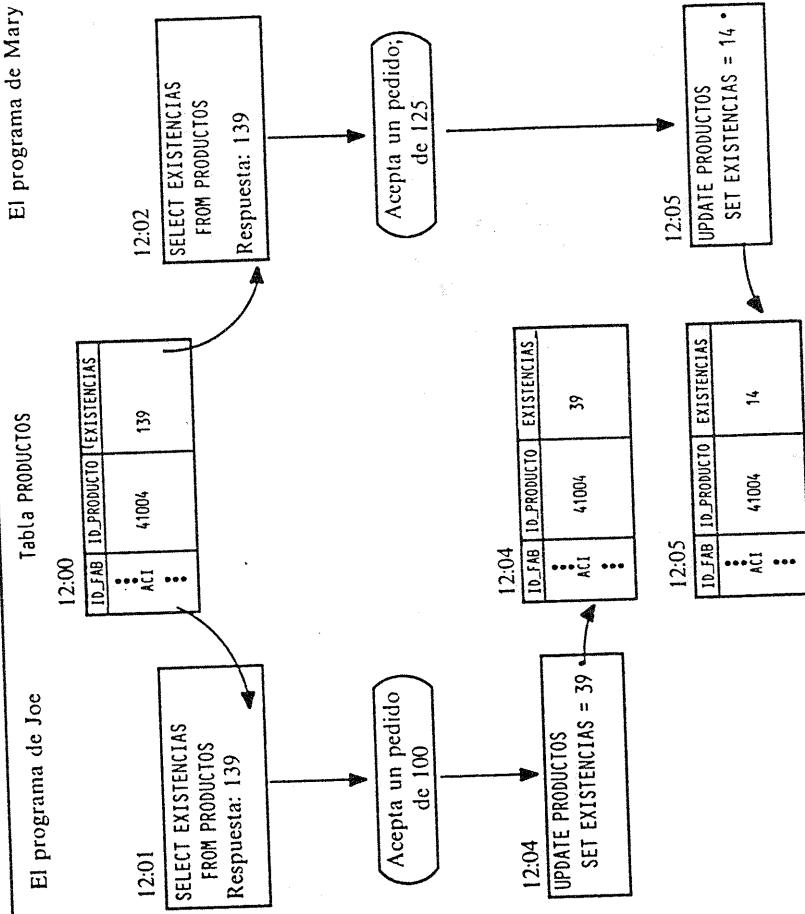
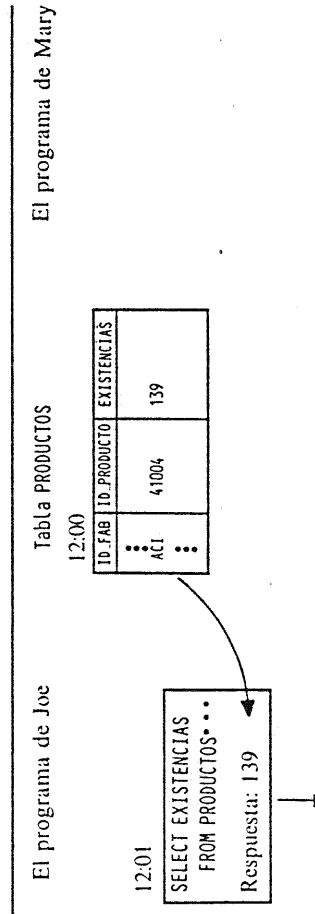


Figura 12.6. El problema de la actualización perdida.

La gestión de los dos pedidos ha dejado obviamente la base de datos en un estado inconsistente. La primera de las dos actualizaciones al archivo PRODUCTOS estuvo bien, pero la segunda no lo fue. La respuesta de los clientes han sido aceptados, pero no hay suficientes artículos en inventario para satisfacer a ambos. Además, la base de datos muestra que aún quedan 14 artículos disponibles para venta. Este ejemplo ilustra el problema de la «actualización perdida» que puede ocurrir cuando dos programas leen los mismos datos de la base de datos, utilizan los datos como base para un cálculo y luego tratan de actualizar los datos.

El problema de los datos no cumplimentados

La Figura 12.7 muestra la misma aplicación de procesamiento de pedidos que la Figura 12.6. Joe comienza de nuevo aceptando un pedido de 100 artículos ACI-41004.



Puesto que el programa de procesamiento de pedidos de Mary pudo examinar la actualización no cumplimentada del programa de Joe, el pedido del cliente de Mary fue rechazado, y el director de compras solicitará más artículos, aun cuando aún hay 139 en almacén. La situación habría sido aún peor si el cliente de Mary hubiera decidido pedir después de todo los 39 artículos disponibles. En este caso, el programa de Mary habría actualizado la tabla PRODUCTOS para que mostrara cero unidades disponibles. Pero cuando se produjera, el ROLLBACK de la transacción de Joe, el DBMS habría establecido el inventario disponible de nuevo en 139 artículos, aun cuando 39 de ellos estuvieran comprometidos con el cliente de Mary. El problema de este ejemplo es que el programa de Mary ha tenido permiso para ver las actualizaciones no cumplimentadas del programa de Joe y ha actuado sobre ellas, produciendo los resultados erróneos.

El problema de los datos inconsistentes

La Figura 12.8 muestra la aplicación de procesamiento de pedidos una vez más. En esta ocasión el director de ventas está ejecutando un programa de informe

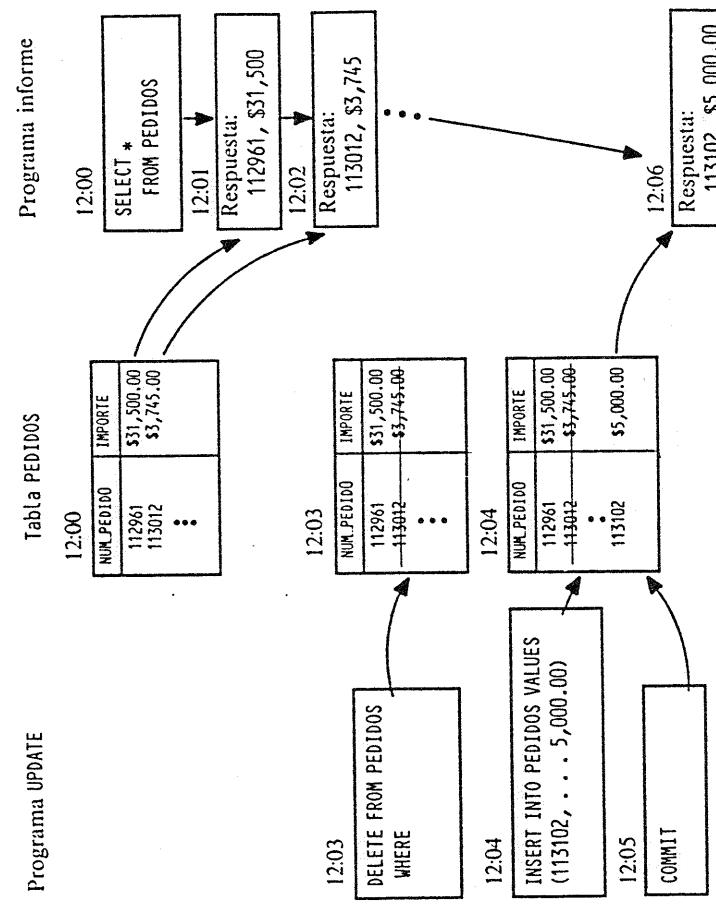


Figura 12.7. El problema de los datos no cumplimentados.

41004 de su cliente. Esta vez, la copia del programa de procesamiento de pedido de Joe consulta la tabla PRODUCTOS, encuentra 139 artículos disponibles y actualiza la tabla PRODUCTOS para mostrar que quedan 39 artículos después del pedido del cliente. A continuación Joe comienza a discutir con el cliente los méritos relativos de los artículos ACI-41004 y ACI-41005. Mientras tanto, el cliente de Mary trata de pedir 125 artículos ACI-41004. La copia del programa de procesamiento de pedidos de Mary consulta la tabla PRODUCTOS, encuentra que sólo hay 39 artículos disponibles y rechaza el pedido. Genera además una nota informando al director de compras para que compre más artículos ACI-41004, de los cuales hay una fuerte demanda. Ahora el cliente de Joe decide que no desea los artículos de Tipo 4 después de todo, y el programa de entrada de pedidos de Joe efectúa un ROLLBACK para abortar su transacción.

Figura 12.8. El problema de los datos inconsistentes.

que recorre la tabla PEDIDOS, imprime una lista de los pedidos de los clientes de Bill Adams y calcula el total. El programa recupera el pedido número 113012, lo imprime, suma sus \$3.745 al total acumulador y continúa el proceso. Mientras tanto, el cliente llama a Joe para cancelar el pedido número 113012 y remite en vez de ello un pedido de \$5.000. El programa de entrada de pedidos de Joe maneja el cambio correctamente, suprimiendo la orden cancelada, insertando el pedido sustituto en la tabla PEDIDOS y cumplimentando la transacción. Unos cuantos segundos más tarde en su procesamiento, el programa de informe encuentra el nuevo pedido, lo imprime y suma sus \$5.000 al total.

El programa de informe ha incluido tanto el pedido cancelado como el pedido sustituto en el informe, y ha añadido *ambas* importes a su total. Evidentemente, éste es un resultado incorrecto. O bien el informe no debería mostrar el pedido antiguo y sí el nuevo, o debería mostrar el antiguo y no el nuevo. Observe que este problema no es el mismo que el de las actualizaciones no cumplimentadas de la Figura 12.7, ya que el programa de informe no ha visto ningún dato no cumplimentado. En su lugar, el programa de informe ha visto algunos datos de *antes* de la transacción del cambio de pedido y algunos datos de *después* de la transacción del cambio de pedido, lo cual produce el resultado erróneo.

Transacciones concurrentes

Como muestran los tres ejemplos de actualizaciones multiusuario, cuando los usuarios comparten el acceso a una base de datos y uno o más de ellos están actualizando datos, hay una situación potencial de corrupción de la base de datos. SQL utiliza su mecanismo de transacción para eliminar esta fuente de corrupción de la base de datos. Además del compromiso «todo o nada» de las sentencias de una transacción, el DBMS basado en SQL efectúa este compromiso con respecto a las transacciones:

Durante una transacción, el usuario verá una vista completamente consistente de la base de datos. El usuario nunca contemplará modificaciones no cumplimentadas de otros usuarios, e incluso los cambios cumplimentados efectuados por otros no afectarán a los datos examinados por el usuario en mitad de una transacción.

Las transacciones son por tanto la esencia del control de recuperación y del control de concurrencia en una base de datos SQL. El compromiso anterior puede ser formulado explícitamente en términos de ejecución de transacciones concurrentes:

Si dos transacciones, A y B, se están ejecutando concurrentemente, el DBMS asegura que los resultados serán los mismos *en cualquier caso* tanto si: a) la

Transacción A se ejecuta primero, seguida de la Transacción B, como si b) la Transacción B se ejecuta primero, seguida de la Transacción A.

Este concepto es conocido como la serialidad de las transacciones. Efectivamente, significa que cada usuario de base de datos puede acceder a la base de datos como si no hubiera otros usuarios accediendo concurrentemente a ella. El hecho de que SQL aisle a un usuario de las acciones de otros usuarios concurrentes no significa, sin embargo, que se pueda ignorar todo acerca de los otros usuarios. De hecho, la situación es en gran medida la opuesta. Puesto que otros usuarios desean actualizar concurrentemente la base de datos, deberían mantenerse las transacciones tan breves y simples como fuera posible, para maximizar la cantidad de procesamiento paralelo que pueda generarse.

Supongamos, por ejemplo, que se ejecuta un programa que efectúa una secuencia de tres largas consultas. Como el programa no actualiza la base de datos, podría parecer que no necesita preocuparse con respecto a las transacciones. Ciertamente, parece innecesario utilizar sentencias COMMIT. Pero de hecho el programa debería utilizar una sentencia COMMIT después de cada consulta. ¿Por qué? Recuerde que SQL inicia *automáticamente* una transacción con la primera sentencia SQL de un programa. Sin una sentencia COMMIT, la transacción continúa hasta que el programa acaba. Además, SQL garantiza que los datos recuperados durante una transacción serán autoconsistentes, no afectados por transacciones de otros usuarios. Esto significa que una vez que el programa recupera una fila de la base de datos, ningún otro usuario puede modificar la fila hasta que la transacción finaliza, ya que el programa podrá tratar de recuperar la fila de nuevo posteriormente dentro de la misma transacción, y el DBMS debe garantizar que se examinen los mismos datos. Por tanto, mientras el programa efectúa sus tres consultas, impedirá a otros usuarios que actualicen porciones cada vez más grandes de la base de datos.

La moraleja de este ejemplo es sencilla: hay que preocuparse *siempre* respeto a las transacciones cuando se escriben programas para una base de datos SQL de producción. Las transacciones deberían ser siempre lo más breve posibles. «Usa COMMIT pronto y usa COMMIT a menudo» es un buen consejo cuando se está utilizando SQL programado.

Existen otros varios esquemas posibles para implementar la concurrencia necesaria en las transacciones SQL, pero todos los productos SQL comerciales utilizan la técnica basada en *cerramientos (lockings)*. El cerramiento se maneja automáticamente por parte del DBMS y es invisible al usuario de SQL. No es necesario entenderlo para utilizar las transacciones SQL. Sin embargo, la comprensión del cerramiento y cómo funciona dentro del DBMS puede ayudar a utilizar las transacciones más eficazmente. El resto de este capítulo discute el cerramiento con cierto detalle.

Cerramiento (*locking*)*

Los principales productos DBMS tales como DB2 y SQL Server utilizan técnicas de cerramiento sofisticadas para manejar transacciones concurrentes de muchos usuarios simultáneos. Sin embargo, los conceptos básicos de cerramiento y transacción son muy sencillos. La Figura 12.9 muestra un sencillo esquema de cerramiento y cómo maneja la contención entre dos transacciones concurrentes.

Cuando la Transacción A de la figura accede a la base de datos, el DBMS bloquea automáticamente cada parte de la base de datos que la transacción consulta o modifica. La Transacción B procede en paralelo, y el DBMS también bloquea las partes de la base de datos a las que ella accede. Si la Transacción B trata de acceder a la parte de la base de datos que ha sido bloqueada por la Transacción A, el DBMS congela la Transacción B, haciéndole que espere a que los datos sean desbloqueados. El DBMS libera los bloqueos mantenidos por la Transacción A solamente cuando ésta finaliza en una operación COMMIT o ROLLBACK. El DBMS «descongela» la Transacción B, permitiéndole que prosiga. La Transacción B puede bloquear esa parte de la base de datos por su propia cuenta, protegiéndola de los efectos de otras transacciones.

Como muestra la figura, la técnica del cerramiento proporciona a una transacción acceso temporal exclusivo a una parte de una base de datos, impidiendo que otras transacciones modifiquen los datos encerrados (bloqueados). El cerramiento resuelve por tanto los tres problemas de transacción concurrente. Impide que las actualizaciones perdidas, los datos no cumplimentados, y los datos inconsistentes puedan corromper la base de datos. Sin embargo, el cerramiento introduce un nuevo problema —puede hacer que una transacción espere durante mucho tiempo mientras las partes de la base de datos a las que desea acceder estén bloqueadas por otras transacciones.

Niveles de cerramiento

El cerramiento puede ser implementado a varios niveles de la base de datos. En su forma menos elaborada, el DBMS puede bloquear la base entera por cada transacción. Esta estrategia de cerramiento sería sencilla de implementar, pero sólo permitiría el proceso de una sola transacción cada vez. Si la transacción incluyera un «tiempo para pensar», tal como el tiempo para discutir un pedido con el cliente, todos los restantes accesos a la base de datos estarían bloqueados durante ese tiempo, conduciendo a un rendimiento inaceptablemente bajo.

Una forma mejorada de cerramiento es el cerramiento a *nivel de tabla*. En este esquema, el DBMS bloquea únicamente las tablas accedidas por una transacción. Otras transacciones pueden acceder concurrentemente a otras tablas.

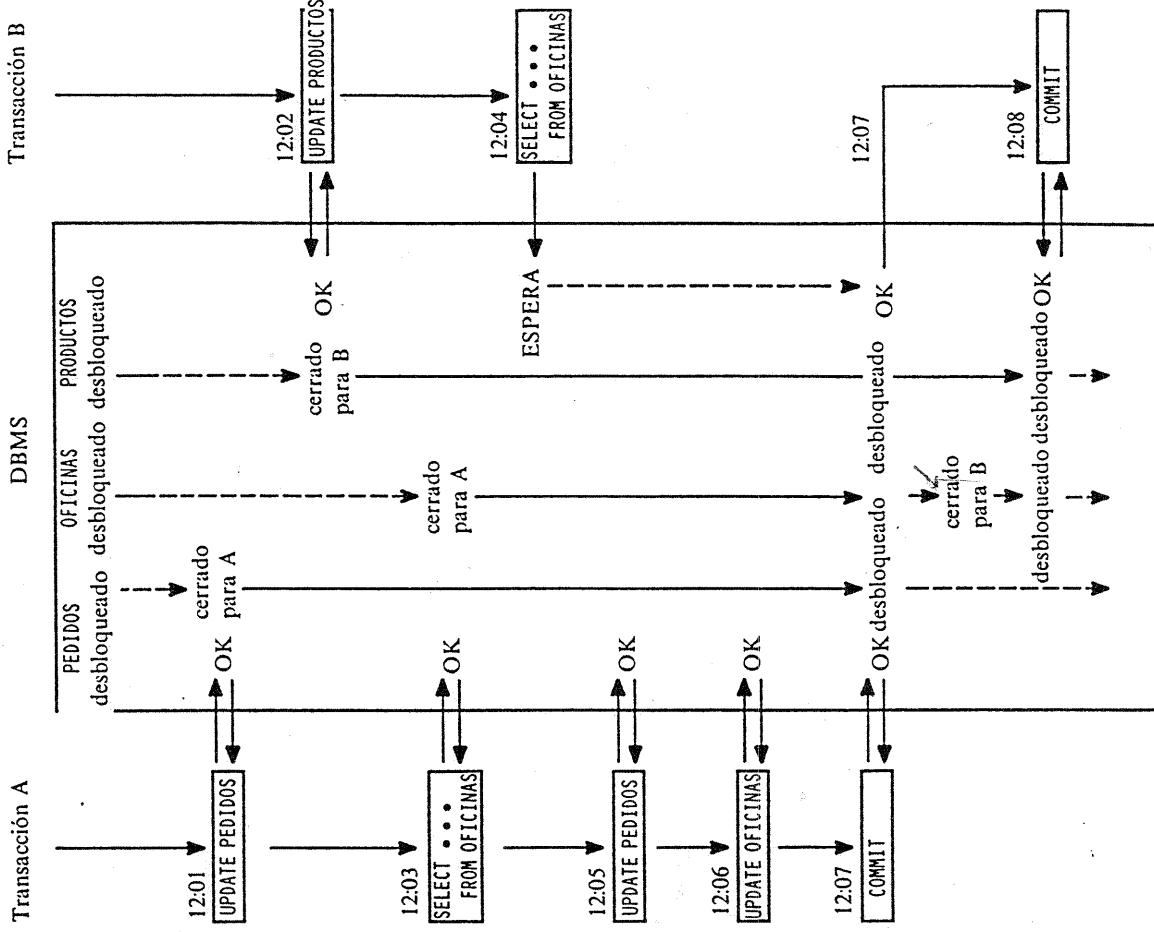


Figura 12.9. Cerramiento con dos transacciones concurrentes.

Esta técnica permite más procesamiento paralelo, pero sigue conduciendo a rendimiento inaceptablemente bajo en aplicaciones tales como entrada de pedidos, en donde muchos usuarios deben compartir el acceso a las mismas tablas. Muchos productos DBMS implementan el cerramiento a nivel de página. En este esquema, el DBMS bloquea los grupos individuales de datos procedentes del disco («páginas») conforme son accedidos por una transacción. Otras transacciones se ven impeditidas de acceder a las páginas bloqueadas, pero pueden acceder (y bloquearlas para ellas mismas) a otras páginas de datos. Habitualmente se utilizan tamaños de páginas de 2KB, 4KB y 16KB. Puesto que una tabla de grandes dimensiones se extenderá a lo largo de cientos o miles de páginas, dos transacciones que tratan de acceder a dos filas diferentes de una tabla accederán generalmente a dos páginas diferentes, permitiendo que las dos transacciones procedan en paralelo.

Unos cuantos productos DBMS han pasado del cerramiento a nivel de fila permitiéndole a los cerramientos a nivel de fila. El cerramiento a nivel de fila permite que dos transacciones concurrentes que acceden a dos filas diferentes de una tabla procedan en paralelo, incluso de disco. Aunque esto puede parecer una posibilidad remota, puede ser un problema real con tablas pequeñas que contienen registros pequeños tales como la tabla OFICINAS de la base de datos ejemplo.

El cerramiento a nivel de fila proporciona un alto grado de ejecución de transacciones en paralelo. Desgraciadamente, el llevar la cuenta de los bloques en las piezas de longitud variable de la base de datos (en otras palabras, en las filas) en lugar de las páginas de tamaño fijo es una tarea mucho más compleja, por lo que el incremento de paralelismo se obtiene a costa de una lógica de cerramiento más sofisticada y de un incremento del recargo. Los vendedores de DBMS que fuerzan el rendimiento del procesamiento de transacciones en línea están sopportando cada vez más el cerramiento a nivel de filas como una opción.

Téóricamente es posible pasar incluso del cerramiento de nivel fila al cerramiento a nivel de datos individuales. En teoría esto proporcionaría incluso más paralelismo que los cerramientos a nivel fila, ya que permitiría acceso concurrente a la misma fila por parte de dos transacciones diferentes, supuesto que accedieran a diferentes grupos de columnas. El recargo en la gestión del cerramiento a nivel de datos, sin embargo, pesa mucho más que sus potenciales ventajas. Ningún DBMS de SQL comercial utiliza cerramiento a nivel de datos. De hecho, el cerramiento es un área de considerable investigación en la tecnología de las bases de datos, y los esquemas de cerramiento utilizados en los productos DBMS comerciales son mucho más sofisticados que el esquema fundamental descrito aquí. Los más sencillos de estos esquemas de cerramiento avanzados, que utilizan cierrres compartidos y exclusivos, se describen en la próxima sección.

Cierres compartidos y exclusivos

Para aumentar el acceso concurrente a una base de datos, la mayoría de los productos DBMS comerciales utilizan un esquema de cerramiento con más de un tipo de cierre. Un esquema que utiliza cierrres compartidos y exclusivos es bastante habitual:

- Un cierre *compartido* se utiliza en el DBMS cuando una transacción desea leer datos de la base de datos. Otra transacción concurrente puede también adquirir un cierre compartido sobre los mismos datos, permitiendo que la otra transacción también lea los datos.
- Un cierre *exclusivo* se utiliza en el DBMS cuando una transacción desea actualizar datos en la base de datos. Cuando una transacción tiene un cierre exclusivo sobre algunos datos, el resto de las transacciones no pueden adquirir ningún tipo de cierre (compartido o exclusivo) sobre los datos.

La Figura 12.10 muestra las reglas para este esquema de cerramiento y las combinaciones permitidas de cierrres que pueden mantener dos transacciones concurrentes. Observe que una transacción puede adquirir un cierre exclusivo solamente si ninguna otra transacción tiene actualmente un cierre compartido o exclusivo sobre los datos. Si una transacción trata de adquirir un cierre no exclusivo sobre los datos, se congela hasta que otras transacciones desbloquean los datos que ella requiere.

La Figura 12.11 muestra cómo el nuevo esquema de cerramiento mejora el acceso concurrente a la base de datos. Los productos DBMS maduros y complejos, tales como DB2, tienen más de dos tipos de cierrres y utilizan diferentes técnicas de cerramiento a diferentes niveles de la base de datos. A pesar de la complejidad creciente, el objetivo del esquema de cerramiento sigue siendo el mismo: impedir la interferencia no deseada entre transacciones mientras se sigue proporcionando el mayor acceso concurrente posible a la base de datos, todo con el mínimo recargo de cerramiento.

Interbloqueos *

Desgraciadamente, el uso de cualquier esquema de cerramiento para soportar transacciones SQL concurrentes conduce a un problema llamado *interbloqueo (deadlock)*. La Figura 12.12 ilustra una situación de interbloqueo. El Programa A actualiza la tabla PEDIDOS, bloqueando por tanto parte de ella. Mientras tanto, el Programa B actualiza la tabla PRODUCTOS bloqueeando parte de ella. Ahora el Programa A trata de actualizar la tabla PRODUCTOS y el Programa B trata de

	Transacción B	
	Despejado	Cierre compartido
Transacción A	Despejado	OK
Cierre compartido	OK	OK
Cierre exclusivo	OK	NO

Figura 12.10. Reglas para cierres compartidos y exclusivos.

actualizar la tabla PEDIDOS, en cada caso tratando de acceder a la parte de la tabla que ha sido previamente cerrada (bloqueada) por el otro programa. Sin intervención externa, cada programa esperará eternamente a que el otro programa cumpla su transacción y desbloquee los datos. Esta situación de la figura es un sencillo interbloqueo entre dos programas, pero pueden ocurrir situaciones más complejas donde tres, cuatro o más programas están en un «ciclo» de cierres, cada uno esperando a datos que están bloqueados por alguno de los otros programas.

Para tratar los interbloqueos, un DBMS incluye típicamente lógica que periódicamente (digamos, una vez por minuto) comprueba los cierres mantenidos por varias transacciones. Cuando detecta un interbloqueo, el DBMS elige arbitrariamente una de las transacciones como «perdedora» del interbloqueo y le da marcha atrás. Esto libera los cierres mantenidos por la transacción perdedora, permitiendo al «ganador» del interbloqueo proseguir. El programa perdedor recibe un código de error informativo que le dice que ha perdido un interbloqueo y que su transacción actual ha sido vuelta atrás. Este esquema para romper interbloques significa que *cualquier* sentencia SQL puede devolver potencialmente un código de error «perdedor de interbloqueo», incluso si no hay nada incorrecto en la sentencia per se. La transacción que intenta la sentencia es vuelta atrás no debido a ningún fallo de ella misma, sino debido a la actividad concurrente de la base de datos. Esto puede parecer injusto, pero en la práctica es mucho mejor que las otras dos alternativas —interbloqueo eterno o corrupción de la base de datos—. Si un error perdedor de interbloqueo ocurre en SQL interactivo, el usuario puede simplemente reescribir la(s) sentencia(s) SQL. En SQL programado, el programa de aplicación debe estar preparado para manejar el código de error perdedor de

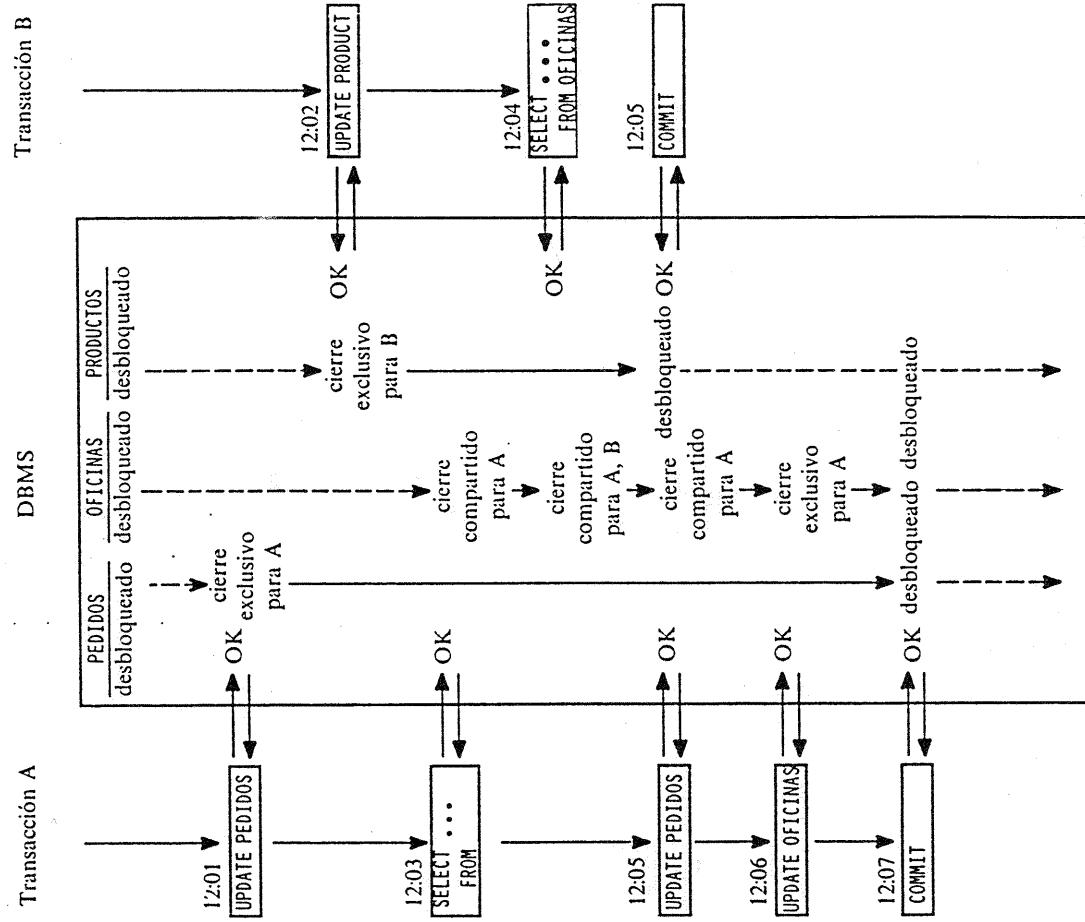


Figura 12.11. Uso de cierres compartidos y exclusivos.

Tipicamente, el programa responderá o bien alertando al usuario o bien reintentando automáticamente la transacción.

La probabilidad de interbloqueos puede reducirse enormemente con un planeamiento cuidadoso de las actualizaciones de la base de datos. Todos los

Técnicas avanzadas de cerramiento *

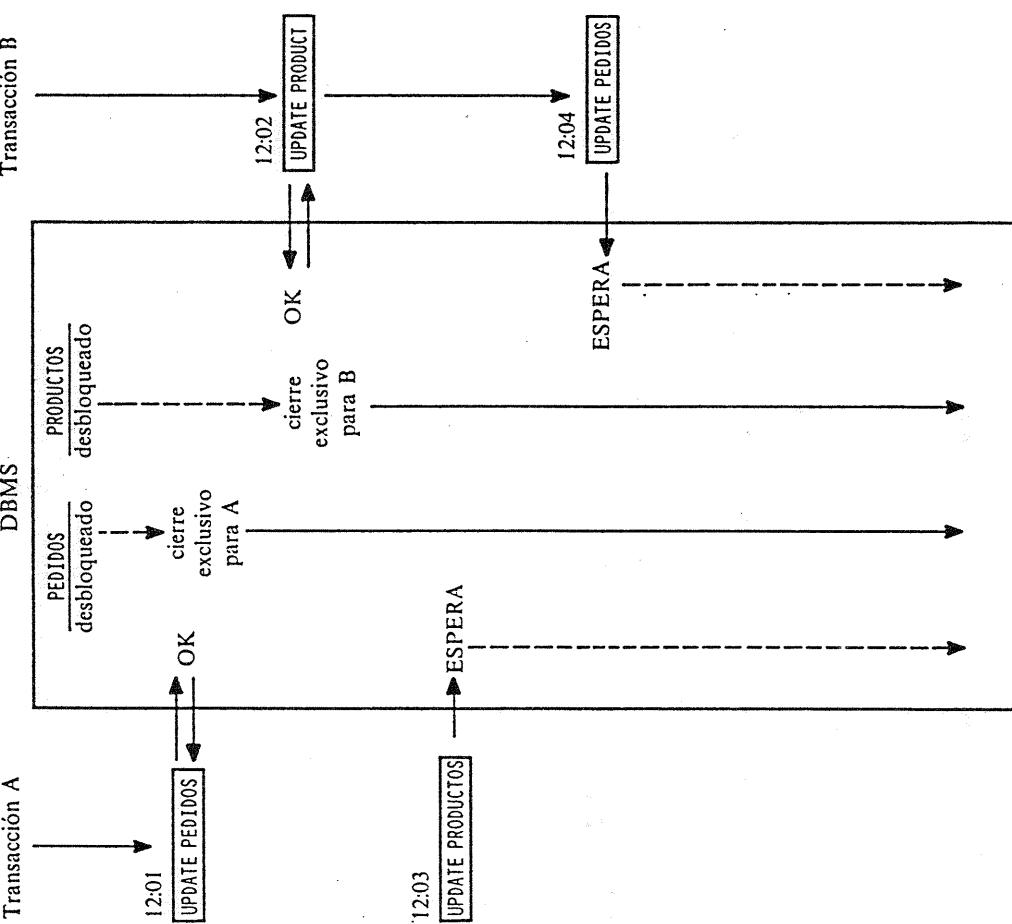


Figura 12.12. Un interbloqueo de transacciones.

programas que actualizan múltiples tablas durante una transacción deberían, si fuera posible, actualizar las tablas en la misma secuencia. Esto permite que los cierres fluyan suavemente a través de las tablas, minimizando la posibilidad de interbloqueos. Además, algunas de las características avanzadas de cerramiento descritas en secciones posteriores de este capítulo pueden ser utilizadas para reducir aún más el número de interbloqueos que aparecen.

Muchos productos de base de datos comerciales ofrecen facilidades avanzadas de cerramiento que van más allá de las proporcionadas por las transacciones SQL estándar. Estas facilidades son, por su naturaleza, no estándar y específicas de cada producto. Sin embargo, varias de ellas, particularmente las disponibles en DB2, han sido implementadas en varios productos SQL comerciales y han logrado el estado de características habituales, si no estándar. Entre ellas se incluyen:

- **Cerramiento explícito.** Un programa puede bloquear explícitamente una tabla entera o alguna otra parte de la base de datos si va a ser repetidamente accedida por el programa.
- **Niveles de aislamiento.** Se puede informar al DBMS que un programa específico no recuperará datos durante una transacción, permitiendo al DBMS liberar los cierres antes de que la transacción finalice.
- **Parámetros de cierre.** El administrador de la base de datos puede ajustar manualmente el tamaño de la «pieza bloqueable» de la base de datos y otros parámetros de cierre para ajustar el rendimiento del esquema.

Cerramiento explícito *. Si una transacción accede repetidamente a una tabla, el recargo de adquirir pequeños cierres sobre muchas partes de la tabla puede ser muy sustancial. Un programa de actualización masiva que recorre todas las filas de la tabla, por ejemplo, cerrará la tabla entera, pieza a pieza, conforme vaya procediendo. Para este tipo de transacción, el programa debería cerrar explícitamente la tabla entera, procesar las actualizaciones y luego desbloquear la tabla. Cerrar la tabla entera tiene tres ventajas:

- Elimina el recargo de cierre fila a fila (o página a página).
- Elimina la posibilidad de que otra transacción cierre parte de la tabla, forzando a la transacción de actualización masiva a esperar.
- Elimina la posibilidad de que otra transacción pueda cerrar parte de la tabla e interbloquear la transacción de actualización masiva, forzándola a ser reiniciada.

Naturalmente, el cierre de la tabla tiene la desventaja de que las otras transacciones que intenten acceder a ella deben esperar mientras la actualización está en proceso. Sin embargo, debido a que la transacción de actualización masiva puede proceder mucho más rápidamente, la productividad global del DBMS puede incrementarse utilizando cierre explícito de la tabla.

En las bases de datos IBM (DB2, SQL/DS y OS/2 Extended Edition), la

sentencia `LOCK TABLE`, mostrada en la Figura 12.13, se utiliza para cerrar explícitamente una tabla entera. Ofrece dos modos de cerramiento:

- El modo **EXCLUSIVE** adquiere un cierre exclusivo sobre la tabla entera. Ninguna otra transacción puede acceder a ninguna parte de la tabla para ningún propósito mientras el cierre se mantenga. Este es el modo que se solicitaría para una transacción de actualización masiva.
- El modo adquiere un cierre compartido sobre la tabla entera. Otras transacciones pueden leer partes de la tabla (es decir, también pueden adquirir cierres compartidos), pero no pueden actualizar ninguna parte de ella. Naturalmente si la transacción que emite la sentencia `LOCK TABLE` actualiza ahora parte de la tabla, seguirá incurriendo en el recargo de adquirir cierres incurriendo en el recargo de adquirir cierres exclusivos sobre las partes de la tabla que actualiza. Este es el modo que se solicitaría si se desea obtener una imagen precisa de la tabla, en un punto particular en el tiempo.

Oracle también soporta una sentencia `LOCK TABLE` al estilo de la de DB2. El mismo efecto puede lograrse en Ingres con una sentencia diferente. Otros varios sistemas de gestión de base de datos, incluyendo SQL Server y SQLBase, no soportan cerramiento explícito en absoluto, eligiendo en su lugar optimizar sus técnicas de cerramiento implícito.

Niveles de aislamiento * Bajo la estricta definición de una transacción SQL, ninguna acción por parte de una transacción en ejecución concurrente se permite que afecte a los datos visibles durante el curso de una transacción dada. Si un programa efectúa una consulta de base de datos durante una transacción, prosigue con otro trabajo y posteriormente efectúa la misma consulta una segunda vez, el mecanismo de transacción SQL garantiza que los datos devueltos por las dos consultas serán idénticos (a menos que la transacción actúe para modificar los datos). Esta capacidad de recuperar fiablemente una fila durante una transacción se denomina modo de *lectura repetible*.

El modo de lectura repetible es muy costoso en términos de cerramiento de la base de datos. Cuando el programa lee cada fila de resultados de la consulta, el DBMS debe bloquear la fila (con un cierre compartido) para impedir que

otras transacciones concurrentes modifiquen la fila. Estos cierres deben mantenerse hasta el final de la transacción, para el caso en que el programa los recupere de nuevo. Generalmente se sabe de antemano que el programa no intentará volver a leer las mismas filas posteriormente en la misma transacción. En este caso, los cierres sobre filas previamente leidas son desaprovechados e innecesarios.

Para eliminar los cierres innecesarios, puede utilizarse un modo de cerramiento alternativo, denominado *estabilidad del cursor*. Este modo funciona así:

- Cuando el programa lee cada fila de resultados de la consulta, adquiere un cierre compartido sobre la fila, lo mismo que con la lectura repetible.
- Si el programa actualiza la fila, el cierre pasa a ser un cierre exclusivo, como es usual, y se mantiene hasta el final de la transacción.
- Si el programa no actualiza la fila y pasa a leer la fila siguiente, el DBMS *desbloquea* la fila inmediatamente, permitiendo que sea accedida por otras transacciones.

El modo de estabilidad de cursor puede incrementar significativamente el acceso concurrente a una base de datos. Debido a que el cierre sobre una fila no modificada se libera tan pronto como el programa pasa de ella, la transacción de otro usuario puede modificar la fila mientras la transacción considerada sigue en proceso. Naturalmente si la transacción de otro usuario se cumplimenta y la considerada relee luego la fila, se verán los cambios que haya producido el otro usuario en la fila. Por tanto el modo de estabilidad de cursor no protege a la transacción considerada frente al problema de datos inconsistentes ilustrado anteriormente en la Figura 12.8. Por esta razón el modo de estabilidad de cursor debe ser utilizado con precaución, y *sólo* cuando la aplicación no esté acumulando totales o efectuando otros cálculos que confíen en un conjunto autoconsistente de filas devueltas por una consulta.

Las bases de datos de los mainframe computadores IBM (CB2 y SQL/DS) ofrecen elección entre los modos de lectura repetible y de estabilidad de cursor. La elección se efectúa durante el proceso de desarrollo del programa, durante el paso BIND descrito en el Capítulo 17. Aunque los modos no forman parte estricta del lenguaje SQL, la elección de modo influye fuertemente en cómo la aplicación puede utilizar los datos recuperados. Por esta razón el modo de lectura repetible es el modo por omisión.

El DBMS Ingres ofrece una capacidad similar a los modos de aislamiento de las bases de datos IBM, pero la proporciona en una forma diferente. Utilizando la sentencia `SET LOCKMODE`, un programa de aplicación puede informar a Ingres de qué tipo de cerramiento utilizar cuando maneje una consulta de base de datos. Las opciones son:

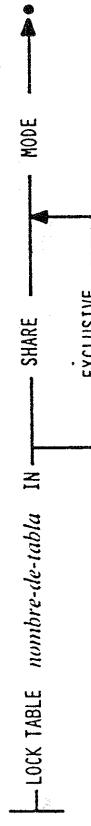


Figura 12.13. Diagrama sintáctico de la sentencia `LOCK TABLE`.

■ **no cerramiento**, que es similar al modo de estabilidad de cursor IBM que acabamos de describir;

■ **cerramiento compartido**, que es similar al modo de lectura repetible de IBM que acabamos de describir,

■ **cerramiento exclusivo**, que proporciona acceso exclusivo a la tabla durante la consulta y ofrece una capacidad semejante a la de la sentencia LOCK TABLE de IBM.

El modo por omisión en Ingres es cerramiento compartido, que corresponde a la lectura repetible en el esquema IBM. Observe, sin embargo, que los modos de cerramiento de Ingres son fijados mediante una sentencia SQL ejecutable. A diferencia de los modos IBM, que deben ser elegidos en tiempo de compilación, los modos Ingres pueden ser elegidos cuando el programa se ejecuta, e incluso pueden ser modificados de una consulta a la siguiente.

Parámetros de cerramiento*. Un DBMS maduro tal como DB2, SQL/DS, Oracle o Ingres emplea técnicas de cerramiento mucho más complejas que las que hemos descrito aquí. El administrador de la base de datos puede mejorar las prestaciones de estos sistemas fijando manualmente parámetros de cerramiento. Los parámetros típicos que pueden ser ajustados incluyen:

■ **Tamaño de cierre**. Algunos productos DBMS ofrecen la opción de cierra a nivel tabla, cierra a nivel página, cierra a nivel fila y otros tamaños de cierre. Dependiendo de la aplicación específica, puede ser adecuado un diferente tamaño de cierre.

■ **Número de cierrres**. Tipicamente un DBMS permite que cada transacción tenga un cierto número finito de cierrres. El administrador de la base de datos puede frecuentemente fijar este límite, elevándolo para permitir transacciones más complejas o rebajándolo para desanimar prematuras escaladas de cierrres.

■ **Escala de cierrres**. Con frecuencia un DBMS «escalará» automáticamente los cierrres, reemplazando muchos cierrres pequeños con un único cierrre mayor (por ejemplo, reemplazando muchos cierrres de nivel página con un cierrre de nivel tabla). El administrador de la base de datos puede tener cierto control sobre este proceso de escalamiento.

■ **Plazo de cierre**. Aun cuando una transacción no se interbloquee con otra transacción, puede tener que esperar mucho tiempo a que otras transacciones liberen sus cierrres. Algunos productos DBMS implementan una característica de *plazo*, en donde una sentencia SQL falla con un código de error si no puede obtener los cierrres que necesita dentro de un cierto plazo de tiempo. El período del plazo puede ser generalmente fijado por el administrador de la base de datos.

Resumen

Este capítulo ha descrito el mecanismo de transacción proporcionado por el lenguaje SQL:

- Una transacción es una unidad lógica de trabajo en una base de datos basada en SQL. Consiste en una secuencia de sentencias SQL que son ejecutadas efectivamente como una sola unidad por el DBMS.
- La sentencia COMMIT señala la terminación con éxito de una transacción, haciendo que todas las modificaciones de la base de datos sean permanentes.
- La sentencia ROLLBACK pide al DBMS que aborde una transacción, dando marcha atrás en todas las modificaciones de la base de datos.
- Las transacciones son la clave para recuperar una base de datos después de un fallo del sistema; sólo las transacciones que estaban cumplimentadas en el momento del fallo permanecen en la base de datos recuperada.
- Las transacciones son la clave para el acceso concurrente en una base de datos multiusuario. Un usuario o programa tiene garantizado que su transacción no se verá interferida por otras transacciones concurrentes.
- Ocasionalmente un conflicto con otra transacción en ejecución concurrente puede hacer que el DBMS dé marcha atrás a una transacción debido a un fallo no propio. Un programa de aplicación que utiliza SQL debe estar preparado para manejar esta situación si se produce.

Estructura de una base de datos

Cuarta parte

Un papel importante de SQL es definir la estructura y organización de una base de datos. Los cuatro capítulos siguientes describen las características SQL que soportan esta función. El Capítulo 13 explica cómo crear una base de datos y sus tablas. El Capítulo 14 describe las vistas, una característica importante de SQL que permite a los usuarios examinar organizaciones alternativas de los datos de la base de datos. Las características de seguridad SQL que protegen los datos almacenados se describen en el Capítulo 15. Finalmente, el Capítulo 16 discute el catálogo del sistema, una colección de tablas de sistema que describe la estructura de una base de datos.

13 Creación de una base de datos

Muchos usuarios SQL no tienen que preocuparse de crear una base de datos; utilizan SQL interactivo o programado para acceder a una base de datos de información corporativa o para acceder a alguna otra base de datos que ya ha sido creada por alguien más. En una típica base de datos corporativa, por ejemplo, el administrador de la base de datos puede dar permisos para recuperar y quizás actualizar los datos almacenados. Sin embargo, el administrador no permitirá crear nuevas bases de datos o modificar la estructura de las tablas existentes.

Conforme usted adquiera más dominio con SQL, probablemente deseará comenzar a crear sus propias tablas privadas para almacenar datos personales tales como resultados de tests técnicos o previsiones de ventas. Si está utilizando una base de datos multusuario, puede desear crear tablas e incluso bases de datos enteras que sean compartidas por otros usuarios. Si está utilizando una base de datos en un computador personal, ciertamente deseará crear sus propias tablas y bases de datos para soportar aplicaciones personales. Este capítulo describe las características del lenguaje SQL que permiten crear bases de datos y tablas y decidir sus estructuras.

El lenguaje de definición de datos

Las sentencias SELECT, INSERT, DELETE, UPDATE, COMMIT y ROLLBACK descritas en la Segunda y Tercera Parte de este libro se refieren todas a la manipulación de los datos de una base de datos. Estas sentencias se denominan colectivamente *Lenguaje de Manipulación de Datos de SQL*, o DML. *Data Manipulation Lan-*

language). Las sentencias DML pueden modificar los datos almacenados en una base de datos, pero no pueden cambiar su estructura. Ninguna de estas sentencias crea o suprime tablas o columnas, por ejemplo.

Los cambios a la estructura de una base de datos son manejados por un conjunto diferente de sentencias SQL, denominadas conjuntamente *Lenguaje de Definición de Datos de SQL*, o DDL (*Data Definition Language*). Utilizando sentencias DDL, se puede:

- definir y crear una nueva tabla
- suprimir una tabla que ya no se necesita
- cambiar la definición de una tabla existente
- definir una tabla virtual (o vista) de datos
- establecer controles de seguridad para una base de datos
- construir un índice para hacer más rápido el acceso a la tabla
- controlar el almacenamiento físico de los datos por parte del DBMS

En su mayor parte, las sentencias DDL aíslan al usuario de los detalles de bajo nivel referentes a cómo los datos están físicamente almacenados en la base de datos. Manipulan objetos abstractos de la base de datos, tales como tablas y columnas. Sin embargo, el DDL no puede evitar completamente las cuestiones referentes al almacenamiento físico, y, por necesidad, las sentencias DDL y las cláusulas que controlan el almacenamiento físico varían de un DBMS a otro.

El núcleo del Lenguaje de Definición de Datos está basado en tres verbos SQL:

- CREATE, que define y crea un objeto de la base de datos
- DROP, que elimina un objeto existente en la base de datos
- ALTER, que modifica la definición de un objeto de la base de datos

En todos los principales productos DBMS basados en SQL, estos tres verbos DDL pueden ser utilizados mientras el DBMS está corriendo. La estructura de la base de datos es por tanto dinámica. El DBMS puede crear, eliminar o alterar la definición de las tablas en la base de datos, por ejemplo, mientras simultáneamente proporciona acceso a la base de datos a sus usuarios. Esta es una ventaja importante de SQL y de las bases de datos relacionales sobre los sistemas anteriores, en los que el DBMS tenía que ser detenido antes de que se pudiera modificar la estructura de la base de datos. Esto significa que una base de datos relacional puede crecer y cambiar fácilmente en el tiempo. El uso de producción de una base de datos puede continuar mientras se le añaden nuevas tablas y aplicaciones.

Aunque el DDL y el DML son dos partes distintas del lenguaje SQL, en la mayoría de los productos DBMS basados en SQL la división es solamente conceptual. Generalmente las sentencias DDL y DML se remiten al DBMS exactamente del mismo modo, y pueden ser libremente entrelazadas en aplicaciones de SQL programado y en sesiones de SQL interactivo. Si un programa o usuario necesita una tabla para almacenar sus resultados temporales, puede crear la tabla, llenarla, manipular los datos y luego eliminarla. De nuevo, ésta es una ventaja importante con respecto a los modelos de datos más primitivos, en donde la estructura de la base de datos quedaba fijada cuando se creaba la base de datos.

Aunque prácticamente todos los productos SQL comerciales soportan el DDL como parte integral del lenguaje SQL, el estándar SQL ANSI/ISO no lo exige. De hecho el estándar actual ANSI/ISO implica una fuerte separación entre el DML y el DDL. Las diferencias entre el estándar ANSI/ISO y el DDL tal como se implementan en los productos SQL más populares se describirán posteriormente en este capítulo.

Creación de una base de datos

En una instalación DBMS grande para mainframe, el administrador de la base de datos es el único responsable de la creación de nuevas bases de datos. En instalaciones DBMS sobre minicomputadores más pequeños, los usuarios individuales pueden permitirse crear sus propias bases de datos personales, pero es mucho más habitual que las bases de datos sean creadas centralizadamente y luego accedidas por los usuarios individuales. Si usted está utilizando un DBMS en computador personal, será probablemente a la vez administrador y usuario de la base de datos, y tendrá que crear personalmente las bases de datos que utilice.

El estándar ANSI/ISO no especifica cómo se crean las bases de datos, y cada producto DBMS adopta un planteamiento ligeramente diferente. Las técnicas utilizadas por algunos de los productos SQL más importantes ilustran las diferencias:

- Oracle crea una base de datos como parte del proceso de instalación software Oracle. En su mayor parte, las tablas de usuario se colocan siempre en esta base de datos única global.
- Ingres incluye un programa de utilidad especial, llamado CREATEDB, que crea una nueva base de datos Ingres. Un programa adicional, DESTROYDB, suprime una base de datos que ya no es necesaria.
- SQL Server incluye una sentencia CREATE DATABASE como parte de su lenguaje de definición de datos. Una sentencia adicional DROP DATABASE destruye bases de

datos creadas previamente. Estas sentencias pueden ser utilizadas con SQL interactivo o programado.

- OS/2 Extended Edition proporciona una utilidad CREATE DATABASE para crear bases de datos y una utilidad DROP DATABASE para eliminar bases de datos. También proporciona herramientas a función especiales que pueden ser utilizadas mediante programas de aplicación para crear y suprimir bases de datos.
- SQLBase utiliza la orden de MS-DOS COPY para crear una nueva base de datos. El usuario simplemente hace una copia duplicada de una plantilla de base de datos vacía suministrada con el software SQLBase. La base de datos puede ser suprimida posteriormente con la orden de MS-DOS DEL.

A causa de estos planteamientos diferentes, es preciso consultar los manuales de cada DBMS particular antes de proceder a crear una base de datos. Tras crear una base de datos vacía, el siguiente paso es rellenarla con tablas, tal como se describe en la sección siguiente.

Definiciones de tablas

La estructura más importante de una base de datos relacional es la tabla. En una base de datos multiusuario, las tablas principales son típicamente creadas una vez por el administrador de la base de datos y utilizadas luego día tras día. Conforme se utiliza la base de datos con frecuencia se encontrará conveniente definir tablas propias para almacenar datos personales o datos extraídos de otras tablas. Estas tablas pueden ser temporales, que duran únicamente una sesión SQL interactiva, o más permanentes, con una duración de semanas o meses. En una base de datos sobre computador personal, la estructura de las tablas es incluso más fluida. Puesto que uno es a la vez usuario y administrador de la base de datos, puede crear o destruir las tablas conforme a las propias necesidades, sin preocuparse del resto de usuarios.

Creación de una tabla (CREATE TABLE)

La sentencia CREATE TABLE, mostrada en la Figura 13.1, define una nueva tabla en la base de datos y la prepara para aceptar datos. Las diferentes cláusulas de la sentencia especifican los elementos de la definición de la tabla. El diagrama sintáctico de la sentencia parece complejo, ya que hay muchas partes de la definición que especificar y muchas opciones para cada elemento. En la práctica, la creación de una nueva tabla es relativamente sencilla.

Cuando se ejecuta una sentencia CREATE TABLE, uno se convierte en el propietario de la tabla recién creada, a la cual se le da el nombre especificado en la

sentencia. El nombre de la tabla debe ser un nombre SQL legal, y no debe entrar en conflicto con el nombre de alguna otra tabla ya existente. La tabla recién creada está vacía, pero el DBMS la prepara para aceptar datos añadidos con la sentencia INSERT.

Definiciones de columnas. Las columnas de la tabla recién creada se definen en el cuerpo de la sentencia CREATE TABLE. Las definiciones de columnas

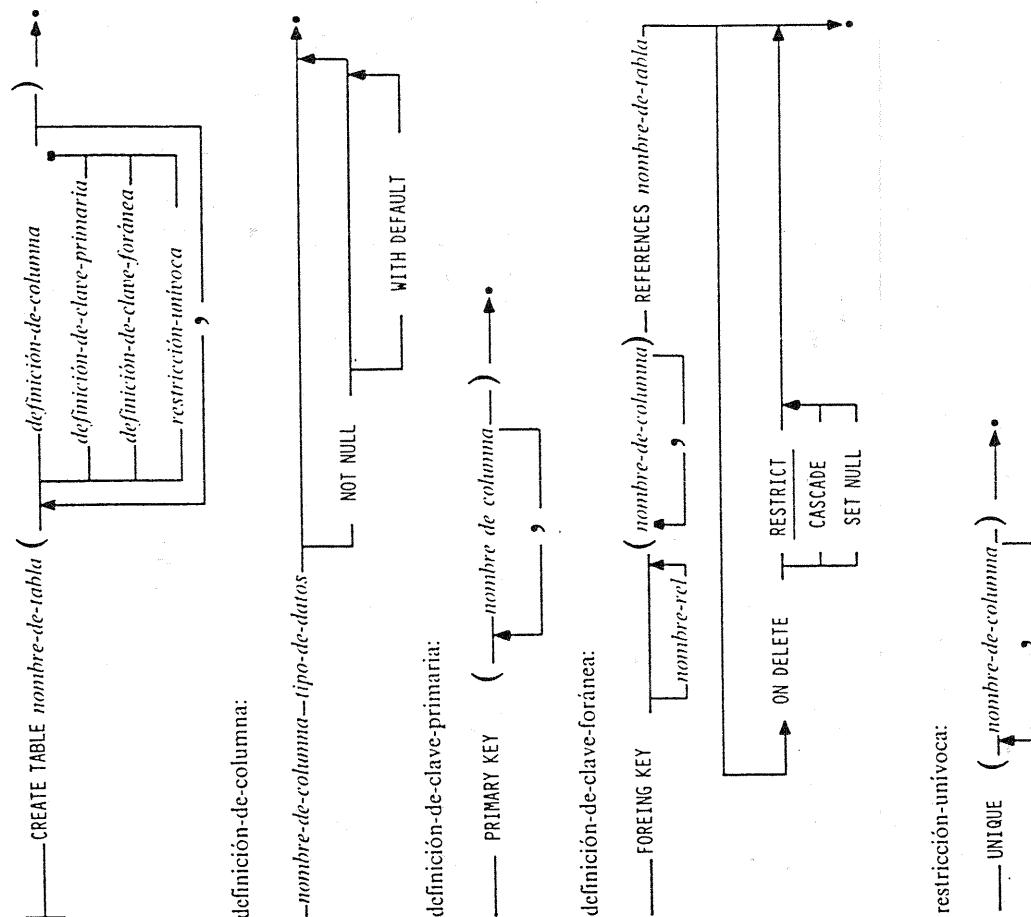


Figura 13.1. Diagrama sintáctico básico de la sentencia CREATE TABLE.

aparecen en una lista separada por comas e incluida entre paréntesis. El orden de las definiciones de las columnas determina el orden de izquierda a derecha de las columnas en la tabla. Cada definición especifica:

- El *nombre* de la columna, que se utiliza para referirse a la columna en sentencias SQL. Cada columna de la tabla debe tener un nombre único, pero los nombres pueden ser iguales a los de las columnas de otras tablas.
- El *tipo de datos* de la columna, que identifica la clase de datos que la columna almacena. Los tipos de datos fueron discutidos en el Capítulo 5. Algunos tipos de datos, tales como VARCHAR y DECIMAL, requieren información adicional, como la longitud o número de lugares decimales de los datos. Esta información adicional se incluye entre paréntesis a continuación de la palabra clave que especifica el tipo de datos.
- Si la columna contiene *datos requeridos*. La cláusula NOT NULL impide que aparezcan valores NULL en la columna; en caso contrario se permiten los valores NULL.
- Un valor *por omisión* opcional para la columna. El DBMS utiliza este valor cuando una sentencia INSERT aplicada a la tabla no especifica un valor para la columna.

He aquí algunas sentencias CREATE TABLE sencillas para las tablas de la base de datos ejemplo:

Define la tabla OFICINAS y sus columnas.

```
CREATE TABLE OFICINAS
  (OFICINA INTEGER NOT NULL,
   CIUDAD VARCHAR(15) NOT NULL,
   REGION VARCHAR(10) NOT NULL,
   DIR INTEGER,
   OBJETIVO MONEY,
   VENTAS MONEY NOT NULL)
```

Define la tabla PEDIDOS y sus columnas.

```
CREATE TABLE PEDIDOS
  (NUM_PEDIDO INTEGER NOT NULL,
   FECHA_PEDIDO DATE NOT NULL,
   CLIE INTEGER NOT NULL,
   REP INTEGER,
   FAB CHAR(3) NOT NULL,
   PRODUCTO CHAR(5) NOT NULL,
   CANT INTEGER NOT NULL,
   IMPORTE MONEY NOT NULL)
```

La sentencia CREATE TABLE varía ligeramente de un producto DBMS a otro, ya que cada DBMS soporta su propio conjunto de tipos de datos y utiliza sus propias palabras claves para identificarlos en las definiciones de columna. Además, Sybase y SQL Server difieren mucho de otros productos DBMS y del estándar ANSI/ISO en su manejo de los valores NULL. El estándar especifica que una columna puede contener valores NULL a menos que específicamente se declare NOT NULL. Sybase y SQL Server utilizan el convenio opuesto, suponiendo que los valores NULL no son permitidos a menos que la columna se declare explícitamente NULL.

Valores por omisión. Tanto el estándar ANSI/ISO como los productos SQL de IBM soportan valores por omisión para las columnas, pero los soporan de maneras diferentes. El estándar ANSI/ISO permite especificar un valor por omisión para cada columna. Por ejemplo, he aquí una sentencia CREATE TABLE ANSI/ISO para la tabla OFICINAS que especifica valores por omisión:

Define la tabla OFICINAS con valores por omisión (sintaxis ANSI/ISO).

```
CREATE TABLE OFICINAS
  (OFICINA INTEGER NOT NULL,
   CIUDAD VARCHAR(15) NOT NULL,
   REGION VARCHAR(10) NOT NULL DEFAULT 'Este',
   DIR INTEGER DEFAULT 106,
   OBJETIVO MONEY DEFAULT NULL,
   VENTAS MONEY NOT NULL DEFAULT 0.00)
```

Con esta definición de tabla, sólo es necesario especificar el número de oficina y la ciudad cuando se inserte una nueva oficina. La región por omisión es Este, el director de la oficina es Sam Clark (el empleado número 106), las ventas son cero y el objetivo es NULL. Observe que el objetivo tomaría el valor por omisión NULL incluso sin la especificación DEFAULT NULL.

Los productos SQL de IBM no permiten especificar un valor por omisión diferente para cada columna. En su lugar, proporcionan un valor por omisión específico para cada tipo de dato soportado. El valor por omisión para los datos numéricos es cero (0), el valor por omisión para los datos VARCHAR es la cadena vacía (''), el valor por omisión para los datos CHAR es todos blancos y el valor por omisión para los datos de fecha y hora es la fecha y hora actual. He aquí una redefinición de la tabla OFICINAS, utilizando la sintaxis IBM para valores por omisión:

Define la tabla OFICINAS con valores por omisión (sintaxis IBM).

```
CREATE TABLE OFICINAS
  (OFICINA INTEGER NOT NULL,
   CIUDAD VARCHAR(15) NOT NULL,
   REGION VARCHAR(10) NOT NULL WITH DEFAULT '',
   DIR INTEGER WITH DEFAULT 106,
   OBJETIVO MONEY,
   VENTAS MONEY NOT NULL)
```

Con esta definición de la tabla, si se inserta una nueva oficina y solamente se especifica el número de la oficina y la ciudad, la columna REGION se inicializará a la cadena vacía, las columnas DIR y VENTAS tendrán valores cero y la columna OBJETIVO será NULL.

Definiciones de clave primaria y foránea. Además de la definición de las columnas de una tabla, la sentencia CREATE TABLE identifica la clave primaria de la tabla y las relaciones de la tabla con otras tablas de la base de datos. Las cláusulas PRIMARY KEY y FOREIGN KEY manejan estas funciones. Estas cláusulas son soportadas por las bases de datos IBM (DB2, SQL/DS y OS/2 Extended Edition) y han sido añadidas a la especificación ANSI/ISO original. Sin embargo, la mayoría de los principales productos SQL no las soportan aún.

La cláusula PRIMARY KEY especifica la columna o columnas que forman la clave primaria de la tabla. Recuerde del Capítulo 4 que esta columna (o combinación de columnas) sirve como identificador único para cada fila de la tabla. El DBMS requiere automáticamente que el valor de clave primaria sea único en cada fila de la tabla. Además, la definición de columna para todas las columnas que forman la clave primaria debe especificar que la columna es NOT NULL.

La cláusula FOREIGN KEY especifica una clave foránea en la tabla y la relación que crea con otra tabla (padre) de la base de datos. La cláusula especifica:

- La columna o columnas que forman la clave foránea, todas las cuales son columnas de la tabla que está siendo creada.
- La tabla que es referenciada por la clave foránea. Esta es la tabla padre en la relación; la tabla que está siendo definida es el hijo.
- Un nombre opcional para la relación. El nombre no se utiliza en ninguna sentencia SQL, pero puede aparecer en mensajes de error y es necesario si se desea poder suprimir la clave foránea posteriormente.
- Una regla de supresión opcional para la relación (RESTRICT, CASCADE o SET NULL, tal como se describió en el Capítulo 11). Si no se especifica regla de supresión, se utiliza la regla RESTRICT.

He aquí una sentencia CREATE TABLE ampliada para la tabla PEDIDOS, que incluye una definición de las claves primaria y foránea que contiene:

```
Define la tabla PEDIDOS con claves primaria y foránea.
CREATE TABLE PEDIDOS
  (NUM_PEDIDO INTEGER NOT NULL,
   FECHA_PEDIDO DATE NOT NULL,
   CLIE INTEGER NOT NULL,
   REP INTEGER,
   FAB CHAR (3) NOT NULL,
```

```
  PRODUCT  CHAR (5) NOT NULL,
  CANT  INTEGER NOT NULL,
  IMPORTE MONEY NOT NULL,
  PRIMARY KEY  (NUM_PEDIDO),
  FOREIGN KEY  PLACEDBY (CLIE)
  REFERENCES  CLIENTES
  ON DELETE  CASCADE,
  FOREIGN KEY  TOMADOPOR (REP)
  REFERENCES  REVENTAS
  ON DELETE  SET NULL,
  FOREIGN KEY  ESPOR (FAB, PRODUCTO)
  REFERENCES  PRODUCTOS
  ON DELETE  RESTRICT)
```

La Figura 13.2 muestra las tres relaciones creadas por esta sentencia y los nombres que se les asigna. En general es buena idea asignar un nombre de relación, ya que ayuda a clarificar la relación creada por la clave foránea. Por ejemplo, cada pedido fue remitido por el cliente cuyo número aparece en la columna CLIE de la tabla PEDIDOS. La relación creada por esta columna ha recibido el nombre de PEDIDOPOR.

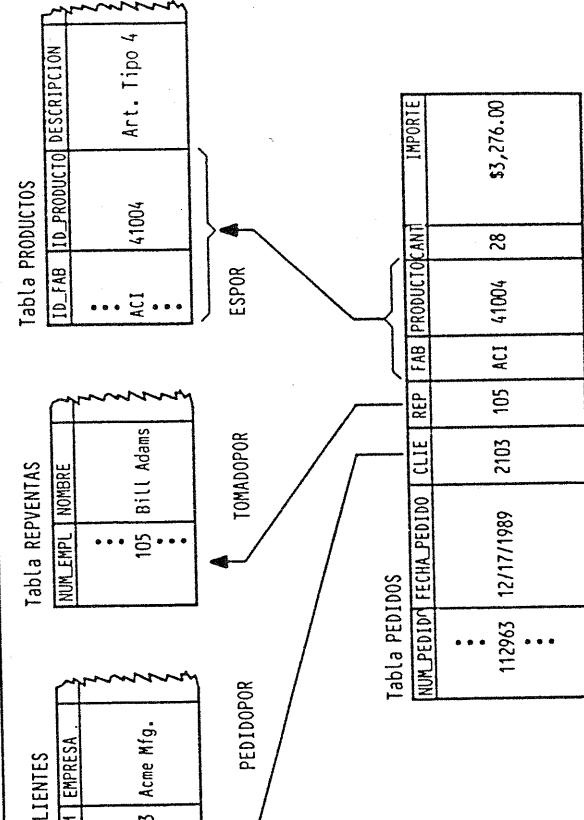


Figura 13.2. Nombre de relaciones en la sentencia CREATE TABLE.

Cuando el DBMS procesa la sentencia CREATE TABLE, compara cada definición de clave foránea con la definición de la tabla que referencia. El DBMS se asegura que la clave foránea y la clave primaria de la tabla referenciada concuerden en el número de columnas que contienen y en sus tipos de datos. La tabla referenciada debe estar ya definida en la base de datos para que esta comparación tenga éxito.

Si dos o más tablas forman un ciclo referencial (como ocurre con las tablas OFICINAS y REPVENTAS), no se puede definir la clave foránea para la tabla que se crea en primer lugar, ya que la tabla referenciada aún no existe. En su lugar, debe crearse la tabla sin definición de clave foránea y añadir la clave foránea posteriormente utilizando la sentencia ALTER TABLE.

Restricciones de unicidad. El estándar ANSI/ISO especifica que las restricciones de unicidad también se definen en la sentencia CREATE TABLE, utilizando la cláusula UNIQUE mostrada en la Figura 13.1. He aquí una sentencia CREATE TABLE para la tabla OFICINAS, modificada para exigir valores de CIUDAD únicos:

Define la tabla OFICINAS con una restricción de unicidad.

```
CREATE TABLE OFICINAS
  (OFICINA INTEGER NOT NULL,
   CIUDAD VARCHAR(15) NOT NULL,
   REGION VARCHAR(10) NOT NULL,
   DIR INTEGER,
   OBJETIVO MONEY,
   VENTAS MONEY NOT NULL
  PRIMARY KEY (OFICINA),
  FOREIGN KEY HAYDIR (DIR)
  REFERENCES REPVENTAS
  ON DELETE SET NULL,
  UNIQUE (CIUDAD))
```

Desgraciadamente, la cláusula UNIQUE no es utilizada por ninguno de los DBMS basados en SQL más importante. En vez de ello, todos los productos SQL populares siguen la práctica de DB2 y hacen a las restricciones de unicidad formar parte de la sentencia CREATE INDEX, que se describe posteriormente en este capítulo.

Si una clave primaria, una clave foránea o una restricción de unicidad afecta a una sola columna, el estándar ANSI/ISO permite una forma «abreviada» de la definición. La clave primaria, la clave foránea o la restricción de unicidad se añaden simplemente al final de la definición de la columna, tal como se muestra en este ejemplo:

Define la tabla OFICINAS con una restricción de unicidad (sintaxis ANSI/ISO).

```
CREATE TABLE OFICINAS
  (OFICINA INTEGER NOT NULL PRIMARY KEY,
   CIUDAD VARCHAR(15) NOT NULL UNIQUE,
   REGION VARCHAR(10) NOT NULL,
   DIR INTEGER REFERENCES REPVENTAS,
   OBJETIVO MONEY,
   VENTAS MONEY NOT NULL)
```

Aunque esta abreviatura forma parte del estándar ANSI/ISO, no es aceptada por los productos SQL de IBM, que han sido los pioneros en el soporte de claves primarias y foráneas, ni por ningún otro DBMS comercial importante hoy día.

Definición de almacenamiento físico. La sentencia CREATE TABLE incluye típicamente una o más cláusulas opcionales que especifican características de almacenamiento físico para la tabla. Generalmente estas cláusulas sólo las utiliza el administrador de la base de datos para optimizar el rendimiento de una base de datos de producción. Por su naturaleza estas cláusulas son muy específicas de un DBMS particular. Aunque son de poco interés práctico para la mayoría de los usuarios de SQL, las diferentes estructuras de almacenamiento físico proporcionadas por los diferentes productos DBMS ilustran sus diferentes aplicaciones y niveles de sofisticación.

La mayoría de las bases de datos sobre computadores personales disponen de mecanismos de almacenamiento físico muy simples. El DBMS dBASE IV, por ejemplo, utiliza un archivo MS-DOS individual para almacenar cada tabla de datos. SQLBase utiliza un solo archivo para almacenar cada base de datos. En cada caso el uso de archivos MS-DOS limita el tamaño de una tabla o una base de datos. También requieren que la tabla o la base de datos entera sean almacenadas en un único volumen de disco físico.

Las bases de datos multiusuario proporcionan típicamente esquemas de almacenamiento físico más sofisticados para mejorar el rendimiento de las bases de datos. Por ejemplo, Ingres permite que el administrador defina múltiples *ubicaciones* nombradas, que son directorios físicos en donde los datos de la base pueden ser almacenados. Las ubicaciones pueden extenderse a través de múltiples volúmenes para aprovechar las operaciones paralelas de entrada/salida sobre los discos. Opcionalmente se puede especificar una o más ubicaciones para la tabla en la sentencia CREATE TABLE de Ingres:

```
CREATE TABLE OFICINAS ( definición-de-tabla )
  WITH LOCATION = (AREA1, AREA2, AREA3)
```

Especificando múltiples ubicaciones, se pueden dispensar los contenidos de una tabla a través de varios volúmenes de disco con objeto de ampliar el acceso en paralelo a la tabla.

SQL Server ofrece un planteamiento análogo, que permite al administrador de la base de datos especificar múltiples *archivos de bases de datos* nominados que son utilizados para almacenar datos. La sentencia CREATE DATABASE de SQL Server puede especificar uno o más archivos de bases de datos:

```
CREATE DATABASE DATOSOP
ON BDARCH1, BDARCH2, BDARCH3
```

El planteamiento SQL Server permite que los contenidos de una base de datos sean distribuidos a través de varios volúmenes de discos, pero proporciona un menor control tabla a tabla que el esquema de Ingres.

Finalmente, DB2 ofrece un esquema muy complejo para gestionar el almacenamiento físico. El conjunto completo de tablas gestionado por un subsistema DB2 puede dividirse en dos o más *bases de datos* DB2. Observe que este uso del término «base de datos» es confuso, ya que una base de datos DB2 es una estructura interna utilizada para gestionar el almacenamiento físico de los datos, y no una estructura visible al usuario. Cada una de estas bases de datos se subdivide a su vez en *espacios de tablas* y *espacios de índices* en DB2, entidades de almacenamiento lógico que almacenan tablas e índices, respectivamente. Los espacios de tablas y los espacios de índices pueden ser asignados a su vez en un *grupo de almacenamiento*, una entidad de almacenamiento físico mantenida por DB2. Finalmente, un grupo de almacenamiento consta de uno o más volúmenes de memoria de acceso directo sobre el maxicomputador. Las bases de datos, los espacios de tablas y los grupos de almacenamiento se gestionan utilizando las sentencias CREATE DATABASE, CREATE TABLESPACE y CREATE STOFGROUP en el DDL de DB2. Cuando se crea una tabla en DB2, puede asignársele opcionalmente a un espacio de tablas específico:

```
CREATE TABLE OFICINAS (definición-de-tabla)
IN BADMIN-ESPACIO
```

Este esquema permite un control preciso, tabla a tabla, con respecto a la ubicación de los datos. DB2 permite que el administrador de la base de datos recupere espacios de tablas individualmente tras un fallo del sistema, y permite también que el administrador inicie y detenga el acceso a bases de datos individuales utilizando sentencias especiales de administración de la base de datos. Por tanto estas estructuras de DB2 funcionan juntas proporcionando una facilidad potente aunque compleja para gestionar el almacenamiento físico de una base de datos DB2.

Eliminación de una tabla (DROP TABLE)

Con el tiempo la estructura de una base de datos crecerá y se modificará. Nuevas tablas serán creadas para representar nuevas entidades, y algunas tablas



Figura 13.3. Diagrama simbólico de la sentencia DROP TABLE.

antiguas ya no serán necesarias. Se puede eliminar de la base de datos una tabla que ya no es necesaria con la sentencia DROP TABLE, mostrada en la Figura 13.3. El nombre de la tabla en la sentencia identifica la tabla a eliminar. Normalmente se eliminará una de las tablas propias del usuario y se utilizará un nombre de tabla no cualificado. Con el permiso adecuado, también se puede eliminar una tabla propiedad de otro usuario especificando un nombre de tabla cualificado. He aquí algunos ejemplos de la sentencia DROP TABLE:

La tabla CLIENTES ha sido sustituida por dos nuevas tablas, INFO_CLIENTE e INFO_CONTA, y ya no se necesita.

```
DROP TABLE CLIENTES
```

Sam te da permiso para eliminar su tabla, llamada CUMPLEAÑOS.

```
DROP TABLE SAM.CUMPLEAÑOS
```

Cuando la sentencia DROP TABLE suprime una tabla de la base de datos, su definición y todos sus contenidos se pierden. No hay manera de recuperar los datos, y habría que utilizar una nueva sentencia CREATE TABLE para volver a crear la definición de la tabla. Debido a sus serias consecuencias, debe utilizarse la sentencia DROP TABLE con mucho cuidado.

Modificación de una definición de tabla (ALTER TABLE)

Después que una tabla ha sido utilizada durante algún tiempo, los usuarios suelen descubrir que desean almacenar información adicional con respecto a las entidades representadas en la tabla. En la base de datos ejemplo, por ejemplo, se podría desechar:

- Añadir el nombre y el número de teléfono de una persona de contacto a cada fila de la tabla CLIENTES, al comenzar a utilizarlo para contactar.
- Añadir una columna de nivel de inventario mínimo a la tabla PRODUCTOS, para que la base de datos pueda alertar automáticamente cuando el stock de un producto particular esté bajo.

- Hacer de la primaria sea
 - Suprimir la d PEDIDOS con foránea que e INFO_CONTA.
 - Cada uno c sentencia ALTER DROP TABLE, ALI permiso adecua cado y alterar l figura, la senten
 - añadir una d
 - definir una c
 - definir una n
 - eliminar una
- Muchos pr alteran otras c

Lista el nombre, las ventas, la oficina, y las ventas de oficina de todo el mundo.

```
SELECT NOMBRE, ADMIN_PP.REVENTAS, OFICINA, ADMIN_PP.OFICINAS
FROM ADMIN_PP.REVENTAS, ADMIN_PP.OFICINAS
```

Para tratar este problema, los productos SQL de IBM proporcionan una capacidad de *sinónimos*. Un sinónimo es un nombre definido para representar el nombre de alguna otra tabla. Un sinónimo se crea utilizando la sentencia CREATE SYNONYM. Si usted fuera el usuario George de la Figura 13.5, por ejemplo, he aquí un par de sentencias CREATE SYNONYM que podría utilizar:

Crea sinónimos para dos tablas propiedad de otro usuario.

```
CREATE SYNONYM REPS
FOR ADMIN_PP.REVENTAS

CREATE SYNONYM OFICINAS
FOR ADMIN_PP.OFICINAS
```

Una vez que se ha definido un sinónimo, puede utilizarse lo mismo que un nombre de tabla en las consultas SQL. La consulta anterior pasa a ser por tanto:

```
SELECT NAME, REP.VENTAS, OFICINAS, OFICINAS.VENTAS
FROM REPS, OFICINAS
```

El uso de sinónimos no cambia el significado de la consulta, y sigue siendo necesario tener permiso para acceder a las tablas de los demás usuarios. No obstante, los sinónimos simplifican las sentencias SQL utilizadas y aparentan que las tablas son propiedad del usuario. Si posteriormente se decide que ya no se desea utilizar los sinónimos, pueden suprimirse con la sentencia DROP SYNONYM:

Suprime los sinónimos creados con anterioridad.

```
DROP SYNONYM REPS
DROP SYNONYM OFICINAS
```

Los sinónimos están disponibles en las bases de datos IBM (DB2, SQL/DS y OS/2 Extended Edition) y en Oracle, SQLBase e Informix. No están especificadas por el estándar SQL ANSI/ISO.

Indices (CREATE/DROP INDEX)

Una de las estructuras de almacenamiento físico proporcionadas por la mayoría de los sistemas de gestión de base de datos basados en SQL es el *índice*. Un

Figura 13.6

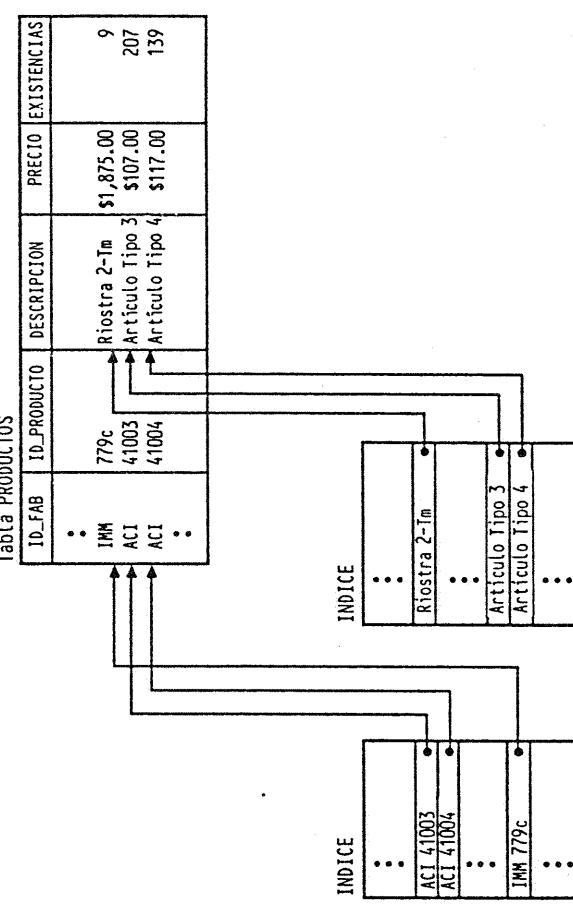


Figura 13.6. Dos índices para la tabla PRODUCTOS.

índice es una estructura que proporciona un acceso rápido a las filas de una tabla en base a los valores de una o más columnas. La Figura 13.6 muestra la tabla PRODUCTOS y dos índices que han sido creados para ella. Uno de los índices proporciona acceso basándose en la columna DESCRIPCION. Los otros proporcionan acceso en base a la clave primaria de la tabla, que es una combinación de las columnas ID_FAB e ID_PRODUCTO.

El DBMS utiliza el índice del mismo modo que usted utilizaría el índice de un libro. El índice almacena valores y punteros a las filas en donde los valores se producen. En el índice los valores de datos están dispuestos en orden ascendente o descendente, de modo que el DBMS puede buscar rápidamente el índice para encontrar un valor particular. A continuación puede seguir el puntero para localizar la fila que contiene el valor.

La presencia o ausencia de un índice es completamente transparente al usuario de SQL que accede a una tabla. Por ejemplo, consideremos esta sentencia SELECT:

Halla la cantidad y precio de los artículos de tipo 4.

```
SELECT EXISTENCIAS, PRECIO
FROM PRODUCTOS
WHERE DESCRIPCION = 'Artículo tipo 4'
```

Figura 13.4. 1

La sentencia no dice si existe un índice sobre la columna DESCRIPCION o no, y el DBMS llevará a cabo la consulta exista o no exista.

Si no hubiera índice para la columna DESCRIPCION, el DBMS se vería forzado a procesar la consulta recorriendo secuencialmente la tabla PRODUCTOS, fila a fila, examinando la columna DESCRIPCION en cada fila. Para asegurarse que ha encontrado todas las filas que satisfacen la condición de búsqueda, tendría que examinar *todas y cada una* de las filas de la tabla. Para una tabla grande con miles o millones de filas, la exploración de la tabla llevaría minutos u horas.

Con un índice para la columna DESCRIPCION, el DBMS puede localizar los datos solicitados con mucho menos esfuerzo. Examina el índice para encontrar el valor solicitado («Los artículos de tipo 4») y luego sigue al puntero para encontrar la fila o filas solicitadas de la tabla. La búsqueda con índices es muy rápida ya que el índice está ordenado y sus filas son muy pequeñas. Pasar del índice a la fila es también muy rápido, ya que el índice informa al DBMS de en qué lugar el disco está localizada la fila.

Como muestra el ejemplo, la ventaja de tener un índice es la de acelerar enormemente la ejecución de las sentencias SQL con condiciones de búsqueda que se refieren a las columnas indexadas. Una desventaja de tener un índice es que consume espacio en disco adicional. Otra desventaja es que el índice debe ser actualizado cada vez que se añade una fila a la tabla y cada vez que la columna indexada se actualiza en una fila existente. Esto impone recargos adicionales sobre las sentencias INSERT y UPDATE para la tabla.

En general es buena idea crear un índice para las columnas que son utilizadas frecuentemente en condiciones de búsqueda. La indexación es también más apropiada cuando las consultas de una tabla son más frecuentes que las inserciones y las actualizaciones. El DBMS *siempre* establece un índice para la clave primaria de una tabla, ya que presupone que el acceso a la tabla se efectuará más frecuentemente a través de la clave primaria. En la base de datos ejemplo, estas columnas son buenas candidatas para índices adicionales:

- La columna EMPRESA en la tabla CLIENTES debería ser indexada si los datos del cliente se recuperan frecuentemente por el nombre de la empresa.
- La columna NOMBRE en la tabla REPENTAS debería ser indexada si los datos referentes a vendedores se recuperan frecuentemente por el nombre del vendedor.
- La columna REP en la tabla PEDIDOS debería ser indexada si los pedidos se recuperan frecuentemente basándose en el vendedor que los recibió.
- La columna CLIE en la tabla de PEDIDOS debería ser analógicamente indexada si los pedidos se recuperan frecuentemente basándose en el cliente que los remitió.
- Las columnas FAB y PRODUCTO, juntas, en la tabla PEDIDOS deberían ser indexadas si los pedidos se recuperan frecuentemente en base al producto solicitado.

Para crear un índice, se utiliza la sentencia CREATE INDEX, mostrada en la Figura 13.7. La sentencia asigna un nombre al índice y especifica la tabla para la cual se crea el índice. La sentencia también especifica la columna o columnas a indexar y si deberían ser indexadas en orden ascendente o descendente. La palabra clave UNIQUE se utiliza en la versión DB2 de la sentencia para especificar que la columna que está siendo indexada debe contener valores únicos. Como se describió anteriormente, el estándar ANSI/ISO hace de la cláusula UNIQUE una parte de la sentencia CREATE TABLE, pero DB2 la asocia con el índice.

He aquí un ejemplo de la sentencia CREATE INDEX que construye un índice para la tabla PEDIDOS basado en las columnas FAB y PRODUCTO y que requiere que las combinaciones de columnas tengan un valor único:

Crea un índice para la tabla pedidos.

```
CREATE UNIQUE INDEX PED_PROD_IDX  
ON PEDIDOS (FAB, PRODUCTO)
```

Además de las cláusulas mostradas en el ejemplo, la sentencia CREATE INDEX incluye casi siempre cláusulas adicionales específicas de cada DBMS que especifican la ubicación en disco para el índice y parámetros para mejora del rendimiento tales como el tamaño de las páginas de índice y el porcentaje de espacio libre que el índice debería permitir para filas adicionales.

Si se crea un índice para una tabla y posteriormente se decide que ya no es necesario, la sentencia DROP INDEX suprime el índice de la base de datos. La sentencia siguiente suprime el índice creado en el ejemplo anterior:

Suprime el índice creado anteriormente.

```
DROP INDEX PED_PROD_IDX
```

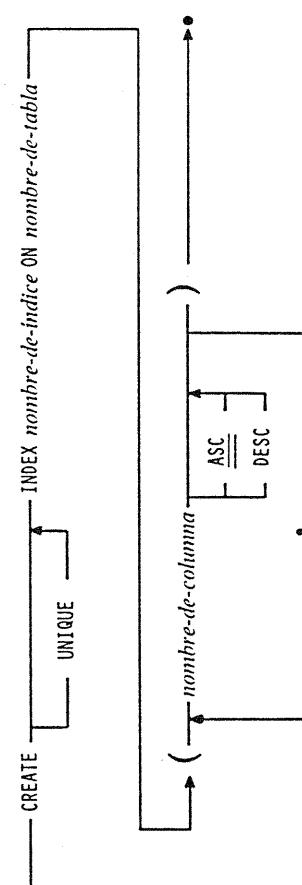


Figura 13.7. Diagrama sintáctico básico de la sentencia CREATE INDEX.

Otros objetos de bases de datos

Los verbos CREATE, DROP y ALTER forman los puentes del Lenguaje de Definición de Datos SQL. Las sentencias basadas en estos verbos se utilizan en todas las implementaciones SQL para manipular tablas, índices y vistas (que se describirán en el Capítulo 14). La mayoría de los productos DBMS basados en SQL más populares también utilizan estos verbos para formar sentencias DDL adicionales que crean, destruyen y modifican otros objetos de bases de datos únicos para ese producto particular DBMS.

En el DBMS Sybase o SQL Server, por ejemplo, la sentencia CREATE TRIGGER crea un disparador. El disparador puede ser suprimido posteriormente de la base de datos utilizando la sentencia DROP TRIGGER. Excepto por el uso de los nombres de verbo estándares y por el convenio de que la siguiente palabra sea el nombre del objeto, estas sentencias son todas muy específicas de cada DBMS y no son estándares. No obstante, dan un aspecto uniforme a los diferentes dialectos SQL. Si usted se encuentra un nuevo DBMS basado en SQL y sabe que soporta un objeto conocido como BLOB, no sería raro que soportara una sentencia CREATE BLOB y otra DROP BLOB. La Tabla 13.1 muestra cómo utilizan algunos de los productos SQL populares los verbos CREATE, DROP y ALTER en sus DDL ampliados.

Estructura de base de datos

El estándar SQL ANSI/ISO especifica una sencilla estructura para los contenidos de una base de datos, mostrada en la Figura 13.8. Cada usuario de la base de datos tiene una colección de tablas que son de su propiedad. Virtualmente todos los productos DBMS importantes soportan este esquema, aunque algunos (tales como Rdb/VMSP y dBASE IV) no soportan el concepto de propiedad de tabla. En estos sistemas todas las tablas de una base de datos son parte de una gran colección.

Sentencias DDL de SQL	Objeto gestionado
<i>Soportado por la mayoría de los productos DBMS.</i>	
CREATE/DROP/ALTER TABLE	tabla
CREATE/DROP/ALTER VIEW	vista
CREATE/DROP/ALTER INDEX	índice
<i>Soportado por DB2</i>	
CREATE/DROP DATABASE	grupo de tablas e índices
CREATE/DROP/ALTER STOGROUP	grupo de almacenamiento físico
CREATE/DROP/SYNONYM	sinónimo de tabla
CREATE/DROP/ALTER TABLESPACE	espacio de tablas (grupo de tablas)
<i>Soportado por SQL/DS</i>	
CREATE/DROP PROGRAM	módulo de acceso a la base de datos
DROP STATEMENT	sentencia en módulo de acceso a la base de datos
ACQUIRE/DROP DBSPACE	grupo de almacenamiento lógico
<i>Soportado por Oracle</i>	
CREATE/DROP/ALTER DATABASE	base de datos
CREATE/DROP/ALTER ROLLBACK SEGMENT	segmento de vuelta atrás
CREATE/DROP/ALTER SEQUENCE	secuencia de claves únicas
CREATE/DROP/ALTER TABLESPACE	espacio de tablas (grupo de tablas)
<i>Soportado por Ingres</i>	
CREATE INTEGRITY	restricción de integridad de tabla
CREATE PERMIT	permiso de acceso a tabla
<i>Soportado por VAX SQL</i>	
CREATE/DROP DATABASE	base de datos
<i>Soportado por Informix</i>	
CREATE/DROP AUDIT	seguimiento auditor
CREATE/DROP DATABASE	base de datos
CREATE/DROP SYNONYM	sinónimo
<i>Soportado por Sybase/SQL Server</i>	
CREATE/DROP/ALTER DATABASE	base de datos
CREATE/DROP DEFAULT	valor de columna por omisión
CREATE/DROP PROCEDURE	procedimiento almacenado
CREATE/DROP RULE	regla de integridad de columna
CREATE/DROP TRIGGER	disparador almacenado

Tabla 13.1. Sentencias DDL en productos basados en SQL populares.

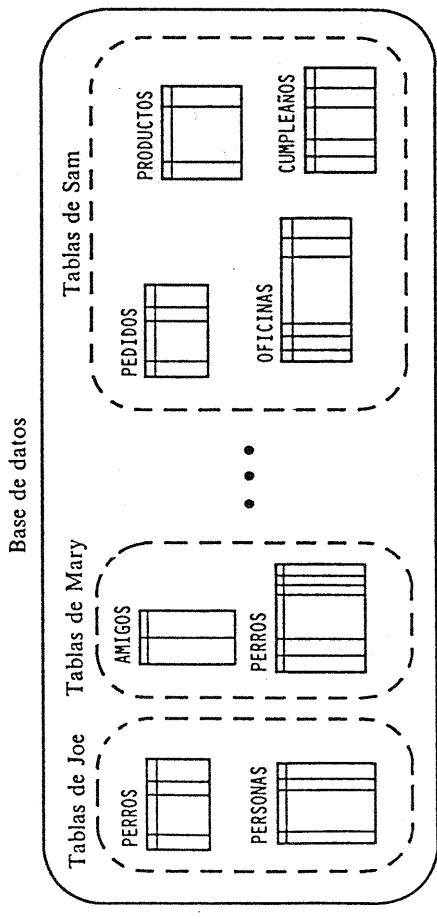


Figura 13.8. Organización ANSI/ISO de una base de datos.

Aunque diferentes productos de sistemas de gestión de base de datos basados en SQL proporcionan la misma estructura dentro de una única base de datos, hay amplia variación en cómo se organizan y estructuran las diferentes bases de datos de cada sistema informático particular. Algunos productos suponen una única base de datos global que almacena a todos los datos del sistema. Otros productos DBMS soportan múltiples bases de datos sobre un único computador, cada base de datos identificada por un nombre. Otros productos DBMS soportan múltiples bases de datos dentro del contexto del sistema de directorios del computador.

Estas variaciones no cambian el modo de utilizar SQL para acceder a los datos dentro de una base de datos. Sin embargo, afectan al modo de organizar los datos —por ejemplo, ¿se mezclan el procesamiento de pedidos y los datos de contabilidad en una sola base de datos o se dividen en dos bases de datos?—.

También afectan al modo en que inicialmente se obtiene acceso a la base de datos —por ejemplo, si hay múltiples bases de datos, es necesario informar al DBMS de cuál se desea utilizar—. Para ilustrar cómo los diferentes productos DBMS manejan estas cuestiones, supongamos que la base de datos ejemplo fuera ampliada para soportar una nómina y una aplicación de contabilidad, además de las tareas de procesamiento de pedidos que ahora soporta.

Arquitectura de base de datos única

La Figura 13.9 muestra una arquitectura de base de datos única en donde el DBMS soporta una base de datos global al sistema. DB2 y Oracle utilizan

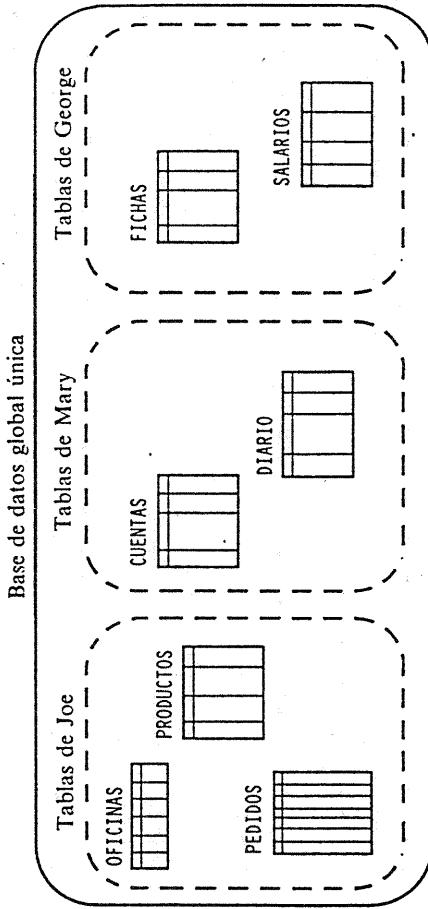


Figura 13.9. Arquitectura de base de datos única.

ambas este planteamiento. Los datos de procesamiento de pedidos, contabilidad y nómina están todos almacenados en tablas dentro de la base de datos. Las principales tablas de esta aplicación están reunidas bajo la propiedad de un único usuario, que probablemente es la persona al cargo de esa aplicación en este computador.

Una ventaja de esta arquitectura es que las tablas de las diferentes aplicaciones pueden referenciarse fácilmente unas a otras. La tabla PERSONAL de la aplicación de nóminas, por ejemplo, puede contener una clave foránea que referencia la tabla REPVENTAS, y las aplicaciones pueden utilizar esta relación para calcular comisiones. Con el permiso adecuado, los usuarios pueden ejecutar consultas que combinan datos de las diferentes aplicaciones.

Una desventaja de esta arquitectura es que la base de datos crecerá enormemente con el tiempo conforme se le añada más y más complicaciones. Una base de datos DB2 u Oracle con varios cientos de tablas no es habitual. Los problemas de gestionar una base de datos de ese tamaño —realización de copias de seguridad, recuperación de datos, análisis de rendimiento, etc.— requieren generalmente un administrador de base de datos a tiempo completo.

En la arquitectura de base de datos única, obtener acceso a la base de datos es muy sencillo —sólo hay una base de datos, por lo que no hay que efectuar decisiones—. Por ejemplo, la sentencia de SQL programado que conecta a un usuario con una base de datos Oracle es CONNECT, y los usuarios hablan en términos de «conectarse a Oracle», en vez de conectarse a una base de datos específica.

De hecho las instalaciones Oracle y DB2 ejecutan frecuentemente dos bases de datos separadas, una para trabajo de producción y otra para comprobación.

Fundamentalmente, sin embargo, todos los datos de producción están recogidos en una única base de datos.

Arquitectura de bases de datos múltiples

La Figura 13.10 muestra una arquitectura de bases de datos múltiples en donde cada base de datos tiene asignado un nombre único. Ingres, Sybase y SQL Server utilizan todas este esquema. Como se muestra en la figura, cada una de las bases de datos de esta arquitectura está dedicada generalmente a una aplicación particular. Cuando se añade una nueva aplicación, lo más normal es crear una nueva base de datos.

La ventaja principal de la arquitectura de bases de datos múltiples con respecto a la arquitectura de base de datos única es que divide a las tareas de gestión de base de datos en partes más pequeñas y más manejables. Cada persona responsable de una aplicación puede ahora ser el administrador de su propia base de datos, con menos preocupaciones con respecto a la coordinación global. Cuando llega el momento de añadir una nueva aplicación, puede desarrollarse en su propia base de datos sin afectar a las bases de datos existentes. También es más probable que los usuarios y los programadores puedan recorrer la estructura general de sus propias bases de datos.

La desventaja principal de la arquitectura de bases de datos múltiples es que las bases de datos individuales pueden convertirse en «islas» de información, desconectadas unas de otras. Tipicamente una tabla de una base de datos no puede contener una referencia de clave foránea a una tabla en una base de datos diferente. Con frecuencia el DBMS no soporta consultas a través de fronteras de bases de datos, haciendo imposible relacionar datos de dos aplicaciones. Si las consultas entre bases de datos son soportadas, pueden imponer un recargo sustancial o exigir la compra al vendedor del DBMS de un software de DBMS distribuido adicional.

Si un DBMS utiliza una arquitectura de bases de datos múltiples y soporta consultas entre bases de datos, debe ampliar las convenciones de designación de tablas y columnas SQL. Un nombre de tabla cualificado debe especificar no solamente el propietario de la tabla, sino también qué base de datos contiene la tabla. Tipicamente el DBMS extiende la «notación punto» para nombres de tabla añadiendo como prefijo del nombre de la base de datos el nombre del propietario, separado por un punto (.). Por ejemplo, en una base de datos Sybase o SQL Server, esta referencia de tabla:

PP.JOE.OFICINAS

se refiere a la tabla OFICINAS propiedad del usuario JOE en la base de datos de procesamiento de pedidos de nombre PP, y esta otra consulta compone la tabla REVENTAS en la base de datos de nómina con la tabla OFICINA:

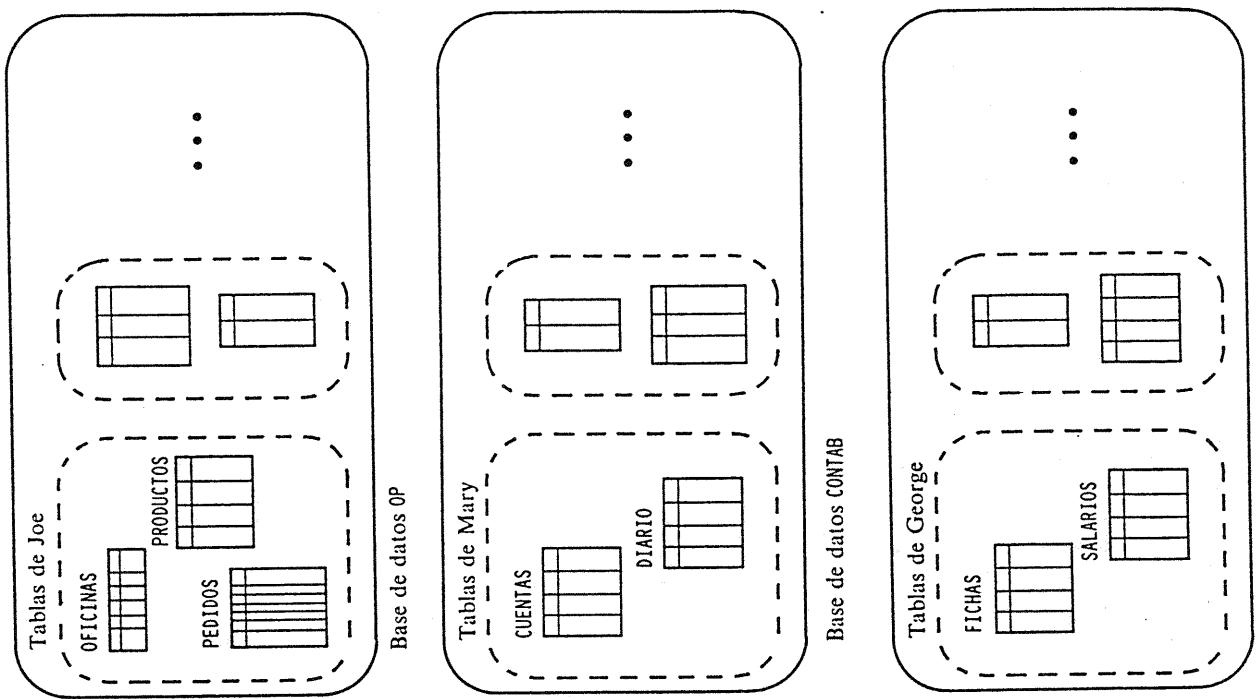


Figura 13.10. Arquitectura de bases de datos múltiples.

```

SELECT PP.JOE.OFICINAS.CIUDAD, NOMINA.GEORGE.REPVENTAS.NOMBRE
FROM PP.JOE.OFICINAS, PAYROLL.GEORGE.REPVENTAS
WHERE PP.JOE.OFICINAS.DIR = PAYROLL.GEORGE.REPVENTAS.NUM_EMPL

```

Afortunadamente, tales consultas entre bases de datos son la excepción en vez de la regla, y normalmente pueden utilizarse nombres de usuario de base de datos por omisión.

Con una arquitectura de bases de datos múltiples, obtener acceso a una base de datos resulta ligeramente más complejo, ya que debe informarse al DBMS de qué base de datos se desea utilizar. El programa SQL interactivo de DBMS visualizará con frecuencia una lista de las bases de datos disponibles o pedirá que se introduzca el nombre de la base de datos junto con el nombre del usuario y su contraseña para permitirle el acceso. Para el acceso en programación, el DBMS extiende generalmente el lenguaje SQL incorporando con una sentencia que conecta el programa a una base de datos particular. La forma Ingres para conectarse a la base de datos de nombre PP es:

```

CONNECT 'PP'
USE DATABASE 'PP'

```

La forma utilizada por Sybase y SQL Server es:

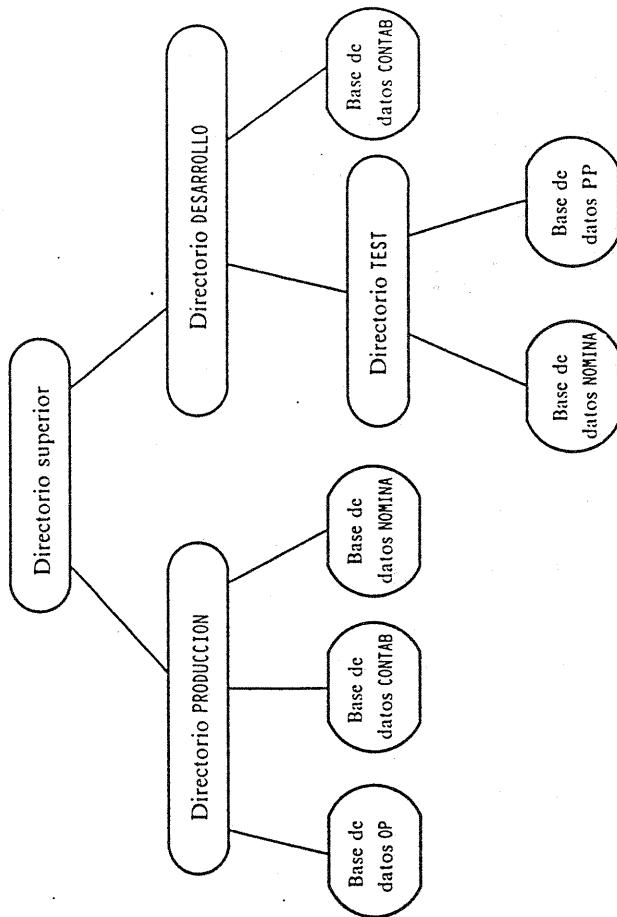


Figura 13.11. Arquitectura de ubicación múltiple.

La arquitectura de ubicación múltiple hace aún más complejo el obtener acceso a una base de datos, ya que debe especificarse tanto el nombre de la base de datos como su localización en la jerarquía de directorios. La sintaxis del SQL VAX para obtener acceso a una base de datos Rdb/VMS es la de la sentencia DECLARE DATABASE. Por ejemplo, esta sentencia DECLARE DATABASE establece una conexión a la base de datos de nombre PP en el directorio de VAX/VMS SYSSROOT:[DESARROLLO.TEST]:

```

DECLARE DATABASE FILENAME 'SYSSROOT:[DESARROLLO.TEST]PP'

```

Si la base de datos está en el directorio actual del usuario (que suele ser el caso), la sentencia se simplifica a:

```

DECLARE DATABASE FILENAME 'PP'

```

Algunos de los productos DBMS que utilizan este esquema permiten al usuario tener acceso a varias bases de datos concurrentemente, incluso cuando

Arquitectura de ubicación múltiple

La Figura 13.11 muestra una arquitectura de ubicación múltiple que soporta múltiples bases de datos y utiliza la estructura de directorio del sistema informático para organizarlas. Rdb/VMS e Informix utilizan ambos este esquema para soportar múltiples bases de datos. Como con la arquitectura de bases de datos múltiples, cada aplicación tiene asignada típicamente su propia base de datos. Como muestra la figura, cada base de datos tiene un nombre, pero es posible que dos bases de datos diferentes en dos directorios diferentes tengan el mismo nombre.

La principal ventaja de la arquitectura de ubicación múltiple es la flexibilidad. Es especialmente adecuada en aplicaciones tales como la ingeniería y diseño, en donde hay muchos usuarios sofisticados del sistema informático, que pueden todos desear utilizar varias bases de datos para estructurar su propia información. Las desventajas de la arquitectura de ubicación múltiple son las mismas que las de la arquitectura de bases de datos múltiples. Además, el DBMS no suele conocer todas las bases de datos que han sido creadas, las cuales pueden extenderse a través de toda la estructura de directorios del sistema. No existe una «base de datos maestra» que lleve la cuenta de todas las bases de datos, y esto hace que la administración centralizada sea muy difícil.

no soportan consultas a través de fronteras de bases de datos. Una vez más, la técnica más habitual utilizada para distinguir entre múltiples bases de datos es la de usar un nombre de tabla «superclificado». Puesto que dos bases de datos en dos directorios diferentes pueden tener el mismo nombre, es necesario además introducir un *alias de base de datos* para eliminar la ambigüedad. Estas sentencias de SQL VAX abren dos bases de datos Rdb/VMS diferentes que resultan tener el mismo nombre:

```
DECLARE DATABASE PP1
FILENAME 'SYS$ROOT:[PRODUCCION]PP1'

DECLARE DATABASE PP2
FILENAME 'SYS$ROOT:[DESARROLLO.TEST]PP'
```

Las sentencias asignan los alias PP1 y PP2 a las dos bases de datos, y estos alias se utilizan para cualificar nombres de tablas en posteriores sentencias del SQL VAX.

Como muestra esta discusión, existe una gran variedad en el modo en que los diferentes productos DBMS organizan sus bases de datos y proporcionan acceso a ellas. Esta área de SQL es una de las menos estándar, y no obstante suele ser la primera que el usuario encuentra cuando se trata de acceder a una base de datos por primera vez. Las inconsistencias también hacen imposible trasladar programas desarrollados para una DBMS a otro transparentemente, aunque el proceso de conversión es generalmente tedioso más que complejo.

DDL y el estándar ANSI/ISO

El estándar SQL ANSI/ISO hace una fuerte distinción entre el Lenguaje de Manipulación de Datos (DML) de SQL y el Lenguaje de Definición de Datos (DDL), definiéndolos como dos lenguajes separados y relativamente poco relacionados. El estándar no requiere que las sentencias DDL sean aceptadas por el DBMS durante la operación normal. De hecho, el estándar permite una estructura de base de datos estática como la utilizada por los productos DBMS jerárquicos y de red antiguos, como los mostrados en la Figura 13.12.

El estándar ANSI/ISO especifica las sentencias DDL como parte de un *esquema de base de datos*, un mapa de la base de datos que muestra su estructura. He aquí una definición de esquema ANSI/ISO para un usuario (llamado *Joe*) (*el autorización en el estándar*) de nombre Joe:

```
CREATE SCHEMA AUTHORIZATION JOE
CREATE TABLE PERSONAS
(NOMBRE VARCHAR(30),
EDAD INTEGER)
```

```
CREATE TABLE LUGARES
(CIUDAD VARCHAR(30)
ESTADO VARCHAR(30))
GRANT ALL PRIVILEGES
ON PERSONAS
TO PUBLIC
GRANT SELECT
ON LUGARES
TO MARY
```

El esquema define las dos tablas y da a algunos otros usuarios permiso para acceder a ellas. Si hay otro usuario de la base de datos, llamado Mary, sus tablas se definen mediante un esquema aparte:

```
CREATE SCHEMA AUTHORIZATION MARY
CREATE TABLE COSAS
(NOMBRE_COSAS VARCHAR(30)
NUM_COSA INTEGER)
CREATE TABLE LUGARES
(CIUDAD VARCHAR(30),
ESTADO VARCHAR(30))
GRANT SELECT, INSERT
ON COSAS
TO JOE
GRANT INSERT, UPDATE
ON LUGARES
TO PUBLIC
```

Estos dos esquemas pueden ser remitidos a un programa de utilidad constructor de base de datos, que construye una base de datos con la estructura especificada. Una vez que se crea la base de datos, sus tablas, usuarios y esquema de seguridad están congelados y son permanentes. Las sentencias DML pueden entonces ser utilizadas para añadir datos a la base de datos o para recuperar datos de ella, pero la estructura no puede ser modificada. Para cambiar la estructura (por ejemplo, para eliminar una tabla o añadir una columna), debe detenerse todo acceso a la base de datos, descargar los datos, remitir un esquema revisado al constructor de base de datos y volver a cargar los datos. Mientras la estructura está siendo modificada, la base de datos no está disponible para acceso.

A pesar del hecho de que el estándar ANSI/ISO permite una estructura de base de datos estática con un DML y un DDL separados, ningún DBMS importante basado en SQL utiliza este enfoque. En vez de ello, todos proporcionan un DDL totalmente dinámico y permiten la libre mezcla de sentencias de definición de datos y manipulación de datos. Las bases de datos de IBM (DB2, SQL/DS y OS/2 Extended Edition) tienen todas esta arquitectura, al igual que Oracle, Informix, SQLBase, Ingres, SQL Server y otras. Los productos SQL que

Puesto que no requiere un DDL dinámico, el estándar ANSI/ISO no especifica las sentencias DROP TABLE o ALTER TABLE. Sin embargo, estas sentencias han estado en los productos SQL IBM desde sus primeras revisiones, y otras implementaciones SQL las han utilizado como parte de su Lenguaje de Definición de Datos. Es muy probable que las versiones futuras del estándar ANSI/ISO requieran un DDL dinámico e incluyan estas sentencias para soportarlo. Mientras tanto, generalmente se puede asumir que estas sentencias son proporcionadas por un DBMS basado en SQL. De hecho, son más portables que muchas de las características que están especificadas en el estándar ANSI/ISO.

Resumen

- Este capítulo ha descrito las características del Lenguaje de Definición de Datos (DDL) de SQL que definen y alteran la estructura de una base de datos:
- La sentencia CREATE TABLE crea una tabla y define sus columnas, clave primaria y clave foránea.
 - La sentencia DROP TABLE elimina una tabla previamente creada de la base de datos.
 - La sentencia ALTER TABLE puede ser utilizada para añadir una columna a una tabla existente y para cambiar las definiciones de clave primaria y clave foránea.
 - Las sentencias CREATE INDEX y DROP INDEX definen índices, los cuales aceleran las consultas de la base de datos pero añaden recargo a las actualizaciones.
 - La mayoría de los productos DBMS soportan otras sentencias CREATE, DROP y ALTER utilizadas con objetos específicos del DBMS.
 - Varios productos DBMS utilizan enfoques muy diferentes para organizar la(s) base(s) de datos que gestionan, y estas diferencias afectan al modo de diseñar las bases de datos y de obtener acceso a ellas.

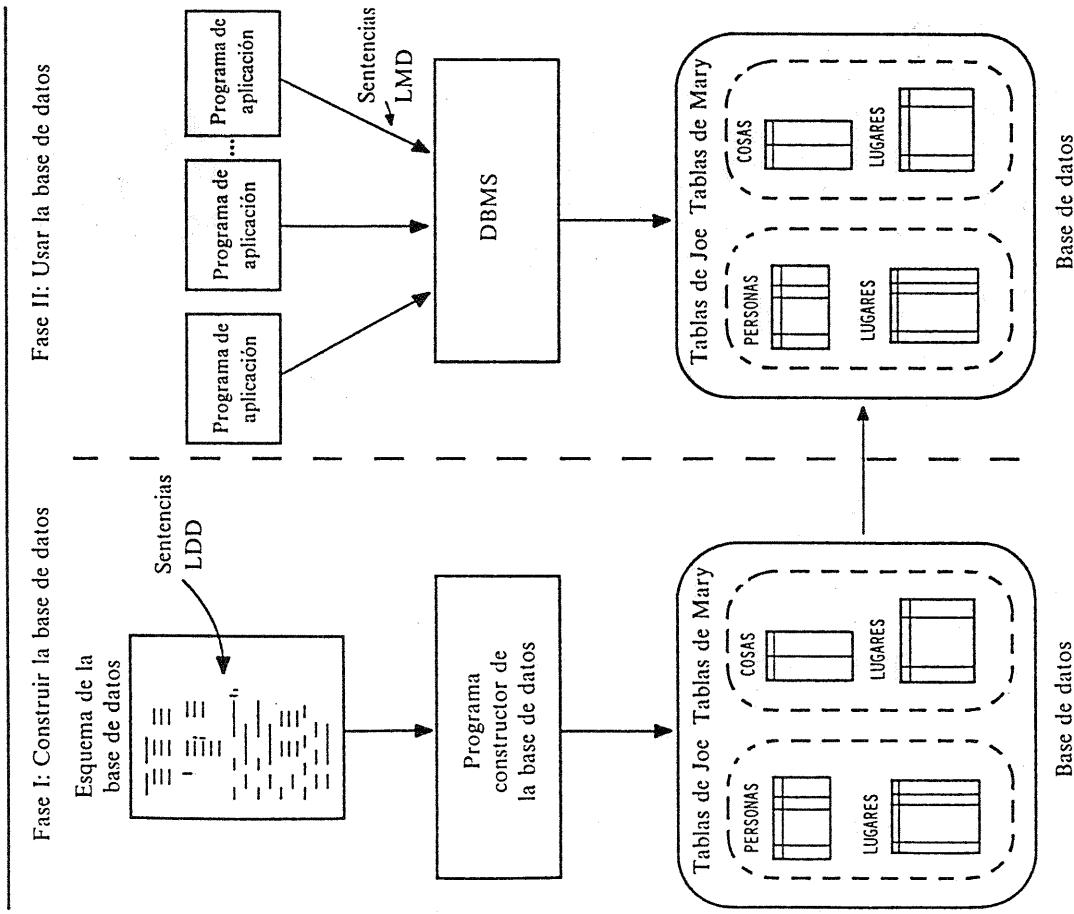


Figura 13.12. Un DBMS con un DDL estático.

no ofrecen un DDL dinámico son típicamente aquellos que tienen el lenguaje SQL «superpuesto» por encima de una estructura de base de datos no relacional más antigua. La versión dBASE IV de SQL es el ejemplo más prominente de este tipo de producto.

14 Vistas

Las tablas de una base de datos definen la estructura y organización de su datos. Sin embargo, SQL también permite mirar a los datos almacenados de otros modos mediante la definición de vistas alternativas de los datos. Una *vista* es una consulta SQL que está permanentemente almacenada en la base de datos y a la que se le asigna un nombre. Los resultados de una consulta almacenada son «visibles» a través de la vista, y SQL permite acceder a estos resultados como si fueran, de hecho, una tabla «real» en la base de datos.

Las vistas son parte importante de SQL, por varias razones:

- Las vistas permiten acomodar el aspecto de una base de datos de modo que diferentes usuarios la vean desde diferentes perspectivas.
- Las vistas permiten restringir acceso a los datos, permitiendo que diferentes usuarios sólo vean ciertas filas o ciertas columnas de una tabla.
- Las vistas simplifican el acceso a la base de datos mediante la presentación de la estructura de los datos almacenados del modo que sea más natural a cada usuario.

Este capítulo describe cómo crear vistas y cómo utilizarlas para simplificar el procesamiento y mejorar la seguridad de una base de datos.

¿Qué es una vista?

Una *vista* es una «tabla virtual» en la base de datos cuyos contenidos están definidos por una consulta tal como se muestra en la Figura 14.1. Para el

usuario de la base de datos, la vista aparece igual que una tabla real, con un conjunto de columnas designadas y filas de datos. Pero a diferencia de una tabla real, una vista no existe en la base de datos como conjunto almacenado de valores. En su lugar, las filas y columnas de datos visibles a través de la vista son los resultados producidos por la consulta que define la vista. SQL crea la ilusión de la vista dándole a ésta un nombre semejante a un nombre de tabla y almacenando la definición de la vista en la base de datos.

La vista que se muestra en la Figura 14.1 es típica. Se le ha dado el nombre DATOSREP y se le ha definido mediante esta consulta de dos tablas:

```
SELECT NOMBRE, CIUDAD, REGION, CUOTA, REPVENTAS, VENTAS
  FROM REPVENTAS, OFICINAS
 WHERE OFICINA.REP = OFICINA
```

Los datos de la vista provienen de las tablas REPVENTAS y OFICINAS. Estas tablas se denominan *tablas fuente* de la vista, ya que son la fuente de los datos que son visibles a través de la vista. Esta vista contiene una fila de información por cada vendedor, ampliada con el nombre de la ciudad y región en donde el vendedor trabaja. Como muestra la figura, la vista aparece como una tabla, y sus contenidos aparecen como los resultados que se obtendrían si realmente se ejecutara la consulta.

Una vez definida una vista, se puede utilizar en una sentencia SELECT, lo mismo que una tabla real, como en esta consulta:

Lista los vendedores que están por encima de su cuota, mostrando el nombre, ciudad y región por cada vendedor.

```
SELECT NOMBRE, CIUDAD, REGION
  FROM DATOSREP
 WHERE VENTAS > CUOTA
```

El nombre de la vista, DATOSREP, aparece en la cláusula FROM como un nombre de tabla, y las columnas de la vista se mencionan en la sentencia SELECT lo mismo que las columnas de una tabla real. Para algunas vistas también se pueden utilizar las sentencias INSERT, DELETE y UPDATE para modificar los datos visibles a través de la vista, como si fueran una tabla real. Por tanto, a todos los efectos prácticos, la vista puede ser utilizada en las sentencias SQL como si fuera una tabla real.

Cómo maneja el DBMS las vistas

Cuando el DBMS encuentra una referencia a una vista en una sentencia SQL, determina la definición de la vista almacenada en la base de datos. Luego el

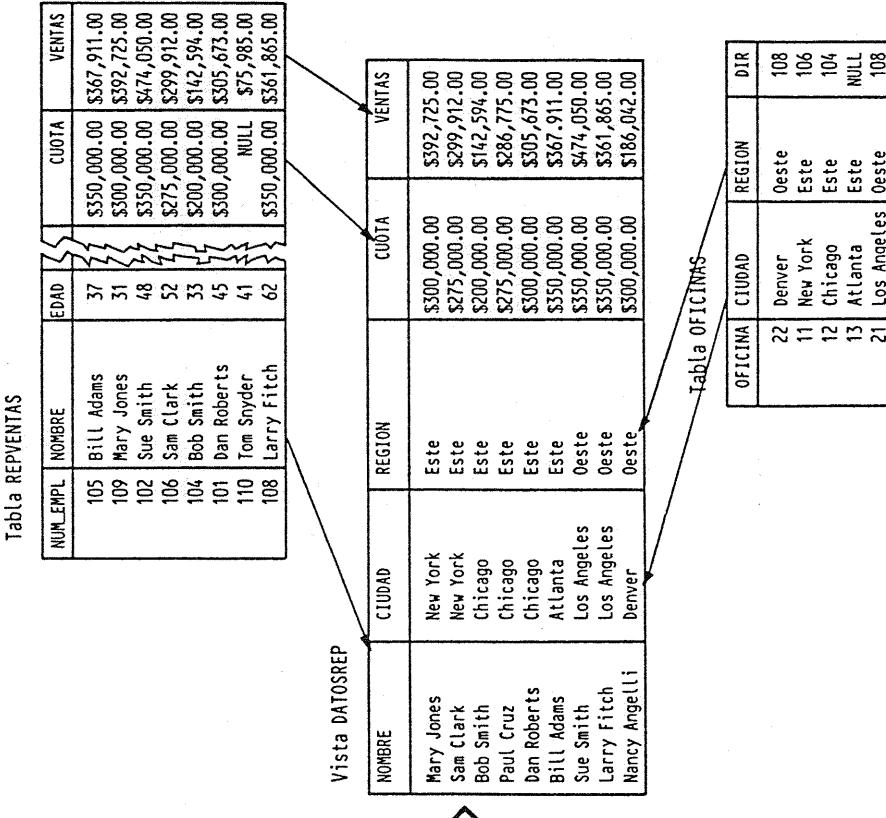


Figura 14.1. Una vista típica con dos tablas fuente.

DBMS traduce la petición que referencia a la vista a una petición *equivalente* con respecto a las tablas fuente de la vista y lleva a cabo la petición equivalente. De este modo el DBMS mantiene la ilusión de la vista mientras mantiene la integridad de las tablas fuente.

Para vistas sencillas, el DBMS puede construir cada fila de la vista «sobre la marcha», extrayendo los datos para la fila de las tablas fuente. Para vistas más complejas, el DBMS debe *materializar* realmente la vista; es decir, el DBMS debe llevar a cabo efectivamente la consulta que define la vista y almacenar sus resultados en una tabla temporal. El DBMS cumplimenta las peticiones de acceso a la vista a partir de esta tabla temporal y descarta la tabla cuando ya no es necesaria. Con independencia de cómo maneje el DBMS realmente una vista particular, el resultado es el mismo para el usuario —la vista puede ser referenciada en sentencias SQL exactamente como si fuera una tabla real de la base de datos.

Ventajas de las vistas

Las vistas proporcionan una variedad de beneficios y pueden ser útiles en muchos tipos diferentes de bases de datos. En una base de datos sobre un computador personal, las vistas son generalmente una conveniencia, definidas para simplificar las peticiones a la base de datos. En una instalación de base de datos de producción, las vistas juegan un papel central en la definición de la estructura de la base de datos para sus usuarios y en el reforzamiento de su seguridad. Las vistas proporcionan estos importantes beneficios:

■ **Seguridad.** Cada usuario puede obtener permiso para acceder a la base de datos únicamente a través de un pequeño conjunto de vistas que contienen los datos específicos que el usuario está autorizado a ver, restringiendo así el acceso al usuario a datos almacenados.

■ **Simplicidad de consulta.** Una vista puede extraer datos de varias tablas diferentes y presentarlos como una única tabla, haciendo que consultas multitable se formulen como consultas de una sola tabla con respecto a la vista.

■ **Simplicidad estructural.** Las vistas pueden «dar a un usuario una visión personalizada» de la estructura de la base de datos presentando ésta como un conjunto de tablas virtuales que tienen sentido para ese usuario.

■ **Aislamiento frente al cambio.** Una vez vista puede presentar una imagen consistente inalterada de la estructura de la base de datos, incluso si las tablas fuente subyacentes se dividen, reestructuran o cambian de nombre.

■ **Integridad de datos.** Si los datos se acceden y se introducen a través de una vista, el DBMS puede comprobar automáticamente los datos para asegurarse que satisfacen restricciones de integridad específicas.

Desventajas de la vistas

Aunque las vistas proporcionan ventajas sustanciales, también hay dos desventajas importantes al utilizar una vista en lugar de una tabla real:

■ **Rendimiento.** Las vistas crean la *apariencia* de una tabla, pero el DBMS debe traducir las consultas con respecto a la vista en consultas con respecto a las tablas fuente subyacentes. Si la vista se define mediante una consulta multitable compleja, entonces incluso una consulta sencilla con respecto a la vista se convierte en una composición complicada, y puede tardar mucho tiempo en completarse.

■ **Restricciones de actualización.** Cuando un usuario trata de actualizar filas de una vista, el DBMS debe traducir la petición a una actualización sobre las filas de las tablas fuente subyacentes. Esto es posible para vistas sencillas, pero vistas más complejas no pueden ser actualizadas; son «de solo lectura».

Estas desventajas significan que no se pueden definir indiscriminadamente vistas y utilizarlas en lugar de las tablas fuente. En vez de ello, en cada caso deben considerarse las ventajas proporcionadas por el uso de una vista ponderándolas con sus desventajas.

Creación de una vista (CREATE VIEW)

La sentencia CREATE VIEW, mostrada en la Figura 14.2, se utiliza para crear una vista. La sentencia asigna un nombre a la vista y especifica la consulta que define la vista. Para crear la vista con éxito, es preciso tener permiso para acceder a todas las tablas referenciadas en la consulta.

La sentencia CREATE VIEW puede asignar opcionalmente un nombre a cada columna en la vista recién creada. Si se especifica una lista de nombres de columnas, debe tener el mismo número de elementos que el número de columnas producido por la consulta. Observe que solamente se especifican los nombres de las columnas; el tipo de datos, la longitud y las otras características de cada

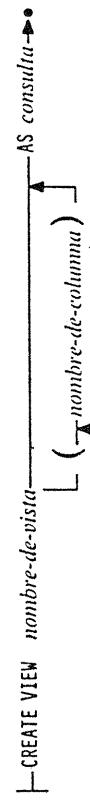


Figura 14.2: Diagrama sintáctico de la sentencia CREATE VIEW

columna se deducen de la definición de las columnas en la tablas fuente. Si la lista de nombres de columnas se omite de la sentencia CREATE VIEW, cada columna de la vista adopta el nombre de la columna correspondiente de la consulta. La lista de nombres de columnas debe ser especificada si la consulta incluye columnas calculadas o si produce dos columnas con nombres idénticos.

Aunque las vistas se crean del mismo modo, en la práctica diferentes tipos de vistas son utilizados típicamente para propósitos diferentes. Las próximas secciones examinan estos tipos de vistas y proporcionan ejemplos de la sentencia CREATE VIEW.

Vistas horizontales

Un uso común de las vistas es restringir el acceso de un usuario a únicamente filas seleccionadas de una tabla. Por ejemplo, en la base de datos ejemplo, puede ser deseable que un director de ventas solamente vea las filas de REPVENTAS correspondientes a los vendedores de la región propia del director. Para lograr esto, se pueden definir dos vistas, del modo siguiente:

Crea una vista que muestre a los vendedores de la región Este.

```
CREATE VIEW REPESTE AS
  SELECT *
    FROM REPVENTAS
   WHERE OFICINA REP IN (11, 12, 13)
```

Crea una vista que muestre a los vendedores de la región Oeste.

```
CREATE VIEW REPOSETE AS
  SELECT *
    FROM REPVENTAS
   WHERE OFICINA REP IN (21, 22)
```

Ahora se puede dar a cada director de ventas permiso para acceder a la vista REPESTE o bien a la vista REPOSETE, negándoles el permiso para acceder a la otra vista y a la propia tabla REPVENTAS. Esto da efectivamente al director de ventas una visión particular de la tabla REPVENTAS, que sólo muestra los vendedores de la región adecuada.

Una vista como REPESTE o REPOSETE se denomina normalmente *vista horizontal*. Como se muestra en la Figura 14.3, una vista horizontal «divide» horizontalmente la tabla fuente para crear la vista. Todas las columnas de la tabla fuente participan en la vista, pero sólo algunas de sus filas son visibles a través de la vista. Las vistas horizontales son adecuadas cuando la tabla fuente contiene datos que relacionan a varias organizaciones o usuarios. Proporcionan una «tabla privada» para cada usuario, compuesta únicamente de las filas necesarias a ese usuario.

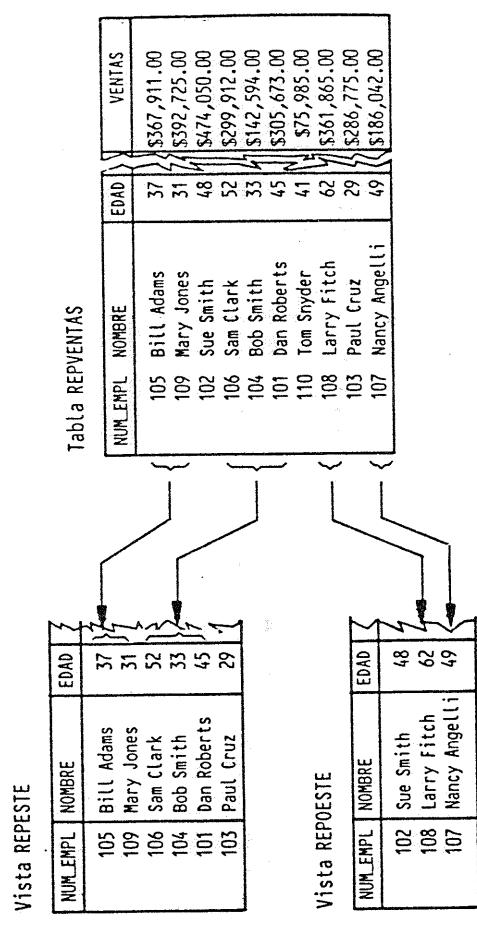


Figura 14.3. Dos vistas horizontales de la tabla REPVENTAS.

He aquí algunos ejemplos más de vistas horizontales:

```
CREATE VIEW OFICINASESTE AS
  SELECT *
    FROM OFICINAS
   WHERE REGION = 'Este'

CREATE VIEW PEDIDOSUE AS
  SELECT *
    FROM PEDIDOS
   WHERE CLIE IN (SELECT NUM_CLIE
                  FROM CLIENTES
                 WHERE NUM_CLIE = 102)
```

Define una vista para Sue Smith (empleando número 102) que contiene solamente los pedidos remitidos por clientes asignados a ella.

fsef

Define una vista que muestre únicamente aquellos clientes que tienen más de \$30.000 en pedidos actualmente registrados.

```
CREATE VIEW CLIENTESVIP AS
SELECT *
FROM CLIENTES
WHERE 30000.00 < (SELECT SUM(TIMPORTE)
                     FROM PEDIDOS
                     WHERE CLIE = NUMCLIE)
```

En cada uno de estos ejemplos, la vista se deriva de una única tabla fuente. La vista se define mediante una consulta SELECT * y por tanto tiene exactamente las mismas columnas que la tabla fuente. La cláusula WHERE determina qué filas de la tabla fuente son visibles en la vista.

Vistas verticales

Otro uso habitual de las vistas es restringir el acceso de un usuario a sólo ciertas columnas de una tabla. Por ejemplo, en nuestra base de datos, el departamento de procesamiento de pedidos puede necesitar acceder al número de empleado, su nombre y la asignación de oficina de cada vendedor, puesto que esta información puede ser necesaria para procesar un pedido correctamente. Sin embargo, no hay necesidad de que el personal de procesamiento de pedidos vea las ventas anuales hasta la fecha del vendedor o su cuota. Esta vista selectiva de la tabla REPVENTAS puede ser construida con la vista siguiente:

Crea una vista mostrando información seleccionada de cada vendedor.

```
CREATE VIEW INFOREP AS
SELECT NUM_EMPL, NOMBRE, OFICINA.REP
FROM REPVENTAS
```

Dando al personal de procesamiento de pedidos acceso a esta vista y denegándole acceso a la propia tabla REPVENTAS, el acceso a datos de cuotas y ventas específicos se restringe efectivamente.

Una vista como la INFOREP se denomina con frecuencia *vista vertical*. Como se muestra en la Figura 14.4, una vista vertical «divide» la tabla fuente verticalmente para crear la vista. Las vistas verticales suelen encontrarse allí donde los datos almacenados en una tabla son utilizados por varios usuarios o grupos de usuarios. Proporcionan una «tabla privada» a cada usuario, compuesta únicamente de las columnas necesarias a ese usuario.

He aquí algunos ejemplos más de vistas verticales:

Define una vista de la tabla OFICINAS para el personal de procesamiento de pedidos que incluye la ciudad, el número de oficina, y la región.

```
CREATE VIEW INFOOFICINA AS
SELECT OFICINA, CIUDAD, REGION
FROM OFICINAS
```

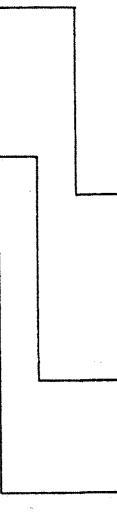
Define una vista de la tabla CLIENTES que incluye únicamente los nombres de los clientes y su asignación a vendedores.

```
CREATE VIEW INFOCLIE AS
SELECT EMPRESA, REP_CLIE
FROM CLIENTES
```

En cada uno de estos ejemplos, la vista se deriva de una tabla fuente única. La lista de selección en la definición de la vista determina qué columnas de la tabla fuente son visibles en la vista. Puesto que se trata de vistas verticales, cada fila de la tabla fuente está representada en la vista, y la definición no incluye una cláusula WHERE.

Vista INFOREP

NUM_EMPL	NOMBRE	OFICINA.REP
105	Bill Adams	13
109	Mary Jones	11
102	Sue Smith	21
106	Sam Clark	11
104	Bob Smith	12
101	Dan Roberts	12
110	Tom Snyder	NULL
108	Larry Fitch	21
103	Paul Cruz	12
107	Nancy Angelli	22



NUM_EMPL	NOMBRE	EDAD	OFICINA.REP	TITULO	CONTRATO	DIRECTOR	CUOTA	VENTAS
105	Bill Adams	37	13	Rep Ventas	02/12/1988	104	\$350,000.00	\$367,911.00
109	Mary Jones	31	11	Rep Ventas	10/12/1989	106	\$300,000.00	\$392,725.00
102	Sue Smith	48	21	Rep Ventas	12/10/1986	108	\$350,000.00	\$474,050.00
106	Sam Clark	52	11	VP Ventas	06/14/1988	NULL	\$275,000.00	\$299,912.00
104	Bob Smith	33	12	Dir Ventas	05/19/1987	106	\$200,000.00	\$142,594.00
101	Dan Roberts	45	12	Rep Ventas	10/20/1986	104	\$300,000.00	\$305,673.00
110	Tom Snyder	41	NULL	Rep Ventas	01/13/1990	101	NULL	\$75,985.00
108	Larry Fitch	62	21	Dir Ventas	10/12/1989	106	\$350,000.00	\$361,865.00
103	Paul Cruz	29	12	Rep Ventas	03/01/1987	104	\$275,000.00	\$286,775.00
107	Nancy Angelli	49	22	Rep Ventas	11/14/1988	108	\$300,000.00	\$186,042.00

Tabla REPVENTAS

Figura 14.4. Una vista vertical de la tabla REPVENTAS.

Vistas con subconjuntos fila/columna

Cuando se define una vista, SQL no restringe a divisiones puramente horizontales o verticales de una tabla. De hecho, el lenguaje SQL no incluye la noción de vistas horizontales y verticales. Estos conceptos meramente ayudan a visualizar cómo la vista presenta la información a partir de la tabla fuente. Es bastante habitual definir una vista que divide una tabla fuente tanto por la dimensión horizontal como por la vertical, como en este ejemplo:

Define una vista que contiene el número de cliente, el nombre de empresa y el límite de crédito de todos los clientes asignados a Bill Adams (empleado número 105).

```
CREATE VIEW CLIEBILL AS
SELECT NUM_CLIE, EMPRESA LIMIT_CREDITO
  FROM CLIENTES
 WHERE REP_CLIE = 105
```

Los datos visibles a través de esta vista son un subconjunto fila/columna de la tabla CLIENTES. Sólo las columnas explicitamente designadas en la lista de selección de la vista y las filas que satisfacen la condición de búsqueda son visibles a través de la vista.

Vistas agrupadas

La consulta especificada en una definición de vista puede incluir una cláusula GROUP BY. Este tipo de vista se denomina *vista agrupada*, ya que los datos visibles a través de ella son el resultado de una consulta agrupada. Las vistas agrupadas efectúan la misma función que las consultas agrupadas; agrupan filas relacionadas de datos y producen una fila de resultados de consulta para cada grupo, sumarizando los datos de ese grupo. Una vista agrupada reúne estos resultados de la consulta agrupada en una tabla virtual, permitiendo efectuar consultas adicionales sobre ella.

He aquí un ejemplo de vista agrupada:

Define una vista que contiene datos de pedidos sumarios para cada vendedor.

```
CREATE VIEW PED_PORREP (QUOTEN, CUANTOS, TOTAL, INF, SUP, MEDIO) AS
SELECT REP, COUNT(*) , SUM(IMPORTE), MIN(IMPORTE), MAX(IMPORTE), AVG(IMPORTE)
  FROM PEDIDOS
 GROUP BY REP
```

Como muestra este ejemplo, la definición de una vista agrupada siempre incluirá una lista de nombres de columna. La lista asigna nombres a las columnas de la vista agrupada que se derivan de funciones de columna, tales como SUM() y MIN(). También puede especificar un nombre modificado para una columna de

agrupación. En este ejemplo, la columna REP de la tabla PEDIDOS pasa a ser la columna QUIEN en la vista PED_PORREP.

Una vez que se define esta vista agrupada, puede ser utilizada para simplificar consultas. Por ejemplo, esta consulta genera un sencillo informe que sumariza los pedidos de cada vendedor:

Muestra el nombre, los números de pedido, el importe total de pedidos y el medio de pedido para cada vendedor.

```
SELECT NOMBRE, CUANTOS, TOTAL, MEDIO
  FROM REVENTAS, PED_PORREP
 WHERE QUIEN = NUM_EMPL
 ORDER BY TOTAL DESC
```

NOMBRE	CUANTOS	TOTAL	MEDIO
Larry Fitch	7	\$58,633.00	\$8,376.14
Bill Adams	5	\$39,327.00	\$7,865.40
Nancy Angelli	3	\$34,432.00	\$11,477.33
Sam Clark	2	\$32,958.00	\$16,479.00
Dan Roberts	3	\$26,628.00	\$8,876.00
Tom Snyder	2	\$23,132.00	\$11,566.00
Sue Smith	4	\$22,776.00	\$5,694.00
Mary Jones	2	\$7,105.00	\$3,552.50
Paul Cruz	2	\$2,700.00	\$1,350.00

A diferencia de una vista horizontal o vertical, las filas en una vista agrupada no tienen una correspondencia una a una con las filas de la tabla fuente. Una vista agrupada no es únicamente un filtro de su tabla fuente que oculta ciertas filas y columnas. Es un sumario de las tablas fuente, y por tanto se requiere una sustancial cantidad de procesamiento DBMS para mantener la ilusión de una tabla virtual para vistas agrupadas.

Las vistas agrupadas pueden ser utilizadas en consultas igual que cualquier otra vista más sencilla. Sin embargo, una vista agrupada no puede ser actualizada. La razón debería ser obvia a partir del ejemplo. ¿Qué significaría «actualizar el tamaño medio de pedido para el vendedor número 105»? Puesto que cada fila de la vista agrupada corresponde a un *grupo* de filas de la tabla fuente, y puesto que las columnas de la vista agrupada contienen generalmente datos calculados, no hay manera de trasladar una petición de actualización a una actualización de las filas de la tabla fuente. Las vistas agrupadas funcionan por tanto como vistas «de sólo lectura», que pueden participar en consultas, pero no en actualizaciones.

Las vistas agrupadas están también sujetas a las restricciones SQL sobre funciones de columna animadas. Recuerde del Capítulo 8 que las funciones de columna animadas, tales como:

MIN(MIN(A))

no son legales en expresiones SQL. Aunque la vista agrupada «oculta» las funciones de columna de su lista de selección al usuario, el DBMS sigue considerándolas y fuerza la restricción. Consideré este ejemplo:

Para cada oficina de ventas, muestra el rango de los tamaños medios de pedido para todas los vendedores que trabajan en la oficina.

```
SELECT OFICINA.REP, MIN(MEDIO), MAX(MEDIO)
  FROM REPVENTAS, PED_PORREP
 WHERE NUM_EMPL = QUIEN
 GROUP BY OFICINA.REP
```

Error: Referencia a función de columna anidada

Esta consulta produce un error, aun cuando parezca perfectamente razonable. Es una consulta de dos tablas que agrupa las filas de la vista PED_PORREP basándose en la oficina a la cual se asigna el vendedor. Pero las funciones de columna MIN() y MAX() en la lista de selección provocan un problema. El argumento de estas funciones de columna, la columna MEDIO, es ella misma el resultado de una función columna. La consulta «efectiva» que está siendo solicitada de SQL es:

```
SELECT OFICINA.REP, MIN(AVG(MEDIO)), MAX(AVG(MEDIO))
  FROM REPVENTAS, PEDIDOS
 WHERE NUM_EMPL = REP
 GROUP BY REP
 GROUP BY OFICINA.REP
```

Esta consulta es ilegal a causa del doble GROUP BY y de las funciones de columna anidadadas. Desgraciadamente, como muestra este ejemplo, una sentencia SELECT agrupada perfectamente razonable puede, de hecho, producir un error si una de sus tablas fuente resulta ser una vista agrupada. No hay manera de anticipar esta situación; debe comprenderse la causa del error cuando SQL informe de él.

Vistas compuestas

Una de las razones más frecuentes para utilizar vistas compuestas es simplificar las consultas multitalia. Especificando una consulta de dos o tres tablas en la definición de vista, se puede crear una vista compuesta que extrae sus datos de dos o tres tablas diferentes y presenta los resultados de la consulta como única tabla virtual. Una vez definida la vista, con frecuencia se puede utilizar una consulta simple de una sola tabla con respecto a la vista para peticiones que en caso contrario requerirían una composición de dos o tres tablas.

Por ejemplo, supongamos que Sam Clark, el vicepresidente de ventas, efect-

túa con frecuencia consultas sobre la tabla PEDIDOS de la base de datos ejemplo. Sin embargo, Sam no quisiera trabajar con números de clientes y empleados. En vez de ello, le gustaría poder utilizar una versión de la tabla PEDIDOS que tenga nombres en vez de números. He aquí una vista que satisface las necesidades de Sam:

Crea una vista de la tabla PEDIDOS con nombres en vez de números.

```
CREATE VIEW INFO_PEDIDO (NUM_PEDIDO, EMPRESA, NOMBREREP, IMPORTE) AS
  SELECT NUM_PEDIDO, EMPRESA, NOMBRE, IMPORTE
    FROM PEDIDOS, CLIENTES, REPVENTAS
   WHERE CLIE = NUM_CLIE
     AND REP = NUM_EMPL
```

Esta vista está definida mediante una composición de tres tablas. Como con la vista agrupada, el procesamiento necesario para crear la ilusión de una tabla virtual para esta vista es considerable. Cada fila de la vista se deriva de una combinación de una fila de la tabla PEDIDOS, una fila de la tabla CLIENTES y una fila de la tabla REPVENTAS.

Aunque tiene una definición relativamente compleja, esta vista puede proporcionar algunos beneficios reales. He aquí una consulta sobre la vista que genera un informe de pedidos, agrupados por vendedor:

```
Muestra los pedidos actuales totales para cada empresa y para cada vendedor.
SELECT NOMBREREP, EMPRESA, SUM(IMPORTE)
  FROM INFO_PEDIDO
 GROUP BY NOMBREREP, EMPRESA
```

NOMBREREP	EMPRESA	SUM(IMPORTE)
Bill Adams	Acme Mfg.	\$35,502.00
Bill Adams	JCP Inc.	\$3,745.00
Dan Roberts	First Corp.	\$3,978.00
Dan Roberts	Holm & Landis	\$150.00
Dan Roberts	Ian & Schmidt	\$22,500.00
Larry Fitch	Midwest Systems	\$3,608.00
Larry Fitch	Orion Corp.	\$7,100.00
Larry Fitch	Zetacorp	\$47,925.00

:

Observe que esta consulta es una sentencia SELECT de una sola tabla, que es considerablemente más simple que la sentencia SELECT de tres tablas para las tablas fuente:

```
SELECT NOMBRE, EMPRESA, SUM(IMPORTE)
  FROM REPVENTAS, PEDIDOS, CLIENTES
```

```

WHERE REP = NUM_EMPL
AND CLIE = NUM_CLIE
GROUP BY NOMBRE, EMPRESA
  
```

Análogamente, es fácil generar un informe de los mayores pedidos, mostrando quién los remitió y quién los recibió, con esta consulta sobre la vista:

Mostrar los mayores pedidos actuales, ordenados por importe.

```

SELECT EMPRESA, IMPORTE, NOMBREREP
FROM INFO_PEDIDO
WHERE IMPORTE > 20000.00
ORDER BY IMPORTE DESC
  
```

EMPRESA	IMPORTE	NOMBREREP
Zetacorp	\$45,000.00	Larry Fich
J.P. Sinclair	\$31,500.00	Sam Clark
Chen Associates	\$31,350.00	Nancy Angelli
Acme Mfg.	\$27,500.00	Bill Adams
Ace International	\$22,500.00	Tom Snyder
Ian & Schmidt	\$22,500.00	Dan Roberts

La vista hace mucho más fácil examinar lo que sucede en la consulta que si fuera expresada como la composición equivalente de tres tablas. Naturalmente el DBMS debe trabajar igual de duro para generar los resultados de la consulta en la consulta de una tabla con respecto a la vista que si generara los resultados para la consulta de tres tablas equivalentes. De hecho, el DBMS debe efectuar un poco más de trabajo para manejar la consulta con respecto a la vista. Sin embargo, para el usuario humano de la base de datos es mucho más fácil escribir y comprender la consulta de una tabla que referencia a la vista.

Actualización de una vista

¿Qué significa insertar una fila de datos en un vista, suprimir una fila de una vista o actualizar una fila de una vista? Para algunas vistas estas operaciones pueden ser traducidas obviamente a operaciones equivalentes respecto a las tablas fuente de la vista. Por ejemplo, consideremos una vez más la vista REPESTE, definida anteriormente en este capítulo:

```

Crea una vista que muestre los vendedores de la región Este.

CREATE VIEW REPESTE AS
SELECT *
FROM REPVENTAS
WHERE OFICINAREP IN (11, 12, 13)
  
```

Se trata de una vista horizontal directa, deducida de una tabla fuente única. Como muestra la Figura 14.5, tiene sentido hablar de insertar una fila en esta vista; significa que la nueva fila debería ser insertada en la tabla REPVENTAS subyacente a partir de la cual se derivó la vista. También tiene sentido suprimir una fila de la vista REPESTE; resultaría en la supresión de la fila correspondiente de la tabla REPVENTAS. Finalmente, actualizar una fila de la vista REPESTE también tiene sentido; actualizaría la fila correspondiente de la tabla REPVENTAS. En cada caso la acción puede ser realizada con respecto a la fila correspondiente de la tabla fuente, preservando la integridad tanto de la tabla fuente como de la vista. Sin embargo, consideremos la vista agrupada PED_PORREP, como se definió anteriormente en este capítulo:

Define una vista que contiene datos de pedidos sumarios para cada vendedor.

```

CREATE VIEW PED_PORREP (QUEN, CUANTOS, TOTAL, INF, SUP, MEDIO) AS
SELECT REP, COUNT(*), SUM(IMPORTE), MIN(IMPORTE),
       MAX(IMPORTE), AVG(IMPORTE)
  FROM PEDIDOS
 GROUP BY REP
  
```

No existe una correspondencia una a una entre las filas de esta vista y las filas de la tabla PEDIDOS subyacente, por lo cual no tiene sentido hablar de insertar, suprimir o actualizar filas de esta vista. La vista PED_PORREP no es actualizable; es una vista de sólo lectura.

La vista REPESTE y la vista PED_PORREP son dos ejemplos extremos en términos de la complejidad de sus definiciones. Hay vistas más complejas que REPESTE donde sigue teniendo sentido actualizar la vista, y hay vistas menos complejas que PED_PORREP en donde las actualizaciones no tienen sentido. De hecho, determinar que vistas pueden ser actualizadas y cuáles no ha sido un importante problema de investigación en base de datos relacional a lo largo de los años.

Tabla REPVENTAS

NUM_EMPL	NOMBRE	EDAD
105	Bill Adams	37
109	Mary Jones	31
102	Sue Smith	48
106	Sam Clark	52
104	Bob Smith	33
101	Dan Roberts	45
110	Tom Snyder	41
108	Larry Fitch	62
103	Paul Cruz	29
107	Nancy Angelli	49

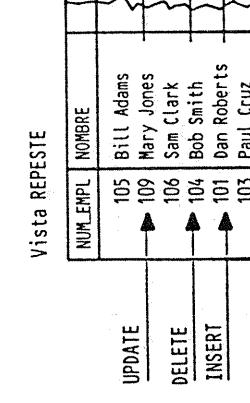


Figura 14.5. Actualización de datos a través de una vista.

Actualizaciones de vistas y el estándar ANSI/ISO

El estándar SQL ANSI/ISO especifica las vistas que deben ser actualizables en una base de datos que reclama conformidad con el estándar. Bajo el estándar, una vista puede ser actualizada si la consulta que la define satisface todas estas restricciones:

- No debe especificarse DISTINCT, es decir, las filas duplicadas no deben ser eliminadas de los resultados de la consulta.
- La cláusula FROM debe especificar solamente una tabla actualizable; es decir, la vista debe tener una única tabla fuente para la cual el usuario tiene los privilegios requeridos. Si la tabla fuente es ella misma una vista, entonces esa vista debe satisfacer estos criterios.
- Cada elemento de selección debe ser una referencia de columna simple; la lista de selección no puede contener expresiones, columnas calculadas o funciones de columna.
- La cláusula WHERE no debe incluir una subconsulta; sólo pueden aparecer condiciones de búsqueda simples fila a fila.
- La consulta no debe incluir una cláusula GROUP BY o HAVING.
- La consulta debe satisfacer las siguientes restricciones:

 - El concepto básico que subyace en estas restricciones es más fácil de recordar que las propias reglas:

Para que una vista sea actualizable, el DBMS debe ser capaz de relacionar cualquier fila de la vista con su fila fuente en la tabla fuente. Análogamente, el DBMS debe ser capaz de relacionar cada columna individual a actualizar con su columna fuente en la tabla fuente.

Si la vista satisface esta comprobación, es posible definir operaciones INSERT, DELETE y UPDATE significativas para la vista en términos de la tabla fuente.

Actualizaciones de vistas en productos SQL comerciales

Las reglas ANSI/ISO sobre actualizaciones de vista son muy restrictivas. Hay muchas vistas que pueden ser actualizadas teóricamente y que no satisfacen todas las restricciones. Además, hay vistas que pueden soportar algunas de las operaciones de actualización pero no otras, y hay vistas que pueden soportar actualizaciones sobre ciertas columnas pero no sobre otras. La mayoría de las implementaciones SQL comerciales tienen reglas de actualización de vistas que son considerablemente más permissivas que el estándar ANSI/ISO. Por ejemplo, consideremos esta vista:

Crea una vista mostrando las ventas, cuota y la diferencia entre ambas para cada vendedor.

```
CREATE VIEW RVENTAS (NUM_EMPL, VENTAS, CUOTA, DIFE) AS
SELECT NUM_EMPL, VENTAS, CUOTA, (VENTAS - CUOTA)
FROM REPVENTAS
```

El estándar ANSI/ISO no permite actualizaciones de esta vista, ya que su cuarta columna es una columna calculada. Sin embargo, observe que cada fila de esta vista puede ser relacionada con una fila única en la tabla fuente (REPVENTAS). Por esta razón DB2 (y otras varias implementaciones SQL comerciales) permitirán operaciones DELETE dentro de esta vista. Además, DB2 permite operaciones UPDATE sobre las columnas NUM_EMPL, VENTAS y CUOTA, ya que se derivan directamente de la tabla fuente. Unicamente la columna DIFE no puede ser actualizada. DB2 no permite la sentencia INSERT para la vista, ya que insertar un valor para la columna DIFE no tendría significado.

Las reglas específicas que determinan si una vista puede ser actualizada o no varían de un producto DBMS a otro, y suelen estar bastante detalladas. Algunas vistas, tales como las basadas en consultas agrupadas, no pueden ser actualizadas por ningún DBMS, ya que las operaciones de actualización simplemente no tienen sentido. Otras vistas pueden ser actualizables en un producto DBMS, parcialmente actualizables en otro producto y no actualizables en un tercero. El mejor modo de determinar la posibilidad de actualización de las vistas en un DBMS particular es consultar la guía de usuario o experimentar con diferentes tipos de vistas.

Comprobación de actualizaciones de vistas (CHECK OPTION)

Si una vista se define mediante una consulta que incluye una cláusula WHERE, sólo las filas que satisfacen la condición de búsqueda son visibles en la vista. Otras filas pueden estar presentes en la tabla fuente de la cual se deriva la vista, pero no son visibles a través de la vista. Por ejemplo, la vista REPESTE descrita anteriormente en este capítulo sólo contiene las filas de la tabla REPVENTAS con valores específicos en la columna OFICINA REP:

Crea una vista que muestre los vendedores de la región Este.

```
CREATE VIEW REPESTE AS
SELECT *
FROM REPVENTAS
WHERE OFICINA REP IN (11, 12, 13)
```

Esta es una vista actualizable según el estándar ANSI/ISO y para la mayoría de

las implementaciones SQL comerciales. Se puede añadir un nuevo vendedor con esta sentencia INSERT:

```
INSERT INTO REPESTE (NUM_EMPL, NOMBRE, OFICINA REP, EDAD, VENTAS)
VALUES (113, 'Jake Kimball', 114, 43, 0.00)
```

El DBMS añadirá la nueva fila a la tabla REVENTAS subyacente, y la fila será visible a través de la vista REPESTE. Pero consideremos qué ocurre cuando se añade un nuevo vendedor con esta sentencia INSERT:

```
INSERT INTO REPESTE (NUM_EMPL, NOMBRE, OFICINA REP, EDAD, VENTAS)
VALUES (114, 'Fred Roberts', 21, 47, 0.00)
```

Esta es una sentencia SQL perfectamente legal, y el DBMS la insertará en la tabla REVENTAS. Sin embargo, la fila recién insertada no satisface la condición de búsqueda para la vista. Su valor de OFICINA REP (21) especifica la oficina de Los Angeles, que está en la región Oeste. Como resultado, si se efectúa esta consulta inmediatamente después de la sentencia INSERT:

```
SELECT NUM_EMPL, NOMBRE, OFICINA REP
FROM REPESTE
```

NUM_EMPL	NOMBRE	OFICINA REP
105	Bill Adams	13
109	Mary Jones	11
106	Sam Clark	11
104	Bob Smith	12
101	Dan Roberts	12
103	Paul Cruz	12
107	Nancy Angelini	22
114	Fred Roberts	21

la fila recién añadida no aparecerá en la vista. Lo mismo ocurre si se cambia la asignación de oficina para uno de los vendedores actualmente en la vista. Esta sentencia UPDATE:

```
UPDATE REPESTE
SET OFICINA REP = 21
WHERE NUM_EMPL = 104
```

modifica una de las columnas para la fila de Bob Smith e inmediatamente hace que desaparezca de la vista. Naturalmente ambas filas «desaparecidas» aparecen en una consulta con respecto a la tabla subyacente:

```
SELECT NUM_EMPL, NOMBRE, OFICINA REP
FROM REVENTAS
-----
```

NUM_EMPL	NOMBRE	OFICINA REP
105	Bill Adams	13
109	Mary Jones	11
102	Sue Smith	21
106	Sam Clark	11
104	Bob Smith	21
101	Dan Roberts	12
110	Tom Snyder	NULL
108	Larry Fitch	21
103	Paul Cruz	12
107	Nancy Angelini	22
114	Fred Roberts	21

El hecho de que las filas desaparezcan de la vista como resultado de una sentencia INSERT o UPDATE es desconcertante, como poco. Probablemente es de desear que el DBMS detecte e impida este tipo de INSERT o UPDATE sobre la vista. SQL permite especificar este tipo de comprobación de integridad para vistas mediante la creación de la vista con una opción de *comprobación*. La opción de comprobación se especifica en la sentencia CREATE VIEW, como se muestra en esta definición de la vista REPESTE:

```
CREATE VIEW REPESTE AS
SELECT *
FROM REVENTAS
WHERE OFICINA REP IN (11, 12, 13)
WITH CHECK OPTION
```

Cuando se solicita la opción de comprobación para una vista, SQL comprueba *automáticamente* cada operación INSERT y cada operación UPDATE sobre la vista para asegurarse que las filas resultantes satisfagan el criterio de búsqueda de la definición de la vista. Si una fila insertada o modificada no satisface la condición, la sentencia INSERT o UPDATE fallaría y la operación no se llevaría a cabo.

La opción de comprobación añade recargo a las operaciones INSERT y UPDATE, pero ayuda a asegurar la integridad de la base de datos. Cuando se crea una vista actualizable como parte de un esquema de seguridad, debería siempre especificarse la opción de comprobación. Esto impide que modificaciones efectuadas a través de la vista afecten a datos que no son accesibles al usuario en primera instancia.

Las vistas permiten redefinir la estructura de una base de datos, proporcionando a cada usuario una vista personalizada de la estructura y los contenidos de la base de datos:

- Una vista es una tabla virtual definida mediante una consulta. La vista parece contener filas y columnas de datos, al igual que una tabla «real», pero los datos visibles a través de la vista son, de hecho, los resultados de la consulta.
- Una vista puede ser un subconjunto simple fila/columna de una única tabla, puede summarizar una tabla (una vista agrupada) o puede extraer sus datos de dos o más tablas (una vista compuesta).
- Una vista puede ser referenciada como una tabla real en una sentencia SELECT, INSERT, DELETE o UPDATE. Sin embargo, las vistas más complejas no pueden ser actualizadas; son vistas de sólo lectura.

- Las vistas suelen utilizarse para simplificar la estructura aparente de una base de datos, para simplificar consultas y para proteger ciertas filas y/o columnas frente al acceso no autorizado.

15 Seguridad SQL

Cuando alguien confía sus datos a un sistema de gestión de base de datos, la seguridad de los datos almacenados es una preocupación primordial. La seguridad es especialmente importante en un DBMS basado en SQL, puesto que SQL interactivo hace el acceso a la base de datos muy sencillo. Los requerimientos de seguridad de una base de datos de producción típica son muchos y muy variados:

- Los datos de cualquier tabla dada deberían ser accesibles a algunos usuarios, pero el acceso a otros debería ser impedido.
- Algunos usuarios deberían tener permitido actualizar datos en una tabla particular; a otros sólo se les debería permitir recuperar datos.
- Para algunas tablas, el acceso debería estar restringido en base a las columnas.
- Algunos usuarios deberían tener denegado el acceso mediante SQL interactivo a una tabla, pero se les debería permitir utilizar programas de aplicación que actualicen la tabla.

El esquema de seguridad SQL, descrito en este capítulo, proporciona estos tipos de protección para datos en una base de datos relacional.

Conceptos de seguridad SQL

La implementación de un esquema de seguridad y el reforzamiento de las restricciones de seguridad son responsabilidad del software DBMS. El lenguaje SQL define un panorama general para la seguridad de la base de datos, y las

sentencias SQL se utilizan para especificar restricciones de seguridad. El esquema de seguridad SQL se basa en tres conceptos principales:

- Los *usuarios* son los actores de la base de datos. Cada vez que el DBMS recupera, inserta, suprime o actualiza datos, lo hace a cuenta de algún usuario. El DBMS permitirá o prohibirá la acción dependiendo de qué usuario esté efectuando la petición.
- Los *objetos de la base de datos* son los elementos a los cuales se puede aplicar la protección de seguridad SQL. La seguridad se aplica generalmente a tablas y vistas, pero otros objetos tales como formularios, programas de aplicación y bases de datos enteras también pueden ser protegidos. La mayoría de los usuarios tendrán permiso para utilizar ciertos objetos de la base de datos, pero tendrán prohibido el uso de otros.
- Los *privilegios* son las acciones que un usuario tiene permitido efectuar para un determinado objeto de la base de datos. Un usuario puede tener permiso para SELECT e INSERT sobre filas en una tabla determinada, por ejemplo, pero puede carecer de permiso para DELETE o UPDATE filas de la tabla. Un usuario diferente puede tener un conjunto diferente de privilegios.

La Figura 15.1 muestra cómo podrían ser utilizados estos conceptos de seguridad en un esquema de seguridad para la base de datos ejemplo. Para establecer un esquema de seguridad en una base de datos, se utiliza la sentencia de SQL GRANT para especificar qué usuarios tienen qué privilegios sobre qué objetos de la base de datos. Por ejemplo, he aquí una sentencia GRANT que permite a Sam Clark recuperar e insertar datos en la tabla OFICINAS de la base de datos ejemplo:

```
Permite a Sam Clark recuperar e insertar datos en la tabla OFICINAS.
GRANT SELECT, INSERT
ON OFICINAS
TO SAM
```

La sentencia GRANT especifica una combinación de un identificador de usuario (SAM), un objeto (la tabla OFICINAS) y privilegios (SELECT e INSERT). Una vez concedidos, los privilegios pueden ser rescindidos más tarde con esta sentencia REVOKE:

```
Revoca los privilegios concedidos anteriormente a Sam Clark.
REVOKE SELECT, INSERT
ON OFICINAS
FROM SAM
```

Las sentencias GRANT y REVOKE se describen con detalle posteriormente en este capítulo.

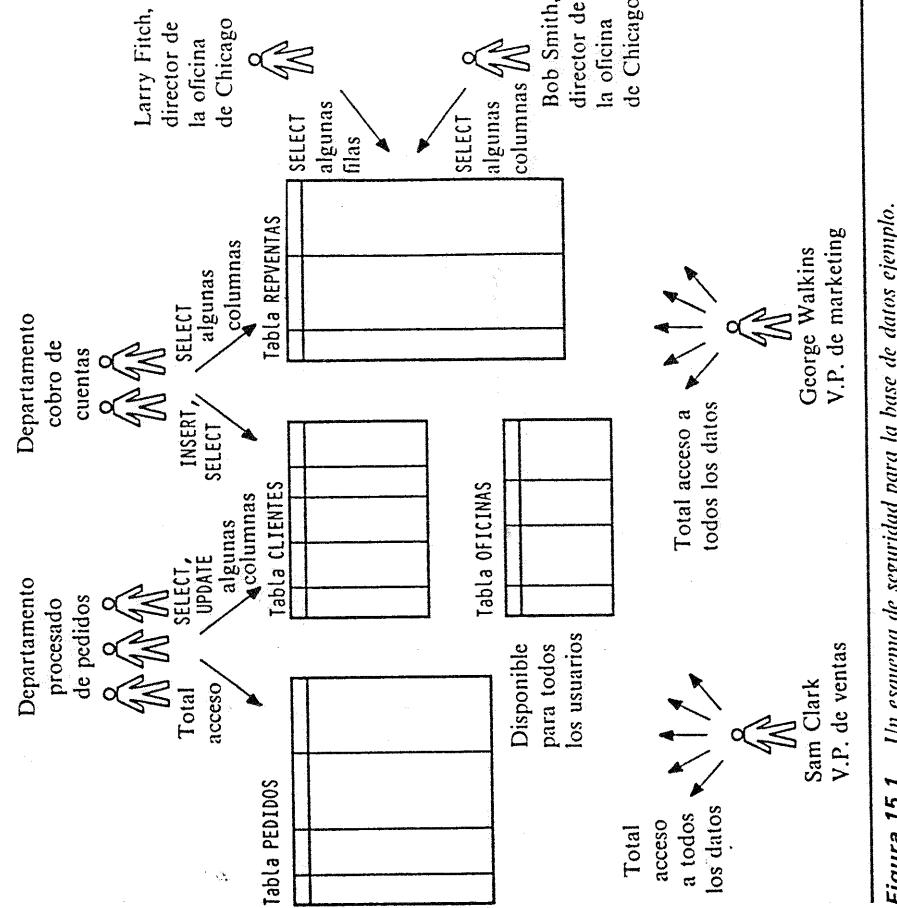


Figura 15.1. Un esquema de seguridad para la base de datos ejemplo.

Identificadores de usuario (id-usuario)

Cada usuario de una base de datos basada en SQL tiene asignado un *id-usuario*, un nombre breve que identifica al usuario dentro del software DBMS. El id-usuario se encuentra en el núcleo de la seguridad SQL. Toda sentencia SQL ejecutada por el DBMS se lleva a cabo a cuenta de un id-usuario específico. El id-usuario determina si la sentencia va a ser permitida o prohibida por el DBMS. En una base de datos de producción los id-usuario son asignados por el administrador. En una base de datos sobre un computador personal, puede haber únicamente un solo id-usuario, que identifica al usuario que ha creado y que tiene la propiedad de la base de datos.

El estándar SQL ANSI/ISO permite que los id-usuario tengan hasta 18 caracteres y requiere que sean nombres SQL válidos, pero muchos productos DBMS comerciales tienen diferentes restricciones. En DB2 y SQL/DS, por ejemplo, los id-usuario no pueden tener más de ocho caracteres. En Sybase y SQL Server, los id-usuario pueden tener hasta 30 caracteres. Si la portabilidad es una preocupación, es mejor limitar los id-usuario a ocho o menos caracteres. La Figura 15.2 muestra a varios usuarios que necesitan acceso a la base de datos ejemplo e id-usuario típicos asignados a ellos. Observe que todos los usuarios en el departamento de procesado de pedidos pueden tener asignados el mismo id-usuario, puesto que tienen todos idénticos privilegios en la base de datos.

El estándar SQL ANSI/ISO utiliza el término id-autorización en vez de id-usuario, y usted encontrará ocasionalmente este término utilizado en otra documentación SQL. Técnicamente, «id-autorización» es un término más preciso, puesto que el papel de la identificación (id) es determinar la autorización o privilegios en la base de datos. Existen situaciones, como en la Figura 15.2, donde tiene sentido asignar el mismo id-usuario a diferentes usuarios. En otras situaciones, una sola persona puede utilizar dos o tres id-usuario diferentes. En una base de datos de producción, los id-autorización pueden ir asociados con programas y grupos de programas, en vez de con usuarios humanos. En cada una de estas situaciones, el «id-autorización» es un término más preciso y menos confuso que «id-usuario». Sin embargo, la práctica más común es asignar un id-usuario diferente a cada persona, y la mayoría de los DBMS basados en SQL utilizan el término «id-usuario» en su documentación.

Validación de usuario. El estándar SQL ANSI/ISO especifica que los id-usuario proporcionan seguridad en la base de datos, pero no dice nada con respecto al mecanismo de asociar un id-usuario con una sentencia SQL. Por ejemplo, cuando se escriben sentencias SQL en una utilidad SQL interactiva ¿cómo determina el DBMS qué id-usuario está asociado con las sentencias? La mayoría de las implementaciones SQL comerciales establecen un id-usuario para cada sesión de base de datos. En SQL interactivo, la sesión comienza cuando arranca el programa SQL interactivo, y dura hasta que se abandona el programa. En un programa de aplicación que utiliza SQL programado, la sesión comienza cuando el programa de aplicación se conecta al DBMS, y finaliza cuando el programa de aplicación termina. Todas las sentencias SQL utilizadas durante la sesión están asociadas con el id-usuario especificado para la sesión. Generalmente debe surgióndose tanto un id-usuario como una contraseña asociada al comienzo de una sesión. El DBMS comprueba la contraseña para verificar que el usuario está, en efecto, autorizado a utilizar el id-usuario que suministra. Aunque los id-usuario y las contraseñas son habituales en la mayoría de los productos SQL, las técnicas específicas utilizadas para especificar el id-usuario y la contraseña varían de un producto a otro.

Algunos productos DBMS implementan su propia seguridad id-usuario/con-

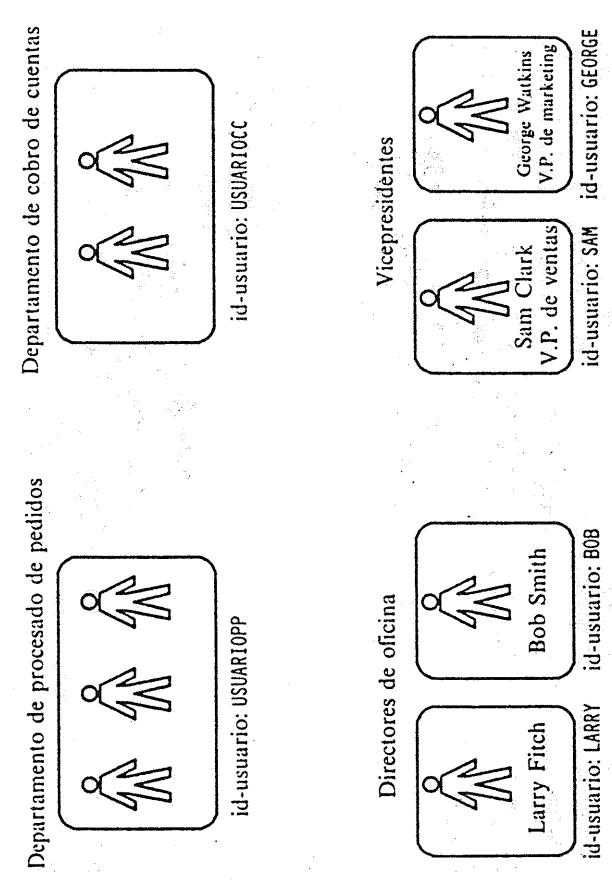


Figura 15.2. Asignaciones de id-usuario para la base de datos ejemplo.

traseña. Por ejemplo, cuando se utiliza el programa SQL interactivo Oracle, llamado SQLPLUS, hay que especificar un nombre de usuario y la contraseña asociada en la orden que arranca el programa, de este modo:

SQLPLUS SCOTT/TIGRE

El programa SQL interactivo de Sybase, llamado ISQL, también acepta un nombre de usuario y contraseña, utilizando este formato de orden:

ISQL /USER=SCOTT /PASSWORD=TIGRE

En cada caso, el DBMS valida el id-usuario (SCOTT) y la contraseña (TIGRE) antes de comenzar la sesión SQL interactiva.

Muchos otros productos DBMS, incluyendo Ingres e Informix, utilizan los nombres de usuario del sistema operativo del computador como id-usuario de la base de datos. Por ejemplo, cuando usted se conecta a un sistema informático VAX/VMS, debe suministrar un nombre de usuario VMS válido y una contraseña para obtener acceso. Para iniciar la utilidad SQL interactiva de Ingres, simplemente tiene que dar la orden:

ISQL BOVENTAS

dónde BDVENTAS es el nombre de la base de datos Ingres que usted desea utilizar. Ingres obtiene automáticamente el nombre de usuario VMS y lo convierte en el id-usuario de Ingres para la sesión. Por tanto usted no tiene que especificar un id-usuario de base de datos y una contraseña aparte. El SQL interactivo de DB2, que corre bajo MVS/TSO, utiliza una técnica análoga. El nombre de presentación ante TSO se convierte automáticamente en el id-usuario de DB2 para la sesión SQL interactiva.

La seguridad también se aplica al acceso programado a una base de datos, de modo que el DBMS debe determinar y validar el id-usuario para cada programa de aplicación que pretenda acceder a la base de datos. De nuevo, las técnicas y reglas para establecer el id-usuario varían de un producto de DBMS a otro. Se describirán en el Capítulo 17, que trata del SQL programado.

Grupos de usuario. En una base de datos de producción grande existen con frecuencia grupos de usuario con necesidades similares. En la base de datos ejemplo, por ejemplo, las tres personas del departamento de procesado de pedidos forman un grupo natural de usuarios, y las dos personas del departamento de cobro de cuentas forman otro grupo natural. Dentro de cada grupo, todos los usuarios tienen necesidades idénticas para acceder a los datos y deberían tener privilegios idénticos.

Bajo el esquema de seguridad de SQL ANSI/ISO, se pueden manejar grupos de usuario con similares necesidades en uno de dos modos:

- Se puede asignar el mismo id-usuario a todas las personas del grupo, como se muestra en la Figura 15.2. Este esquema simplifica la administración de la seguridad, ya que permite especificar los privilegios de acceso a datos una vez para el único id-usuario. Sin embargo, bajo este esquema las personas que comparten el id-usuario no pueden distinguirse unas de otras en las visualizaciones del operador del sistema ni en los informes del DBMS.

- Se puede asignar un id-usuario diferente a cada persona del grupo. Este esquema permite diferenciar a los usuarios en los informes producidos por el DBMS, y permite establecer privilegios diferentes para los usuarios individuales con posterioridad. Sin embargo, deben especificarse privilegios para cada usuario individualmente, haciendo la administración de la seguridad tediosa y propensa a errores.

El esquema que se elija dependerá de los compromisos en cada base de datos y aplicación particular. Sybase y SQL Server ofrecen una tercera alternativa para manejar grupos de usuarios similares. Soportan id-grupo, que identifica grupos de id-usuario relacionados. Los privilegios pueden ser concedidos tanto a id-usuarios individuales como a id-grupo, y un usuario puede efectuar una acción de base de datos si se le permite por su privilegio bien como id-usuario o bien como id-grupo. Los id-

grupos simplifican por tanto la administración de privilegios proporcionados a grupos de usuarios. Sin embargo, no son estándar y son específicos de SQL Server y Sybase.

DB2 también soporta grupos de usuarios, pero adopta un planteamiento diferente. El administrador de la base de datos en DB2 puede configurar al DB2 de modo que cuando alguien conecte por primera vez a DB2 y suministre su id-usuario (conocido como *id-autorización primario*), DB2 examina automáticamente un conjunto de id-usuarios adicionales (conocido como *id-autorización secundario*) que puede utilizarse. Cuando DB2 comprueba posteriormente los privilegios, examina los privilegios de todos los id-autorización, primario y secundario. El administrador de la base de datos en DB2 establece normalmente los id-autorización secundarios para que sean los mismos que los nombres de grupo de usuario utilizados por RACF, la facilidad de seguridad en mainframe IBM. Así que el mecanismo de DB2 proporciona efectivamente id-grupos, pero lo hace sin añadir al mecanismo id-usuario.

Objetos de seguridad

Las protecciones de seguridad SQL se aplican a *objetos* específicos contenidos en una base de datos. El estándar ANSI/ISO especifica dos tipos de objetos de seguridad —tablas y vistas—. Así cada tabla y cada vista pueden ser individualmente protegidas. El acceso a una tabla o vista puede ser permitido por ciertos id-usuario y prohibido para otros id-usuario.

La mayoría de los productos SQL comerciales soportan objetos de seguridad adicionales. En una base de datos SQL Server, por ejemplo, un procedimiento almacenado es un objeto importante de la base de datos. El esquema de seguridad SQL determina qué usuarios pueden crear y suprimir procedimientos almacenados y qué usuarios tienen permiso ejecutarlos. En el DB2 de IBM, los espacios de tablas físicos en donde se almacenan las tablas son tratados como objetos de seguridad. El administrador de la base de datos puede dar permiso a algunos id-usuario para crear nuevas tablas en un espacio de tablas particular y denegar ese permiso a otros id-usuario. Otras implementaciones SQL soportan otros objetos de seguridad.

Privilegios

El conjunto de acciones que un usuario puede efectuar sobre un objeto de base de datos se denomina los *privilegios* para el objeto. El estándar SQL ANSI/ISO especifica cuatro privilegios para tablas y vistas:

- El privilegio SELECT permite recuperar datos de una tabla o vista. Con este

privilegio, se puede especificar la tabla o vista en la cláusula `FROM` de una sentencia `SELECT` o subconsulta.

■ El privilegio `INSERT` permite insertar nuevas filas en una tabla o vista. Con este privilegio, se puede especificar la tabla o vista en la cláusula `INTO` de una sentencia `INSERT`.

■ El privilegio `DELETE` permite eliminar filas de datos de una tabla o vista. Con este privilegio, se puede especificar la tabla o vista en la cláusula `FROM` de una sentencia `DELETE`.

■ El privilegio `UPDATE` permite modificar filas de datos en una tabla o vista. Con este privilegio, se puede especificar la tabla o vista como tabla destino en una sentencia `UPDATE`. El privilegio `UPDATE` puede restringirse a columnas específicas de la tabla o vista, permitiendo actualizaciones de estas columnas, pero desautorizando actualizaciones de cualesquier otras columnas.

Estos cuatro privilegios son soportados por virtualmente todos los productos SQL comerciales.

Privilegios de propiedad. Cuando se crea una tabla con la sentencia `CREATE TABLE`, quien lo hace se convierte en su propietario y recibe totales privilegios para la tabla (`SELECT, INSERT, UPDATE, DELETE, ALTER`) y cualesquier otros privilegios soportados por el DBMS. Otros usuarios no tiene inicialmente privilegios sobre la tabla recién creada. Si se les va a proporcionar acceso a la tabla, hay que concederles específicamente privilegios, utilizando la sentencia `GRANT`.

Si usted crea una vista con la sentencia `CREATE VIEW`, se convierte en el propietario de la vista, pero no recibe necesariamente privilegios totales sobre ella. Para crear la vista con éxito debe ya tener el privilegio `SELECT` sobre cada una de las tablas fuente para la vista; por tanto el DBMS le proporciona el privilegio `SELECT` para la vista automáticamente. Para cada uno de los otros privilegios (`INSERT, DELETE` y `UPDATE`), el DBMS proporciona el privilegio sobre la vista solamente si se dispone de ese mismo privilegio sobre *todas* las tablas fuente para la vista.

Otros privilegios. Muchos productos DBMS comerciales ofrecen privilegios adicionales sobre tablas y vistas además de los privilegios `SELECT, INSERT, DELETE` y `UPDATE` especificados en el estándar SQL ANSI/ISO. Por ejemplo, las bases de datos sobre mainicomputadores de IBM (DB2 y SQL/DS) y Oracle soportan un privilegio `ALTER` y un privilegio `INDEX` para las tablas. Un usuario con el privilegio `ALTER` sobre una tabla particular puede utilizar la sentencia `ALTER TABLE` para modificar la definición de la tabla; un usuario con el privilegio `INDEX` puede crear un índice para la tabla con la sentencia `CREATE INDEX`. En productos DBMS que no soportan los privilegios `ALTER` e `INDEX`, únicamente el propietario puede utilizar las sentencias `ALTER TABLE` y `CREATE INDEX`.

Otro ejemplo de privilegios adicionales sobre tablas y vistas es el privilegio `SELECT` extendido ofrecido por Sybase y SQL Server. Al igual que el privilegio `UPDATE` de ANSI/ISO, este privilegio `SELECT` puede ser especificado para columnas individuales de una tabla o vista. Por tanto un usuario puede tener permitido recuperar ciertas columnas de una tabla mientras que otro usuario está restringido a otras columnas.

Frecuentemente se soportan privilegios adicionales para objetos de seguridad DBMS diferentes a tablas y vistas. Por ejemplo, Sybase y SQL Server soportan un privilegio `EXECUTE` para procedimientos almacenados, que determina si un usuario tiene permitido ejecutar un procedimiento de tablas, que determina si un usuario tiene permitido usar un espacio de tablas, que determina si un usuario puede crear tablas en un espacio de tablas específico. Estos privilegios son típicamente administrados utilizando las sentencias `GRANT` y `REVOKE`, del mismo modo que los privilegios estándar ANSI/ISO.

Vistas y seguridad SQL

Aemás de las restricciones sobre acceso a tabla proporcionadas por los privilegios SQL, las vistas también juegan un papel clave en la seguridad SQL. Definiendo cuidadosamente una vista y proporcionando un permiso de usuario para acceder a ella pero no a sus tablas fuente, se puede restringir efectivamente el acceso de un usuario a únicamente columnas y filas seleccionadas. Las vistas ofrecen por tanto un modo de ejercitarse un control muy preciso sobre qué datos se hacen visibles a qué usuarios.

Por ejemplo, supongamos que se desea forzar esta regla de seguridad en la base de datos ejemplo:

El personal del departamento de cobro de cuentas debe poder recuperar los números y nombres de los empleados y los números de oficina de la tabla `REPENTAS`, pero los datos referentes a ventas y cuotas no deben tenerlos disponibles.

Se puede implementar esta regla de seguridad definiendo una vista del modo siguiente:

```
CREATE VIEW INFOREP AS
  SELECT NUM_EMPL, NOMBRE, OFICINA REP
    FROM REPENTAS
```

y dando el privilegio `SELECT` para la vista al id-usuario `USUARIOCC`, como se muestra en la Figura 15.3. Este ejemplo utiliza una vista vertical para restringir el acceso a columnas específicas.

Las vistas horizontales son también efectivas para forzar reglas de seguridad como ésta:

Los directores de venta de cada región deben tener acceso completo a los datos de REPVENTAS referentes a los vendedores asignados a esa región.

Como se muestra en la Figura 15.4, se pueden definir dos vistas, VISTASESTE y VISTASOESTE, que contengan los datos de REPVENTAS referentes a cada una de las dos regiones, y luego conceder a cada director de oficina acceso a la vista adecuada.

Naturalmente las vistas pueden ser mucho más complejas que los sencillos subconjuntos de filas y columnas de una tabla única mostrados en estos ejemplos. Definiendo una vista con una consulta agrupada, se puede dar a un usuario acceso a datos sumarios, pero no a las filas detalladas de la tabla subyacente. Una vista también puede combinar datos de dos o más tablas, proporcionando precisamente los datos necesitados por un usuario particular y denegando acceso a todo otro dato. La utilidad de las vistas para implementar seguridad SQL está limitada por las dos restricciones fundamentales descritas anteriormente en el Capítulo 14:

■ **Restricciones de actualización.** El privilegio SELECT puede ser utilizado con vistas de sólo lectura para limitar la recuperación de datos, pero los privilegios INSERT, DELETE y UPDATE no tienen significado para estas vistas. Si un usuario debe actualizar los datos visibles en una lista de sólo lectura, el usuario debe obtener permiso para actualizar las tablas subyacentes, y debe utilizar las sentencias INSERT, DELETE y UPDATE que refieren a esas tablas.

■ **Rendimiento.** Puesto que el DBMS traduce cada acceso a una vista en un acceso correspondiente a sus tablas fuente, las vistas pueden añadir recargos significativos a las operaciones de base de datos. Las vistas no pueden ser utilizadas indiscriminadamente para restringir el acceso a la base de datos sin causar que el rendimiento global de la base de datos descienda.

Concesión de privilegios (GRANT)

La sentencia GRANT, mostrada en la Figura 15.5, se utiliza para conceder privilegios de seguridad sobre objetos de la base de datos a usuarios específicos. Normalmente la sentencia GRANT es utilizada por el propietario de una tabla o vista para proporcionar a otros usuarios acceso a los datos. Como se muestra en la figura, la sentencia GRANT incluye una lista específica de los privilegios a conceder, el nombre de la tabla a la cual se aplican los privilegios y el id-usuario al cual los privilegios son concedidos.

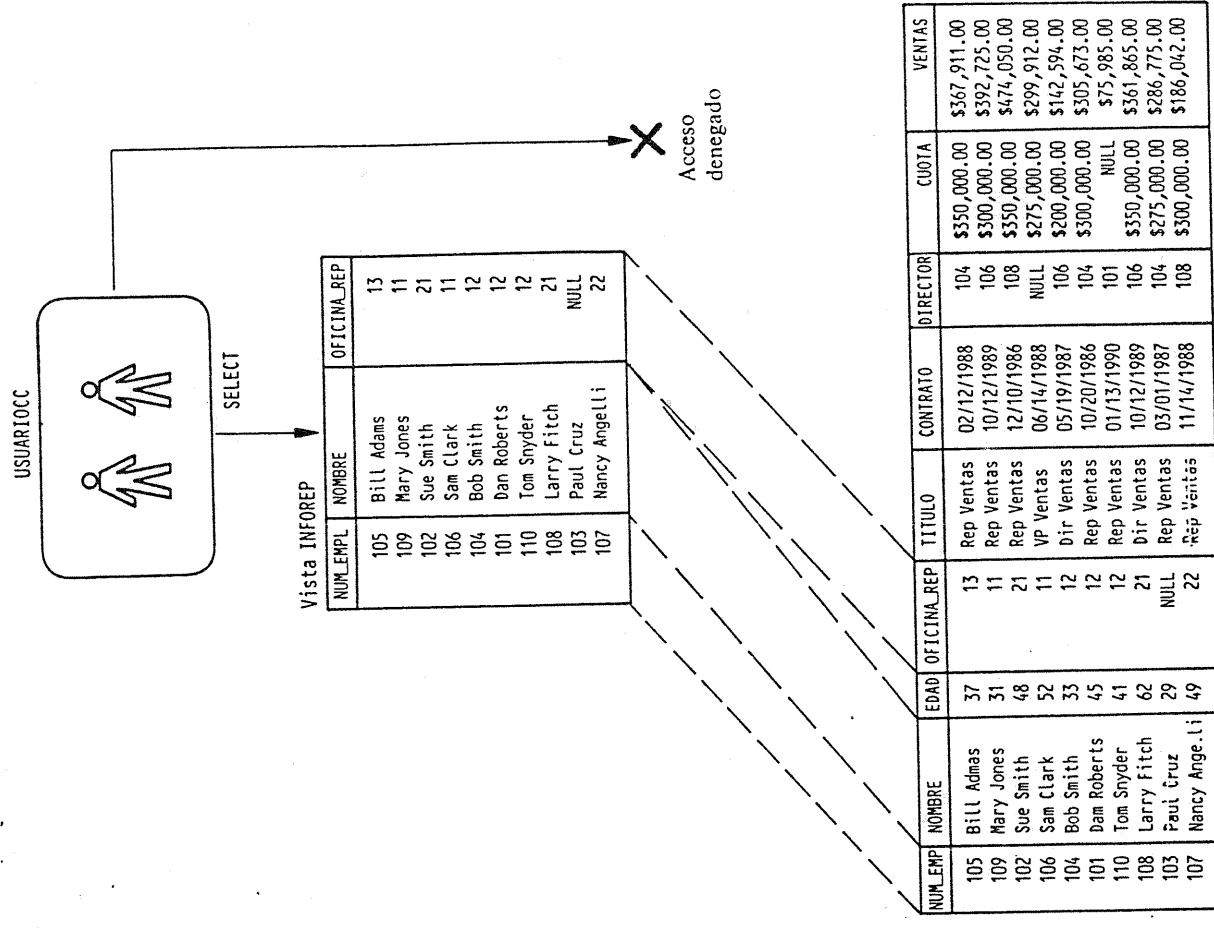


Figura 15.3. Utilización de una vista para restringir acceso a una columna.

La sentencia GRANT mostrada en el diagrama sintáctico se conforma al estándar SQL ANSI/ISO. Muchos productos DBMS siguen la sintaxis de la sentencia GRANT de DB2, que es más flexible. La sintaxis DB2 permite especificar una lista de id-usuarios y una lista de tablas, haciendo más simple conceder muchos privilegios de una vez. He aquí algunos ejemplos de sencillas sentencias GRANT para la base de datos ejemplo:

Proporciona a los usuarios de proceso de pedidos acceso completo a la tabla PEDIDOS.

```
GRANT SELECT, INSERT, DELETE, UPDATE
  ON PEDIDOS
  TO USUARIOPP
```

Permite a los usuarios de cobro de cuentas a recuperar datos del cliente y añadir nuevos clientes a la tabla CLIENTES, pero proporciona a los usuarios de proceso de pedidos acceso de sólo lectura.

```
GRANT SELECT, INSERT
  ON CLIENTES
  TO USUARIOCC
```

```
GRANT SELECT
  ON CLIENTES
  TO USUARIOPP
```

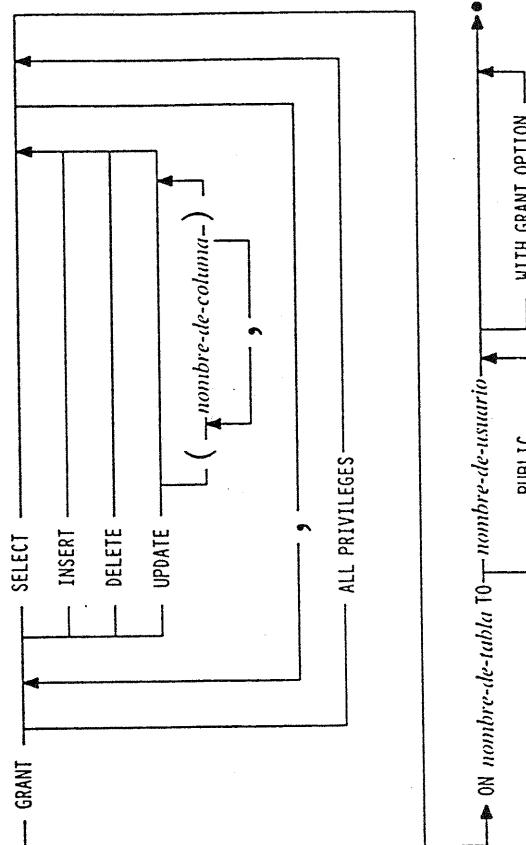


Figura 15.4. Utilización de vistas para restringir acceso a filas.

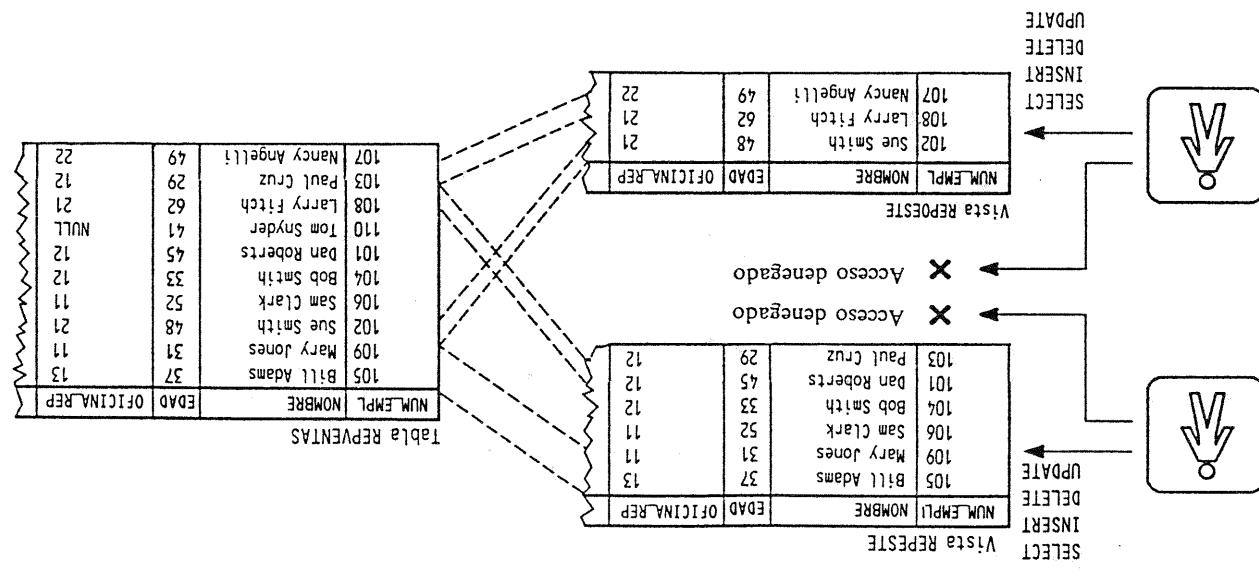


Figura 15.5. Diagrama sintáctico de la sentencia GRANT.

Permite a Sam Clark insertar o suprimir una oficina.

```
GRANT INSERT, DELETE
  ON OFICINAS
  TO SAM
```

Por conveniencia, la sentencia GRANT dispone de dos atajos que se pueden utilizar cuando se conceden muchos privilegios o cuando se conceden a muchos usuarios. En vez de listar específicamente todos los privilegios disponibles para un objeto particular, se puede utilizar las palabras claves ALL PRIVILEGES. Esta sentencia GRANT da a Sam Clark, el vicepresidente de ventas, acceso total a la tabla REPVENTAS:

Da privilegios totales sobre la tabla REPVENTAS a Sam Clark.

```
GRANT ALL PRIVILEGES
  ON REPVENTAS
  TO SAM
```

En vez de conceder privilegios a todos los usuarios de la base de datos uno a uno, se puede utilizar la palabra clave PUBLIC para conceder un privilegio a todo usuario autorizado de la base de datos. Esta sentencia GRANT permite a cualquiera recuperar datos de la tabla OFICINAS:

Da a todos los usuarios acceso SELECT a la tabla OFICINAS.

```
GRANT SELECT
  ON OFICINAS
  TO PUBLIC
```

Observe que esta sentencia GRANT concede acceso a todos los usuarios autorizados presentes y futuros, no solamente a los id-usuarios actualmente conocidos al DBMS. Esto elimina la necesidad de conceder explícitamente privilegios a los nuevos usuarios conforme son autorizados.

Privilegios de columna

El estándar SQL ANSI/ISO permite conceder el privilegio UPDATE para columnas individuales de una tabla o vista. Las columnas actualizables se listan tras la palabra clave UPDATE y se encierran entre paréntesis. He aquí una sentencia UPDATE que permite al departamento de procesado de pedidos actualizar solamente las columnas del nombre de la empresa y del vendedor en la tabla CLIENTES:

Permite a los usuarios de procesado de pedidos modificar los nombres de empresa y las asignaciones de vendedores.

```
GRANT UPDATE (EMPRESA, REP_CLIENTE)
  ON CLIENTES
  TO USUARIOFP
```

Si la lista de columnas se omite, el privilegio UPDATE se aplica a todas las columnas de la tabla o vista, como en este ejemplo:

Permite a los usuarios de cobro de cuentas modificar cualquier información de cliente.

```
GRANT UPDATE
  ON CLIENTES
  TO USUARIOC
```

El estándar ANSI/ISO no permite una lista de columnas para el privilegio SELECT; requiere que el privilegio SELECT se aplique a todas las columnas de una tabla o vista. En la práctica, ésta no es una restricción importante. Para conceder acceso a columnas específicas se define primero una vista sobre la tabla que incluya sólo a esas columnas, y luego se concede el privilegio SELECT para la vista únicamente, como describimos anteriormente en este capítulo. Sin embargo, las vistas definidas únicamente con propósitos de seguridad pueden embarullar la estructura de una base de datos que de otra manera resultaría sencilla. Por esta razón algunos productos DBMS permiten una lista de columnas para el privilegio SELECT. Por ejemplo, la siguiente sentencia GRANT es legal para los productos DBMS Sybase, SQL Server e Informix:

Proporciona a los usuarios de cobro de cuentas acceso de sólo lectura a las columnas de número y nombre de empleado y oficina de ventas de la tabla REPVENTAS.

```
GRANT SELECT (NUM_EMPL, NOMBRE, OFICINA.REP)
  ON REPVENTAS
  TO USUARIOC
```

Esta sentencia GRANT elimina la necesidad de la vista INFOREP definida en la Figura 15.3, y en la práctica puede eliminar la necesidad de muchas vistas en una base de datos de producción. Sin embargo, el uso de una lista de columnas para el privilegio SELECT es único en ciertos dialectos SQL, y no está permitida por el estándar ANSI/ISO ni por los productos SQL de IBM.

Paso de privilegios (GRANT OPTION)

Cuando usted crea un objeto de base de datos y se convierte en su propietario, usted es la única persona que puede conceder privilegios para autorizar el objeto. Cuando concede privilegios a otros usuarios, éstos tienen permitido el

uso del objeto, pero no pueden transmitir esos privilegios a otros usuarios. De este modo el propietario de un objeto mantiene un control muy estricto sobre quién tiene permisos para utilizar el objeto y sobre qué formas de acceso se permiten.

Ocasionalmente puede desearse permitir a otros usuarios que concedan privilegios sobre un objeto que no es de su propiedad. Por ejemplo, consideremos una vez más las vistas REPOESTE y REPOESTE de la base de datos ejemplo. Sam Clark, el vicepresidente de ventas, creó estas vistas y tiene propiedad sobre ellas. El puede proporcionar al director de la oficina de Los Angeles, Larry Fitch, permiso para utilizar la vista REPOESTE con esta sentencia GRANT:

```
GRANT SELECT  
ON REPOESTE  
TO LARRY  
WITH GRANT OPTION
```

¿Qué ocurre si Larry desea conceder permiso a Sue Smith (id-usuario SUE) para acceder a los datos de REPOESTE debido a que ella está realizando algunas previsiones de ventas para la oficina de Los Angeles? Con la sentencia GRANT precedente, él no puede proporcionar el privilegio requerido. Únicamente Sam Clark puede conceder el privilegio, puesto que es el propietario de la vista.

Si Sam desea dar a Larry discreción sobre quién puede utilizar la vista REPOESTE, puede utilizar esta variante de la sentencia GRANT anterior:

```
GRANT SELECT  
ON REPOESTE  
TO LARRY  
WITH GRANT OPTION
```

Debido a la cláusula WITH GRANT OPTION, esta sentencia GRANT comporta, junto con los privilegios especificados, el derecho a conceder esos privilegios a otros usuarios.

Larry puede ahora emitir esta sentencia GRANT:

```
GRANT SELECT  
ON REPOESTE  
TO SUE
```

la cual permite a Sue Smith recuperar datos de la vista REPOESTE. La Figura 15.6 ilustra gráficamente el flujo de privilegios, primero de Sam Clark a Larry, y luego de Larry a Sue. Debido a que la sentencia GRANT emitida por Larry no incluía la cláusula WITH GRANT OPTION, la cadena de permisos finaliza con Sue; ella puede recuperar los datos de REPOESTE, pero no puede conceder acceso a otro usuario. Sin embargo, si la concesión de privilegios de Larry a Sue hubiera incluido la opción de concesión, la cadena podría continuar a otro nivel, permitiendo a Sue que concediera acceso a otros usuarios.

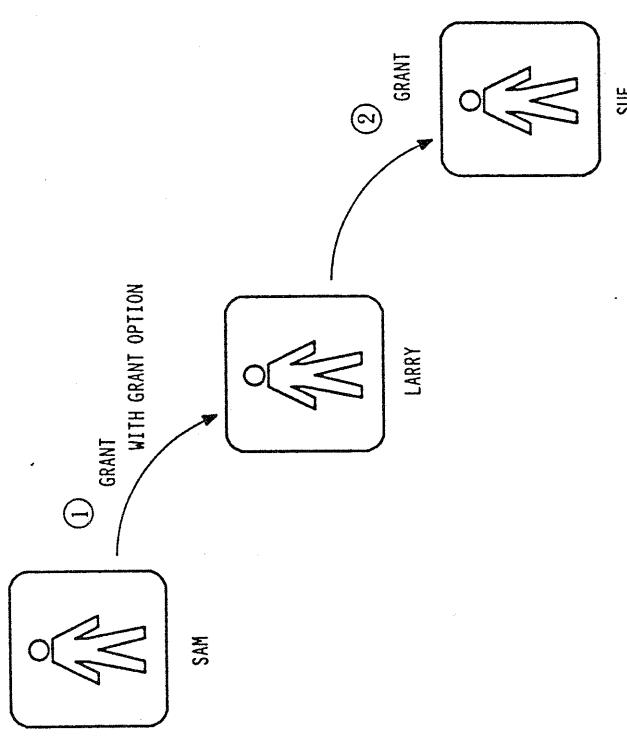


Figura 15.6. Utilización de la GRANT OPTION.

Alternativamente, Larry podría construir una vista para Sue que incluyera únicamente los vendedores de la oficina de Los Angeles y después le proporcionaría acceso a esa vista:

```
CREATE VIEW REPSLA AS  
SELECT *  
FROM REPOESTE  
WHERE OFICINA = 21  
  
GRANT ALL PRIVILEGES  
ON REPSLA  
TO SUE
```

Larry será el propietario de la vista REPSLA, pero no es el propietario de la vista REPOESTE de la cual ésta nueva se ha derivado. Para mantener la seguridad efectiva, el DBMS requiere que Larry no solamente tenga el privilegio SELECT sobre REPOESTE, sino que también exige que tenga la opción de concesión para ese privilegio antes de permitirle conceder el privilegio SELECT sobre REPSLA a Sue. Una vez que a un usuario se le han concedido ciertos privilegios con la

opción de concesión, ese usuario puede conceder estos privilegios y la opción de concesión a otros usuarios. Aquellos otros usuarios pueden, a su vez, continuar concediendo tanto los privilegios como la opción de concesión. Por esta razón se debería utilizar gran cuidado cuando se proporcionen a otros usuarios la opción de concesión. Observe que la opción de concesión se aplica únicamente a los privilegios específicos designados en la sentencia GRANT. Si se desean conceder ciertos privilegios con la opción de concesión y conceder otros privilegios sin ella, deben utilizarse dos sentencias GRANT separadas, como en este ejemplo:

Permite a Larry Fitch recuperar, insertar, actualizar y suprimir datos de la tabla REPOESTE, y también se le permite conceder permiso de recuperación a otros usuarios.

```
GRANT SELECT
   ON REPOESTE
   TO LARRY
   WITH GRANT OPTION
GRANT INSERT, DELETE, UPDATE
   ON REPOESTE
   TO LARRY
```

Revocación de privilegios (REVOKE)

En la mayoría de las bases de datos basadas en SQL, los privilegios que se han concedido con la sentencia GRANT pueden ser retirados con la sentencia REVOKE, mostrada en la Figura 15.7. La sentencia REVOKE tiene una estructura que se asemeja estrechamente a la sentencia GRANT, especificando un conjunto específico de privilegios a ser revocados, para un objeto de base de datos específico, para uno o más id-usuarios.

Una sentencia REVOKE puede retirar todos o parte de los privilegios que previamente se han concedido a un id-user. Por ejemplo, consideremos esta secuencia de sentencias:

Concede y luego revoca algunos privilegios sobre la tabla REPVENTAS.

```
GRANT SELECT, INSERT, UPDATE
   ON REPVENTAS
   TO USUARIOCC, USUARIOPP
REVOKE INSERT, UPDATE
   ON REPVENTAS
   FROM USUARIOPP
```

Los privilegios INSERT y UPDATE sobre la tabla REPVENTAS son primero concedidos a los dos usuarios y luego revocados a uno de ellos. Sin embargo, el privilegio

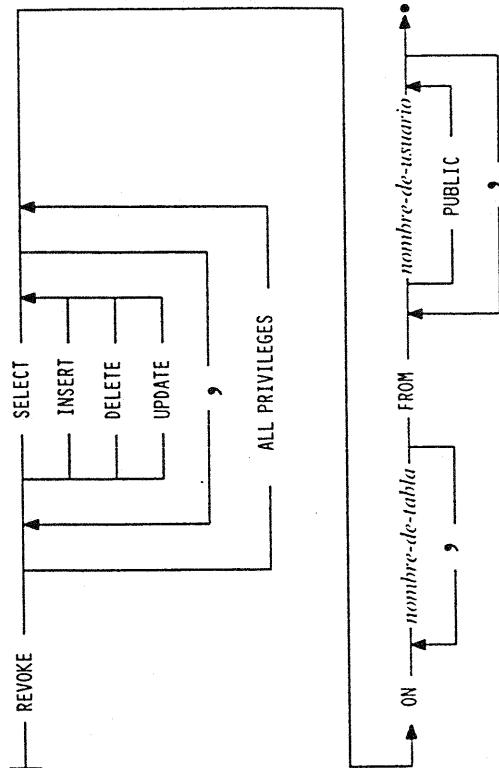


Figura 15.7. Diagrama sintáctico de la sentencia REVOKE.

SELECT permanece para ambos id-usuarios. He aquí algunos otros ejemplos de la sentencia REVOKE:

Retira todos los privilegios concedidos anteriormente sobre la tabla OFICINAS.

```
REVOKE ALL PRIVILEGES
   ON OFICINAS
   FROM USUARIOCC
```

Retira los privilegios UPDATE y DELETE a dos id-usuarios.

```
REVOKE UPDATE, DELETE
   ON OFICINAS
   FROM USUARIOCC, USUARIOPP
```

Retira todos los privilegios sobre la tabla OFICINAS que fueron anteriormente concedidos a todos los usuarios.

```
REVOKE ALL PRIVILEGES
   ON OFICINAS
   FROM PUBLIC
```

Cuando usted emite una sentencia REVOKE, solamente puede retirar aquellos privilegios que usted haya concedido previamente a otro usuario. Ese usuario puede tener además privilegios que le fueron concedidos por otros usuarios; esos

privilegios no quedan afectados por la sentencia REVOKE que usted haya emitido. Observe específicamente que si dos usuarios diferentes conceden el mismo privilegio sobre el mismo objeto a un usuario y uno de ellos posteriormente revoca el privilegio, la concesión del segundo usuario seguirá permitiendo al usuario acceder al objeto. Este manejo de «concesiones solapadas» de privilegios se ilustra en la secuencia ejemplo siguiente:

Supongamos que Sam Clark, el vicepresidente de ventas, proporciona a Larry Fitch privilegios SELECT para la tabla REPVENTAS, y privilegios SELECT y UPDATE para la tabla PEDIDOS, utilizando las sentencias siguientes:

```
GRANT SELECT  
ON REPVENTAS  
TO LARRY  
  
GRANT SELECT, UPDATE  
ON PEDIDOS  
TO LARRY
```

Unos pocos días más tarde George Watkins, el vicepresidente de marketing, proporciona a Larry los privilegios SELECT y DELETE para la tabla PEDIDOS, y el privilegio SELECT para la tabla CLIENTES, utilizando estas sentencias:

```
GRANT SELECT, DELETE  
ON PEDIDOS  
TO LARRY  
  
GRANT SELECT  
ON CLIENTES  
TO LARRY
```

Observe que Larry ha recibido privilegios sobre la tabla PEDIDOS desde dos fuentes diferentes. De hecho, el privilegio SELECT sobre la tabla PEDIDOS se le ha concedido por parte de ambas fuentes. Unos pocos días después, Sam revoca los privilegios que previamente concedió a Larry para la tabla PEDIDOS:

```
REVOKE SELECT, UPDATE  
ON PEDIDOS  
FROM LARRY
```

Después que el DBMS procesa la sentencia REVOKE, Larry sigue reteniendo el privilegio SELECT sobre la tabla REPVENTAS, los privilegios SELECT y DELETE sobre la tabla PEDIDOS y el privilegio SELECT sobre la tabla CLIENTES, pero ha perdido el privilegio UPDATE sobre la tabla PEDIDOS.

REVOKE y la opción de concesión

Cuando se conceden privilegios con la opción de concesión y posteriormente se revocan esos privilegios, la mayoría de los productos DBMS revocarán *automáticamente* todos los privilegios derivados de la concesión original. Consideremos una vez más la cadena de privilegios de la Figura 15.6, que va de Sam Clark, el vicepresidente de ventas, a Larry Fitch, el director de la oficina de Los Angeles, y luego a Sue Smith. Si Sam revoca ahora los privilegios de Larry para la vista REQUEST, el privilegio de Sue es revocado automáticamente también.

Esta situación se complica más si dos o más usuarios tienen privilegios concedidos y uno de ellos posteriormente revoca los privilegios. Consideremos la Figura 15.8, una ligera variación del último. Aquí Larry recibe el privilegio SELECT con la opción de concesión de Sam (el vicepresidente de ventas) y George (el vicepresidente de marketing), y luego concede privilegios a Sue. Esta vez cuando Sam revoca los privilegios de Larry, la concesión de privilegios por parte de George permanece. Además, los privilegios de Sue también permanecen, ya que pueden ser derivados de la concesión de George.

Sin embargo, consideremos otra variante sobre la cadena de privilegios con los eventos ligeramente reordenados, como se muestra en la Figura 15.9. Aquí Larry recibe el privilegio con la opción de concesión por parte de Sam, concede el privilegio a Sue y *luego* recibe la concesión, con la opción de concesión, por parte de George. Esta vez cuando Sam revoca los privilegios de Larry, los resultados son ligeramente diferentes, y pueden variar de un DBMS a otro. Como en la Figura 15.8, Larry tiene el privilegio SELECT sobre la vista RE0ESTE, ya que la concesión de George sigue intacta. Pero en una base de datos DB2 o SQL/DS, Sue perderá automáticamente su privilegio SELECT sobre la tabla. ¿Por qué? Porque la concesión de Larry a Sue se derivaba claramente de la concesión de Sam a Larry, la cual acaba de ser revocada. No podría haberse derivado de la concesión de George a Larry, ya que esa concesión no había tenido lugar aun cuando se efectuó la concesión de Larry a Sue.

En un producto diferente de DBMS, los privilegios de Sue podrían permanecer intactos, debido a que la concesión de George a Larry sigue intacta. Por tanto la secuencia temporal de sentencias GRANT y REVOKE, y no tan solo los privilegios, puede determinar lo lejos que los efectos de una sentencia REVOKE se continúan en cascada. La concesión y revocación de privilegios con la opción de concesión deben ser manejadas muy cuidadosamente, para asegurar que los resultados sean los que se pretenden.

REVOKE y el estándar ANSI/ISO

El estándar SQL ANSI/ISO especifica la sentencia GRANT como parte del Lenguaje de Definición de Datos (DDL) de SQL. Recuerde del Capítulo 13 que el

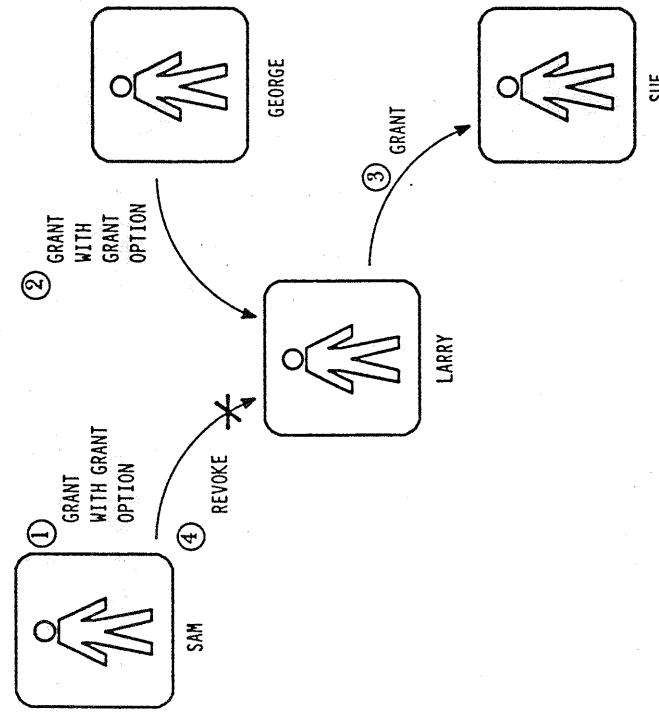


Figura 15.8. Revocación de privilegios concedidos por dos usuarios.

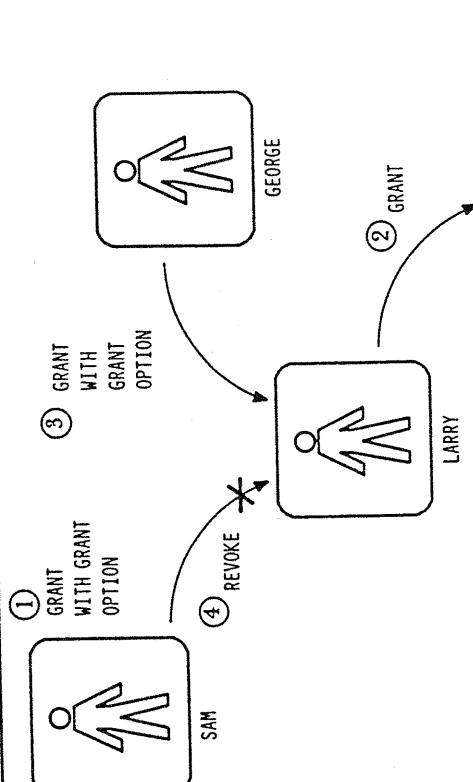


Figura 15.9. Revocación de privilegios en una secuencia diferente.

Resumen

El lenguaje SQL se utiliza para especificar las restricciones de seguridad en una base de datos basada en SQL:

- El esquema de seguridad de SQL está construido alrededor de privilegios (acciones permitidas) que pueden ser concedidos sobre objetos específicos de la base de datos (tales como tablas y vistas), a id-usuarios específicos o grupos de usuarios.
- Las vistas también juegan un papel esencial en la seguridad de SQL, puesto que pueden ser utilizadas para restringir acceso a filas o columnas específicas de una tabla.
- La sentencia GRANT se utiliza para conceder privilegios, los privilegios que usted concede a un usuario con la opción de concesión pueden a su vez ser concedidos por ese usuario a otros.
- La sentencia REVOKE se utiliza para revocar privilegios previamente concedidos con la sentencia GRANT.

El estándar ANSI/ISO trata al DDL como una definición estática de una base de datos, y no requiere que el DBMS permita modificaciones dinámicas a la estructura de la base de datos. Este planteamiento se aplica a la seguridad de la base de datos también. Bajo el estándar ANSI/ISO, la accesibilidad a tablas y vistas en la base de datos se determina mediante una serie de sentencias GRANT incluidas en el esquema de la base de datos. No existe mecanismo para cambiar el esquema de seguridad una vez definida la estructura de la base de datos. La sentencia REVOKE está por tanto ausente del estándar SQL ANSI/ISO, lo mismo que la sentencia DROP TABLE falta del estándar.

A pesar de su ausencia del estándar ANSI/ISO, la sentencia REVOKE es proporcionada por prácticamente todos los productos DBMS comerciales basados en SQL. Al igual que con las sentencias DROP y ALTER, el dialecto DB2 de SQL se ha establecido efectivamente como el estándar para la sentencia REVOKE.

16 El catálogo de sistema

Un sistema de gestión de base de datos debe llevar la cuenta de gran cantidad de información referente a la estructura de una base de datos con el fin de efectuar sus funciones de gestión de datos. En una base de datos relacional, esta información está almacenada típicamente en el *catálogo de sistema*, una colección de tablas del sistema que el DBMS mantiene para su propio uso. La información del catálogo del sistema describe las tablas, las vistas, las columnas, los privilegios y otras características estructurales de la base de datos.

Aunque el DBMS mantiene el catálogo de sistema principalmente para sus propios fines internos, las tablas del sistema son generalmente accesibles a los usuarios de la base de datos también a través de consultas SQL estándar. Una base de datos relacional es por tanto autodescriptiva; utilizando consultas sobre las tablas del sistema, se puede pedir a la base de datos que describa su propia estructura. Los «front-ends» (*front-end*s) de bases de datos de propósito general, tales como las herramientas de consulta y los escritores de informes, utilizan esta característica autodescriptiva para generar listas de tablas y columnas para selección de usuario, simplificando el acceso a la base de datos.

Este capítulo describe los catálogos de sistema proporcionados por varios productos DBMS populares, basados en SQL y la información que el catálogo contiene.

¿Qué es el catálogo de sistema?

El catálogo de sistema es una colección de tablas especiales en una base de datos que son propiedad, están creadas y son mantenidas por el propio DBMS. Estas

tablas del sistema contienen datos que describen la estructura de la base de datos. Las tablas del catálogo de sistema son automáticamente creadas al crear la base de datos. Generalmente se recogen todas juntas bajo un «id-usuario de sistema» especial con un nombre como SYSTEM, SYSIBM, MASTER o DBA.

El DBMS se refiere constantemente a los datos del catálogo de sistema cuando procesa las sentencias SQL. Por ejemplo, para procesar una sentencia SELECT de dos tablas, el DBMS debe:

- verificar que las dos tablas designadas existen realmente
- asegurar que el usuario tiene permiso para acceder a ellas
- comprobar si existen las columnas referenciadas en la consulta
- resolver los nombres de columna no cualificados a una de las tablas
- determinar el tipo de datos de cada columna

Almacenando la información estructural en tablas del sistema, el DBMS puede utilizar sus propios métodos y lógica de acceso para recuperar rápida y eficientemente la información que necesita en la realización de esas tareas.

Si las tablas del sistema sólo fueran utilizadas internamente por el DBMS, serían de poco interés para los usuarios de la base de datos. Sin embargo, el DBMS hace generalmente que las tablas del sistema también estén disponibles para acceso a los usuarios. Las consultas de usuario respecto de los catálogos del sistema están casi siempre permitidas en las bases de datos sobre minicomputadores y computadores personales. Estas consultas son también soportadas en productos DBMS sobre mainframe y minicomputadores, pero el administrador de la base de datos puede restringir el acceso al catálogo de sistema como medida adicional de seguridad de la base de datos. Consultando los catálogos de sistema, usted puede descubrir información acerca de la estructura de una base de datos, incluso si nunca la ha utilizado antes.

El acceso de los usuarios al catálogo de sistema es de solo lectura. El DBMS impide a los usuarios actualizar o modificar directamente las tablas del sistema, ya que tales modificaciones destruirán la integridad a la base de datos. En su lugar, el propio DBMS tiene a su cargo insertar, eliminar y actualizar filas de las tablas del sistema cuando modifica la estructura de una base de datos. Las sentencias DDL tales como CREATE, ALTER, DROP, GRANT y REVOKE producen cambios en las tablas del sistema como consecuencia de sus acciones.

N/A

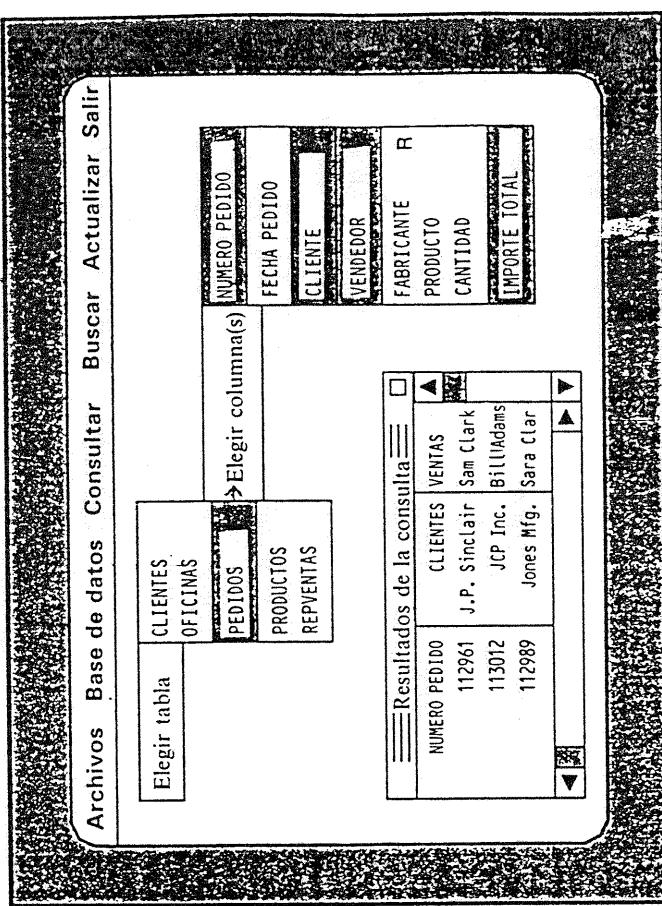


Figura 16.1. Una herramienta de consulta amistosa.

se muestra en la Figura 16.1. El objetivo de tal herramienta es permitir a los usuarios un acceso simple y transparente a la base de datos sin tener que aprender el lenguaje SQL. Tipicamente, la herramienta conduce al usuario a través de una serie de pasos como éstos:

1. El usuario proporciona un nombre y una contraseña para el acceso a la base de datos.
2. La herramienta de consulta visualiza una lista de las tablas disponibles.
3. El usuario elige una tabla, haciendo que la herramienta de consulta visualice una lista de las columnas que contiene.
4. El usuario elige las columnas de interés, quizás aceptando sus nombres con pulsaciones de tecla cuando éstos aparecen en la pantalla de un PC.
5. El usuario elige columnas de otras tablas o restringe los datos a ser recuperados con una condición de búsqueda.
6. La herramienta de consulta recupera los datos solicitados y los visualiza en la pantalla de usuario.

El catálogo y sus herramientas de consulta

Uno de los beneficios más importantes del catálogo de sistema es que hace posible herramientas de consulta de fácil actualización («amistosas»), tal como

Una herramienta de consulta de propósito general con la de la Figura 16.1 debe ser capaz de conocer, dinámicamente las tablas y columnas existentes en la base de datos. La herramienta realiza consultas sobre el catálogo de sistema con este propósito.

E/ catálogo y el estándar ANSI//ISO

El estándar SQL ANSI/ISO actual no especifica la estructura y los contenidos del catálogo de sistema. De hecho, el estándar no requiere la existencia de un catálogo de sistema en absoluto. Sin embargo, todos los productos DBMS basados en SQL más importantes proporcionan un catálogo de sistema de una forma u otra. La estructura del catálogo y las tablas que contiene varían considerablemente de un producto DBMS a otro.

Debido a la creciente importancia de las herramientas de base de datos de propósito general que deben acceder al catálogo de sistema, el estándar SQL2 propuesto incluye una especificación de un catálogo de sistema mínimo. La especificación está basada generalmente en las tablas de sistema DB2, pero elimina los contenidos específicos de DB2. Si este intento de normalizar el catálogo del sistema va a tener éxito o no sigue siendo una cuestión abierta.

Contenidos del catálogo

Cada tabla del catálogo de sistema contiene información referente a una sola base de elemento estructural de la base de datos. Aunque los detalles varían, casi todos los productos SQL comerciales incluyen tablas del sistema que describen cada uno de estos cinco tipos de datos.

- Tablas.* El catálogo describe cada tabla de la base de datos, identificando su nombre, su propietario, el número de columnas que contiene, su tamaño, etc.

Columnas. El catálogo describe cada columna de la base de datos, proporcionando el nombre de la columna, la tabla a la que pertenece, su tipo de datos, su tamaño, si están permitidos los NULL.

Usuarios. El catálogo describe a cada usuario autorizado de la base de datos, incluyendo el nombre, una forma cifrada de la contraseña del usuario y otros datos.

Vistas. El catálogo describe cada vista definida en la base de datos, incluyendo su nombre, el nombre de su propietario, la consulta que define la vista, etc.

Privilegios. El catálogo describe cada grupo de privilegios concedidos en la base de datos, incluyendo los nombres del donante y el donatario, los privilegios concedidos, el objetivo sobre el cual se han concedido los privilegios, etc.

2) Es posible saber quién es el creador de cada una de las tablas existentes en la base de datos.

La Tabla 16.1 muestra los nombres de las tablas de sistema que proporcionan esta información en cada uno de los principales productos DBMS basados en SQL. El resto de este capítulo describe algunas tablas de sistema típicas con más detalle y proporciona ejemplos de acceso al catálogo de sistema.

Información sobre tablas

Cada uno de los principales productos SQL tiene una tabla de sistema que lleva la cuenta de las tablas existentes en la base de datos. En OS/2 Extended Edition y en DB2, esta tabla del sistema se denomina SYSTABLES. Contiene una fila por cada tabla o vista definida en la base de datos. La Tabla 16.2 muestra las columnas en la tabla SYSTABLES de OS/2.

Nombre de columna	Tipo de datos	Información
NAME	VARCHAR(18)	Nombre de la tabla o vista.
CREATOR	CHAR(18)	Propietario de la tabla o vista.
TYPE	CHAR(1)	'T' para tabla; 'V' para vista.
CTIME	TIMESTAMP	Fecha/hora de creación de tabla.
REMARKS	VARCHAR(254)	Hasta 254 caracteres de comentarios.
PACKED_DESC	LONG VARCHAR	Forma interna de la descripción de tabla.
VIEW_DESC	LONG VARCHAR	Forma interna de definición de vista.
COLCOUNT	SMALLINT	Número de columnas de la tabla.
FID	SMALLINT	Id de archivo interno del archivo que contiene los datos.
TID	SMALLINT	Id de tabla interno de la tabla.
CARD	INTEGER	Cardinalidad (número de filas).
NPAGES	INTEGER	Número de páginas de almacenamiento utilizadas.
FPAGES	INTEGER	Número total de páginas de almacenamiento
OVERFLOW	INTEGER	Número de registros de rebosé.

Tabla 16.2. La tabla SYSTABLES (OS/2 Extended Edition)

Usted puede utilizar consultas como las de los ejemplos siguientes para encontrar información referente a las tablas en una base de datos OS/2. Consultas análogas, utilizando diferentes nombres de tabla y columna, pueden ser utilizadas para obtener la misma información de otros productos DBMS.

Lista los nombres y propietarios de todas las tablas en la base de datos.

```
SELECT CREATOR, NAME
      FROM SYSIBM.SYSTABLES
     WHERE TYPE = 'T'
```

Lista los nombres de todas las tablas y vistas en la base de datos.

```
SELECT NAME,
       CTIME
      FROM SYSIBM.SYSTABLES
     WHERE TYPE = 'T'
       AND CREATOR = USER
```

Lista los nombres y fechas de creación de mis tablas initamente.

```
SELECT NAME, CTIME
      FROM SYSIBM.SYSTABLES
     WHERE TYPE = 'T'
```

La versión DB2 de la tabla de sistema SYSTABLES es muy similar a la versión OS/2, pero existen algunas diferencias sutiles que se derivan de las diferencias características entre ambos productos. En OS/2 Extended Edition el usuario que crea un tabla se convierte automáticamente en su propietario. La columna CREATOR en la tabla SYSTABLES de OS/2 identifica así *tan sólo* al propietario *como* al creador de una tabla. En DB2, un usuario puede crear una tabla y ceder la propiedad de ella a otro usuario. Por esta razón el catálogo de DB2 incluye una columna CREATOR, especificando el propietario de una tabla, y una columna CREATED_BY, especificando el usuario que realmente creó la tabla. Analogamente, la columna FID (file id —id de archivo—) del catálogo OS/2, que identifica al archivo OS/2 que contiene los datos de una tabla, se sustituye en el catálogo del DB2 por una columna que especifica el espacio de tablas (una entidad de almacenamiento físico en DB2) en donde se almacena la tabla.

En una base de datos SQL/DS, la tabla del sistema SYSCATALOG efectúa la misma función que la tabla SYSTABLES de OS/2 y de DB2. La tabla SYSCATALOG también contiene una fila por cada tabla o vista de la base de datos. La tabla SYSCATALOG registra el nombre (TNAME), el creador (CREATOR), el número de columnas (NCOLS), etcétera, para cada tabla o vista. El equivalente en SQL Server a la tabla SYSTABLES de OS/2 es una tabla de sistema llamada SYSOBJECTS, descrita en la Tabla 16.3. La tabla SYSOBJECTS almacena información acerca de las tablas y vistas en SQL Server, y de otros objetos de SQL Server tales como procedimientos almacenados, reglas y disparadores. Observe también cómo la tabla SYSOBJECTS utiliza un número de identificación interno en lugar de un nombre para identificar al propietario de la tabla.

Es interesante observar que incluso los productos SQL de IBM, que han estado bajo una fuerte presión para ofrecer características uniformes y estándares, tienen diferentes definiciones de la misma tabla de sistema. Las diferencias entre las tablas de IBM y el catálogo de SQL Server son aún más pronunciadas. Estas diferencias de catálogo incluso entre productos DBMS producidos por el mismo fabricante, se derivan de diferencias en las características de DBMS, de las diferencias en el hardware y el sistema operativo subyacentes y de operaciones históricas.

Nombre de columna	Tipo de datos	Información
name	SYSNAME	Nombre del objeto.
id	INT	Número id interno del objeto.
uid	SMALLINT	Id-usuario del propietario del objeto.
type	CHAR(2)	Código de tipo de objeto *.
refdate	DATETIME	Reservado para uso futuro.
cdate	DATETIME	Fecha/hora en que el objeto fue creado.
exdate	DATETIME	Reservado para uso futuro.
deltrig	INT	Id de procedimiento de disparador DELETE.
instrig	INT	Id de procedimiento de disparador INSERT.
updtrig	INT	Id de procedimiento de disparador UPDATE.
seltrig	INT	Reservado para uso futuro.
* S = tabla de sistema, U = tabla de usuario, V = vista, L = registro (log), P = procedimiento almacenado, R = regla, D = omisión, TR = disparador		

Tabla 16.3. Columnas seleccionadas de la tabla SYSOBJECTS (SQL Server).

Información sobre columnas

Todos los principales productos SQL tienen una tabla de sistema que lleva la cuenta de las columnas de la base de datos. Los productos SQL de IBM llaman

Nombre de columna	Tipo de datos	Información
NAME	VARCHAR(18)	Nombre de la columna.
TBNAME	VARCHAR(18)	Nombre de la tabla o vista que contiene la columna.
TBCREATOR	CHAR(8)	Propietario de la tabla o vista.
REMARKS	VARCHAR(254)	Hasta 254 caracteres de comentario.
COLTYPE	CHAR(8)	Tipo de datos de la columna (por ejemplo, "INTEGER")
NULLS	CHAR(1)	'Y' si se permiten NULLS; 'N' si no.
CODEPAGE	SMALLINT	Conjunto de caracteres IBM (para datos de texto).
DBSCODEPAGE	SMALLINT	Utilizado internamente por DBMS
LENGTH	SMALLINT	Longitud de la columna.
SCALE	SMALLINT	Escala de la columna (para columnas decimales).
COLNO	SMALLINT	Posición de la columna dentro de la tabla.
COLCARD	INTEGER	Número de valores distintos en la columna.
HIGH2KEY	VARCHAR(16)	Segundo valor más alto en la columna.
LOW2KEY	VARCHAR(16)	Segundo valor más bajo en la columna.
AVGCOLLEN	INTEGER	Longitud media de la columna.

Tabla 16.4. La tabla SYSCOLUMNS (OS/2 Extended Edition).

todos a esta tabla de sistema SYSCOLUMNS. Hay una fila en la tabla SYSCOLUMNS por cada columna de cada tabla o vista de la base de datos. La Tabla 16.4 muestra la definición de la tabla de sistema SYSCOLUMNS en OS/2 Extended Edition.

La mayor parte de la información de la tabla SYSCOLUMNS almacena la definición de una columna —su nombre, su tipo de datos, su longitud, si puede aceptar valores NULL o no, etc—. Además, la tabla SYSCOLUMNS incluye información referente a la distribución de valores de datos hallados en cada columna. Esta información estadística ayuda al DBMS a decidir cómo llevar a cabo una consulta de modo óptimo.

Se pueden utilizar consultas como las de los ejemplos siguientes para hallar información referente a las columnas en una base de datos OS/2 Extended Edition. Consultas similares, utilizando nombres de columna y tablas diferentes, pueden utilizarse para obtener la misma información en otros productos DBMS.

Lista los nombres y tipos de datos de las columnas de mi tabla OFICINA.

```
SELECT NAME, COLTYPE
      FROM SYSTEM.SYSCOLUMNS
    WHERE TBNAME = 'OFICINAS'
      AND TBCREATOR = USER
        WHERE COLTYPE = 'DATE'
```

Halla todas las columnas de la base de datos que tienen el tipo de datos DATE.

```
SELECT TBCREATOR, TBNAME, SYSIBM.SYSCOLUMNS.NAME, COLTYPE, LENGTH
      FROM SYSTEM.SYSCOLUMNS, SYSIBM.SYSTABLES
    WHERE TBCREATOR = CREATOR
      AND TBNAME = SYSIBM.SYSTABLES.NAME
      AND (COLTYPE = 'VARCHAR' OR COLTYPE = 'CHAR')
        AND LENGTH > 10
          AND TYPE = 'V'
```

Lista el propietario, el nombre de columna, el tipo de datos y la longitud de todas las columnas de texto superiores a diez caracteres definidas en vistas.

■ Los nombres de las columnas en las dos tablas son completamente diferentes, aun cuando contienen datos análogos.

■ El catálogo de OS/2 Extended Edition utiliza una combinación del nombre del propietario y del nombre de la tabla para identificar la tabla que contiene

una columna determinada; el catálogo de SQL Server utiliza un número id de tabla interno, que es una clave foránea a su tabla SYSOBJECTS.

- El catálogo de OS/2 Extended Edition especifica tipos de datos en formato texto (por ejemplo, «CHAR(10)»); el catálogo de SQL Server utiliza códigos de tipo de datos enteros.

Otras diferencias reflejan las diferentes capacidades proporcionadas por los dos productos DBMS:

- OS/2 Extended Edition permite especificar hasta 254 caracteres de comentarios referentes a cada columna; SQL Server no dispone de esta posibilidad.
- SQL Server permite especificar una regla que define los datos válidos para una columna; OS/2 Extended Edition no dispone de esta posibilidad.
- OS/2 Extended Edition utiliza un valor por omisión estándar para cada tipo de datos (0 para enteros, por ejemplo); SQL Server requiere que se escriba un procedimiento almacenado que genere un valor por omisión, y almacena el número id del procedimiento almacenado en la tabla SYSOLUMNS.

Nombre de columna	Tipo de datos	Información
id	INT	Id interno de la tabla que contiene la columna.
number	SMALLINT	Cero para definiciones de columna.
col_id	TINYINT	Número id de columna interno.
status	TINYINT	¿Se permiten NULLs?
type	TINYINT	Código de tipo de datos para la columna.
length	TINYINT	Largo de la columna.
offset	TINYINT	Desplazamiento de columna físico desde el comienzo de la fila.
usertype	SMALLINT	Código de tipo de datos para tipos de datos definidos por usuario.
cdefault	INT	Id de procedimiento almacenado para valor por omisión.
domain	INT	Id de procedimiento almacenado para regla de columna.
name	INT	Nombre de columna.
printfmt	VARCHAR(255)	Formato de visualización para la columna.

Tabla 16.5 La tabla SYSOLUMNS (SQL Server)

Información sobre vistas

Las definiciones de las vistas en una base de datos se almacenan en el catálogo de sistema. El catálogo de OS/2 Extended Edition contiene dos tablas de

Nombre de columna	Tipo de datos	Información
NAME	VARCHAR(18)	Nombre de la vista.
CREATOR	CHAR(8)	Propietario de la vista.
SEQNO	SMALLINT	Número de secuencia de esta fila (1, 2, etc.).
CHECK	CHAR(1)	'Y' si la vista tiene opción de chequeo; 'N' si no.
TEXT	VARCHAR(3900)	Hasta 3900 caracteres de definición de la vista.

Tabla 16.6 La tabla SYVIEWS (OS/2 Extended Edition)

sistemas que siguen la pista a las vistas. La tabla SYVIEWS, que se describe en la Tabla 16.6, contiene la definición de texto SQL de cada vista. Si la definición supera a 3900 caracteres, se almacena en múltiples filas, con números de secuencia 1, 2, 3, etc.

La tabla SYVIEWDEP, contenida en la Tabla 16.7, describe cómo depende cada vista de otras tablas o vistas. Existe una fila en la tabla por cada dependencia, de modo que una vista con tres tablas fuente estará representada por tres filas. Utilizando estas dos tablas, se pueden ver las definiciones de las vistas en la base de datos y se puede determinar rápidamente qué tablas de la base de datos sirven como fuentes para una vista. El modo más fácil de determinar los nombres de las propias vistas es consultar el catálogo SYSTABLES, que incluye tanto tablas de vista como tablas base:

Nombre de columna	Tipo de datos	Información
BNAME	VARCHAR(18)	Nombre de tabla o vista de la cual depende la vista.
NAME	CHAR(8)	Propietario de la tabla o vista especificada en BNAME.
CREATOR	CHAR(1)	'T' si BNAME designa una tabla; 'V' si una vista.
SYSTABLES	CHAR(1)	Nombre de vista dependiente.
TYPE = 'V'	CHAR(8)	Propietario de vista dependiente.

Tabla 16.7 La tabla SYVIEWDEP (OS/2 Extended Edition)

Comentarios y etiquetas

Los productos SQL de IBM permiten asociar hasta 254 caracteres de comentarios con cada tabla, vista y columna definida en la base de datos. Los comentarios

rios permiten almacenar una breve descripción de la tabla o el elemento de datos en el catálogo de sistema. Además de los comentarios, se puede especificar una etiqueta de hasta 30 caracteres para cada tabla, vista y columna. La etiqueta se utiliza con encabezamiento por omisión en los informes y salidas de la base de datos.

Los comentarios y etiquetas se almacenan en las tablas de sistema SYSTABLE y SYSOLUMNS del catálogo. A diferencia de los otros elementos de las definiciones de tabla y columna, los comentarios y etiquetas no se especifican mediante la sentencia CREATE TABLE. En vez de ello se utilizan las sentencias COMMENT y LABEL, mostradas en la Figura 16.2, para especificar los comentarios y etiquetas en el catálogo del sistema.

He aquí algunos ejemplos de las sentencias COMMENT y LABEL:

Define comentarios y una etiqueta para la tabla OFICINAS.

```
COMMENT ON TABLE OFICINAS
IS 'Esta tabla almacena datos referentes a nuestras oficinas de ventas'
```

```
LABEL ON TABLE OFICINAS
IS 'Oficinas de ventas'
```

Asocia algunos comentarios con las columnas OBJETIVO y VENTAS de la tabla OFICINAS.

```
COMMENT ON OFICINAS
(OBJETIVO IS 'Este es el objetivo de ventas anual para la oficina'
VENTAS IS 'Estas son las ventas anuales hasta la fecha para la oficina')
```

Define una etiqueta para la columna OBJETIVO.

```
LABEL ON COLUMN OFICINAS.OBJETIVO
IS 'Objetivo de ventas'
```

Información sobre relaciones

Con la introducción de la integridad referencial en la Versión 2 de DB2, el catálogo de DB2 fue ampliado para describir claves primarias, claves foráneas y las relaciones padre/hijo que éstas crean. La descripción se extiende a lo largo de cuatro tablas del catálogo de sistema denominadas SYSTABLES, SYSOLUMNS, SYSFOREIGNKEYS y SYSRELS, las dos primeras de las cuales ya fueron descritas en las secciones anteriores.

Toda relación padre/hijo entre dos tablas de la base de datos está representada mediante una fila en la tabla de sistema SYSRELS, descrita en la Tabla 16.8. La fila identifica los nombres de las tablas padre e hijo, el nombre de la relación y la regla de supresión para la relación. Puede ser consultada para determinar las relaciones de la base de datos:

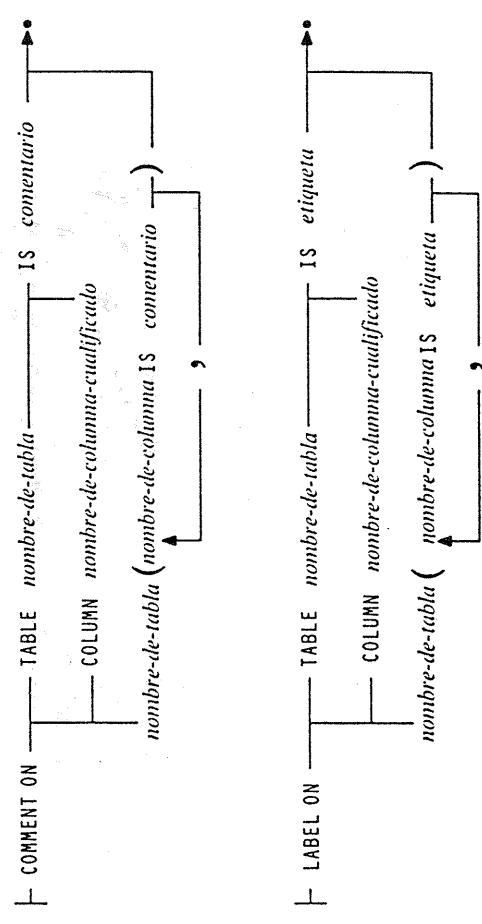


Figura 16.2. Diagramas sintácticos de las sentencias COMMENT y LABEL.

Nombre de columna	Tipo de datos	Información
CREATOR	CHAR(8)	Propietario de la tabla hijo en la relación.
TBNAME	VARCHAR(18)	Nombre de la tabla hijo.
RELNAME	CHAR(8)	Nombre de la relación.
REFTBNAME	VARCHAR(18)	Propietario de la tabla padre en la relación.
REFBCREATOR	CHAR(8)	Propietario de la tabla padre.
COLCOUNT	SMALLINT	Número de columnas en la clave foránea.
DELETERULE	CHAR(1)	'C', 'R', 'N', para CASCADE, RESTRICT, SET NULL
TIMESTAMP	TIMESTAMP	Fecha/hora cuando se definió la relación.

Tabla 16.8. Columnas seleccionadas de la tabla SYSRELS (DB2)

Lista todas las relaciones padre/hijo entre mis tablas, mostrando el nombre de la relación, el nombre de la tabla padre, el nombre de la tabla hijo y la regla de supresión para cada una.

```
SELECT RELNAME, REFTBNAME, TBNAME, DELETERULE
FROM SYSIBM.SYSRELS
WHERE CREATOR = USER
```

Lista todas las tablas relacionadas con la tabla REPVENTAS ya sea como padre ya sea como hijo.

```
SELECT REFTBNAME
  FROM SYSIBM.SYSRELS
 WHERE CREATOR = 'USER'
   AND TBNNAME = 'REPVENTAS'
UNION
SELECT TBNNAME
  FROM SYSIBM.SYSRELS
 WHERE REFBCREATOR = 'USER'
   AND REFTBNAME = 'REPVENTAS'
```

Cada clave foránea que crea una relación está descrita en una o más filas de la tabla de sistema SYSFOREIGNKEYS, mostrada en la Tabla 16.9. Existe una fila en esta tabla de sistema por cada columna en cada clave foránea definida en la base de datos. Un número de secuencia define el orden de las columnas en una clave foránea compuesta. Se puede utilizar esta tabla de sistema para determinar los nombres de las columnas que enlazan una tabla con su padre, utilizando una consulta como ésta:

Informa qué columnas de la tabla PEDIDOS la enlaza con la tabla PRODUCTOS en la relación de nombre ESPOR.

```
SELECT COLNAME, COLSEQ
  FROM SYSIBM.SYSFOREIGNKEYS
 WHERE CREATOR = 'USER'
   AND RELNAME = 'ESPOR'
 ORDER BY COLSEQ
```

La clave primaria de una tabla está descrita en las tablas de sistema SYSTABLES y SYSCOLUMNS, mostradas previamente en las Tablas 16.2 y 16.4. Si una tabla

Nombre de columna	Tipo de datos	Información
CREATOR	CHAR(8)	Propietario de la tabla que contiene la clave foránea.
TBNNAME	VARCHAR(8)	Nombre de la tabla que contiene la clave foránea.
RELNAME	CHAR(8)	Nombre de la relación creada por la clave foránea.
COLNAME	VARCHAR(18)	Nombre de una columna en la clave foránea.
COLNO	SMALLINT	Posición de la columna en la clave foránea (1, 2, etcétera.).
COLSEQ	SMALLINT	Posición de la columna en la clave foránea (1, 2, etcétera.).

Tabla 16.9. Columnas seleccionadas de la tabla SYSFOREIGNKEYS (DB2).

tiene una clave primaria, la columna KEYCOLUMNS en su fila de la tabla de sistema SYSTABLES es distinta de cero e informa de cuántas columnas forman la clave primaria (1 para una clave simple; 2 ó más para una clave compuesta). En la tabla de sistema SYSCOLUMNS, las filas para las columnas que forman la clave primaria tienen un valor distinto de cero en su columna KESEQ. El valor de esta columna indica la posición (1, 2, etc.) de la columna de clave primaria dentro de la clave primaria.

Se puede consultar la tabla SYSCOLUMNS para descubrir la clave primaria de una tabla:

Lista las columnas que forman la clave primaria de la tabla PRODUCTOS.

```
SELECT NAME, KESEQ, COLTYPE, REMARKS
  FROM SYSIBM.SYSCOLUMNS
 WHERE TBLCREATOR = 'USER'
   AND KESEQ > 0
     AND TBNAME = 'PRODUCTOS'
 ORDER BY KESEQ
```

Información sobre usuarios

El catálogo de sistema contiene generalmente una tabla que identifica a los usuarios que tienen autorizado el acceso a la base de datos. El DBMS puede utilizar esta tabla de sistema para validar el nombre de usuario y su contraseña cuando un usuario intenta conectarse por primera vez a la base de datos. La tabla puede almacenar también otros datos referentes al usuario.

SQL Server almacena información de usuario en la tabla de sistema SYSUSERS, mostrada en la Tabla 16.10. Cada fila de esta tabla describe un solo usuario o un grupo de usuarios en el esquema de seguridad de SQL Server. El catálogo de sistema de DB2 adopta un planteamiento ligeramente diferente. Incluye una tabla de sistema SYSUSERAUTH, mostrada en la Tabla 16.11, que almacena información básica de usuario e información referente a los privilegios de usuario en la base de datos. Las filas de SYSUSERAUTH describen tanto usuarios «reales» como programas de aplicación que acceden a la base de datos.

He aquí dos consultas equivalentes que listan los usuarios autorizados para SQL Server y DB2:

```
Listar todos los id-usuario conocidos a SQL Server.
SELECT NAME
  FROM SYSUSERS
 WHERE UID < > 610
```

Lista todos los id-usuario conocidos a DB2.

```
SELECT DISTINCT GRANTEE
  FROM SYSIBM.SYSUSERAUTH
 WHERE GRANTEETYPE = '1'
```

Nombre de columna	Tipo de datos	Información
suid	SMALLINT	Número id-usuario servidor interno.
uid	SMALLINT	Número id-usuario interno en esta base de datos.
gid	SMALLINT	Nombre id-grupo de usuario interno en la base de datos.
name	SYSNAME	Nombre de usuario o grupo.

Tabla 16.10. Columnas seleccionadas de la tabla SYSUSERS (SQL Server)

Información sobre privilegios

Además de almacenar información acerca de la estructura de la base de datos, el catálogo de sistema almacena generalmente la información necesaria al DBMS para forzar la seguridad de la base de datos. Como se describió anteriormente en el Capítulo 15, los diferentes productos DBMS ofrecen diferentes variaciones sobre el esquema de privilegios básico de SQL. Estas variaciones se reflejan en la estructura de los catálogos de sistema para los diferentes productos DBMS.

DB2 tiene uno de los esquemas más completos para privilegios de usuario, que alcanza hasta las columnas individuales de una tabla. La Tabla 16.12 muestra los catálogos de sistema de DB2 que almacenan información referente a privilegios y describe brevemente el papel de cada uno.

Nombre de columna	Tipo de datos	Información
GRANTOR	CHAR(8)	El usuario que concedió los privilegios.
GRANTEE	CHAR(8)	Usuario o aplicación que recibió los privilegios.
TIMESTAMP	CHAR(12)	Fecha/hora en que los privilegios se concedieron.
DATEGRANTED	CHAR(6)	Fecha en que los privilegios se concedieron.
TIMEGRANTED	CHAR(8)	Hora en que los privilegios se concedieron.
GRANTEETYPE	CHAR(1)	'*' para usuario; 'P' para programa de aplicación.
CREATEDBAUTH	CHAR(1)	¿Puede crear el usuario nuevas bases de datos?
SYSADMAUTH	CHAR(1)	¿Tiene el usuario autoridad de administrador del sistema?
SYSOPRAUTH	CHAR(1)	¿Tiene el usuario autoridad de operador del sistema?

Tabla 16.11. Columnas seleccionadas de la tabla SYSUSERAUTH (DB2)

Tabla de sistema	Funcióñ
SYSTABAUTH	Implementa el esquema de privilegios básico informando de qué usuarios tienen permiso para acceder a qué tablas, para qué operaciones (SELECT, INSERT, DELETE, UPDATE, ALTER e INDEX).
SYSCOLAUTH	Implementa privilegios de actualización a nivel de columna informando de qué usuarios tienen permiso para actualizar qué columnas de qué tablas.
SYSDBAAUTH	Determina qué usuarios tienen permiso para crear tablas y ejecutar diferentes programas de utilidad de DB2 (por ejemplo, para cargar datos en masa).
SYSPLANAUTH	Determina qué usuarios tienen permiso para ejecutar qué programas de aplicación que acceden a la base de datos.
SYSRESAUTH	Determina qué usuarios tienen permiso para crear tablas y asignar espacio en diferentes espacios de tablas, grupos de almacenamiento, etc.
SYSUSERAUTH	Determina qué usuarios tienen administración del sistema, son operadores del sistema, tienen mantenimiento del sistema y otros privilegios.

Tabla 16.12. Tablas de DB2 que implementan los permisos

El esquema de autorización utilizado por SQL Server es más fundamental y racional que el de DB2. Trata a las bases de datos, las tablas, los procedimientos almacenados, los disparadores y otras entidades uniformemente como objetos a los cuales se aplican los privilegios. Esta estructura racional se refleja en la tabla de sistema SYSPROTECTS, mostrada en la Tabla 16.13, que implementa el esquema de privilegios completo para SQL Server. Cada fila de la tabla representa una única sentencia GRANT o REVOKE que ha sido emitida.

Nombre de columna	Tipo de datos	Información
id	INT	Id interno del objeto protegido.
uid	SMALLINT	Id interno del usuario o grupo con privilegio.
action	TINYINT	Código numérico de privilegio (por ejemplo, SELECT = 193).
protect type	TINYINT	Código numérico para concesión o revocación.
columns	VARBINARY(32)	Mapa de bits para privilegios de actualización a nivel de columna.

Tabla 16.13. La tabla SYSPROTECTS (SQL Server)

Otras informaciones

El catálogo de sistema es un reflejo de las capacidades y características del DBMS que lo utiliza. Debido a las muchas extensiones de SQL y a las características adicionales ofrecidas por los productos DBMS más populares, sus catálogos de sistema siempre contienen varias tablas propias del DBMS. He aquí algunos ejemplos:

- DB2 y Oracle soportan sinónimos (nombres alternativos para las tablas). Las definiciones de sinónimos se almacenan en una tabla de sistema llamada SYSSYNONYMS.
- SQL Server soporta múltiples bases de datos designadas. Tiene una tabla de sistema llamada SYSDATABASES que identifica las bases de datos gestionadas por un mismo servidor.
- Ingres soporta tablas que están distribuidas a través de varios volúmenes de discos. Su tabla de sistema IIMULTILOCATIONS lleva la cuenta de las ubicaciones de las tablas multivolumen.

Resumen

El catálogo de sistema es una colección de tablas de sistema que describen la estructura de una base de datos relacional:

- El DBMS mantiene los datos en las tablas del sistema, actualizándolos cuando la estructura de la base de datos cambia.
- Un usuario puede consultar las tablas del sistema para obtener información referente a tablas, columnas y privilegios en la base de datos.
- Herramientas de consulta frontales utilizan las tablas del sistema para ayudar a los usuarios a manejar la base de datos de una manera más amistosa.
- Los nombres y la organización de las tablas del sistema difieren ampliamente de un producto DBMS a otro; incluso los productos SQL de IBM tienen diferencias en sus catálogos de sistema.

Programación con SQL

Quinta parte

17 SQL incorporado

SQL es un *lenguaje de modo dual*. Es a la vez un lenguaje de base de datos interactivo utilizado para consultas y actualizaciones ad hoc, y un lenguaje de base de datos programado utilizado por programas de aplicación para acceso a la base de datos. En su mayor parte, el lenguaje SQL es idéntico en ambos modos. La naturaleza de modo dual de SQL tiene varias ventajas:

- Es relativamente fácil para los programadores aprender a escribir programas que acceden a la base de datos.
- Las capacidades disponibles a través del lenguaje de consulta interactivo están también automáticamente disponibles para los programas de aplicación.
- Las sentencias SQL a utilizar en un programa pueden ser probadas por primera vez utilizando SQL interactivo y luego codificadas en el programa.
- Los programas pueden trabajar con tablas de datos y resultados de consulta en vez de navegar a través de la base de datos.

Este capítulo resume los tipos de SQL programado ofrecidos por los principales productos basados en SQL y luego describe el SQL programado utilizado por los productos SQL de IBM, denominado *SQL incorporado*.

Técnicas de SQL programado

SQL es un lenguaje, y puede ser utilizado para programación, pero sería incorrecto decir de SQL que es un lenguaje de programación. SQL carece incluso de

las características más primitivas de los lenguajes de programación «reales». No dispone de provisiones para declarar variables, ni de la sentencia `GOTO`, ni de la sentencia `IF` para comprobar condiciones, ni de las sentencias `FOR`, `DO` o `WHILE` para construir bucles, ni de estructura de bloques, etc. SQL es un *sublenguaje* de base de datos que maneja tareas de gestión de base de datos de propósito especial. Para escribir un programa que acceda a una base de datos, debe comenzarse con un lenguaje de programación convencional, tal como COBOL, PL/I, FORTRAN, Pascal o C, y luego «añadir SQL al programa».

El estándar SQL ANSI/ISO se refiere exclusivamente a este uso programado de SQL. El estándar ni siquiera incluye la sentencia `SELECT` programada, descrita en los Capítulos 6 al 9. Únicamente especifica la sentencia `SELECT` programada, descripción posteriormente en este capítulo.

Los vendedores de bases de datos SQL comerciales ofrecen dos técnicas básicas para utilizar SQL dentro de un programa de aplicación:

■ **SQL incorporado.** En este enfoque, las sentencias SQL están incorporadas directamente al código fuente del programa, entremezcladas con las otras sentencias del lenguaje de programación. Se utilizan sentencias especiales de SQL incorporado para recuperar datos en el programa. Un precompilador de SQL especial acepta el código fuente combinado y, junto con otras herramientas de programación, lo convierte en un programa ejecutable.

■ **Interfaz de programa de aplicación.** En este enfoque, el programa se comunica con el DBMS a través de un conjunto de llamadas de función denominadas *interfaz de programa de aplicación*, o API (*application program interface*). El programa transmite sentencias SQL al DBMS a través de las llamadas API y utiliza las llamadas API para recuperar resultados de consulta. Este planteamiento no exige un precompilador especial.

Los productos SQL de IBM utilizan un enfoque de SQL incorporado, y éste es el que ha sido adoptado por la mayoría de los productos SQL comerciales. El estándar ANSI/ISO especificaba originalmente un «lenguaje de módulo» separado para SQL programado, pero en 1989 el estándar fue ampliado para incluir una definición de SQL incorporado para los lenguajes de programación Ada, C, COBOL, FORTRAN, Pascal y PL/I. Sin embargo, SQL Server y el DBMS Sybase del cual se deriva utilizan ambos el enfoque API exclusivamente, lo cual hace también el producto SQLBase de Gupta Technologies. La Tabla 17.1 resume los interfaces de programación ofrecidos por los principales productos DBMS basados en SQL.

Las técnicas básicas de SQL incorporado, llamadas SQL *estática*, se describen en este capítulo. Algunas características avanzadas de SQL incorporado, llamadas SQL *dinámico*, se discuten en el Capítulo 18. El API de SQL proporcionado por Sybase y SQL se discutirá en el Capítulo 19.

DBMS	API	Sopporte de lenguaje incorporado
DB2	No *	APL, Assembler, BASIC, COBOL, FORTRAN, PL/I
SQL/DS	No	APL, Assembler, BASIC, COBOL, FORTRAN, PL/I
Oracle	Sí	Ada, C, COBOL, FORTRAN, Pascal, PL/I
Ingres	No	Ada, BASIC, C, COBOL, FORTRAN, Pascal, PL/I
Sybase	Sí	none
Informix	No	Ada, C, COBOL
OS/2 EE	No	C
SQLBase	Sí	none

* Interfaz de función no disponible para programación de aplicaciones; sin embargo, la Facilidad Call Attach de DB2 proporciona un interfaz de función para programadores de sistema.

Tabla 17.1. Interfaces de SQL programado para productos SQL populares.

Procesamiento de sentencias DBMS

Para entender cualquiera de las técnicas de SQL programado es útil comprender un poco mejor cómo procesa el DBMS las sentencias de SQL. Para procesar una sentencia SQL, el DBMS recorre una serie de cinco pasos, mostrados en la Figura 17.1:

1. El DBMS comienza *analizando* la sentencia SQL. Descomponer la sentencia en palabras individuales, asegurándose que la sentencia tenga un verbo válido y cláusulas legales, etc. Los errores sintácticos y las faltas de deletreo pueden ser detectadas en este paso.
2. El DBMS *valida* la sentencia. Comprueba la sentencia con respecto al catálogo del sistema. ¿Existen todas las tablas designadas en la sentencia en palabras individuales, asegurándose que la sentencia tenga un verbo válido y cláusulas legales, etc. Los errores sintácticos y las faltas de deletreo pueden ser detectadas en este paso.
3. El DBMS *explora* la sentencia. Explora los diferentes modos de llevar a cabo la sentencia. ¿Puede ser utilizado un índice para acelerar la búsqueda? ¿Debería el DBMS aplicar primero una condición de búsqueda a la Tabla A y luego componerla con la Tabla B, o debería comenzar con la composición y utilizar la condición de búsqueda después? ¿Puede evitarse una búsqueda secuencial a través de una tabla o reducirse a un subconjunto de la tabla? Tras explorar las alternativas, el DBMS elige una de ellas.
4. El DBMS *optimiza* la sentencia. Optimiza las tablas designadas en la sentencia en la base de datos. ¿Existen todas las columnas y sus nombres no ambiguos? ¿Tiene el usuario los privilegios necesarios para ejecutar la sentencia? Los errores semánticos se detectan en este paso.

datos. Para una consulta multitable compleja, el optimizador puede explorar más de una docena de maneras diferentes de llevar a cabo la consulta. Sin embargo, el coste en tiempo de proceso del computador al efectuar la consulta del modo «equivocado» es generalmente tan alto comparado con el coste de efectuarlo en el modo «correcto» (o al menos un modo «mejor») que el tiempo empleado en la optimización se ve más que compensado con el incremento en la velocidad de ejecución de la consulta.

Cuando se escribe una sentencia SQL en SQL interactivo, el DBMS recorre los cinco pasos mientras el usuario espera a la respuesta. El DBMS tiene poca opción en esta materia —no sabe qué sentencia se va a escribir a continuación hasta el momento en que se escribe, y por tanto no puede adelantar procesos en el tiempo—. En SQL programado, sin embargo, la situación es bastante diferente. Algunos de los primeros pasos pueden ser efectuados en *tiempo de compilación*, cuando el programador está desarrollando el programa. Esto deja que únicamente los pasos posteriores sean efectuados en *tiempo de ejecución*, cuando el programa sea ejecutado por un usuario. Cuando se utiliza SQL programado, todos los productos DBMS tratan de trasladar tanto proceso como es posible al tiempo de compilación, ya que una vez que se ha desarrollado la versión final del programa, puede ser ejecutado miles de veces por usuarios en una aplicación de producción. En particular, el objetivo es trasladar la optimización al tiempo de compilación siempre que sea.

Conceptos de SQL incorporado

La idea central de SQL incorporado es mezclar las sentencias de lenguaje SQL directamente en un programa escrito en un lenguaje de programación «anónimo», tal como C, Pascal, COBOL, FORTRAN, PL/I o Assembler. SQL incorporado utiliza las siguientes técnicas para incorporar las sentencias SQL:

- Las sentencias SQL se entremezlan con las sentencias del lenguaje principal en el programa fuente. Este «programa fuente con SQL incorporado» se remite a un *precompilador* de SQL, que procesa las sentencias SQL.
- Las variables del lenguaje de programación anfitrión pueden ser referenciadas en las sentencias SQL incorporado, permitiendo que valores calculados por el programa sean utilizados por las sentencias SQL.
- Las variables del lenguaje de programación son también utilizadas por las sentencias de SQL incorporado para recibir los resultados de consultas SQL, permitiendo al programa utilizar y procesar los valores recuperados.
- Variables de programa especiales son utilizadas para asignar valores NULL a columnas de la base de datos y para soportar la recuperación de valores NULL de la base de datos.

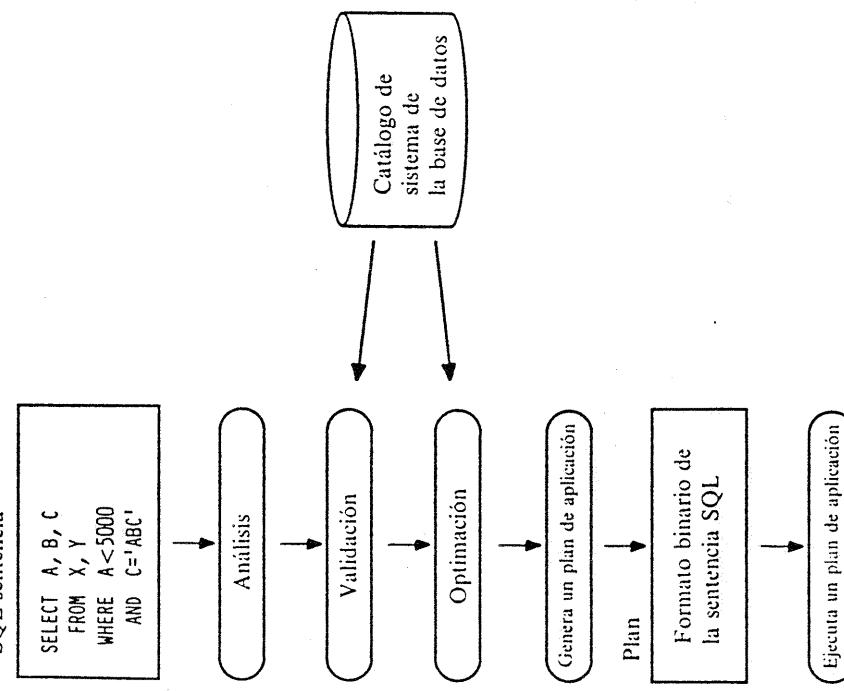


Figura 17.1. Cómo procesa el DBMS una sentencia SQL.

para llevar a cabo la sentencia; es el equivalente DBMS del «código ejecutable».

5. Finalmente, el DBMS lleva a cabo la sentencia mediante la ejecución del plan de aplicación.

Observe que los pasos de la Figura 17.1 varían en la cuantía de acceso a base de datos que requieren y la cantidad de tiempo de CPU que toman. El análisis de una sentencia SQL no requiere acceso a la base de datos, y típicamente puede ser realizada muy rápidamente. La optimización, por otra parte, es un proceso muy intensivo en CPU y requiere acceso al catálogo de sistema de la base de

- Varias nuevas sentencias SQL que son propias del SQL incorporado se añaden al lenguaje SQL interactivo, para permitir el procesamiento fila a fila de resultados de consulta.

La Figura 17.2 muestra un sencillo programa de SQL incorporado, escrito en C. El programa ilustra muchas, pero no todas las técnicas de SQL incorporado. El programa solicita al usuario un número de oficina, recuperar la ciudad, región, ventas y objetivo de la oficina y los visualiza en la pantalla.

No se preocupe si el programa le parece extraño, o si no puede comprender todas las sentencias que contiene antes de leer el resto de este capítulo. Una de las desventajas del planteamiento SQL incorporado es que el código fuente de un programa se convierte en una «mezcla» impura de dos lenguajes diferentes, haciendo el programa difícil de entender sin un previo entrenamiento tanto en SQL como en el lenguaje de programación. Otra desventaja es que SQL incorporado utiliza construcciones del lenguaje SQL no utilizadas en SQL interactivo, tales como la sentencia `WHENEVER` y la cláusula `INTO` de la sentencia `SELECT` —ambas utilizadas en este programa.

Desarrollo de un programa de SQL incorporado

Un programa de SQL incorporado contiene una mezcla de sentencias SQL y sentencias del lenguaje de programación, por lo que no puede ser remitido directamente a un compilador del lenguaje de programación. En vez de ello, debe recorrer un proceso de desarrollo de varios pasos, como se muestra en la Figura 17.3. Los pasos de la figura son realmente los utilizados por las bases de datos de IBM (DB2, SQL/DS y OS/2 Extended Edition), pero todos los productores que soportan SQL incorporado utilizan un proceso similar:

1. El programa fuente incorporado se remite al *precompilador* de SQL, una herramienta de programación. El precompilador examina el programa, encuentra las sentencias de SQL incorporado y las procesa. Se requiere un precompilador diferente por cada lenguaje de programación soportado por el DBMS. Los productos SQL comerciales ofrecen típicamente precompiladores para uno o más lenguajes, incluyendo C, Pascal, COBOL, FORTRAN, Ada, PL/I, RPG y varios lenguajes ensambladores.
2. El precompilador produce dos archivos como salida. El primer archivo es el programa fuente, despojado de las sentencias de SQL incorporado. En su lugar, el precompilador sustituye llamadas a las rutinas DBMS «privadas», que proporcionan el enlace en tiempo de ejecución entre el programa y el DBMS. Tipicamente, los nombres y las secuencias de llamada de estas rutinas son conocidas únicamente por el precompilador y el DBMS; no son

```

main()
{
    exec sql include sqlca;
    exec sql begin declare section;
        int numoficina; /* número de oficina (del usuario) */
        char nombreciudad[16]; /* nombre de ciudad recuperado */
        char nombreregion[11]; /* nombre de región recuperado */
        float valobjetivo; /* objetivo y ventas recuperados */
    exec sql end declare section;

    /* Prepara el procesamiento de errores */
    exec sql whenever sqlerrortodo goto error_acceso;
    exec sql whenever not found goto mal_numero;

    /* Pide al usuario que escriba un número de oficina */
    printf("Introduzca un número de oficina: ");
    scanf("%d", &numoficina);

    /* Ejecuta la consulta SQL */
    exec sql select ciudad, region, objetivo,
        from oficinas
        where oficina = numoficina
        into :nombreciudad, :nombreregion, :valobjetivo, :valventas;

    /* Visualiza los resultados */
    printf("Ciudad: %s\n", nombreciudad);
    printf("Región: %s\n", nombreregion);
    printf("Objetivo: %f\n", valobjetivo);
    printf("Ventas: %f\n", valventas);
    exit(0);

error_acceso:
    printf("SQL error: %d\n", sqlca.sqlcode);
    exit(0);
}

mal_numero:
printf("Número de oficina no válido.\n");
exit(0);
}

```

Figura 17.2. Típico programa de SQL incorporado.

1. El interfaz público con el DBMS. El segundo archivo es una copia en el programa fuente. Este archivo se denomina a veces *módulo de petición a base de datos* o DBRM (*database request module*).
3. La salida de archivo fuente del precompilador se remite al compilador estándar para el lenguaje de programación anfitrión (como por ejemplo al

Programa fuente SQL incorporado

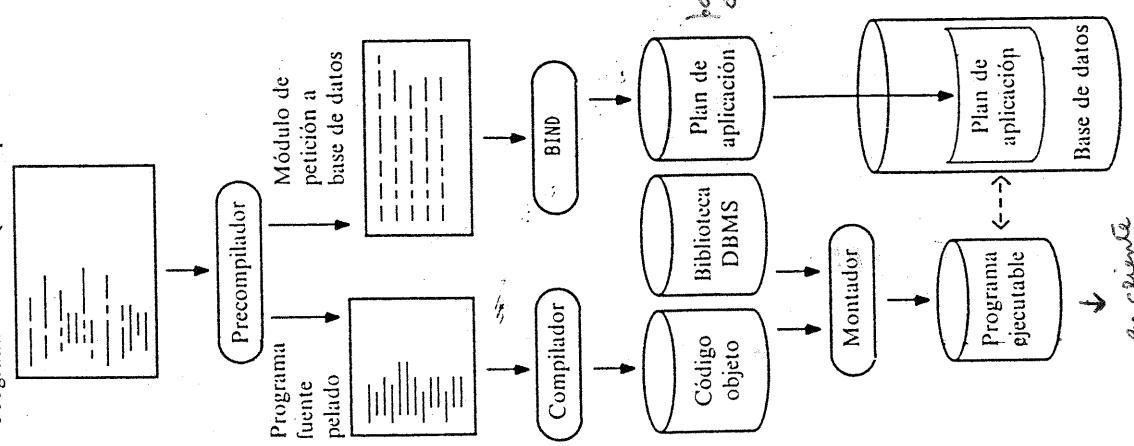


Figura 17.3. El proceso de desarrollo de SQL incorporado.

compilador de C o COBOL). El compilador procesa el código fuente y produce código objeto como salida. Obsérve que este paso no tiene nada en particular que ver con el DBMS o con SQL.

4. El *montador* acepta los módulos objeto generados por el compilador, los enlaza con diferentes rutinas de biblioteca y produce un programa ejecutable. Las rutinas de biblioteca montadas en el programa ejecutable incluyen las rutinas «privadas» del DBMS descritas en el Paso 2.
5. El módulo de petición a base de datos generado por el precompilador es remitido a un programa especial BIND. Este programa examina las sentencias de SQL, las analiza, las valida y las optimiza y produce un plan de aplicación para cada sentencia. El resultado es un plan de aplicación combinado para el programa entero, representando una versión ejecutable por el DBMS de las sentencias de SQL incorporado. El programa BIND almacena el plan en la base de datos, asignándole generalmente el nombre del programa de aplicación que lo creó.

Los pasos de desarrollo de programa de la Figura 17.3 se correlacionan con los pasos de proceso de la sentencia DBMS de la Figura 17.1. En particular, el precompilador maneja generalmente el análisis de la sentencia (el primer paso), y la utilidad BIND maneja la verificación, la optimización y la generación de plan (los pasos segundo, tercero y cuarto). Por tanto los cuatro primeros pasos de la Figura 17.1 tienen todos lugar en tiempo de compilación cuando se utiliza SQL incorporado. Sólo el quinto paso, la ejecución efectiva del plan de aplicación, sigue realizándose en tiempo de ejecución.

El proceso de desarrollo de SQL incorporado convierte el programa fuente original con SQL incorporado en dos partes ejecutables:

- un *programa ejecutable*, almacenado en un archivo en el computador con el mismo formato que cualquier programa ejecutable.
- un *plan de aplicación ejecutable*, almacenado dentro de la base de datos en el formato esperado por el DBMS.

El ciclo de desarrollo de SQL incorporado puede parecer complicado, y es más complejo que el desarrollo de un programa C o COBOL estándar. En la mayoría de los casos, todos los pasos de la Figura 17.3 están automatizados mediante un único procedimiento de órdenes, por lo que los pasos individuales son invisibles al programador de aplicaciones. El proceso tiene varias ventajas importantes desde el punto de vista de un DBMS:

- La mezcla de sentencias SQL y de lenguaje de programación en el programa fuente con SQL incorporado es una manera efectiva de fundir los dos lenguajes. El lenguaje de programación ampliará proporciona flujo de control, varia-

bles, estructura de bloques y funciones de entrada/salida; SQL maneja el acceso a la base de datos y no tiene que preocuparse de estas otras construcciones.

- El uso de un precompilador significa que el trabajo intensivo en cálculo de análisis y optimización puede tener un lugar *durante el ciclo de desarrollo*. El programa ejecutable resultante es muy eficiente en su uso de los recursos de CPU.

- El módulo de petición a base de datos producido por el precompilador proporciona portabilidad de aplicaciones. Un programa de aplicación puede ser escrito y comprobado en un sistema, luego su programa ejecutable y el DBRM pueden ser trasladados a otro sistema. Después que el programa BIND del nuevo sistema cree el plan de aplicación y lo instale en la base de datos, el programa de aplicación puede utilizarlo sin tener que ser recompilado de nuevo.

- El interfaz efectivo en tiempo de ejecución del programa con las rutinas privadas de DBMS está totalmente oculto al programador de aplicación. El programador trabaja con SQL incorporado a nivel de código fuente y no tiene que preocuparse de otros interfaces más complejos.

Ejecución de un programa de SQL incorporado

Recuerde de la Figura 17.3 que el proceso de desarrollo de SQL incorporado produce dos componentes ejecutables, el propio programa ejecutable y el plan de aplicación del programa, almacenado en la base de datos. Cuando se ejecuta un programa de SQL incorporado, estos dos componentes son reunidos para hacer el trabajo de la aplicación:

1. Cuando se pide al sistema informático que ejecute el programa, el computador carga el programa ejecutable de la manera habitual y comienza a ejecutar sus instrucciones.
2. Una de las primeras llamadas generadas por el precompilador es una llamada a una rutina DBMS que encuentra y carga el plan de aplicación para el programa.
3. Por cada sentencia de SQL incorporado, el programa invoca una rutina DBMS privada, solicitando ejecución de la sentencia correspondiente en el plan de aplicación. El DBMS encuentra la sentencia, ejecuta esa parte del plan y luego devuelve control al programa.
4. La ejecución continúa de esta manera, con el programa ejecutable y el DBMS cooperado para llevar a cabo la tarea definida mediante el programa fuente original de SQL incorporado.

Seguridad en tiempo de ejecución. Cuando se utiliza SQL interactivo, el DBMS fuerza su seguridad basándose en el id-usuario que se suministra al programa SQL interactivo. Usted puede escribir cualquier sentencia SQL que deseé, pero los privilegios concedidos al id-usuario determinan si el DBMS ejecutará o no la sentencia que haya escrito. Cuando usted ejecuta un programa que utiliza SQL incorporado, existen *dos* id-usuario a considerar:

- el id-usuario de la persona que *desarrolló* el programa, o más específicamente, la persona que ejecutó el programa BIND para crear el plan de aplicación
- el id-usuario de la persona que está ahora *ejecutando* el programa y el plan de aplicación correspondiente

Puede parecer extraño considerar el id-usuario de la persona que ejecutó el programa BIND, pero de hecho DB2 y otros varios productos SQL comerciales utilizan ambos id-usuario en su esquema de seguridad. Para ver cómo funciona el esquema de seguridad, supongamos que el usuario JOE ejecuta el programa de mantenimiento de pedidos MANTPED, que actualiza las tablas PEDIDOS, VENTAS y OFICINAS. El plan de aplicación para el programa MANTPED fue originalmente tratado con BIND por el id-usuario ADMINPP, que pertenece al administrador de procesado de pedidos.

En el esquema DB2, cada plan de aplicación es un objeto de la base de datos, protegido por la seguridad de DB2. Para ejecutar un plan, JOE debe tener el privilegio EXECUTE para él. Si no lo tiene, la ejecución falla inmediatamente. Cuando el programa MANTPED se ejecuta, sus sentencias INSERT, UPDATE y DELETE incorporadas actualizan la base de datos. Los privilegios del usuario ADMINPP determinan si el plan tendrá permitido realizar estas actualizaciones. Observe que el plan puede actualizar las tablas incluso si JOE no tiene los privilegios necesarios. Sin embargo, las actualizaciones que pueden ser efectuadas son *ínicamente* aquellas que han sido explícitamente codificadas en las sentencias de SQL incorporadas al programa. Por tanto DB2 proporciona un control muy fino sobre la seguridad de la base de datos. Los privilegios de los usuarios para acceder a tablas pueden ser muy limitados, sin disminuir su capacidad para utilizar programas «enlazados».

No todos los productos DBMS proporcionan protección de seguridad para los planes de aplicación. Para aquellos que no lo hacen, los privilegios del usuario que ejecuta un programa de SQL incorporado determinan los privilegios del plan de aplicación del programa. Bajo este esquema, el usuario debe tener privilegios para efectuar todas las acciones acometidas por el plan, o el programa fallará. Si el usuario no va a tener estos mismos permisos en un entorno SQL interactivo, el acceso al propio programa SQL interactivo debe estar restringido, lo cual es una desventaja en este enfoque.

Replantamiento automático. Observe que un plan de aplicación está optimizado para la estructura de base de datos tal como existe en el momento en que el plan es colocado en la base de datos por el programa BIND. Si la estructura cambia más tarde (por ejemplo, si se suprime un índice o se elimina una columna de una tabla), cualquier plan de aplicación que referencia las estructuras modificadas puede resultar inválido. Para manejar esta situación, el DBMS almacena, con el plan de aplicación, una copia de las sentencias SQL originales que la produjeron. El DBMS también lleva la cuenta de todos los objetos de base de datos de los cuales depende cada plan de aplicación. Si cualquiera de estos objetos es modificado por una sentencia DDL, el DBMS marca automáticamente el plan como «inválido». La próxima vez que el programa trata de utilizar el plan, el DBMS detecta la situación y *replantea automáticamente* las sentencias para producir una nueva imagen de planteo (*bind*). Este replanteamiento automático es completamente transparente al programa de aplicación, excepto que una sentencia SQL puede tardar mucho más en ejecutarse cuando su plan es replanteado que cuando el plan es simplemente ejecutado. Aunque el DBMS replantea automáticamente un plan cuando una de las estructuras de la que depende se modifica, el DBMS no puede detectar automáticamente cambios en la base de datos que puedan hacer posible un mejor plan. Por ejemplo, si el plan utiliza un recorrido secuencial de una tabla debido a que no existía un índice adecuado cuando fue planteado inicialmente, es posible que una sentencia CREATE INDEX posterior pueda crear un índice adecuado. Para aprovechar la nueva estructura, debe ejecutarse explícitamente el programa BIND para reclarorar el plan.

Sentencias simples de SQL incorporado

Las sentencias más simples de SQL que se pueden incorporar a un programa son aquellas que son autocontenidoas y que no producen resultados de consulta. Por ejemplo, consideremos esta sentencia de SQL interactivo:

Suprime todos los vendedores con ventas inferiores a \$150.000.

```
DELETE FROM REVENTAS
WHERE VENTAS < 150000.00
```

Las Figuras 17.4, 17.5 y 17.6 muestran tres programas que efectúan la misma tarea que esta sentencia SQL interactiva, utilizando SQL incorporado. El programa de la Figura 17.4 está escrito en C, el programa de la Figura 17.5 está escrito en COBOL y el programa de la Figura 17.6 está escrito en FORTRAN. Aunque los programas son extremadamente sencillos, ilustran las características más básicas de SQL incorporado.

```
main()
{
    exec sql include sqca;
    exec sql declare repventas table
        (num_emp integer not null,
        nombre varchar(15) not null,
        edad integer,
        oficina rep integer,
        titulo varchar(10),
        contrato date not null,
        director integer,
        cuota money,
        ventas money not null);

    /* Visualiza un mensaje para el usuario */
    printf("Supresión de los vendedores con baja cuota.\n");

    /* Ejecuta la sentencia SQL */
    exec sql delete from repventas
        where ventas < 150000.00;

    /* Visualiza otro mensaje */
    printf("Supresión acabada.\n");
    exit();
}
```

Figura 17.4. Un programa de SQL incorporado escrito en C.

- Las sentencias de SQL incorporado aparecen en medio de las sentencias del lenguaje de programación anfitrión. Generalmente no importa si la sentencia SQL está escrita en mayúsculas o minúsculas; la práctica más habitual es seguir el estilo del lenguaje anfitrión.
- Todas las sentencias de SQL incorporado comienzan con un *introduction* que las señala como sentencias SQL. Los productos SQL de IBM utilizan el introducción EXEC SQL para la mayoría de los lenguajes anfitriones, y el estándar ANSI/ISO ha sido actualizado para que también lo especifique. Algunos productos de SQL incorporado utilizan aún otros introductores.

- Si una sentencia SQL se extiende a lo largo de varias líneas, se utiliza el método propio del lenguaje anfitrión para continuación de sentencias. Para los programas en COBOL, PL/I y C, no se requiere un carácter de continuación especial. Para programas FORTRAN, la línea segunda y las posteriores de la sentencia deben tener un carácter de continuación en la columna 6.
- Toda sentencia de SQL incorporado finaliza con un *terminator* que señala el final de la sentencia SQL. El terminador varía con el estilo del lenguaje

```

IDENTIFICATION DIVISION.
PROGRAM-ID. PRUEBA.
ENVIRONMENT DIVISION.
DATA DIVISION.
FILE SECTION.
WORKING-STORAGE SECTION.
EXEC SQL INCLUDE SQLCA.
EXEC SQL DECLARE REVENTAS TABLE
  (NUMEMPL INTEGER NOT NULL,
   NOMBRE VARCHAR(15) NOT NULL,
   EDAD INTEGER,
   OFICINA.REP INTEGER,
   TITULO VARCHAR(10),
   CONTRATO DATE NOT NULL,
   DIRECTOR INTEGER,
   CUOTA MONEY,
   VENTAS MONEY NOT NULL)
PROCEDURE DIVISION.
  * VISUALIZA UN MENSAJE PARA EL USUARIO
  DISPLAY 'Supresión de los vendedores con baja cuota.'
  *
  EXECUTA LA SENTENCIA SQL
    EXEC SQL DELETE FROM REVENTAS
      WHERE CUOTA < 150000
  END-EXEC.
END-EXEC.

```

Figura 17.5. Un programa de SQL incorporado escrito en COBOL.

```

PROGRAM PRUEBA
 100 FORMAT('A35')
  EXEC SQL INCLUDE SQLCA
  EXEC SQL DECLARE REVENTAS TABLE
    (NUMEMPL INTEGER NOT NULL,
     NOMBRE VARCHAR(15) NOT NULL,
     EDAD INTEGER,
     OFICINA.REP INTEGER,
     TITULO VARCHAR(10),
     CONTRATO DATE NOT NULL,
     DIRECTOR INTEGER,
     CUOTA MONEY,
     VENTAS MONEY NOT NULL)
  *
  * VISUALIZA UN MENSAJE PARA EL USUARIO
  * WRITE (6,100) 'Supresión de los vendedores con baja cuota.'
  *
  EXECUTA LA SENTENCIA SQL
    EXEC SQL DELETE FROM REVENTAS
      WHERE CUOTA < 150000
  RETURN
END

```

Figura 17.6. Un programa de SQL incorporado en FORTRAN.

```

main()
{
  exec sql include sqlca;
  /* Crea una nueva tabla REGIONES */
  exec sql create table regiones
    (nombre char(15),
     ciudad.op char(15),
     director integer,
     objetivo money,
     ventas money,
     primary key nombre,
     foreign key director
     references repventas);
  printf("Tabla creada.\n");
  /* Inserta dos filas: una por cada región */
  exec sql insert into regiones
    values ('Este', 'New York', 106, 0.00, 0.00);
  exec sql insert into regiones
    values ('Oeste', 'Los Angeles', 108, 0.00, 0.00);
  printf("Tabla rellena.\n");
}

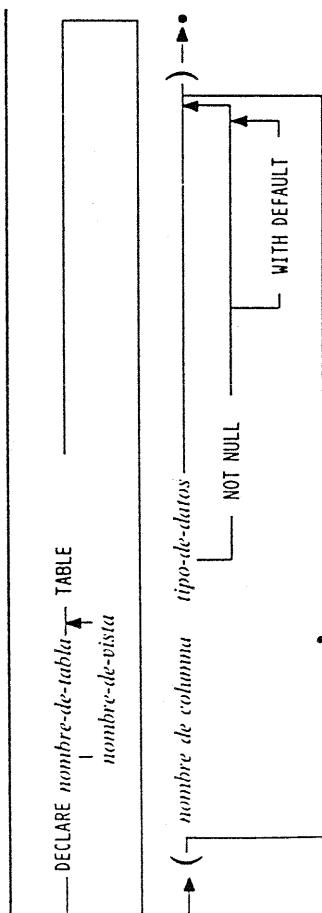
```

Figura 17.7. Utilización de SQL incorporado para crear una tabla.

La técnica de incorporación mostrada en las tres figuras funciona para cualquier sentencia SQL que, a) no dependa de los valores de las variables del lenguaje anfítrion para su ejecución y b) no recupere datos de la base de datos. Por ejemplo, el programa C de la Figura 17.7 crea una nueva tabla REGIONES e inserta dos filas en ella, utilizando exactamente las mismas características de SQL incorporado que el programa de la Figura 17.4. Por consistencia, todos los restantes ejemplos de programas del libro utilizarán el lenguaje de programación C, excepto cuando esté ilustrándose una característica particular de un determinado lenguaje anfítrion.

Declaración de tablas

La sentencia `DECLARE TABLE`, mostrada en la Figura 17.8, declara una tabla que será referenciada por una o más sentencias de SQL incorporado en el programa. Se trata de una sentencia opcional que ayuda al precompilador en su tarea de análisis y validación de las sentencias de SQL incorporado. Utilizando la sentencia `DECLARE TABLE`, el programa especifica explícitamente sus consideraciones respecto a las columnas de la tabla y a los tipos y tamaños de los datos. El precompilador comprueba las referencias a tablas y columnas en el programa para asegurarse que se conforman a la declaración de tabla.



Ejemplo 17.8 Diagrama sintáctico de la sentencia DECLARATE.

Los programas de las Figuras 17.4, 17.5 y 17.6 utilizan todos la sentencia `DECLARE TABLE`. Es importante observar que la sentencia aparece únicamente con propósitos de documentación y para uso del precompilador. No es una sentencia ejecutable, y no es necesario declarar explícitamente las tablas antes de referirse a ellas en sentencias de DML o DDL incorporadas. Sin embargo, la utilización de la sentencia `DECLARE TABLE` hace los programas más autodocumentados y más simples de mantener. Los productos SQL de IBM soportan todos la sentencia `DECLARE TABLE`, pero la mayoría de los restantes productos SQL no la proporcionan aún, y sus dorecompiladores generarán un mensaje de error si se utiliza.

Gestión de errores

Cuando se escribe una sentencia SQL interactiva que provoca un error, el programa SQL interactivo visualiza un mensaje de error, aborta la sentencia y solicita que se escriba una nueva sentencia. En SQL incorporado, el manejo de errores pasa a ser responsabilidad del programa de aplicación. Realmente, las competencias de SQL incorporado pueden producir dos tipos distintos de errores:

- **Erros en tiempo de compilación.** Comas mal colocadas, palabras claves de SQL mal escritas y errores similares en sentencias de SQL incorporado son detectados por el precompilador de SQL y reportados al programador. El programador puede arreglar los errores y recompilar el programa de aplicación.
 - **Erros en tiempo de ejecución.** Las referencias a columnas no existentes o la falta de permiso para actualizar una tabla sólo pueden detectarse en tiempo de ejecución. Errores como éstos deben ser detectados y manejados por el programa de aplicación.

El DBMS informa de los errores en tiempo de ejecución al programa de aplicación a través de un *Área de Comunicaciones SQL*, o SQLCA (*SQL Communications Area*). La SQLCA es una estructura de datos que contiene variables de error e indicadores de estado. Mediante el examen de la SQLCA, el programa de aplicación puede determinar el éxito o el fallo de las sentencias de SQL incorporando su actuación correspondientemente.

raio y actual correspondiente.

Observe en las Figuras 17.4, 17.5, 17.6 y 17.7 que la primera sentencia de SQL incorporada al programa es `INCLUDE SQLCA`. Esta sentencia dice al precompilador de SQL que incluya un área de comunicaciones SQL en este programa. Los contenidos específicos de la SQLCA varían ligeramente de un producto de DBMS a otro, pero el SQLCA siempre proporciona el mismo tipo de información. La Figura 17.9 muestra la definición del SQLCA situado por las bases de datos IBM (DB2, SQL/DS y OS/2 Extended Edition). La parte más importante de la SQLCA, la variable `SQLCODE`, es soportada por todos los productos SQL incorporados y está especificada en el estándar ANSI/ISO.

La variable SQLCODE. Cuando ejecuta cada sentencia de SQL incorporado, el DBMS fija el valor de la variable SQLCODE en SQLCA para que indique el estado de terminación de la sentencia:

- Un valor de `SALCODE` negativo indica un error serio que impidió que la sentencia se ejecutase correctamente. Por ejemplo, un intento de actualizar una vista de sólo lectura produciría un valor de `SQLCODE` negativo. Cada error en tiempo de ejecución que pueda ocurrir tiene asignado un valor negativo diferente.

- Un valor de SQLCODE positivo indica una condición de aviso. Por ejemplo, el truncamiento o redondeo de un dato recuperado por un programa produciría un aviso. Cada aviso de tiempo de ejecución que pueda ocurrir tiene asignado un valor positivo diferente.

```

struct sqlca {
    unsigned char    sqlcaid[8];      /* La cadena "SQLCA" */
    long             sqlabc;          /* Longitud de SQLCA, en bytes */
    short            sqlcode;         /* Código de estado SQL */
    short            sqlerrml;        /* Longitud del array de datos sqlerrmc */
    unsigned char    sqlermc[70];     /* nombre(s) de objeto(s) que produce(n) error */
    unsigned char    sqlerip[8];
    unsigned char    sqterr[6];
    long             sqlwarn[8];
    unsigned char    sqltext[8];
    unsigned char    sqlwarnr[8];
};

#define SQLCODE  sqlca.sqlcode
/* Código de estado SQL */

/* Una 'W' en cualquiera de los campos SQLWARN indica una condición de aviso;
   si no es así, contienen un blanco */

#define SQLWARN sqlca.sqlwarn[0]
#define SQLWARN1 sqlca.sqlwarn[1]    /* Indicador principal de aviso */
#define SQLWARN2 sqlca.sqlwarn[2]    /* Cadena truncada */
#define SQLWARN3 sqlca.sqlwarn[3]    /* Nulos (NULL) eliminados de función de columna */
#define SQLWARN4 sqlca.sqlwarn[4]    /* Demasiados o demasiado pocas variables principales */
#define SQLWARN5 sqlca.sqlwarn[5]    /* UPDATE/DELETE preparado sin WHERE */
#define SQLWARN6 sqlca.sqlwarn[6]    /* Incompatibilidad SQL/DS - DB2 */
#define SQLWARN7 sqlca.sqlwarn[7]    /* Fecha no válida en expr. aritmética */
/* Reservado */

```

Figura 17.9. El área de comunicaciones SQL (sqlca) de las bases de datos IBM.

```

;
exec sql delete from repventas
      where cuota < 150000;
if (sqlca.sqlcode < 0)
  goto rutina.error;
;
;

rutina.error:
  printf("SQL error: %ld\n", sqlca.sqlcode);
  exit(0);
;
;
```

Figura 17.10. Un extracto de un programa C con comprobación de errores SQLCODE.

- WHENEVER SQLERROR dice al precompilador que genere código para manejar errores (SQLCODE negativo).
- WHENEVER SQLWARNING dice al precompilador que genere código para manejar avisos (SQLCODE positivo).
- WHENEVER NOT FOUND dice al precompilador que genere código que maneje un aviso particular —el aviso generado por el DBMS cuando el programa trata de recuperar resultados de consultas cuando ya no hay más—. Este uso de la sentencia WHENEVER es específico de las sentencias SELECT singulares y de la sentencia FETCH, y se describirá posteriormente en este capítulo.

```

;
01 MENS_IMPRENSA          PIC X(11) Value 'SQL error: '.
02 PLANTILLA               PIC S7(9).
02 CODIGO_IMPR              PIC S7(9).

;
EXEC SQL DELETE FROM REPVENTAS
      WHERE CUOTA < 150000
END-EXEC.
IF SQLCODE NOT = ZERO GOTO RUTINAPERRO.

;
ROUTINAPERRO:
  MOVE SQLCODE TO CODIGO_IMPR.
  DISPLAY MENS_IMPRENSA.
;
;
```

Figura 17.11. Extracto de un programa COBOL con comprobación de errores SQLCODE.

Puesto que toda sentencia de SQL incorporado ejecutable puede generar potencialmente un error, un programa bien escrito comprobará el valor de SQLCODE después de cada sentencia de SQL incorporado ejecutable. La Figura 17.10 muestra un trozo de un programa C que comprueba el valor de SQLCODE. La Figura 17.11 muestra un trozo análogo de un programa COBOL.

La sentencia WHENEVER. Para un programador rápidamente resulta tedioso escribir programas que explícitamente comprueben el valor de SQLCODE después de cada sentencia de SQL incorporada. Para simplificar el manejo de errores, SQL incorporado soporta la sentencia WHENEVER, mostrada en la Figura 17.12. La sentencia WHENEVER es una directiva para el precompilador de SQL, y no una sentencia ejecutable. Dice al precompilador que genere *automáticamente* código de gestión de errores a continuación de cada sentencia ejecutable de SQL incorporado y especifica lo que el código generado debería hacer.

Se puede utilizar la sentencia WHENEVER para decir al precompilador cómo manejar tres condiciones de excepción diferentes:

Para cualquiera de estas tres condiciones, se puede decir al precompilador que genere código que tome una de estas dos acciones:

- WHENEVER/GOTO dice al precompilador que genere un salto a la *etiqueta* especificada, que debe ser una etiqueta de sentencia o un número de sentencia en el programa.
- WHENEVER/CONTINUE dice al precompilador que deje al programa continuar hasta la siguiente sentencia del lenguaje anfítrion.

La sentencia WHENEVER es una directiva al precompilador, y su efecto puede ser sustituido por el de otra sentencia WHENEVER que aparezca posteriormente en el texto del programa. La Figura 17.13 muestra un extracto de un programa con tres sentencias WHENEVER y cuatro sentencias de SQL ejecutables. En ese programa, un error en cualquiera de las dos sentencias DELETE da lugar a un salto a error1 debido a la primera sentencia WHENEVER. Un error en la sentencia UPDATE incorporada lleva directamente a las sentencias siguientes del programa. Un error en la sentencia INSERT incorporada resulta en un salto a error2. Como muestra este ejemplo, el principal uso de la forma WHENEVER/CONTINUE de la sentencia es cancelar el efecto de una sentencia WHENEVER previa.

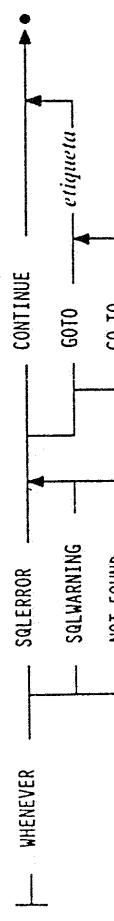


Figura 17.12. Diagrama sintáctico de la sentencia WHENEVER.

La sentencia WHENEVER hace mucho más simple la gestión de errores en SQL incorporado y es más habitual en un programa de aplicación que la comprobación directa de SQLCODE. Recuerde, sin embargo, que después que aparezca una sentencia WHENEVER/GOTO, el precompilador generará un test y un salto a la etiqueta especificada por *cuada* sentencia de SQL incorporado posterior. Usted debe disponer el programa de modo que la etiqueta especificada sea un objetivo válido para el salto desde esas sentencias de SQL incorporado, o utilizar otra sentencia WHENEVER para especificar un destino diferente o cancelar los efectos del WHENEVER/GOTO.

Utilización de variables principales

Los programas de SQL incorporado que aparecían en las figuras anteriores no proporcionaban interacción real entre las sentencias de programación y las

```
:
exec sql whenever sqlerror goto error1;
exec sql delete from repventas
  where cuota < 150000;
exec sql delete from clientes
  where limite_credito < 20000;
exec sql whenever sqlerror continue;
exec sql update repventas
  set cuota = quota * 1.05;
exec sql whenever sqlerror goto error2;
exec sql insert into repventas (num_empl, nombre, cuota)
  values (116, 'Jan Hamilton', 100000.00);
:
error1:
printf('SQL DELETE error: %ld\n', sqlca.sqlcode);
exit0;
:
error2:
printf('SQL INSERT error: %ld\n', sqlca.sqlcode);
exit0;
:
;
```

Figura 17.13. Utilización de la sentencia WHENEVER.

sentencias de SQL incorporado. En la mayoría de las aplicaciones, será de desear utilizar el valor de una o más variables del programa en las sentencias de SQL incorporado. Por ejemplo, supongamos que usted deseara escribir un programa para aumentar o disminuir todas las cuotas de ventas en alguna cantidad determinada. El programa debería pedir al usuario el importe de la variación y luego utilizar una sentencia UPDATE incorporada para cambiar la columna CUOTA en la tabla REPVENTAS.

SQL incorporado soporta esta capacidad a través del uso de *variables principales*. Una variable principal es una variable de programa declarada en el lenguaje anfitrión (por ejemplo, una variable COBOL o una variable C) que es referenciada en una sentencia de SQL incorporado. Para identificar la variable principal, el nombre de la variable está prefijado por dos puntos (:); cuando aparece en una sentencia de SQL incorporado, los dos puntos permiten al

precompilador distinguir fácilmente entre variables principales y objetos de la base de datos (tales como tablas o columnas) que puedan tener el mismo nombre.

La Figura 17.14 muestra un programa C que implementa la aplicación de ajuste de cuotas utilizando una variable principal. El programa pide al usuario la cuantía del ajuste y almacena el valor introducido en la variable denominada importe. Esta variable principal es referenciada en la sentencia UPDATE incorporada. Conceptualmente, cuando se ejecuta la sentencia UPDATE, se obtiene el valor de la variable importe, y ese valor se sustituye en la variable principal en la sentencia SQL. Por ejemplo, si se introduce la cantidad 500 en respuesta a la petición, el DBMS ejecuta efectivamente esta sentencia UPDATE.

```
exec sql update repventas
      set cuota = cuota + 500;
```

Una variable principal puede aparecer en una sentencia de SQL incorporada en cualquier lugar en que pueda aparecer una constante. En particular, una variable principal puede ser utilizada en una expresión de asignación:

```
exec sql update repventas
      set cuota = cuota + :importe;
```

Una variable principal puede aparecer en una condición de búsqueda:

```
exec sql delete from repventas
      where cuota < :importe;
```

Una variable principal también puede ser utilizada en la cláusula VALUES de una sentencia de inserción:

```
exec sql insert into repventas (num_empl, nombre, cuota)
      values (116, 'Bill Roberts', :importe);
```

En cada caso, observe que la variable principal forma parte de la *entrada del programa* al DBMS; forma parte de la sentencia SQL remitida al DBMS para ejecución. Posteriormente en este capítulo, veremos como las variables principales se utilizan también para recibir *salida* del DBMS; reciben resultados de consulta devueltos por el DBMS al programa.

Observe que una variable principal no *puede* ser utilizada en lugar de un identificador SQL. Este uso de la variable principal nombrecol es ilegal:

```
char *nombrecol = 'cuota';
exec sql insert into repventas (num_empl, nombre, :nombrecol)
      values (116, 'Bill Roberts', 0.00);
```

```
main()
{
    exec sql include sqlda;
    exec sql begin declare section; /* importe (del usuario) */
    float importe;
    exec sql end declare section;

/* Pide al usuario que escriba el aumento/disminución de la cuota */
printf("En cuánto se elevan/reajan las cuotas: ");
scanf("%f", &importe);

/* Actualiza la columna CUOTA en la tabla REVENTAS */
exec sql update repventas
      set cuota = cuota + importe;

/* Comprueba los resultados de la ejecución de la sentencia */
if (sqlsta.sqlcode != 0)
    printf("Error durante la actualización.\n");
else
    printf("Actualización correcta.\n");

exit(0);
}
```

Figura 17.14. Utilización de variables principales.

Declaración de variables principales. Cuando se utiliza una variable principal en una sentencia de SQL incorporado, debe declararse la variable utilizando el método normal para declaración de variables en el lenguaje de programación anfitrión. Por ejemplo, en la Figura 17.14, la variable principal importe está declarada utilizando la sintaxis normal del lenguaje C (float importe). Cuando el precompilador procesa el código fuente del programa, anota el nombre de cada variable que encuentra, junto con su tipo de datos y tamaño. El precompilador utiliza esta información para generar código correcto posteriormente cuando encuentra un uso de la variable como variable principal en una sentencia SQL.

Las dos sentencias de SQL incorporado BEGIN DECLARE SECTION y END DECLARE SECTION encierran las declaraciones de variables principales, como se muestra en la Figura 17.14. Estas dos sentencias son propias de SQL incorporado, y no son ejecutables. Son directivas para el precompilador, que le informan de cuándo debe «prestar atención» a las declaraciones de variables y cuándo puede ignorarlas.

En un programa de SQL incorporado sencillo, puede ser posible reunir todas las declaraciones de variables principales en una «sección de declaración». Generalmente, sin embargo, las variables principales deben ser declaradas en varios puntos dentro del programa, especialmente en lenguajes con estructura de

bloques tales como C, Pascal y PL/I. En este caso cada declaración de variables principales debe ir encerrada entre un par de sentencias BEGIN DECLARE SECTION/END DECLARE SECTION.

Las sentencias BEGIN DECLARE SECTION y END DECLARE SECTION son relativamente nuevas en el lenguaje SQL incorporado. Están especificadas en el estándar SQL ANSI/ISO, y DB2 las requiere en SQL incorporado para C, que fue introducido en la Versión 2 de DB2. Sin embargo, DB2 y muchos otros productos DBMS no precisaban históricamente secciones de declaración, y muchos precompiladores de SQL no soportan aún las sentencias BEGIN DECLARE SECTION y END DECLARE SECTION. En este caso el precompilador examina y procesa todas las declaraciones de variables en el programa anfitrión.

Cuando se utiliza una variable principal, el precompilador puede limitar la flexibilidad de la declaración de la variable en el lenguaje de programación anfitrión. Por ejemplo, consideremos el siguiente código fuente en lenguaje C:

```
#define LONGGRANBUF 256
.
.
.
exec sql begin declare section;
char granbuffer[LONGGRANBUF+1];
exec sql end declare section;
```

Esta es una declaración C válida de la variable granbuffer. Sin embargo, si se trata de utilizar granbuffer como variable principal en una sentencia de SQL incorporado como ésta:

```
exec sql update repventas
      set cuota = 300000
      where nombre = :granbuffer;
```

muchos precompiladores generarán un mensaje de error, quejándose de declaración ilegal de granbuffer. El problema es que muchos precompiladores no reconocen constantes simbólicas como LONGGRANBUF. Esto es tan sólo un ejemplo de las consideraciones especiales aplicables cuando se utiliza SQL incorporado y un precompilador. Afortunadamente, los precompiladores ofrecidos por los principales vendedores de DBMS están siendo mejorados continuamente, y el número de problemas por casos especiales como éste está disminuyendo.

Variables principales y tipos de datos. Los tipos de datos soportados por un DBMS basado en SQL y los tipos de datos soportados por un lenguaje de programación tal como C o FORTRAN son a menudo bastante diferentes. Estas diferencias afectan a las variables principales, debido a que éstas juegan un doble papel. Por un lado, una variable principal es una variable de programa, declarada utilizando los tipos de datos del lenguaje de programación y manipulada por sentencias del lenguaje de programación. Por otro lado, una variable

```
      exec sql begin declare section;
      int varprin1      = 106;
      char *varprin2    = "Joe Smith";
      float varprin3    = 150000.00;
      char *varprin4    = "01-JUN-1990";
      exec sql end declare section;

      exec sql update repventas
              set director = :varprin1
              where num_empl = 102;

      exec sql update repventas
              set nombre = :varprin2
              where num_empl = 102;

      exec sql update repventas
              set cuota = :varprin3
              where num_empl = 102;

      exec sql update repventas
              set contrato = :varprin4
              where num_empl = 102;
      :
```

Figura 17.15. Variables principales y tipos de datos.

principal es utilizada en sentencias de SQL incorporado que contienen datos de la base de datos.

Consideremos las cuatro sentencias UPDATE incorporadas de la Figura 17.15. En la primera sentencia UPDATE, la columna DIRECTOR tiene un tipo de datos INTEGER, por lo que varprin1 debería declararse como variable entera de C. En la segunda sentencia, la columna NOMBRE tiene un tipo de dato VARCHAR, por lo que varprin2 debería contener datos de cadenas de caracteres. El programa debería declarar varprin2 como un array de caracteres de C, y la mayoría de los productos DBMS esperarán que los datos del array terminen en un carácter nulo (0). En la tercera sentencia UPDATE, la columna CUOTA tiene un tipo de datos decimal empaqetado. Para la mayoría de los productos DBMS, se puede declarar varprin3 como una variable de coma flotante de C, y el DBMS traducirá automáticamente el valor de coma flotante al formato MONEY del DBMS. Finalmente, en la cuarta sentencia UPDATE, la columna CONTRATO tiene un tipo de dato DATE en la base de datos. Para la mayoría de los productos DBMS, habría que declarar varprin4 como un array de caracteres de C, y rellenar el array con un formato de texto de fecha aceptable al DBMS. Como muestra la Figura

ra 17.13, los tipos de datos de las variables principales deben ser elegidos cuidadosamente para que se correspondan con el uso a que están destinados en las sentencias de SQL incorporado. La Tabla 17.2 muestra los tipos de datos más frecuentes utilizados en SQL y los tipos de datos correspondientes en varios lenguajes de programación. Observe, sin embargo, que en muchos casos no existe una correspondencia única entre los tipos. Además, cada producto de DBMS tiene sus propias idiosincrasias en cuanto a tipos de datos y sus propias reglas de conversión de tipos cuando utilizan variables principales. Antes de confiar en un comportamiento específico de conversión de datos, consulte la

documentación de su producto DBMS particular y lea cuidadosamente la descripción del lenguaje de programación particular que esté usted utilizando.

Variabes principales y valores NULL. La mayoría de los lenguajes de programación no proporcionan soporte al estilo SQL para valores desconocidos o que faltan. Una variable C, COBOL o FORTAN, por ejemplo, siempre tiene un valor. No existe el concepto de que un valor sea NULL o falte. Esto provoca un problema cuando se desean almacenar valores NULL en la base de datos o recuperar valores NULL de la base de datos utilizando SQL programado. SQL incorporado resuelve este problema permitiendo que cada variable principal tenga una variable *indicadora* asociada. En una sentencia de SQL incorporado, la variable principal y la variable indicadora *juntas* especifican un único valor en estilo SQL, del modo siguiente:

Tipos SQL	Tipos C	Tipos COBOL	Tipos FORTRAN	Tipos PL/I
SMLINT	short	PIC S9(4) COMP	INTEGER*2	FIXED BIN(15)
INTEGER	int	PIC S9(9) COMP	INTEGER*4	FIXED BIN(31)
FLOAT	float	COMP-1	REAL*4	BIN FLOAT(21)
DOUBLE	double	COMP-2	REAL*8	BIN FLOAT(53)
DECIMAL(p,s)	double ¹	PIC S9(p-s)V9(.s) COMP-3	REAL*8 ¹	FIXED DEC(p,s)
MONEY(p,s)	double ¹	PIC S9(p-s)V9(.s) COMP-3	REAL*8 ¹	FIXED DEC(p,s)
CHAR(n)	char x[n+1] ²	PIC X(n)	CHAR*n	CHAR(n) VAR
VARCHAR(n)	struct { short len; char x[n+1]}	01 X 02 LEN PIC S9(4) COMP char x[n+1]	CHARACTER*11 ³ CHARACTER*8 ⁵ CHARACTER*27 ⁵	CHAR(11) ⁵ CHAR(8) ⁵ CHAR(27) ⁵
DATE	char x[12] ⁵	PIC X(11) ⁵		
TIME	char x[9] ⁵	PIC X(8) ⁵		
TIMESTAMP	char x[28] ⁵	PIC X(27) ⁵		

Notas:

¹ El lenguaje anfítrion no soporta datos decimales empaquetados; la conversión a datos en coma flotante puede provocar errores de truncamiento o redondeo.

² El DBMS puede devolver o no una cadena C con un carácter terminador nulo.

³ Algunos productos DBMS (incluyendo DB2, OS/2 EE y Oracle) utilizan esta estructura VARCHAR, otros productos (incluyendo Informix e Ingres) soportan cadenas C normales terminadas en nulo.

⁴ El lenguaje anfítrion no soporta cadenas de longitud variable; todos los datos de cadenas son tratados como cadenas de longitud fija rellenas con blancos.

⁵ Supone que los tipos de datos de fecha/hora se convierten a cadenas de caracteres en los formatos "dd-mm-aaaa", "hh:mm:ss" y "dd-mm-aaaa hh:mm:ss". Algunos productos DBMS devuelven los datos de fecha/hora en representación binaria.

Cuando se especifica una variable principal en una sentencia de SQL incorporado, se le puede seguir inmediatamente con el nombre de la variable indicadora correspondiente. Ambos nombres de variables van precedidos por dos puntos. He aquí la variable principal *importe* con la variable indicadora *ind_importe*.

```
exec sql update repventas
      set cuota = :importe :ind_importe, ventas = :importe2
      where cuota < 20000.00;
```

Si *ind_importe* tiene un valor no negativo cuando se ejecuta la sentencia UPDATE, el DBMS trata a la sentencia como si fuera la siguiente:

```
exec sql update repventas
      set cuota = :importe, ventas = :importe2
      where cuota < 20000.00;
```

Tabla 17.2. Tipos de datos en SQL y en lenguajes anfítriones.

Si `ind_importe` tiene un valor negativo cuando se ejecuta la sentencia `UPDATE`, el DBMS tratará a la sentencia como si fuera:

```
exec sql update repventas
    set cuota = NULL, ventas = :importe2
  where cuota < 20000.00
```

La pareja variable principal/variable indicadora puede aparecer en la cláusula de asignación de una sentencia `UPDATE` incorporada (como se muestra aquí), o en la cláusula de valores de una sentencia `INSERT` incorporada. No se puede utilizar una variable indicadora en una condición de búsqueda, por lo que esta sentencia de SQL incorporado es ilegal:

```
exec sql delete from repventas
    where cuota = :importe :ind_importe;
```

La prohibición existe por la misma razón que la palabra clave `NULL` no está permitida en la condición de búsqueda —no tiene sentido examinar si `CUOTA` y `NULL` son iguales, puesto que la respuesta siempre será `NULL` (desconocida)—. En vez de utilizar la variable indicadora, debe utilizarse un test explícito `IS NULL`. Este par de sentencias de SQL incorporado consigue el objetivo pretendido por la sentencia ilegal precedente:

```
if (ind_importe < 0) {
    exec sql delete from repventas
        where cuota is null;
}
else {
    exec sql delete from repventas
        where cuota = :importe;
}
```

Las variables indicadoras son especialmente útiles cuando se recuperan datos de la base de datos en un programa y los valores recuperados pueden ser `NULL`. Este uso de las variables indicadoras se describirá posteriormente en este capítulo.

Recuperación de datos en SQL incorporado

Utilizando las características de SQL incorporado descritas hasta ahora, se puede incorporar cualquier sentencia SQL interactiva *excepto* la sentencia `SELECT` en un programa de aplicación. La recuperación de datos con un programa SQL incorporado requiere algunas extensiones especiales a la sentencia `SELECT`. La razón para estas extensiones es que existe una disparidad fundamental entre el lenguaje SQL y los lenguajes de programación tales como C y COBOL: una

consulta SQL produce una *tabla* entera de resultados de consulta, pero la mayoría de los lenguajes de programación solamente pueden manipular elementos de datos individuales o registros individuales de datos (filas).

SQL incorporado debe construir un «puente» entre la lógica a nivel de tabla de la sentencia `SELECT` de SQL y el procesamiento fila a fila de C, COBOL y otros lenguajes de programación anfistriones. Por esta razón, SQL incorporado divide las consultas en dos grupos:

- *Consultas de una fila*, en donde se espera que los resultados de la consulta contengan una única fila de datos. Examinar al límite de crédito de un cliente o recuperar las ventas y cuotas de un vendedor particular son ejemplos de este tipo de consulta.
- *Consultas multifila*, en donde se espera que los resultados de la consulta puedan contener cero, una o muchas filas de datos. Listar los pedidos con importes superiores a \$20.000 o recuperar los nombres de todos los vendedores que están por encima de sus cuotas son ejemplos de este tipo de consulta. SQL interactivo no distingue entre estos dos tipos de consulta; la misma sentencia `SELECT` interactiva las maneja a ambas. En SQL incorporado, sin embargo, los dos tipos de consultas son manejados de forma muy diferente. Las consultas de una fila son más sencillas de manejar, y se discuten en la próxima sección. Las consultas multifila se discuten más tarde en este capítulo.

Consultas monofila

Muchas consultas SQL útiles devuelven una única fila de resultados. Las consultas de una fila son especialmente comunes en programas de procesamiento de transacciones, en donde un usuario introduce un número de cliente o un número de pedido y el programa recupera los datos relevantes referentes al cliente o el pedido. En SQL incorporado, las consultas monofila son manejadas por la sentencia `SELECT singular` mostrada en la Figura 17.16. La sentencia `SELECT singular` tiene una sintaxis muy parecida a la de la sentencia `SELECT interactiva`. Tiene una cláusula `SELECT`, una cláusula `FROM` y cláusula opcional `WHERE`. Debido a que la sentencia `SELECT singular` devuelve una sola fila de datos, no hay necesidad de cláusulas `GROUP BY`, `HAVING` u `ORDER BY`. La cláusula `INTO` especifica las variables principales que van a recibir los datos recuperados por la sentencia.

La Figura 17.17 muestra un sencillo programa con una sentencia `SELECT singular`. El programa pide al usuario un número de empleado y luego recupera el nombre, la cuota y las ventas del vendedor correspondiente. El DBMS coloca los tres elementos de datos recuperados en las variables principales `nombre`, `cuota` y `ventas`, respectivamente.

Recuerde que las variables principales utilizadas en las sentencias INSERT, DELETE y UPDATE de los ejemplos anteriores eran variables principales de entrada. Por contra, las variables principales especificadas en la cláusula INTO de la sentencia SELECT singular son variables principales de *salida*. Cada variable principal designada en la cláusula INTO recibe una única columna de la fila de resultados de la consulta. Los elementos de la lista de selección y las variables principales correspondientes están emparejadas en secuencia, tal como aparecen en sus respectivas cláusulas, y el número de columnas de resultados debe ser el mismo que el número de variables principales. Además, el tipo de datos de cada variable principal debe ser compatible con el tipo de datos de la columna correspondiente de resultados de la consulta.

Como discutimos anteriormente, la mayoría de los productos DBMS manejarán automáticamente conversiones «razonables» entre los tipos de datos DBMS y los tipos de datos soportados por el lenguaje de programación. Por ejemplo, la mayoría de los productos DBMS convertirán los datos MONEY recuperados de la base de datos en datos decimales empaquetados (COMP-5) antes de almacenarlos en una variable COBOL, o en datos de coma flotante antes de almacenarlos en una variable C. El precompilador utiliza su conocimiento del tipo de datos de la variable principal para manejar la conversión correctamente.

```

main()
{
    exec sql begin declare section;
    int numrep; /* número de empleado (del usuario) */
    char nombrerep[16]; /* nombre de vendedor recuperado */
    float cuotarep; /* cuota recuperada */
    float ventasrep; /* ventas recuperadas */

    exec sql end declare section;

    /* Pide al usuario que escriba el número de empleado */
    printf("Introduzca el número del vendedor: ");
    scanf("%d", &numrep);

    /* Ejecuta la consulta SQL */
    exec sql select nombre, cuota, ventas
        from repventas
        where numempl = :numrep
        into :nombrerep, :cuotarep, :ventasrep;

    /* Visualiza los datos recuperados */
    if (sqlca.sqlcode == 0) {
        printf("Nombre: %s\n", nombrerep);
        printf("Cuota: %f\n", cuotarep);
        printf("Ventas: %f\n", ventasrep);
    }
    else if (sqlca.sqlcode == 100)
        printf("No existe vendedor con ese número de empleado.\n");
    else
        printf("SQL error: %ld\n", sqlca.sqlcode);

    exit(0);
}

```

Figura 17.17. Utilización de la sentencia SELECT singular.

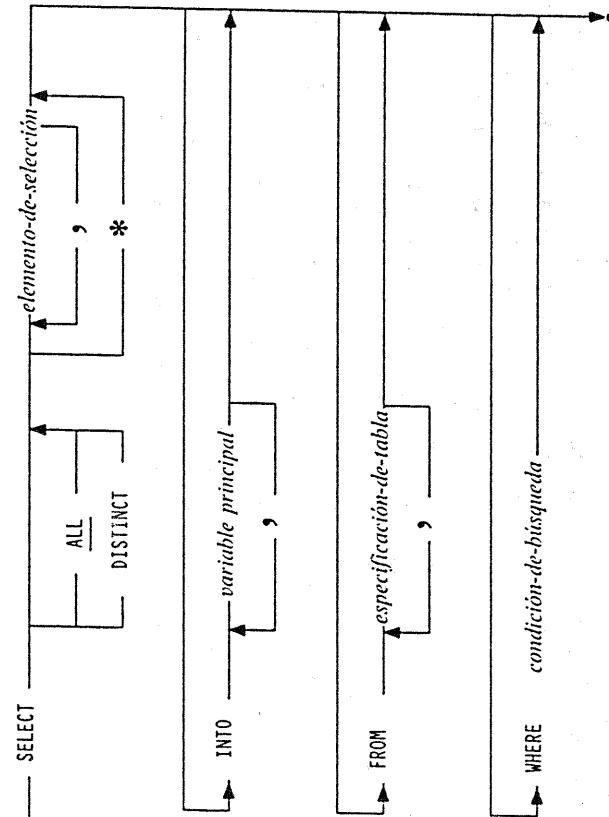


Figura 17.16. Diagrama sintáctico de la sentencia SELECT singular.

Los datos texto de longitud variable también deben ser convertidos antes de almacenarlos en una variable principal. Tipicamente, un DBMS convierte los datos VARCHAR en cadenas terminadas en nulo para programas C y en cadenas de longitud variable (con la cuenta del número de caracteres al comienzo) para programas en Pascal. Para programas COBOL y FORTRAN, la variable principal debe ser declarada generalmente como una estructura de datos con un campo entero «cuenta» y un array de caracteres. El DBMS devuelve los caracteres efectivos en el array de caracteres y devuelve la longitud de los datos en el campo de cuenta de la estructura.

Si un DBMS soporta datos fecha/hora u otros tipos de datos, son necesarios otras conversiones. Algunos productos DBMS devuelven sus representaciones internas de fecha/hora en una variable principal entera. Otros convierten los

datos fecha/hora a fo
cadenas. La Tabla 17.
proporcionadas por l
mentación de SQL in
mación específica.

La condición NOT F
sentencia SELECT sin
de terminación:

- Si se recupera con variables principales recuperados.
- Si la consulta principal no tiene resultados.
- Si la consulta no tiene un aviso especial NOT SQLCODE devuelve

El estándar SQL no especifica un v
mayoría de los re
otros productos S

Recuperación de
datos pueden con
nar un modo par
aplicación. Para
indicadoras en la
de la sentencia IF

Cuando se es
seguida inmediatamente
Figura 17.18 mu
utiliza la variab
Puesto que las c
ción de la tabla
necesarias varia
Después que

programación. DB2 soporta estructuras C y PL/I, pero no soporta las estructuras de COBOL o de lenguaje ensamblador, por ejemplo.

Variables principales de entrada y salida. Las variables principales proporcionan comunicación en dos sentidos entre el programa y el DBMS. En el programa mostrado en la Figura 17.18, las variables principales numrep y nombrerrep ilustran los dos papeles diferentes jugados por las variables principales:

- La variable principal numrep es una variable principal de *entrada*, utilizada para transferir datos desde el programa al DBMS. El programa asigna un valor a la variable antes de ejecutar la sentencia incorporada, y ese valor pasa a formar parte de la sentencia SELECT que va a ejecutar el DBMS. El DBMS no hace nada por alterar el valor de la variable.
- La variable principal nombrerrep es una variable principal de *salida*, utilizada para transferir datos desde el DBMS al programa. El DBMS asigna un valor a esta variable cuando ejecuta la sentencia SELECT incorporada. Después que la sentencia ha sido ejecutada, el programa puede utilizar el valor resultante.

Las variables principales de entrada y salida se declaran del mismo modo y se especifican utilizando la misma notación de dos puntos dentro de una sentencia SQL incorporada. Sin embargo, con frecuencia es útil pensar en términos de variables principales de entrada y de salida cuando se está codificando efectivamente un programa de SQL incorporado. Las variables principales de entrada pueden ser utilizadas en cualquier sentencia SQL en el lugar en que pueda aparecer una constante. Las variables principales de salida solamente se utilizan con la sentencia SELECT singular y con la sentencia FETCH, que se describirá posteriormente en este capítulo.

Consultas multifila

Cuando una consulta produce una tabla entera de resultados, SQL incorporado debe proporcionar un modo para que el programa de aplicación procese los resultados una fila cada vez. SQL incorporado soporta esta capacidad mediante la definición de un nuevo concepto SQL, denominado *cursor*, y la adición de varias sentencias al lenguaje SQL interactivo. He aquí una revisión de la técnica de SQL incorporado para procesamiento de consultas multifilas y las nuevas sentencias que requiere:

1. La sentencia DECLARE CURSOR especifica la consulta a efectuar y asocia un nombre de cursor con la consulta.
2. La sentencia OPEN pide al DBMS que comience a ejecutar la consulta y a

generar resultados. Posiciona el cursor por delante de la primera fila de resultados.

3. La sentencia FETCH avanza el cursor a la primera fila de resultados de la consulta y recupera sus datos en variables principales para uso del programa de aplicación. Posteriormente sentencias FETCH recorren los resultados de la consulta fila a fila, avanzando el cursor a la fila siguiente de resultados y recuperando sus datos en las variables principales.
4. La sentencia CLOSE finaliza el acceso a los resultados de la consulta y rompe la asociación entre el cursor y los resultados.

La Figura 17.20 muestra un programa que utiliza SQL incorporado para efectuar una sencilla consulta multifila. Las referencias numeradas de la figura corresponden con los números de los pasos anteriores. El programa recupera y visualiza, en orden alfabético, el nombre, la cuota y las ventas anuales hasta la fecha de los vendedores cuyas ventas exceden a la cuota. La consulta SQL interactiva que imprime esta información es:

```
SELECT NOMBRE, CUOTA, VENTAS
  FROM REPVENTAS
 WHERE VENTAS > CUOTA
   ORDER BY NOMBRE
```

Observe que esta consulta aparece, palabra por palabra, en la sentencia DECLARE CURSOR incorporada de la Figura 17.20. La sentencia también asocia el nombre de cursor cursrep con la consulta. Este nombre de cursor se utiliza posteriormente en la sentencia OPEN CURSOR para iniciar la consulta y posicionar el cursor delante de la primera fila de resultados de la consulta.

La sentencia FETCH dentro del bucle for accede a la siguiente fila de resultados cada vez que se ejecuta el bucle. La cláusula INTO de la sentencia SELECT singular funciona de modo semejante a la cláusula INTO de la sentencia SELECT multifila. Especifica las variables principales que van a recibir los datos accedidos —una variable principal por cada columna de resultados de la consulta—. Como en los ejemplos anteriores, se utiliza una variable indicadora (*indicadorep*) cuando un dato accedido puede contener valores NULL.

Cuando no existen más filas de resultados a los que acceder, el DBMS devuelve el aviso NOT FOUND en respuesta a la sentencia FETCH. Este es exactamente el mismo código de aviso que se devuelve cuando la sentencia SELECT singular no recupera una fila de datos. En este programa, la sentencia WHENEVER NOT FOUND hace que el precompilador genere código que comprueba el valor de SQLCODE después de la sentencia FETCH. Este código generado salta a la etiqueta hecho cuando se presenta la condición NOT FOUND, y a la etiqueta error si se produce un error. Al final del programa, la sentencia CLOSE finaliza la consulta y termina el acceso del programa a los resultados de la consulta.

```

main()
{
    exec sql include sqlda;
    exec sql begin declare section;
    char nombrerep[16];           /* nombre de vendedor recuperado */
    float cuotarep;              /* cuota recuperada */
    float ventasrep;             /* ventas recuperadas */
    short ind_cuotarep;          /* indicador de cuota nula */
    exec sql end declare section;

    /* Declara el cursor para la consulta */
    exec sql declare cursrep cursor for
        select nombre, cuots, ventas
        from repventas
        where ventas > cuota
        order by nombre;

    /* Prepara el procesamiento de errores */
    whenever sqlerror goto error;
    whenever not found goto hecho;

    /* Abre el cursor para iniciar la consulta */
    exec sql open cursrep;

    /* Recorre con un bucle cada fila de resultados */
    for ( ; ; ) {
        /* Accede a la siguiente fila de resultados */
        exec sql fetch cursrep
        into :nombrerep, :cuotarep, :ventasrep;

        /* Vistaiza los datos recuperados */
        printf("Nombre: %s\n", nombrerep);
        printf("Ventas: %f\n", ventasrep);
        if (ind_cuotarep < 0)
            printf("La cuota es nula (NULL)\n");
        else
            printf("Cuota: %f\n", cuotarep);

        /* Consulta completa; cierra el cursor */
        exec sql close cursrep;
        exit();
    }

    /* Consulta completa; cierra el cursor */
    exec sql close cursrep;
    exit();
}

```

Cursors Como ilustra el programa de la Figura 17.20, un cursor de SQL incorporado se comporta en gran manera como un nombre de fichero o «indizador de archivo» en un lenguaje de programación tal como C o COBOL. Al igual que un programa abre un archivo para acceder a sus contenidos, un cursor se abre para ganar acceso a los resultados de la consulta. Análogamente, el programa cierra un archivo para finalizar su acceso y cierra un cursor para finalizar el acceso a los resultados de la consulta. Finalmente, al igual que un indicador de archivo lleva la cuenta de la posición actual del programa dentro de un archivo abierto, un cursor lleva la cuenta de la posición actual del programa dentro de los resultados de la consulta. Estos paralelismos entre la entrada/salida de archivos y los cursorSQL hacen el concepto de cursor relativamente fácil de entender a los programadores de aplicación.

A pesar de los paralelismos entre archivos y cursor, existen también algunas diferencias. La apertura de un cursor SQL implica generalmente mucho más recargo que la apertura de un archivo, ya que la apertura del cursor provoca realmente que el DBMS comience a efectuar la consulta asociada. Además, los cursorSQL únicamente soporitan el movimiento secuencial hacia adelante a través de los resultados de la consulta, al igual que el procesamiento secuencial de archivos. En la mayoría de las implementaciones SQL actuales, no existen analogías de cursor con el acceso aleatorio proporcionado a los registros individuales de un archivo.

Los cursorSQL proporcionan una gran flexibilidad para el procesamiento de consultas en un programa SQL incorporado. Mediante la declaración y apertura de múltiples cursor, el programa puede procesar varios conjuntos de resultados de consultas en paralelo. Por ejemplo, el programa podría recuperar algunas filas de resultados, visualizarlas en la pantalla para el usuario y luego responder a una petición del usuario de datos más detallados mediante la emisión de una segunda consulta. Las próximas secciones describen con detalle las cuatro sentencias de SQL incorporado que definen y manipulan cursor.

La sentencia DECLARE CURSOR. La sentencia DECLARE CURSOR, mostrada en la Figura 17.21, define una consulta a efectuar. La sentencia también asocia un nombre de cursor con la consulta. El nombre del cursor debe ser un identificador SQL válido. Se utiliza para identificar la consulta y sus resultados en otras sentencias de SQL incorporado. El nombre del cursor no es específicamente una variable del lenguaje anfítrion; se declara mediante la sentencia DECLARE CURSOR, y no en una declaración del lenguaje anfítrion.

La sentencia SELECT en la sentencia DECLARE CURSOR define la consulta asociada con el cursor. La sentencia de selección puede ser cualquier sentencia SELECT de

DECLARE nombre-de-cursor CURSOR FOR sentencia-de-selección →

Figura 17.20. Procesamiento de consulta multípila.

Figura 17.21. Diagrama sintáctico de la sentencia DECLARE CURSOR.

SQL interactivo, como se describió en los Capítulos 6 y 9. En particular, la sentencia SELECT debe incluir una cláusula FROM y puede incluir opcionalmente cláusulas WHERE, GROUP BY, HAVING y ORDER BY. La sentencia SELECT también puede incluir el operador UNION, descrito en el Capítulo 6. Por tanto una consulta de SQL incorporado puede utilizar cualquiera de las capacidades de consultas que están disponibles en el lenguaje SQL interactivo.

La consulta especificada en la sentencia DECLARE CURSOR puede incluir también variables principales de entrada. Estas variables principales realizan exactamente la misma función que en las sentencias incorporadas INSERT, DELETE, UPDATE y SELECT singular. Una variable principal de entrada puede aparecer dentro de la consulta en cualquier lugar en que pueda aparecer una constante. Observe que las variables principales de salida *no pueden* aparecer en la consulta. A diferencia de la sentencia SELECT singular, la sentencia SELECT dentro de la sentencia DECLARE CURSOR no tiene cláusula INTO y no recupera ningún dato. La cláusula INTO aparece como parte de la sentencia FETCH, descrita posteriormente en este capítulo.

Como su nombre implica, la sentencia DECLARE CURSOR es una declaración de cursor. En la mayoría de las implementaciones SQL, incluyendo los productos SQL de IBM, esta sentencia es una directiva para el precompilador SQL; no es una sentencia ejecutable, y el precompilador no produce ningún código para ella. Como todas las declaraciones, la sentencia DECLARE CURSOR debe aparecer físicamente en el programa antes de las sentencias que referencian el cursor que ésta declara. La mayoría de las implementaciones SQL tratan el nombre del cursor como un nombre global que puede ser referenciado dentro de cualquier procedimiento, función o subrutina que aparezca detrás de la sentencia DECLARE CURSOR.

Merece la pena anotar que no todas las implementaciones SQL tratan a la sentencia DECLARE CURSOR estrictamente como una sentencia declarativa, y esto puede conducir a problemas sutiles. Algunos precompiladores SQL generan realmente código para la sentencia DECLARE CURSOR (bien declaraciones del lenguaje anfítrion, o llamadas al DBMS, o ambas), dándole algunas de las cualidades de una sentencia ejecutable. Para estos precompiladores, la sentencia DECLARE CURSOR no solamente debe preceder físicamente a las sentencias OPEN, FETCH y CLOSE que referencian su cursor, sino que a veces debe preceder a estas sentencias en el flujo de ejecución o estar colocada en el mismo bloque que las otras sentencias.

En general se puede evitar problemas con la sentencia DECLARE CURSOR si se siguen estos consejos:

- Coloque la sentencia DECLARE CURSOR justo antes de la sentencia OPEN relativa al cursor. Esta colocación asegura la correcta secuencia física de sentencias pone las sentencias DECLARE CURSOR y OPEN en el mismo bloque y asegura que el flujo de control pasa por la sentencia DECLARE CURSOR, si es necesario. También ayuda a documentar qué consulta está siendo solicitada por la sentencia OPEN.

■ Asegúrese que las sentencias FETCH y CLOSE relativas al cursor siguen a la sentencia OPEN tanto físicamente como en el flujo de control.

La sentencia OPEN. La sentencia OPEN, mostrada en la Figura 17.22, «abre» conceptualmente la tabla de resultados de la consulta para acceso por parte del programa de aplicación. En la práctica, la sentencia OPEN hace realmente que el DBMS procese la consulta, o al menos que comience a procesarla. La sentencia OPEN ocasiona por tanto que el DBMS efectúe el mismo trabajo que una sentencia SELECT interactiva, deteniéndose justo antes del punto en donde produce la primera fila de resultados de la consulta.

El único parámetro de la sentencia OPEN es el nombre del cursor a abrir. Este cursor debe haber sido declarado previamente mediante una sentencia DECLARE CURSOR. Si la consulta asociada con el cursor contiene un error, la sentencia OPEN producirá un valor SQLCODE negativo. La mayoría de los errores de procesamiento de consultas, tales como una referencia a una tabla desconocida, un nombre de columna ambiguo o un intento de recuperar datos de una tabla sin el permiso adecuado, serán reportados como un resultado de la sentencia OPEN. En la práctica, muy pocos errores ocurren durante las sentencias FETCH subsiguientes.

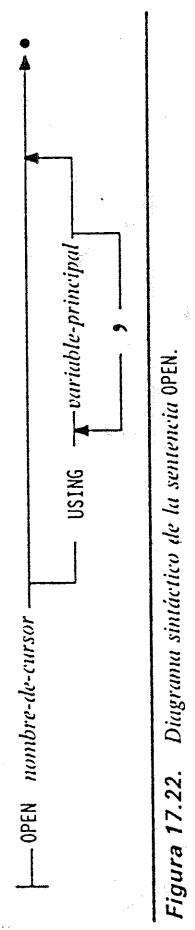


Figura 17.22. Diagrama sintáctico de la sentencia OPEN.

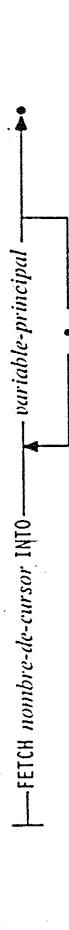


Figura 17.23. Diagrama sintáctico de la sentencia FETCH.

Una vez abierto, un cursor permanece en el estado abierto hasta que sea cerrado con la sentencia CLOSE. El DBMS también cierra todos los cursos abiertos automáticamente al final de una transacción (es decir, cuando el DBMS ejecuta una sentencia COMMIT o ROLLBACK). Después que el cursor ha sido cerrado, puede ser reabierto mediante la ejecución de la sentencia OPEN una segunda vez. Observe que el DBMS reinicia la consulta «desde el principio» cada vez que ejecuta la sentencia OPEN.

La sentencia FETCH. La sentencia FETCH, mostrada en la Figura 17.23, recupera la siguiente fila de resultados de la consulta para su uso por parte del programa de aplicación. El cursor designado en la sentencia FETCH especifica qué fila de resultados va a ser accedida. Debe identificar un cursor previamente abierto mediante la sentencia OPEN.

La sentencia FETCH extrae la fila de elementos de datos sobre lista de variables principales, que están especificadas en la cláusula INTO de la sentencia. Cada variable principal puede ir asociada con una variable indicadora para manejar la recuperación de datos NULL. El comportamiento de la variable indicadora y los valores que puede adoptar son idénticos a los descritos para la sentencia SELECT singular anteriormente en este capítulo. El número de variables principales de la lista debe ser igual al número de columnas de los resultados de la consulta, y los tipos de datos de las variables principales deben ser compatibles, columna a columna, con las columnas de resultados.

Como se muestra en la Figura 17.24, la sentencia FETCH mueve el cursor a través de los resultados de la consulta, fila a fila, de acuerdo con las siguientes reglas:

- La sentencia OPEN posiciona el cursor *delante* de la primera fila de resultados. En este sentido, el cursor no tiene fila actual.
- La sentencia FETCH avanza el cursor a la *siguiente* fila disponible de resultados, si hay alguna. Esta fila se convierte en la fila actual del cursor.
- Si una sentencia FETCH avanza el cursor más allá de la última fila de resultados, la sentencia FETCH devuelve un aviso NOT FOUND. En este estado, el cursor de nuevo no tiene fila actual.
- La sentencia CLOSED finaliza el acceso a los resultados de la consulta y coloca el cursor en un estado cerrado.

Si no hay filas de resultados, la sentencia OPEN sigue posicionando el cursor *delante* de los resultados (no existentes) y vuelve con éxito. El programa no puede detectar que la sentencia OPEN ha producido un conjunto vacío de resultados de consulta. Sin embargo, la primera sentencia FETCH produce el aviso NOT FOUND y posiciona el cursor tras el final de los resultados (no existentes) de la consulta.

La sentencia CLOSE. La sentencia CLOSE, mostrada en la Figura 17.15, «cierra» conceptualmente la tabla de resultados de la consulta creada por la sentencia OPEN, dando por acabado el acceso al programa de aplicación. Su único parámetro es el nombre del cursor asociado con los resultados de la consulta, que debe ser un cursor previamente abierto mediante una sentencia OPEN. La sentencia CLOSE puede ser ejecutada en cualquier instante después que el cursor haya sido abierto. En particular, no es necesario hacer FETCH de todas las filas de resultados antes de cerrar el cursor, aunque éste será generalmente el caso. Todos los

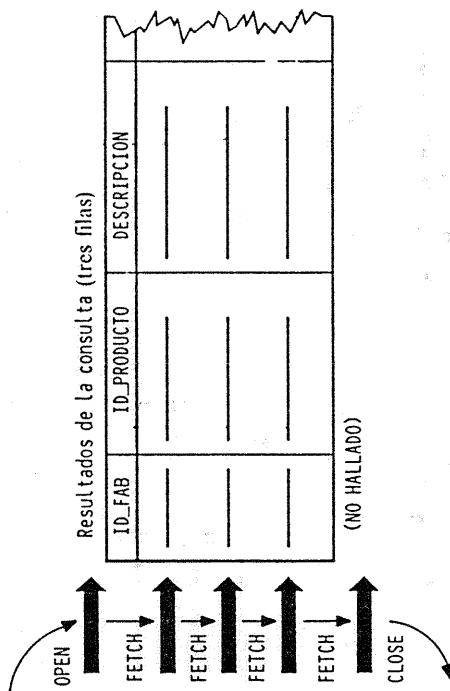


Figura 17.24. Posicionamiento de cursor con OPEN, FETCH y CLOSE.



Figura 17.25. Diagrama sintáctico de la sentencia CLOSE.

cursor es automáticamente cerrado al final de una transacción. Una vez cerrado un cursor, sus resultados de consulta ya no están disponibles al programa de aplicación.

Cursosores con desplazamiento. El estándar SQL ANSI/ISO especifica que un cursor solamente puede moverse hacia adelante a través de los resultados de la consulta. Los productos SQL de IBM y la mayoría de los productos SQL comerciales soportan esta forma de cursor. Si el programa desea volver a recuperar una fila una vez que el cursor ya la ha pasado, el programa debe CERRAR (CLOSE) el cursor y re-ABRIRLO (OPEN) (haciendo que el DBMS efectúe la consulta de nuevo), y luego ACCEDER (FETCH) a las filas hasta que alcance la fila deseada.

Algunos productos SQL comerciales han ampliado el concepto de cursor con el concepto de un *cursor con desplazamiento*. Los cursosres con desplazamiento han sido también propuestos como parte del estándar SQL2 ANSI. A diferencia de los cursosres estándar, un cursor con desplazamiento proporciona acceso aleatorio a las filas de los resultados de la consulta. El programa especifica

ca qué fila desea recuperar una extensión de la sentencia `FETCH`, mostrada en la Figura 17.26:

- `FETCH FIRST` recupera la primera fila de resultados.
- `FETCH LAST` recupera la última fila de resultados.

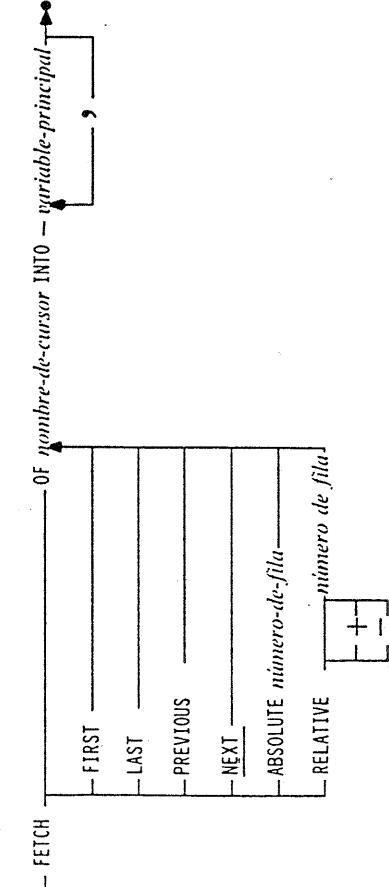


Figura 17.26. Sentencia `FETCH` extendida para cursor con desplazamiento.

- `FETCH PREVIOUS` recupera la fila de resultados que precede inmediatamente a la fila actual del cursor.

- `FETCH NEXT` recupera la fila de resultados que sigue inmediatamente a la fila actual del cursor. Este es el comportamiento por omisión si no se especifica movimiento y corresponde al movimiento estándar del cursor.

- `FETCH ABSOLUTE` recupera una fila específica mediante su número de fila.
- `FETCH RELATIVE` mueve el cursor hacia adelante o hacia atrás un número específico de filas relativo a su posición actual.

Los cursosres con desplazamiento pueden ser especialmente útiles en programas que permiten a un usuario inspeccionar los contenidos de la base de datos. En respuesta a la petición del usuario para mover hacia adelante o hacia atrás a través de los datos una fila o una pantalla cada vez, el programa puede simplemente acceder a las filas requeridas de los resultados de consulta.

Supresiones y actualizaciones basadas en cursor

Los programas de aplicación utilizan con frecuencia cursosres para permitir al usuario inspeccionar una tabla de datos fila a fila. Por ejemplo, el usuario puede pedir ver todos los pedidos remitidos por un cliente particular. El programa declara un cursor para una consulta de la tabla `PEDIDOS` y visualiza cada pedido en la pantalla, posiblemente en un formulario generado por el computador, a la espera de una señal del usuario para avanzar a la fila siguiente. La inspección continua de esta manera hasta que el usuario alcanza el final de los resultados. El cursor sirve como un puntero a la fila actual de resultados de la consulta. Si la consulta extrae sus datos de una única tabla, y no es una consulta sumario, como en este ejemplo, el cursor apunta implícitamente a una fila de una tabla de la base de datos, ya que cada fila de resultados es extraída de una única fila de la tabla.

Mientras inspecciona los datos, el usuario puede apuntar a datos que deberían ser modificados. Por ejemplo, la cantidad pedida de uno de los pedidos puede ser incorrecta, o el cliente puede desear suprimir uno de los pedidos. En esta situación, el usuario desea actualizar o suprimir «este» pedido. La fila no está identificada por la usual condición de búsqueda de SQL; en su lugar, el programa utiliza el cursor como puntero para indicar qué fila particular va a ser actualizada o suprimida.

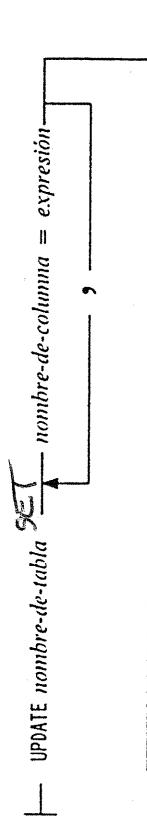
`— DELETE FROM nombre-de-tabla WHERE CURRENT OF nombre-de-cursor —►`

Figura 17.27. Diagrama sintáctico de la sentencia `DELETE` posicionada.

SQL incorporado soporta esta capacidad mediante versiones especiales de las sentencias `DELETE` y `UPDATE`, denominadas sentencias `DELETE posicionada` y `UPDATE posicionada`, respectivamente.

La sentencia `DELETE` posicionada, mostrada en la Figura 17.27, suprime una única fila de una tabla. La fila suprimida es la fila actual de un cursor que referencia la tabla. Para procesar la sentencia, el DBMS localiza la fila de la tabla base que corresponde a la fila actual del cursor y suprime esa fila de la tabla base. Después que la fila se ha suprimido, el cursor no tiene fila actual. En vez de ello, el cursor está efectivamente posicionado en el «espacio vacío» dejado por la fila suprimida, esperando ser avanzado a la siguiente fila por una sentencia `FETCH` posterior.

La sentencia `UPDATE` posicionada, mostrada en la Figura 17.28, actualiza una única fila de una tabla. La fila actualizada es la fila actual de un cursor que

**Figura 17.28.** Diagrama sintáctico de la sentencia UPDATE posicionada.

referencia a la tabla. Para procesar la sentencia, el DBMS localiza la fila de la tabla base que corresponde a la fila actual del cursor y actualiza esa fila según se especifique en la cláusula SET. Después que la fila se ha actualizado, el cursor permanece en la fila actual. La Figura 17.29 muestra un programa de inspección de pedidos que utiliza las sentencias UPDATE y DELETE posicionadas:

1. El programa solicita primero al usuario un número de cliente y luego consulta la tabla de PEDIDOS para localizar todos los pedidos remitidos por ese cliente.
2. Conforme recupera cada fila de resultados de la consulta, visualiza la información de pedido en la pantalla y pregunta al usuario qué hacer a continuación. que pasa directamente al siguiente pedido.
3. Si el usuario escribe una «S», el programa no modifica el pedido actual, sino que pasa directamente al siguiente pedido.
4. Si el usuario escribe una «E», el programa elimina (suprime) el pedido actual utilizando una sentencia DELETE posicionada.
5. Si el usuario escribe una «A», el programa solicita del usuario una nueva cantidad e importe y luego actualiza estas dos columnas del pedido actual utilizando una sentencia UPDATE posicionada.
6. Si el usuario escribe una «F», el programa detiene la consulta y finaliza.

Aunque rudimentario comparado con un programa de aplicación real, el ejemplo de la Figura 17.29 muestra toda la lógica y sentencias de SQL incorporado requeridas para implementar una aplicación de inspección con actualizaciones de la base de datos basadas en cursor.

El estándar SQL ANSI/ISO especifica que las sentencias DELETE y UPDATE posicionadas solamente pueden ser utilizadas con cursos que satisfacen estos criterios muy estrictos:

- La consulta asociada con el cursor debe extraer sus datos de una única tabla

```

main()
{
    exec sql include sqlc;
    exec sql begin declare section;
    int numclie; /* nombre de cliente introducido por el usuario */
    int numped; /* número de pedido recuperado */
    char fechaped[12]; /* fecha de pedido recuperada */
    char fabped[]; /* fabricante recuperado */
    char productoped[6]; /* id.producto recuperado */
    int cantped; /* cantidad de pedido recuperada */
    float importeped; /* importe de pedido recuperado */

    exec sql end declare section;

    /* Declara el cursor para la consulta */
    exec sql declare cursped cursor for
        select num_ped, fecha_ped, fab, producto, cant, importe
        from pedidos
        where clie = num_clie
        order by num_pedido
        for update of cant, importe;

    /* Pide al usuario que escriba un número de cliente */
    printf("Introduzca un número de cliente: ");
    scanf("%d", &numclie);

    /* Prepara el procesamiento de errores */
    whenever sqlerror goto error;
    whenever not found goto hecho;

    /* Abre el cursor para iniciar la consulta */
    exec sql open cursped;

    /* Recorre con un bucle cada fila de resultados */
    for ( ; ; ) {
        /* Accede a la siguiente fila de resultados */
        exec sql fetch cursped
            into numped, fechaped, :productoped,
            :cantped, :importeped;

        /* Visualiza los datos recuperados */
        printf("Número de pedido: %d\n", numped);
        printf("Fecha de pedido: %s\n", fechaped);
        printf("Fabricante: %s\n", fabped);
    }
}

```

Figura 17.29. Utilización de las sentencias DELETE y UPDATE posicionadas.

fuente; es decir, solamente debe haber una tabla designada en la cláusula FROM de la consulta especificada en la sentencia DECLARE CURSOR.

- La consulta no puede especificar una cláusula ORDER BY; el cursor no debe identificar un conjunto ordenado de resultados de la consulta.

- La consulta no puede especificar la palabra clave DISTINCT.
- La consulta no debe incluir una cláusula GROUP BY o una cláusula HAVING.
- El usuario debe tener el privilegio UPDATE o DELETE (según sea adecuado) sobre la tabla base.

```

printf("Producto: %s\n", productoped);
printf("Cantidad: %d\n", cantped);
printf("Importe total: %f\n", importeped);

/* Pide al usuario que elija una acción para este pedido */
printf("Elige una acción (Siguiente/Eliminar/Actualizar/fin): ");
getchar();
getchar();

switch (inbuf[0]) {
    case 'S':
        /* Continúa con el siguiente pedido */
        break;
    case 'E':
        /* Elimina el pedido actual */
        exec sql delete from pedidos
            where current of cursped;
        break;
    case 'A':
        /* Actualiza la orden actual */
        printf("Introduce nueva cantidad: ");
        scanf("%d", &cantped);
        printf("Introduce nuevo importe: ");
        scanf("%f", &importeped);
        exec sql update pedidos
            set cant = cantped, importe = :importeped
            where current of cursped;
        break;
    case 'F':
        /* Deja de recuperar pedidos y finaliza */
        goto hecho;
}

exec sql close cursped;
exec sql commit;
exit();
}

error:
printf("SQL error: %ld\n", sqlca.sqlcode);
exit();
}

```

Las bases de datos de IBM (DB2, SQL/DS y OS/2 Extended Edition) amplian las restricciones ANSI/ISO en un punto más. Requieren que el cursor sea explícitamente declarado como cursor actualizable en la sentencia DECLARE CURSOR. La forma ampliada de IBM de la sentencia DECLARE CURSOR se muestra en la Figura 17.30. Además de declarar un cursor actualizable, la cláusula FOR UPDATE puede especificar opcionalmente columnas particulares que puedan ser actualizadas mediante el cursor. Si la lista de columnas está especificada en las declaraciones del cursor, las sentencias UPDATE posicionadas para el cursor solamente pueden actualizar esas columnas.

En la práctica, todas las implementaciones SQL comerciales que soportan sentencias DELETE y UPDATE posicionadas siguen el enfoque SQL de IBM. Es una gran ventaja para el DBMS conocer de antemano si un cursor va a ser utilizado para actualizaciones o si sus datos sólo serán leídos, ya que el procesamiento de sólo lectura es más sencillo. La cláusula FOR UPDATE proporciona este aviso por adelantado y puede ser considerado un estándar de hecho del lenguaje SQL incorporado.

Cursores y procesamiento de transacciones

El modo en que el programa maneja sus cursosres puede tener un impacto importante sobre el rendimiento de la base de datos. Recuerde del Capítulo 12 que el modelo de transacción SQL garantiza la consistencia de los datos durante una transacción. En términos de cursor, esto significa que el programa puede declarar un cursor, abrirlo, acceder a los resultados de la consulta, cerrarlo, volverlo a abrir y acceder a los resultados de la consulta de nuevo —y tener garantizado que los resultados de la consulta serán idénticos las dos veces—. El programa también puede acceder a la misma fila a través de dos cursosres diferentes y tiene garantizado que los resultados serán idénticos. De hecho, los datos se garantiza que permanecen consistentes hasta que el programa emite un COMMIT o un ROLLBACK para finalizar la transacción. Debido a que la consistencia no está garantizada entre transacciones, tanto las sentencias COMMIT como ROLLBACK cierran automáticamente todos los cursosres abiertos.

Internamente, el DBMS proporciona esta garantía de consistencia mediante el cerramiento de todas las filas de los resultados, impidiendo que otros usuarios puedan modificarlos. Si la consulta produce muchas filas de datos, una parte

Figura 17.29. Utilización de las sentencias DELETE y UPDATE posiciones (continuación).

donde las sentencias de SQL están incorporadas al programa de aplicación, entremezcladas con las sentencias de un lenguaje de programación anfitrión tal como C o COBOL.

- Las sentencias de SQL incorporado son procesadas mediante un precompilador SQL especial. Las sentencias comienzan con un introductor especial (generalmente EXEC SQL) y finalizan con un terminador, que varía de un lenguaje anfitrión a otro.

- Las variables del programa de aplicación, denominadas variables principales, pueden ser utilizadas en sentencias de SQL incorporado en cualquier lugar en que pueda aparecer una constante. Estas variables principales de entrada acomodan la sentencia de SQL incorporado a la situación particular.
- Las variables principales también se utilizan para recibir resultados de consultas. Los valores de estas variables principales de salida pueden luego ser procesadas por el programa de aplicación.

- Las consultas que producen una sola fila de datos son manejadas por la sentencia SELECT singular de SQL incorporado, la cual especifica tanto la consulta como las variables principales que reciben los datos recuperados.
- Las consultas que producen múltiples filas de resultados son manejadas mediante cursos en SQL incorporado. La sentencia DECLARE CURSOR define la consulta, la sentencia OPEN comienza el procesamiento de la consulta, la sentencia CLOSE finaliza el procesamiento de la consulta.
- Las sentencias UPDATE y DELETE posicionadas pueden ser utilizadas para actualizar o suprimir la fila actualmente seleccionada por un cursor.

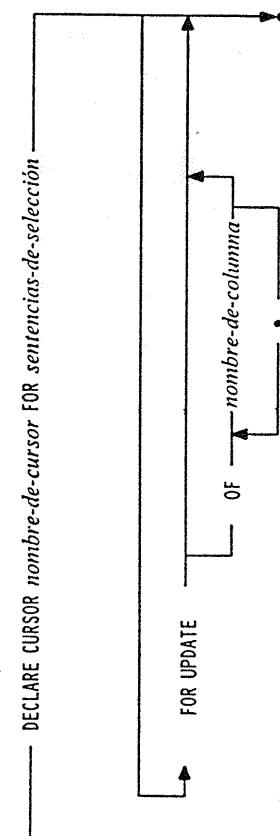


Figura 17.30. Sentencias DECLARACIÓN CURSOR con cláusula FOR UPDATE.

importante de una tabla puede estar cerrada por el cursor. Además, si el programa espera entrada de usuario después de acceder a cada fila (por ejemplo, para permitir al usuario verificar los datos visualizados en la pantalla), partes de la base de datos pueden estar cerradas durante mucho tiempo. En un caso extremo, el usuario podría salir a cenar en medio de una transacción, cerrando el paso a otros usuarios durante una hora o más.

Para minimizar la cantidad de cerramiento requerido, deberían seguirse estos consejos cuando se escriban programas de consulta interactiva:

- Mantener las transacciones tan breves como sea posible.
- Emitir una sentencia COMMIT inmediatamente después de cada consulta y tan pronto como sea posible después que el programa haya completado una actualización.
- Evitar programas que requieran una gran cantidad de interacción de usuario o que inspeccionen muchas filas de datos.
- Si usted sabe que el programa no tratará de volver a acceder a una fila de datos después que el cursor la haya pasado, replantee el plan de aplicación utilizando el modo estabilidad de cursor descrito en el Capítulo 12. Esto permite al DBMS desbloquear una fila tan pronto como se emita la siguiente sentencia FETCH.

Resumen

Además de su papel como lenguaje de base de datos interactivo, SQL se utiliza para acceso programado a bases de datos relacionales:

- La técnica más común para el uso programado de SQL es SQL incorporado,

18 *SQL dinámico**

Las características de programación de SQL incorporado descritas en el capítulo precedente son colectivamente conocidas como *SQL estático*. SQL estático es adecuado para escribir todos los programas típicamente requeridos en una aplicación de procesamiento de datos. Por ejemplo, en la aplicación de procesamiento de pedidos de la base de datos ejemplo, se puede utilizar SQL estático para escribir programas que manejen la entrada de pedidos, las actualizaciones de pedido, las indagaciones sobre pedidos, las indagaciones sobre clientes y los programas que producen todo tipo de informes. En cada uno de estos programas, el patrón de acceso a la base de datos es decidido por el programador y «codificado de modo fijo» en el programa como una serie de sentencias de SQL incorporado.

Existe una importante clase de aplicaciones, sin embargo, en donde el patrón de acceso a la base de datos no puede determinarse de antemano. Una herramienta de consulta gráfica o un escritor de informes, por ejemplo, debe ser capaz de decidir en tiempo de ejecución qué sentencias SQL utilizará para acceder a la base de datos. Una hoja de cálculo de computador personal que soporta acceso a una base de datos anfitrión debe también ser capaz de enviar una consulta al DBMS anfitrión para su ejecución «sobre la marcha». Estos programas y otros frontales de bases de datos de propósito general no pueden ser escritos utilizando técnicas de SQL estático. Requieren una forma avanzada de SQL incorporado, denominada *SQL dinámico*, descrito en este capítulo.

Limitaciones de SQL estático

Como el nombre «SQL estático» implica, un programa construido utilizando las características de SQL incorporado descritas en el Capítulo 17 (variables princi-

pales, cursores y las sentencias DECLARE CURSOR, OPEN, FETCH y CLOSE) tienen un patrón fijo, predeterminado, de acceso a la base de datos. Para cada sentencia de SQL incorporada al programa, las tablas y columnas referenciadas por esa sentencia están determinadas de antemano por el programador y codificadas de modo inalterable en la sentencia de SQL incorporado. Las variables principales de entrada proporcionan cierta flexibilidad en SQL estático, pero no alteran fundamentalmente su naturaleza estática. Recuerde que una variable principal puede aparecer en cualquier lugar en donde está permitida una constante en una sentencia SQL. Usted puede utilizar una variable principal para alterar una condición de búsqueda:

```
exec sql select nombre, cuota, ventas
  from repventas
 where cuota > importe_límite;
```

También puede utilizar una variable principal para cambiar los datos insertados o actualizados en una base de datos:

```
exec sql update repventas
  set cuota = cuota + :aumento
 where cuota > :importe_límite;
```

Como ilustra esta discusión si un programa debe ser capaz de determinar en tiempo de ejecución qué sentencias SQL va a utilizar, o qué tablas y columnas va a referenciar, SQL estático es inadecuado para la tarea. SQL dinámico solventa estas limitaciones.

SQL dinámico ha sido soportado por los productos SQL de IBM desde su introducción, y está soportado por Oracle, Ingres, VAX SQL, Informix y muchos otros productos DBMS. Sin embargo, SQL dinámico no está especificado en el estándar SQL ANSI/ISO; el estándar define únicamente SQL estático. La ausencia de SQL dinámico en el estándar es irónica, dada la noción popular de que SQL ANSI permite construir herramientas frontales de base de datos que son portables a través de muchos productos DBMS diferentes. De hecho, tales herramientas frontales deben ser construidas utilizando SQL dinámico.

En ausencia de un estándar ANSI/ISO, DB2 es el estándar de hecho para SQL dinámico. Las otras bases de datos de IBM (SQL/DS y OS/2 Extended Edition) son casi idénticas a DB2 en su soporte de SQL dinámico, y la mayoría de los restantes productos SQL también siguen el estándar DB2.

Conceptos de SQL dinámico

El concepto central de SQL dinámico es sencillo: no codificar de forma fija una sentencia de SQL incorporado en el código fuente del programa. En vez de ello, permitir que el programa construya el texto de una sentencia SQL en una de sus áreas de datos en tiempo de ejecución y luego traspasar el texto de la sentencia al DBMS para su ejecución «sobre la marcha». Aunque los detalles son bastante complejos, todo SQL dinámico se construye sobre este sencillo concepto, y es buena idea tenerlo presente.

Para comprender SQL dinámico y cómo se compara con SQL estático, es útil considerar una vez más el proceso que el DBMS sigue para ejecutar una sentencia SQL, originalmente mostrado en la Figura 17.1 y repetido aquí en la Figura 18.1. Recuerde del Capítulo 17 que una sentencia de SQL estático recorre los cuatro primeros pasos del proceso en tiempo de compilación. La utilidad BIND almacena el plan de aplicación para la sentencia en la base de datos como parte del proceso de desarrollo del programa. Cuando la sentencia de SQL estático se ejecuta en tiempo de ejecución, el DBMS simplemente ejecuta el plan de aplicación almacenado.

En SQL dinámico, la situación es bastante diferente. La sentencia SQL a ejecutar no es conocida hasta el momento de la ejecución, por lo que el DBMS no puede preparar la sentencia con antelación. Cuando el programa es ejecutado efectivamente, el DBMS recibe el texto de la sentencia a ejecutar dinámicamente (denominado *cadena de sentencia*) y recorre los cinco pasos mostrados en la Figura 18.1 en tiempo de ejecución.

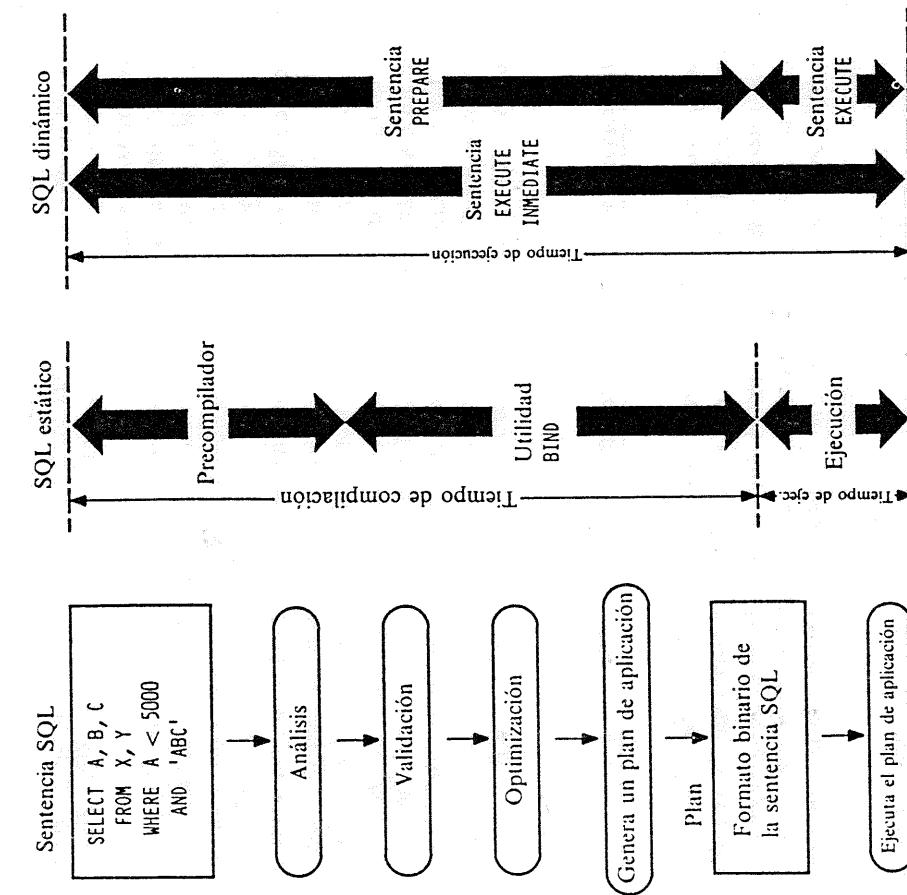


Figura 18.1. Cómo procesa el DBMS una sentencia SQL.

Como cabría esperar, SQL dinámico es menos eficiente que SQL estático. Por esta razón, SQL estático se utiliza siempre que es posible, y la mayoría de los programadores de aplicación nunca necesitan aprender nada acerca de SQL dinámico. Sin embargo, SQL dinámico está creciendo en importancia conforme el acceso a las bases de datos se orienta sobre una arquitectura cliente/servidor, frontal/supporte. El acceso a base de datos está comenzando a aparecer en aplicaciones de computadores personales tales como hojas de cálculo, procesadores de texto y programas de gráficos, y las herramientas frontales de acceso a datos son una de las áreas más activas en el desarrollo de base de datos. Todas estas aplicaciones requieren las características de SQL dinámico.

EXECUTE IMMEDIATE variable-principal

Figura 18.2. Diagrama sintáctico de la sentencia EXECUTE IMMEDIATE.

Ejecución dinámica de sentencia (EXECUTE IMMEDIATE)

La forma más sencilla de SQL dinámico es la proporcionada por la sentencia EXECUTE IMMEDIATE, mostrada en la Figura 18.2. Esta sentencia transmite el texto de una sentencia de SQL dinámico al DBMS y pide al DBMS que ejecute la sentencia dinámica inmediatamente. Para utilizar esta sentencia, el programa recorre los siguientes pasos:

1. El programa construye una sentencia SQL como una cadena de texto en una de sus áreas de datos (generalmente denominada un *buffer*). La sentencia puede ser casi cualquier sentencia SQL que no recupere datos.
2. El programa transfiere la sentencia SQL al DBMS junto con la sentencia EXECUTE IMMEDIATE.
3. El DBMS ejecuta la sentencia y fija SQLCODE para que indique el estado de terminación, exactamente igual que si la sentencia hubiera estado codificada de forma fija utilizando SQL estático.

La Figura 18.3 muestra un sencillo programa C que sigue estos pasos. El programa pide al usuario un nombre de tabla y una condición de búsqueda SQL, y construye el texto de una sentencia DELETE basada en las respuestas del usuario. El programa utiliza la sentencia EXECUTE IMMEDIATE para ejecutar la sentencia DELETE. Este programa no puede utilizar una sentencia DELETE incorporada de SQL estático, ya que ni el nombre de la tabla ni la condición de búsqueda son conocidos antes que el usuario los introduzca en tiempo de ejecución. Debe utilizarse SQL dinámico. Si se ejecuta el programa de la Figura 18.3 con estas entradas:

```
Introduzca un nombre de tabla: personal
Introduzca una condición de búsqueda: cuota < 20000
Exit en la supresión de personal.
```

el programa transfiere este texto de sentencia al DBMS:

```
delete from personal
where cuota < 20000
```

Si ejecuta el programa con estas entradas:

```
Introduzca un nombre de tabla: pedidos
Introduzca una condición de búsqueda: clie = 2105
Exit en la supresión de pedidos.

main()
{
    /* Este programa suprime filas de una tabla especificada por el
    usuario según una condición de búsqueda también especificada
    por el usuario.

    */

    exec sql include sqlca;
    exec sql begin declare section;
    char bufsent[301];
    exec sql end declare section;

    char nombretbl[101];
    strcpy(bufsent, "delete from ");
    char condbusca[101];
    /* Comienza a construir la sentencia DELETE en bufsent */
    /* Pide al usuario que escriba un nombre de tabla; lo añade al texto de la sentencia DELETE */
    printf("Introduzca un nombre de tabla: ");
    gets(nombretbl);
    gets(condbusca);
    strcat(bufsent, nombretbl);
    strcat(bufsent, condbusca);

    /* Pide al usuario que escriba una condición de búsqueda; la añade al texto */
    printf("Introduzca una condición de búsqueda: ");
    gets(cond_busca);
    if (strlen(cond_busca) > 0) {
        strcat(bufsent, " where ");
        strcat(bufsent, cond_busca);
    }

    /* Ahora manda al DBMS que ejecute la sentencia */
    exec sql execute immediate :bufsent;
    if (sqlca.sqlcode < 0)
        printf("SQL error: %ld\n", sqlca.sqlcode);
    else
        printf("Exit en la supresión %s.\n", nombretbl);
    exit();
}
```

el programa transfiere este texto de sentencia al DBMS:

```
delete from pedidos
where clie = 2105;
```

La sentencia EXECUTE IMMEDIATE proporciona por tanto al programa gran flexibilidad en el tipo de sentencia DELETE que ejecuta.

La sentencia EXECUTE IMMEDIATE utiliza exactamente una variable principal —la variable que contiene la cadena de la sentencia SQL entera—. La propia cadena de sentencia no puede incluir referencias a variables principales, pero tampoco hay necesidad de ello. En vez de utilizar una sentencia de SQL estático con una variable principal como ésta:

```
exec sql delete from pedidos
    where clie = :num_clie;
```

un programa de SQL dinámico logra el mismo efecto construyendo la sentencia entera en un buffer y ejecutándola:

```
spint(buffer, 'delete from pedidos where clie = %d', num_clie);
exec sql execute immediate :buffer;
```

La sentencia EXECUTE IMMEDIATE es la forma más sencilla de SQL dinámico, pero es muy versátil. Puede utilizarse para ejecutar dinámicamente la mayoría de las sentencias DML, incluyendo INSERT, UPDATE, COMMIT y ROLLBACK. También puede utilizarse EXECUTE IMMEDIATE para ejecutar dinámicamente la mayoría de las sentencias DDL, incluyendo las sentencias CREATE, DROP, GRANT y REVOKE.

Sin embargo, la sentencia EXECUTE IMMEDIATE tiene una limitación significativa. No puede utilizarse para ejecutar dinámicamente una sentencia SELECT, ya que no proporciona un mecanismo para procesar resultados de consultas. Al igual que SQL estático requiere cursor y sentencias de propósito especial (DECLARE CURSOR, OPEN, FETCH y CLOSE) para consultas programadas, SQL dinámico utiliza cursor y algunas nuevas sentencias de propósito especial para manejar consultas dinámicas. Las características de SQL dinámico que soportan consultas dinámicas se discuten posteriormente en este capítulo.

Ejecución dinámica en dos pasos

La sentencia EXECUTE IMMEDIATE proporciona un soporte en un paso para la ejecución dinámica de sentencias. Como se describió anteriormente, el DBMS recorre los cinco pasos de la Figura 18.1 para ejecutar dinámicamente la senten-

Figura 18.3. Utilización de la sentencia EXECUTE IMMEDIATE.

cia. El recargo de este proceso puede ser significativo si el programa ejecuta muchas sentencias dinámicas, y es un derroche si las sentencias a ejecutar son muy similares.

Para tratar esta situación, SQL dinámico ofrece un método alternativo en dos pasos para ejecutar las sentencias SQL dinámicamente. He aquí una visión general de la técnica de los dos pasos:

1. El programa construye una cadena de sentencia SQL en un buffer, lo mismo que hace para la sentencia EXECUTE IMMEDIATE. Un signo de interrogación (?) puede ser el sustituto de una constante en cualquier lugar del texto de la sentencia para indicar que el valor de la constante será suministrado más tarde. La interrogación se denomina *marcador de parámetro*.
2. La sentencia PREPARE pide al DBMS que analice, valide y optimice la sentencia y que genere un plan de aplicación para ella. El DBMS fija el SQLCODE para que indique los errores encontrados en la sentencia y retiene el plan de aplicación para su ejecución posterior. Observe que el DBMS *no* ejecuta el plan en respuesta a la sentencia PREPARE.
3. Cuando el programa desea ejecutar la sentencia previamente preparada, utiliza la sentencia EXECUTE y transfiere al DBMS un valor por cada marcador de parámetro. El DBMS sustituye los valores de los parámetros, ejecuta el plan de aplicación previamente generado y pone SQLCODE para que indique el estado de terminación.
4. El programa puede utilizar la sentencia EXECUTE repetidamente, suministrando diferentes valores de parámetros cada vez que la sentencia dinámica se ejecute.

La Figura 18.4 muestra un programa C que utiliza estos pasos. El programa es un programa de actualización de tablas de propósito general. Solicita al usuario un nombre de tabla y dos nombres de columna y construye una sentencia UPDATE para la tabla que tiene este aspecto:

```
update nombre-de-tabla
set segundo-nombre-de-columna = ?
where primer-nombre-de-columna = ?
```

La entrada del usuario determina por tanto la tabla y columna a actualizar y la condición de búsqueda utilizada. El valor de la comparación de búsqueda y el valor del dato actualizado son especificados como parámetros, a suministrar posteriormente cuando la sentencia UPDATE sea ejecutada realmente.

Tras construir el texto de la sentencia UPDATE en su buffer, el programa pide al DBMS que la compile con la sentencia PREPARE. El programa entra entonces en un bucle, solicitando al usuario que introduzca pares de valores parámetro para efectuar una secuencia de actualizaciones de tablas. El diálogo con el usuario

```
main()
{
    /* Este es un programa de actualización de propósito general. Puede ser
       utilizado para cualquier actualización en donde una columna numérica
       vaya a ser actualizada en todas las filas en las que la segunda columna
       numérica tenga un valor especificado. Por ejemplo, puede utilizarse para
       actualizar las cuotas de vendedores seleccionados o para actualizar los
       límites de créditos de clientes seleccionados. */

    include sqlca;
    begin declare section;
    exec sql
    begin declare section;
    char buf$[30];
    char valor_busca;
    float valor_nuevo_valor;
    exec sql end declare section;
    /* tabla a actualizar */
    /* nombre de la columna de búsqueda */
    /* texto SQL a ejecutar */
    /* valor de parámetro para búsqueda */
    /* valor de parámetro para actualización */
    /* valor de parámetro para actualización */

    /* tabla a actualizar */
    /* nombre de la columna de actualización */
    /* nombre de la columna de actualización */
    /* respuesta si/no del usuario */

    /* Pide al usuario que escriba nombres de tabla y columnas */
    printf("Introduzca el nombre de la tabla a actualizar: ");
    gets(nombretbl);
    printf("Introduzca el nombre de la columna a buscar: ");
    gets(colbusca);
    printf("Introduzca el nombre de la columna a actualizar: ");
    gets(colactual);

    /* Construye la sentencia SQL en un buffer; manda al DBMS que la compile */
    sprintf(buf$, "update %s set %s = ? where %s = ?;", nombretbl, colbusca, colactual);
    /* Introduzca un nuevo valor para %s: */
    /* Introduzca un nuevo valor para %s: */
    /* respuesta error: %d\n", sql.ca.sqlcode); */

    /* Itera pidiendo parámetros al usuario y efectuando actualizaciones */
    for ( ; ; )
    {
        printf("\nIntroduzca un valor de búsqueda para %s: ", colbusca);
        scanf("%f", &valor_busca);
        printf("Introduzca un nuevo valor para %s: ", colactual);
        scanf("%f", &nuevo_valor);
    }
}
```

Figura 18.4. Utilización de las sentencias PREPARE y EXECUTE

ejecución dinámica en dos pasos de PREPARE y EXECUTE ayuda a eliminar algunas de las desventajas de rendimiento de SQL dinámico.

La sentencia PREPARE

```
/* Manda al DBMS que ejecute la sentencia UPDATE */
execute misent using :valor_busca, :nuevo_valor;
③
if (sqlca.sqlcode) {
    printf("EXECUTE error: %ld\n", sqlca.sqlcode);
    exit();
}

/* Pregunta al usuario si hay otra actualización */
printf("Otra (s/n)? ");
④
gets (si_no);
if (si_no[0] == 'n')
    break;
}

printf("\nActualizaciones completadas.\n");
exit();
}
```

Figura 18.4. Utilización de las sentencias PREPARE y EXECUTE (continuación).

muestra cómo se podría utilizar el programa de la Figura 18.4 para actualizar las cuotas de vendedores seleccionados:

```
Introduzca el nombre de la tabla a actualizar: personal
Introduzca el nombre de la columna a buscar: num_empl
Introduzca el nombre de la columna a actualizar: cuota
Introduzca un valor de búsqueda para num_empl: 106
Introduzca un nuevo valor para cuota: 150000.00
Otro (s/n)? s
Introduzca un valor de búsqueda para num_empl: 107
Introduzca un nuevo valor para cuota: 215000.00
Otro (s/n)? n
Actualizaciones completadas.
```

Este programa es un excelente ejemplo de una situación en donde la ejecución dinámica en dos pasos es adecuada. El DBMS compila la sentencia UPDATE dinámica únicamente una vez, pero la ejecuta tres veces, una vez por cada conjunto de valores parámetros introducidos por el usuario. Si el programa hubiera sido escrito utilizando EXECUTE IMMEDIATE, la sentencia UPDATE dinámica tendría que haber sido compilada tres veces y ejecutada tres veces. Por tanto la

ejecución dinámica en dos pasos de PREPARE y EXECUTE ayuda a eliminar algunas de las desventajas de rendimiento de SQL dinámico.

La sentencia PREPARE, mostrada en la Figura 18.5, es propia de SQL dinámico. Acepta una variable principal que contiene una cadena de sentencia SQL y transfiere la sentencia al DBMS. El DBMS compila el texto de la sentencia y la prepara para ejecución mediante la generación de un plan de aplicación. El DBMS fija la variable SQLCODE para que indique los errores detectados en el texto de la sentencia. Como describimos anteriormente, la cadena de sentencia puede contener un marcador de parámetro, indicado mediante un signo de interrogación, en cualquier lugar en donde pueda aparecer una constante. El marcador de parámetro señala al DBMS que el valor del parámetro será suministrado más tarde, cuando la sentencia sea realmente ejecutada.

Como resultado de la sentencia PREPARE, el DBMS asigna el *nombre de sentencia* especificado a la sentencia preparada. El nombre de sentencia es un identificador SQL, igual que un nombre de cursor. El nombre de sentencia se especifica en las sentencias EXECUTE subsiguientes cuando se desea ejecutar la sentencia. El DBMS retiene la sentencia preparada y el nombre de sentencia asociado hasta el final de la transacción actual (es decir, hasta la siguiente sentencia COMMIT o ROLLBACK). Si se desea ejecutar la misma sentencia dinámica posteriormente durante otra transacción, hay que prepararla de nuevo. La sentencia PREPARE puede ser utilizada para preparar casi cualquier sentencia DML o DDL ejecutable, incluyendo la sentencia SELECT. Las sentencias de SQL que son directivas de precompilador (tales como las sentencias WHENEVER o DECLARE CURSOR) no pueden ser preparadas, naturalmente, ya que no son ejecutables.



Figura 18.5. Diagrama sintáctico de la sentencia PREPARE.

La sentencia EXECUTE

La sentencia EXECUTE, mostrada en la Figura 18.6, es propia de SQL dinámico. Pide al DBMS que ejecute una sentencia previamente preparada con la sentencia PREPARE. Puede ejecutar cualquier sentencia que pueda ser preparada, con

una excepción. Al igual que la sentencia EXECUTE IMMEDIATE, la sentencia EXECUTE no puede ser utilizada para ejecutar una sentencia SELECT, ya que carece de un mecanismo para manejar resultados de consultas.

Si una sentencia dinámica que va a ser ejecutada contiene uno o más marcadores de parámetros, la sentencia EXECUTE debe proporcionar un valor a cada uno de los parámetros. Los valores pueden ser suministrados de dos maneras diferentes, descritas en las dos secciones siguientes.

EXECUTE con variables principales. El modo más fácil de transmitir valores de parámetros a la sentencia EXECUTE es mediante la especificación de una lista de variables principales en la cláusula USING. La sentencia EXECUTE sustituye los valores de las variables principales, en secuencia, por los marcadores de parámetros en el texto de la sentencia preparada. Las variables principales sirven por tanto como variables principales de entrada para la sentencia ejecutada dinámicamente. Esta técnica fue utilizada en el programa mostrado en la Figura 18.4.

El número de variables principales de la cláusula USING debe coincidir con el número de marcadores de parámetros en la sentencia dinámica, y el tipo de datos de cada variable principal debe ser compatible con el tipo de dato requerido por el parámetro correspondiente. Cada variable principal de la lista puede tener también una variable indicadora asociada. Si la variable indicadora contiene un valor negativo cuando la sentencia EXECUTE se procesa, al marcador de parámetro correspondiente se le asigna el valor NULL.

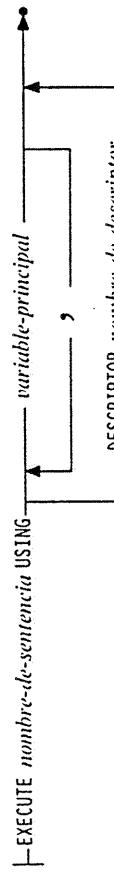


Figura 18.6. Diagrama sintáctico de la sentencia EXECUTE.

EXECUTE con SQLDA. La segunda manera de transmitir parámetros a la sentencia EXECUTE es con una estructura de datos especial de SQL dinámico denominada *área de datos de SQL*, o SQLDA (*SQL Data Area*). Debe utilizarse un SQLDA para transferir parámetros cuando no se conozca el número de parámetros a transferir y sus tipos de datos en el momento en que se escribe el programa. Por ejemplo, supongamos que se desease modificar el programa de actualización de propósito general de la Figura 18.4 de modo que el usuario pudiera seleccionar más de una columna a actualizar. Se podría fácilmente modificar el programa para generar una sentencia UPDATE con un número de variables de asignaciones, pero la lista de variables principales en la sentencia EXECUTE presenta un proble-

```

struct sqlda {
    unsigned char sqldaid[8];
    long sqldabcf;
    short sqlni;
    short sqld;
    struct sqlvar {
        short sqltype;
        short sqllen;
        unsigned char *sqldata;
        short *sqlind;
    } sqlname;
    short le;
    unsigned char da;
} sqlvar[1];
;

```

Figura 18.7. El área de datos SQL (SQLDA) para bases de datos IBM.

ma; debe ser reemplazada con una lista de longitud variable. SQLDA proporciona un modo de especificar tal lista de parámetros de longitud variable.

La Figura 19.7 muestra la estructura del SQLDA utilizado por las bases de datos de IBM (DB2, SQL/DS y OS/2 Extended Edition). La mayoría de los restantes productos DBMS también utilizan el formato SQLDA de IBM o uno muy similar. El SQLDA es una estructura de datos de tamaño variable con dos partes distintas:

- La *parte variable* está localizada al comienzo del SQLDA. Sus campos identifican la estructura de datos como un SQLVAR y especifican el tamaño de este SQLDA particular.
- La *parte fija* es un array de una o más estructuras de datos SQLVAR.

Cuando se utiliza SQLDA para transferir parámetros a una sentencia EXECUTE, debe haber una estructura SQLVAR por cada parámetro. Los campos de la estructura SQLVAR describen los datos que se transfieren a la sentencia EXECUTE como valor de parámetro:

- El campo SQLTYPE contiene un *código de tipo de dato* entero que especifica el tipo de dato del parámetro transferido. Por ejemplo, el código de tipo de dato DB2 es 500 para un entero de dos bytes, 496 para un entero de cuatro bytes y 448 para una cadena de caracteres de longitud variable.

se transfiere una cadena de caracteres como parámetro, SQLLEN contiene el número de caracteres de la cadena.

■ El campo SQLDATA es un puntero al *área de datos* dentro del programa que contiene el valor del parámetro. El DBMS utiliza este puntero para hallar el valor del dato cuando ejecuta la sentencia SQL dinámica. Los campos SQLTYPE y SQLLEN dicen al DBMS qué tipo de dato está siendo apuntado y su longitud.

El campo SQLIND es un puntero a un entero de dos bytes que se utiliza como variable indicadora para el parámetro. El DBMS examina la variable indicadora para determinar si se está transfiriendo un valor NULL. Si no se está utilizando una variable indicadora para un parámetro particular, el campo SQLIND debe ser puesto a cero.

Los otros campos de las estructuras SQLVAR y SQLDA no son utilizados para transferir valores de parámetros a la sentencia EXECUTE. Son utilizados cuando se emplea un SQLDA para recuperar datos de la base de datos, como se describirá posteriormente en este capítulo.

La Figura 18.8 muestra un programa de SQL dinámico que utiliza un SQLDA para especificar parámetros de entrada. El programa actualiza la tabla REPVEN-ITAS, pero permite al usuario seleccionar a comienzo del programa qué columnas van a actualizarse. Luego entra en un bucle, solicitando al usuario un número de empleado, y pidiéndole un nuevo valor por cada columna a actualizar. Si el usuario escribe un asterisco (*) en respuesta al requerimiento de «nuevo valor», el programa asigna a la columna correspondiente un valor NULL.

Puesto que el usuario puede seleccionar diferentes columnas cada vez que ejecuta el programa, este programa debe utilizar un SQLDA para transferir los valores de los parámetros a la sentencia EXECUTE. El programa ilustra la técnica general para la utilización de un SQLDA, indicada por las referencias numéricas de la Figura 18.8.

1. El programa asigna un SQLDA suficientemente grande para contener una estructura SQLVAR por cada parámetro a transferir. Fija el campo SQLN para indicar cuántos SQLVARs pueden acomodarse.
 2. Por cada parámetro a transferir, el programa rellena una de las estructuras SQLVAR con información descriptiva del parámetro.
 3. El programa determina el tipo de dato de un parámetro y coloca el código de tipo de dato correcto en el campo SQLTYPE.

```

main()
{
    /* Este programa actualiza columnas especificadas por el usuario de la
    tabla REPVENTAS. Primero pide al usuario que seleccione las columnas a
    actualizar, y luego solicita repetidamente el numero de empleado de un
    vendedor y nuevos valores para las columnas seleccionadas.
    */

#define CNTCOL 6
/* seis columnas en la tabla REPVENTAS */

exec sql include sqlda;
exec sql include sqdca;
exec sql begin declare section;
char bufsent[12001] /* texto SQL a ejecutar */;
exec sql end declare section;

char *malloc()
struct {
    char inductor[31];
    char nombre[31];
    short codigotipo;
    short longbuf;
    char selector;
} columnas[] = {
    {"Nombre", "Nombre", "Inductor para esta columna * /",
     "OFICINA.REP", "nombre para esta columna * /",
     "DIRECTOR", "/* nombre de tipo de dato */",
     "Contrato", "/* longitud de su buffer */",
     "Cuenta", "/* indicador 'seleccionado' (s/n) */",
     "Ventas", "/* indicador 'seleccionado' (s/n) */",
     "Oficina", "/* indicador para valores de parámetros */",
     "varparm", "/* SOLDA para el parámetro actual */",
     "crtparm", "/* SOLVAR para el parámetro presentes */",
     "num_empl", "/* número de empleado introducido por el us-",
     "int i;      "/* Índice para el array columnas[] */",
     "int jj;    "/* Índice para el array sqlvar en sqlda[] */",
     "char inbuf[101]; "/* Entrada efectuada por el usuario */"
};

/* Pide al usuario que seleccione columnas a actualizar */
printf("***** Programa de Actualización de Vendedores ***\r\n");
entparam = 1;
for (i = 0; i < CNTCOL; i++) {

    /* Pregunta sobre esta columna */
    printf("Actualizar la columna %s (s/n)? ");
    qeqs(inbuf);
}

```

Figura 18.8. Utilización de EXECUTE con un SQLDA.

```

/* Asigna una estructura SQLDA para transferir valores de parámetros */
if (inbuf[0] == 'y') {
    columnas[i].seleccio = 'y';
    cntparm += 1;
}

/* Manda la sentencia UPDATE dinámica completa */
exec sql prepare sentactual from :bufsent;
if (sqlca.sqlcode < 0) {
    printf("PREPARE error: %d\n", sqlca.sqlcode);
    exit();
}

/* Ahora itera, pidiendo parámetros y haciendo UPDATES */
for ( ; ; ) {
    /* Pide al usuario el número del vendedor a actualizar */
    printf("\nIntroduzca el Número de Empleado del Vendedor: ");
    scanf("%d", &num_empl);

    if (num_empl == 0) break;

    /* Obtiene nuevos valores para la columna seleccionada */
    for (j = 0; j < (cntparm-1); j++) {
        varparm = adparm + j;
        printf("Introduzca un nuevo valor para %s: ", varparm->sqlname.data);
        gets(inbuf);
        if (inbuf[0] == '*') {
            /* Si el usuario introduce '*' pone un valor NULL en la columna */
            *varparm->sqind = -1;
            continue;
        }
        else {
            /* En caso contrario, pone el indicador al valor no-NULL */
            *varparm->sqind = 0;
        }
    }

    /* Comienza a construir la sentencia UPDATE en el buffer de sentencias */
    strcpy(bufsent, "update pedidos set ");
    ①
    /* Itera a través de las columnas, procesando las seleccionadas */
    for (i = 0; i < num_col; i++) {
        /* Pasa por alto las columnas no seleccionadas */
        if (columnas[i].seleccio == "n") {
            continue;
        }

        /* Añade una asignación a la sentencia UPDATE dinámica */
        if (cntparm > 0) strcat(bufsent, ", ");
        strcat(bufsent, columnas[i].nombre);
        strcat(bufsent, " = ?");

        /* Asigna espacio para datos y variable indicadora, y */
        /* rellena la SQLVAR con información para esta columna */
        varparm = adparm + i;
        varparm->sqltype = columnas[i].codigotipo;
        varparm->sqllen = columnas[i].longbuf;
        varparm->sqldata = malloc(columnas[i].Longbuf);
        varparm->sqind = malloc(2);
        strcpy(varparm->sqlname.datos, columnas[i].inductor);
        j + 1;
    }
}

```

```

/* Manda al DBMS que compile la sentencia UPDATE dinámica completa */
exec sql prepare sentactual from :bufsent;
if (sqlca.sqlcode < 0) {
    printf("PREPARE error: %d\n", sqlca.sqlcode);
    exit();
}

/* Ahora itera, pidiendo parámetros y haciendo UPDATES */
for ( ; ; ) {
    /* Pide al usuario el número del vendedor a actualizar */
    printf("\nIntroduzca el Número de Empleado del Vendedor: ");
    scanf("%d", &num_empl);

    if (num_empl == 0) break;

    /* Obtiene nuevos valores para la columna seleccionada */
    for (j = 0; j < (cntparm-1); j++) {
        varparm = adparm + j;
        printf("Introduzca un nuevo valor para %s: ", varparm->sqlname.data);
        gets(inbuf);
        if (inbuf[0] == '*') {
            /* Si el usuario introduce '*' pone un valor NULL en la columna */
            *varparm->sqind = -1;
            continue;
        }
        else {
            /* En caso contrario, pone el indicador al valor no-NULL */
            *varparm->sqind = 0;
        }
    }

    /* Comienza a construir la sentencia UPDATE en el buffer de sentencias */
    strcpy(bufsent, "update pedidos set ");
    ①
    /* Itera a través de las columnas, procesando las seleccionadas */
    for (i = 0; i < num_col; i++) {
        /* Pasa por alto las columnas no seleccionadas */
        if (columnas[i].seleccio == "n") {
            continue;
        }

        /* Añade una asignación a la sentencia UPDATE dinámica */
        if (cntparm > 0) strcat(bufsent, ", ");
        strcat(bufsent, columnas[i].nombre);
        strcat(bufsent, " = ?");

        /* Asigna espacio para datos y variable indicadora, y */
        /* rellena la SQLVAR con información para esta columna */
        varparm = adparm + i;
        varparm->sqltype = columnas[i].codigotipo;
        varparm->sqllen = columnas[i].longbuf;
        varparm->sqldata = malloc(columnas[i].Longbuf);
        varparm->sqind = malloc(2);
        strcpy(varparm->sqlname.datos, columnas[i].inductor);
        j + 1;
    }
}

/* Rellena la última SQLVAR para el parámetro de la cláusula WHERE */
strcat(bufsent, "where numempl = ?");
varparm = adparm + cntparm;
varparm->sqltype = 496;
varparm->sqllen = 4;
varparm->sqldata = &num_empl;
varparm->sqind = 0;
adparm = cntparm;

```

Figura 18.8. Utilización de EXECUTE con un SQLDA (continuación).

Figura 18.8. Utilización de EXECUTE con un SQLDA (continuación).

```

*** Programa de Actualización de Vendedores ***

/* Ejecuta la sentencia */
exec sql execute sentactual using :adparm;
if (sqlca.sqlcode < 0) {
  printf('EXECUTE error: %ld\n', sqlca.sqlcode);
  exit();
}

/* Todas las actualizaciones finalizadas */
exec sql execute immediate "commit work";
if (sqlca.sqlcode)
  printf('COMMIT error: %d\n', sqlca.sqlcode);
else
  printf('\nTodas las actualizaciones cumplidas.\n');

exit();
}

```

Figura 18.8. Utilización de EXECUTE con un SQLDA (continuación).

4. El programa determina la longitud del parámetro y la coloca en el campo SQLLEN.
5. El programa asigna memoria para contener el valor del parámetro y coloca la dirección de la memoria asignada en el campo SQLDATA.
6. El programa asigna memoria para contener una variable indicadora para el parámetro y coloca la dirección de la variable indicadora en el campo SQLIND.
7. Ahora, el programa fija el campo SQLD en la cabecera de SQLDA para que indique cuántos parámetros van a transferirse. Esto dice al DBMS cuántas estructuras SQLVAR dentro del SQLDA contienen datos válidos.
8. El programa solicita al usuario valores de datos y los coloca en las áreas de datos asignadas en los Pasos 5 y 6.
9. El programa utiliza una sentencia EXECUTE con la cláusula USING DESCRIPTOR para transferir valores de parámetros mediante el SQLDA.

Observe que este programa particular copia la «cadena inductora» para cada valor de parámetro en la estructura SQLNAME. El programa hace esto únicamente por su propia conveniencia; el DBMS ignora la estructura SQLNAME cuando utiliza el SQLDA para transferir parámetros. He aquí un ejemplo de diálogo del usuario con el programa de la Figura 18.8:

```

*** Programa de Actualización de Vendedores ***

Actualizar la columna Nombre (s/n)? s
Actualizar la columna Oficina (s/n)? s
Actualizar la columna Director (s/n)? n
Actualizar la columna Contrato (s/n)? n
Actualizar la columna Cuota (s/n)? s
Actualizar la columna Ventas (s/n)? n

Introduzca el Número de Empleado del Vendedor: 106
Introduzca un nuevo valor para Nombre: Sue Jackson
Introduzca un nuevo valor para Oficina: 22
Introduzca un nuevo valor para Cuota: 175000.00

Introduzca el Número de Empleado del Vendedor: 104
Introduzca un nuevo valor para Nombre: Joe Smith
Introduzca un nuevo valor para Oficina: *
Introduzca un nuevo valor para Cuota: 275000.00

Introduzca el Número de Empleado del Vendedor: 0
Todas las actualizaciones cumplidas.

Basándose en las respuestas del usuario a las preguntas iniciales, el programa genera esta sentencia UPDATE dinámica y la prepara:

update repventas
  set nombre = ?, oficina = ?, cuota = ?
  where num_empl = ?

La sentencia especifica cuatro parámetros, y el programa asigna un SQLDA suficientemente grande para manejar cuatro estructuras SQLVAR. Cuando el usuario suministra el primer conjunto de valores de parámetros, la sentencia UPDATE dinámica se convierte en:

update repventas
  set nombre = 'Sue Jackson', oficina = 22, cuota = 175000.00
  where num_empl = 106

y con el segundo conjunto de valores de parámetros, resulta ser:

update repventas
  set nombre = 'Joe Smith', oficina = NULL, cuota = 275000.00
  where num_empl = 104

```

Este programa es un tanto complejo, pero es sencillo comparado con una utilidad de actualización de base de datos de propósito general real. También ilustra todas las características de SQL dinámico requeridas para ejecutar dinámicamente sentencias con un número variable de parámetros.

La sentencia DECLARE STATEMENT

La sentencia **DECLARE STATEMENT**, mostrada en la Figura 18.9 es propia de SQL dinámico. Declara uno o más nombres de sentencias que van a ser utilizados dentro del programa. Esta sentencia es una directiva opcional al precompilador SQL y se utiliza únicamente con propósitos de documentación. No es necesario declarar explícitamente nombres de sentencia antes de utilizarlos. Sin embargo, la utilización de la sentencia **DECLARE STATEMENT** hace que el programa esté más autodocumentado y sea más fácil de mantener. Si se utiliza la sentencia **DECLARE**

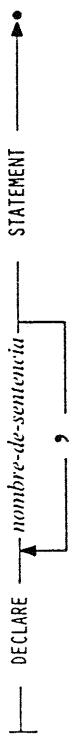


Figura 18.9. Diagrama sintáctico de la sentencia **DECLARE STATEMENT**.

STATEMENT, debería aparecer en el programa delante de cualesquiera sentencias **DECLARE CURSOR** o **PREPARE** que refieran el(s) nombre(s) que declara. La Figura 18.10 muestra un extracto de programa que ilustra su uso.

Los productos SQL de IBM soportan todos la sentencia **DECLARE STATEMENT**, pero la mayoría de los restantes productos SQL no la soportan aún, y sus precompiladores generarán un mensaje de error si usted la utiliza.

```

: /* Prepara una sentencia DELETE dinámica */
exec sql prepare sentsupr from :bufsent;
:
: /* Prepara una sentencia UPDATE dinámica */
exec sql prepare sentsactual from :bufsent;
:
: /* Ejecuta una de las dos sentencias */
if loption = 1)
  exec sql execute sentsactual usign :var1, var2;
else
  exec sql execute sentsupr;
:

```

Figura 18.10. Utilización de la sentencia **DECLARE STATEMENT**.

Consultas dinámicas

Las sentencias **EXECUTE IMMEDIATE**, **PREPARE** y **EXECUTE** tal como se han descrito hasta ahora soportan la ejecución dinámica de la mayoría de las sentencias SQL. Sin embargo, no pueden soportar consultas dinámicas, ya que carecen de un mecanismo para recuperar los resultados de las consultas. Para soportar consultas dinámicas, SQL combina las características de SQL dinámico de las sentencias **PREPARE** y **EXECUTE** con extensiones a las sentencias de procesamiento de consultas de SQL estático, y añade una nueva sentencia. He aquí un panorama general de cómo efectúa un programa una consulta dinámica:

1. Una versión dinámica de la sentencia **DECLARE CURSOR** declara un cursor para la consulta. A diferencia de la sentencia **DECLARE CURSOR** estática, que incluye una sentencia **SELECT** codificada fija, la forma dinámica de la sentencia **DECLARE CURSOR** especifica el nombre de sentencia que estará asociado con la sentencia **SELECT** dinámica.
2. El programa construye una sentencia **SELECT** válida en un buffer, lo mismo que construiría una sentencia **UPDATE** o **DELETE** dinámica. La sentencia **SELECT** puede contener marcadores de parámetros como los utilizados en otras sentencias de SQL dinámico.
3. El programa utiliza la sentencia **PREPARE** para transferir la cadena de sentencia al DBMS, el cual analiza, valida y optimiza la sentencia y genera un plan de aplicación. Esto es idéntico al procesamiento **PREPARE** utilizado por otras sentencias de SQL dinámico.
4. El programa utiliza la sentencia **DESCRIBE** para solicitar una descripción de los resultados de la consulta que serán producidos por la consulta. El DBMS devuelve una descripción columna a columna de los resultados de la consulta en un Área de Datos (SQLDA) suministrado por el programa, diciendo al programa cuántas columnas de resultados hay, y el nombre, el tipo de datos y la longitud de cada columna. La sentencia **DESCRIBE** se utiliza exclusivamente para consultas dinámicas.
5. El programa utiliza las descripciones de columnas en el SQLDA para asignar un bloque de memoria para recibir cada columna de resultados. El programa puede asignar también espacio para una variable indicadora para la columna. El programa coloca la dirección del área de datos y la dirección de la variable indicadora en el SQLDA para decir al DBMS dónde debe devolver los resultados de la consulta.
6. Una versión dinámica de la sentencia **OPEN** pide al DBMS que comience a ejecutar la consulta y transfiere valores para los parámetros especificados en la sentencia **SELECT** dinámica. La sentencia **OPEN** posiciona el cursor delante de la primera fila de resultados.

Puesto que el programa ha llenado previamente los campos SQLDATA y SQLIND del array SQLVAR, el DBMS sabe dónde colocar cada columna de datos recuperada.

Como muestra este ejemplo, gran parte de la programación requerida por una consulta dinámica tiene que ver con la preparación del SQLDA y la asignación de almacenamiento para el SQLDA y los datos recuperados. El programa también debe clasificar los diferentes tipos de datos que pueden ser devueltos por la consulta y manejar cada uno de ellos corteadamente, teniendo en cuenta la posibilidad de que los datos retornados puedan ser NULL. Estas características del programa de ejemplo son típicas de las aplicaciones de producción que utilizan consultas dinámicas. A pesar de la complejidad, la programación no es demasiado difícil en C, Pascal o PL/I. Los lenguajes tales como COBOL y FORTRAN, que carecen de la capacidad de asignar almacenamiento dinámicamente y de trabajar con estructuras de datos de longitud variable, no pueden ser utilizados para procesamiento de consultas dinámicas.

Las siguientes secciones discutirán la sentencia DESCRIBE y las versiones dinámicas de las sentencias DECLARE CURSOR, OPEN y FETCH.

La sentencia DESCRIBE

La sentencia DESCRIBE, mostrada en la Figura 18.12, es propia de las consultas dinámicas. Se utiliza para solicitar al DBMS una descripción de una consulta dinámica. La sentencia DESCRIBE es utilizada después que la consulta dinámica ha sido compilada con la sentencia PREPARE, pero antes de que sea ejecutada con la sentencia OPEN. La consulta a describir es identificada mediante su nombre de sentencia. El DBMS devuelve la descripción de la consulta en un SQLDA suministrado por el programa.

El SQLDA es una estructura de longitud variable con un array de una o más estructuras SQLVAR, como se describió anteriormente en este capítulo y se mostró en la Figura 18.7. Antes de transferir el SQLDA a la sentencia DESCRIBE, el programa debe rellenar el campo SQLN en la cabecera del SQLDA, informando al DBMS de cuán grande es el array SQLVAR en este SQLDA particular. Como primer paso del procesamiento de DESCRIBE, el DBMS rellena el campo SQLD en la cabecera del SQLDA con el número de columnas de resultados de la consulta. Si el tamaño del array SQLVAR (según se especifica en el campo SQLN) es demasiado pequeño para contener todas las descripciones de columnas, el DBMS no rellena el resto del

SQLDA. En caso contrario, el DBMS rellena un estructura SQLVAR por cada columna de resultados de la consulta, en orden de izquierda a derecha. Los campos de cada SQLVAR describen la columna correspondiente:

- La estructura SQLNAME especifica el nombre de la columna (con el nombre en el campo DATA y la longitud del nombre en el campo LENGTH). Si la columna se deriva de una expresión, el campo SQLNAME no es utilizado.
- El campo SQLTYPE especifica un código de tipo de datos entero para la columna. Los códigos de tipo de datos utilizados por los diferentes productos DBMS varían. Para los productos SQL de IBM, el código de tipo de datos indica a la vez el tipo de datos y si se permiten valores NULL, como se muestra en la Tabla 18.1.
- El campo SQLLEN especifica la longitud de la columna. Para tipos de datos de longitud variable (tales como VARCHAR), la longitud reportada es la máxima longitud de datos; la longitud de las columnas en filas individuales de resultados de la consulta no excederán a esa longitud. Para DB2 (y muchos otros productos SQL), la longitud devuelta para un tipo de datos DECIMAL especifica a la vez el tamaño del número decimal (en el byte superior) y la escala del número (en el byte inferior).
- Los campos SQLDATA y SQLIND no son llenados por el DBMS. El programa de aplicación rellena estos campos con las direcciones del buffer de datos y la variable indicadora para la columna antes de utilizar el SQLDA posteriormente en una sentencia FETCH.

Tipo de datos	NULL permitido	NOT NULL
CHAR	452	453
VARCHAR	448	449
LONG VARCHAR	456	457
SMALLINT	500	501
INTEGER	496	497
FLOAT	480	481
DECIMAL	484	485
DATE	384	385
TIME	388	389
TIMESTAMP	392	393
GRAPHIC	468	469
VARGRAPHIC	464	465

Tabla 18.1. Códigos de tipo de datos SQLDA para DB2.

Una complicación al utilizar la sentencia DESCRIBE es que el programa puede no conocer de antemano cuántas columnas de resultados habrá, y por tanto

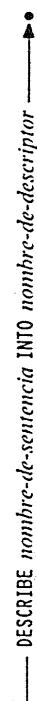


Figura 18.12. Diagrama sintáctico de la sentencia DESCRIBE.

puede no conocer cuán grande debe ser un SQLDA para recibir la descripción. Una de las tres estrategias siguientes es típicamente utilizada para asegurarse que el SQLDA tenga espacio suficiente para las descripciones devueltas:

- Si el programa ha generado la lista de selección de la consulta, puede mantener una cuenta actualizada de los elementos de selección conforme los genera. En este caso, el programa puede asignar un SQLDA con el número exacto de estructuras SQLVAR para recibir las descripciones de columnas. Este enfoque fue utilizado en el programa mostrado en la Figura 18.11.

■ Si es inconveniente para el programa contar el número de elementos de la lista de selección, puede describir (DESCRIBE) inicialmente la consulta dinámica en un SQLDA mínimo con un array SALVAR de un elemento. Cuando la sentencia DESCRIBE retorna, el valor de SQLD dice al programa cuán grande debe ser el SQLDA. El programa puede entonces asignar un SQLDA del tamaño correcto y volver a ejecutar la sentencia DESCRIBE, especificando el nuevo SQLDA. No hay límites al número de veces que una sentencia preparada puede ser descrita.

■ Alternativamente, el programa puede asignar un SQLDA con un array SQLVAR suficientemente grande para acomodar una consulta típica. Una sentencia DESCRIBE que utilice este SQLDA lo hará con éxito la mayor parte de las veces. Si el SQLDA resulta ser demasiado pequeño para la consulta, el valor de SQLD informa al programa de cuán grande debe ser el SQLDA, y puede asignar uno mayor y describir (DESCRIBE) la sentencia de nuevo en SQLDA.

La sentencia DESCRIBE es utilizada normalmente para consultas dinámicas, pero se puede pedir al DBMS que describa cualquier sentencia previamente preparada. Esta característica es útil, por ejemplo, si un programa necesita procesar una sentencia SQL desconocida escrita por un usuario. El programa puede preparar (PREPARE) y describir (DESCRIBE) la sentencia y examinar el campo SQLD en el SQLDA. Si el campo SQLD es cero, el texto de la sentencia no era una consulta, y la sentencia EXECUTE puede ser utilizada para ejecutarla. Si el campo SQLD es positivo, el texto de la sentencia era una consulta, y debe utilizarse la secuencia de sentencias OPEN/FETCH/CLOSE para ejecutarla.

La sentencia DECLARE CURSOR

La sentencia DECLARE CURSOR dinámica, mostrada en la Figura 18.13, es una variación de la sentencia DECLARE CURSOR estática. Recuerde del Capítulo 17 que la sentencia DECLARE CURSOR estática especifica literalmente una consulta mediante la inclusión de la sentencia SELECT como una de sus cláusulas. Por contraste, la sentencia DECLARE CURSOR dinámica especifica la consulta indirectamente, mediante la especificación del nombre de sentencia asociado con la consulta por la sentencia PREPARE.

Al igual que la sentencia DECLARE CURSOR estática, la sentencia DECLARE CURSOR dinámica es una directiva para el precompilador de SQL en vez de una sentencia ejecutable. Debe aparecer antes que cualesquiera otras referencias al cursor que declara. El nombre de cursor declarado por esta sentencia, se utiliza en sentencias OPEN, FETCH y CLOSE subsiguientes para procesar los resultados de la consulta dinámica.

►—DECLARE nombre-del-cursor CURSOR FOR nombre-de-sentencia →●

Figura 18.13. Diagrama sintáctico de la sentencia DECLARE CURSOR dinámica.

La sentencia OPEN dinámica

La sentencia OPEN dinámica, mostrada en la Figura 18.14, es una variación de la sentencia OPEN estática. Hace que el DBMS comience a ejecutar una consulta y posiciona el cursor asociado justo delante de la primera fila de resultados de la consulta. Cuando la sentencia OPEN se completa con éxito, el cursor está en un estado abierto y listo para ser utilizado en una sentencia FETCH.

El papel de la sentencia OPEN para consultas dinámicas sigue un paralelismo con el papel de la sentencia EXECUTE para otras sentencias de SQL dinámico. Tanto las sentencias EXECUTE como OPEN hacen que el DBMS ejecute una sentencia previamente compilada por la sentencia PREPARE. Si el texto de la consulta dinámica incluye uno o más marcadores de parámetros, entonces la sentencia OPEN, al igual que la sentencia EXECUTE, debe suministrar valores para esos parámetros. La consulta USING se utiliza para especificar valores de parámetros, y tiene un formato idéntico en las sentencias EXECUTE y OPEN.

Si el número de parámetros que puede aparecer en una consulta dinámica se conoce de antemano, el programa puede transferir los valores de los parámetros a través de una lista de variables principales en la cláusula USING de la sentencia OPEN. Como en la sentencia EXECUTE, el número de variables principales debe coincidir con el número de parámetros, el tipo de datos de cada variable principal debe ser compatible con el tipo requerido por el parámetro correspondiente, y por cada variable principal puede especificarse una variable indicadora si es necesario. La Figura 18.15 muestra un extracto de programa en donde la consulta dinámica tiene tres parámetros cuyos valores son especificados mediante variables principales.

Si el número de parámetros no se conoce hasta el momento de la ejecución, el programa debe transmitir los valores de los parámetros utilizando una estructura SQLDA. Esta técnica para transferir valores de parámetros fue descrita para la sentencia EXECUTE anteriormente en este capítulo. La misma técnica se aplica a la

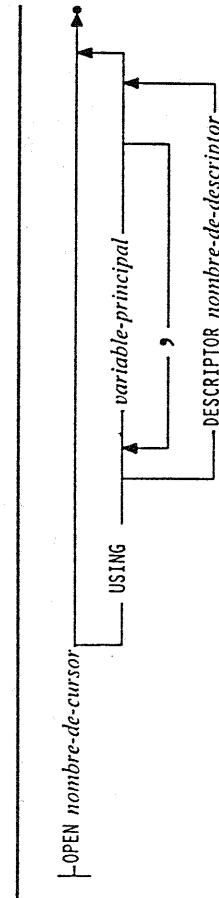


Figura 18.14. Diagrama sintáctico de la sentencia OPEN dinámica.

sentencia OPEN. La Figura 18.16 muestra un extracto de programa semejante al de la Figura 18.15, excepto que utiliza un SQLDA para transferir parámetros.

Observe cuidadosamente que el SQLDA utilizado en la sentencia OPEN no tiene *absolutamente nada* que ver con el SQLDA utilizado en las sentencias DESCRIBE y FETCH:

■ El SQLDA de la sentencia OPEN se utiliza para transferir valores de parámetros al DBMS para ejecución de consulta dinámica. Los elementos de su array SQLVAR corresponden a los *marcadores de parámetros* en el texto de la sentencia dinámica.

- El SQLDA de las sentencias DESCRIBE y FETCH recibe las descripciones de las columnas de resultados de consulta *desde* el DBMS e informa al DBMS de dónde colocar los resultados de consulta recuperados. Los elementos de su array SQLVAR corresponden a las *columnas de resultados* producidos por la consulta dinámica.

La sentencia FETCH dinámica

La sentencia FETCH dinámica, mostrada en la Figura 18.17, es una variación de la sentencia FETCH estática. Avanza el cursor a la siguiente fila disponible de resultado.

/* El programa ha generado y preparado previamente una sentencia SELECT como ésta.

```

SELECT A, B, C ... FROM REVENTAS
WHERE NUM_EMPL IN (? , ? , ... ?)
  
```

con un número variable de parámetros por especificar. El número de parámetros para esta ejecución está almacenado en la variable numparm.

*/

/* El programa ha generado y preparado previamente una sentencia SELECT como ésta:

```

SELECT, A, B, C ... FROM REVENTAS
WHERE VENTAS BETWEEN ? AND ?
  
```

con dos parámetros por especificar

/* Pida al usuario los valores externos y efectúa la consulta */
printf("Introduzca el extremo inferior del rango de ventas: ");
scanf("%f" , &lim_inf);
printf("Introduzca el extremo superior del rango de ventas: ");
scanf("%f" , &lim_sup);

/* Abre el cursor para iniciar la consulta, transfiriendo los parámetros */
exec sql open cursoracc using :lim_end, :lim_sup;

Figura 18.15. Sentencia OPEN con transferencia de parámetros mediante variables principales.

```

char *malloc()
SQLDA *adparm;
SQLVAR *varparm;
long longParm[10];
/* Asigna un SQLDA para transferir valores de parámetros */
adparm = (SQLDA *) malloc(sizeof(SQLDA) + entparm * sizeof(SQLVAR));
adparm -> sqnIn = entparm;

/* Pide al usuario los valores de los parámetros */
for (i = 0; i < cnparm; i++)
printf("Introduzca el número de empleado: ");
scanf("%ld" , &(valor_parm[i]));
adparm -> sqltype = solvar + i;
varparm = adparm -> solvar + i;
varparm -> sqltype = 46;
varparm -> sqlen = 4;
varparm -> sqldata = &(valor_parm[i]);
varparm -> sqlind = 0;

/* Abre el cursor para iniciar la consulta, transfiriendo los parámetros */
exec sql open cursoracc using :adparm;
  
```

Figura 18.16. Sentencia OPEN con transferencia de parámetros mediante SQLDA.

sentencia CLOSE finaliza el acceso a los resultados de la consulta. Cuando un programa cierra un cursor para una consulta dinámica, el programa debería normalmente de asignar también los recursos asociados con la consulta dinámica, incluyendo:

- El SQLDA asignado para la consulta dinámica y utilizado en las sentencias DESCRIBE y FETCH.
 - Un segundo posible SQLDA, utilizado para transferir valores de parámetros a la sentencia OPEN.
 - Las áreas de datos asignados para recibir cada columna de resultados recuperados mediante una sentencia FETCH.
 - Las áreas de datos asignadas como variables indicadoras para las columnas de resultados, de resultados,
- Puede no ser necesario de asignar estas áreas de datos si el programa va a terminar inmediatamente después de la sentencia CLOSE.

Dialectos de SQL dinámico

Como otras partes del lenguaje SQL, SQL dinámico varía de un producto de DBMS a otro. De hecho, las diferencias en el soporte de SQL dinámico son más serias que para SQL estático, ya que SQL dinámico expone más de las «inteligencias» del DBMS subyacente —los tipos de datos, los formatos de los datos, etcétera—. Estas diferencias hacen imposible escribir un frontal de base de datos de propósito general que sea portable a través de diferentes productos DBMS. En vez de ello, los programas frontales de bases de datos deben incluir una «capa de traducción» para cada producto de DBMS que soportan con objeto de acomodar las diferencias.

Una descripción detallada de las características de SQL dinámico soportadas por todos los principales productos DBMS está más allá del alcance de este libro. Sin embargo, es instructivo examinar el soporte de SQL dinámico proporcionado por SQL/DS y por Oracle como ejemplo de las clases de diferencias y extensiones a SQL dinámico que se pueden encontrar en un DBMS particular.

SQL dinámico en SQL/DS

SQL/DS soporta todas las características dinámicas soportadas por DB2 y descritas en las secciones precedentes de este capítulo. Además, SQL/DS soporta una característica denominada *SQL dinámico extendido*. Con SQL dinámico

Figura 18.17. Diagrama sintáctico de la sentencia FETCH dinámica.



dos y recupera los valores de sus columnas en las áreas de datos del programa. Recuerde del Capítulo 17 que la sentencia FETCH estática incluye una cláusula INTO con un lista de variables principales que reciben los valores de las columnas recuperadas. En la sentencia FETCH dinámica, la lista de variables principales está remplazada por un SQLDA.

Antes de utilizar la sentencia FETCH dinámica, es responsabilidad del programa de aplicación proporcionar áreas de datos para recibir los datos recuperados y las variables indicadoras por cada columna. El programa de aplicación debe rellenar también los campos SQLDATA, SQLIND y SQLLEN de la estructura SQLVAR por cada columna, del modo siguiente:

- El campo SQLDATA debe apuntar al área de datos para el dato recuperado.
- El campo SQLLEN debe especificar la longitud del área de datos a que apunta el campo SQLDATA. Este valor debe ser correctamente especificado para asegurar que el DBMS no copie los datos recuperados más allá del final del área de datos.

■ El campo SQLIND debe apuntar a una variable indicadora para la columna (un entero de dos bytes). Si no se utiliza variable indicadora para una columna particular, el campo SQLIND para la estructura SQLVAR correspondiente debería ser puesto a cero.

Normalmente, el programa de aplicación asigna un SQLDA, utiliza la sentencia DESCRIBE para obtener una descripción de los resultados, asigna almacenamiento para cada columna de los resultados y establece los valores de SQLDATA y SQLIND, todo ello antes de abrir el cursor. Este mismo SQLDATA es transferido entonces a la sentencia FETCH. Sin embargo, no es obligatorio utilizar el mismo SQLDA, o que el SQLDA especifique las mismas áreas de datos para cada sentencia FETCH. Es perfectamente aceptable que el programa de aplicación cambie los punteros SQLDATA y SQLIND entre dos sentencias FETCH, recuperando dos filas sucesivas sobre posiciones diferentes.

La sentencia CLOSE dinámica

La forma dinámica de la sentencia CLOSE es idéntica en sintaxis y funcionalidad a la sentencia CLOSE estática mostrada en la Figura 17.25. En ambos casos, la

extendido, se puede escribir un programa, que prepare una cadena de sentencia y almacene permanentemente la sentencia compilada en la base de datos. La sentencia compilada puede ser ejecutada entonces muy eficientemente, bien por el mismo programa o bien por un programa diferente, sin tener que ser preparada de nuevo. Por tanto SQL dinámico extendido proporciona parte de las ventajas de rendimiento de SQL estático en un contexto de SQL dinámico.

Las sentencias preparadas en una base de datos SQL/DS se almacenan en un *módulo de acceso*, que es una colección nominada de sentencias compiladas. Los usuarios de SQL/DS pueden tener sus propios conjuntos de módulos de acceso, protegidos por privilegios de SQL/DS. Para crear un módulo de acceso vacío, se utiliza la sentencia CREATE PROGRAM de SQL/DS, para especificar un nombre de hasta ocho caracteres:

```
CREATE PROGRAM SENTSPP
```

Posteriormente se puede suprimir el módulo de acceso de la base de datos con la sentencia DROP PROGRAM:

```
DROP PROGRAM SENTSPP
```

Observe que aunque las sentencias se denominan CREATE PROGRAM y DROP PROGRAM, realmente operan sobre módulos de acceso. Con frecuencia, sin embargo, el conjunto de sentencias compiladas almacenadas en un nódulo de acceso son, de hecho, el conjunto de sentencias utilizadas por un único programa.

Una vez que se ha creado el módulo de acceso, un programa puede almacenar sentencias compiladas en él y ejecutar esas sentencias compiladas. Para este propósito se utilizan versiones extendidas especiales de las sentencias de SQL dinámico PREPARE, DROP, DESCRIBE, EXECUTE, DECLARE CURSOR, OPEN, FETCH y CLOSE, mostradas en la Figura 18.18. Estas sentencias son soportadas por el precompilador SQL/DS para uso en programas amfífronos escritos en lenguaje ensamblador de IBM S/370.

Para compilar una cadena de sentencia SQL y almacenar la sentencia compilada en un módulo de acceso, el programa debe utilizar la sentencia PREPARE. SQL/DS asigna a la sentencia compilada un *id-sentencia* único (un número de 32 bits) y devuelve el *id-sentencia* en una variable principal al programa. Este id-sentencia es utilizado por todas las restantes sentencias de SQL dinámico extendido para identificar la sentencia compilada. Una sentencia individual puede ser suprimida del módulo de acceso con la sentencia DROP STATEMENT.

Para ejecutar una sentencia almacenada, el programa utiliza una sentencia EXECUTE :ID_SENT IN :NOMBRE_MODULO USING DESCRIPTOR : AD_PARM

El programa transfiere el nombre del módulo de acceso y el *id-sentencia* de la

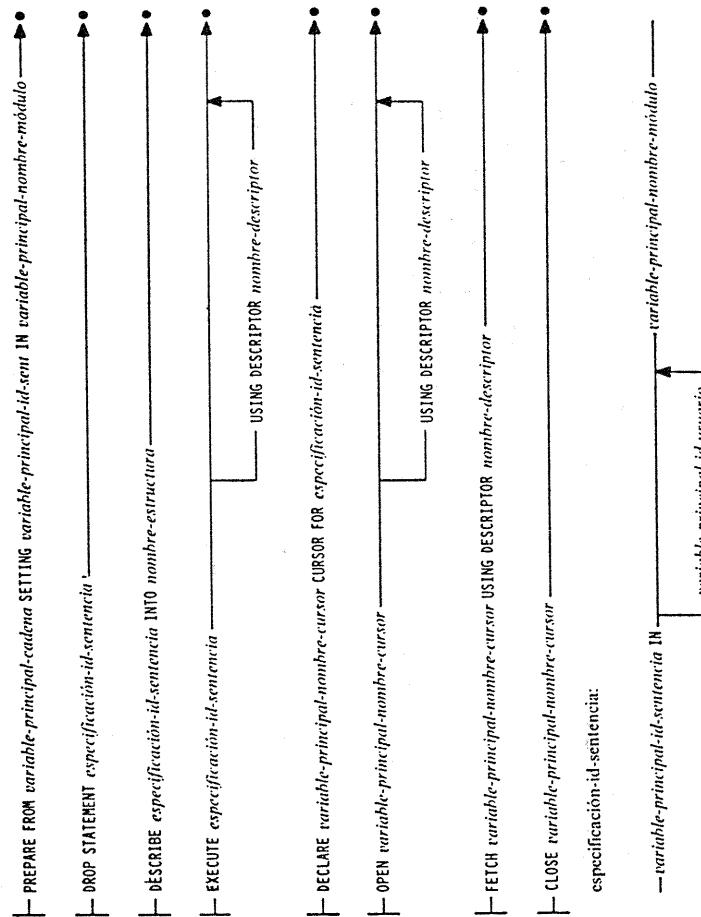


Figura 18.18. Sentencias de SQL dinámico extendido en SQL/DS.

sentencia a ejecutar en un par de variables principales (:NOMBREMODULO e :ID_SENT). También transfiere parámetros para la sentencia dinámica a través de un SQLDA (:AD_PARM), como se describió anteriormente en este capítulo. Al igual que la sentencia EXECUTE «estándar», la sentencia EXECUTE dinámica extendida no puede ser utilizada para ejecutar consultas.

Para ejecutar una consulta almacenada, el programa utiliza una sentencia DECLARATE CURSOR extendida como ésta para asociar un nombre de cursor con la consulta:

```
DECLARE :NOMBRE_CURS CURSOR FOR :ID_SENT IN :NOMBRE_MODULO
```

Observe que el nombre del cursor no está fijado en la sentencia DECLARATE CURSOR, sino que es transferido como una cadena de caracteres en una variable principal (:NOMBRE_CURSOR). Análogamente, la consulta asociada con el cursor no está codificada de forma fija en la sentencia DECLARATE CURSOR (como en SQL estático) ni está especificada mediante un nombre de sentencia (como en SQL dinámico). En vez de ello, la sentencia está especificada mediante la utilización de variables principi-

Conversiones de tipos de datos. Los formatos de tipos de datos que DB2 utiliza para recibir valores de parámetros y devolver resultados de consulta son los soportados por los mainframe computadores de arquitectura IBM S/370 que ejecutan DB2. Puesto que fue diseñado como DBMS portable, Oracle utiliza sus propios formatos internos de tipos de datos. Oracle convierte automáticamente entre sus formatos de datos internos y los del sistema informático sobre el que esté ejecutándose cuando recibe valores de parámetros desde el programa y cuando devuelve resultados de consulta al programa.

El programa puede utilizar el SQLDA de Oracle para controlar la conversión de tipo de datos efectuada por Oracle. Por ejemplo, supongamos que el programa utiliza la sentencia DESCRIBE para describir los resultados de una consulta dinámica y descubre (a partir del código de tipo de datos en el SQLDA) que la primera columna contiene datos numéricos. El programa puede solicitar conversión del dato numérico mediante modificación del código de tipo de dato en el SQLDA antes de acceder al dato. Si el programa coloca el código de tipo de datos de cadena de caracteres en el SQLDA, por ejemplo, Oracle convertirá la primera columna de resultados y los devolverá al programa como una cadena de dígitos. La característica de conversión de tipo de datos del SQLDA de Oracle proporciona una excelente portabilidad, a través de diferentes sistemas informáticos y a través de diferentes lenguajes de programación. Una característica análoga es soportada por otros varios productos DBMS, pero no por los productos SQL de IBM.

Resumen

Este capítulo ha descrito SQL dinámico, una forma avanzada de SQL incorporado. SQL dinámico es raramente necesario para escribir aplicaciones simples de procesamiento de datos, pero es crucial para construir frontales de bases de datos de propósito general. SQL estático y SQL dinámico presentan un compromiso clásico entre eficiencia y flexibilidad, que puede ser resumido del modo siguiente:

- **Simplicidad.** SQL estático es relativamente simple; incluso sus características más complejas, los cursor, pueden ser fácilmente comprendidas en términos de conceptos familiares de entrada y salida de archivos. SQL dinámico es complejo, requiriendo generación dinámica de sentencias, estructuras de datos de longitud variable y asignación de memoria.

- **Rendimiento.** SQL estático se compila en un plan de aplicación en tiempo de compilación; SQL dinámico debe ser compilado en tiempo de ejecución. Como resultado, el rendimiento de SQL estático es generalmente mucho mejor que el de SQL dinámico.

■ **Flexibilidad.** SQL dinámico permite a un programa decidir en tiempo de ejecución qué sentencias SQL específicas ejecutará. SQL estático requiere que todas las sentencias SQL sean codificadas de antemano, cuando se escribe el programa, limitando la flexibilidad del programa.

SQL dinámico utiliza un conjunto de sentencias de SQL incorporado extensamente para soportar sus características dinámicas:

- La sentencia EXECUTE IMMEDIATE transfiere el texto de una sentencia de SQL dinámico al DBMS, el cual la ejecuta inmediatamente.
- La sentencia PREPARE transfiere el texto de una sentencia de SQL dinámico al DBMS, el cual la compila en un plan de aplicación pero no la ejecuta. La sentencia dinámica puede incluir marcadores de parámetros cuyos valores se especifican cuando se ejecuta la sentencia.
- La sentencia EXECUTE pide al DBMS que ejecute una sentencia dinámica previamente compilada por una sentencia PREPARE. También suministra valores de parámetros para la sentencia que va a ser ejecutada.
- La sentencia DESCRIBE devuelve una descripción de una sentencia dinámica previamente preparada en un SQLDA. Si la sentencia dinámica es una consulta, la descripción incluye una descripción de cada columna de resultados de la consulta.
- La sentencia DECLARE CURSOR para una consulta dinámica especifica la consulta mediante el nombre de sentencia asignado a ella cuando fue compilada por la sentencia PREPARE.
- La sentencia OPEN para una consulta dinámica transfiere valores de parámetros para la sentencia SELECT dinámica y solicita ejecución de la consulta.
- La sentencia FETCH para una consulta dinámica accede a una fila de resultados en áreas de datos de programas especificadas mediante una estructura SQLDA.
- La sentencia CLOSE para una consulta dinámica finaliza el acceso a los resultados de la consulta.

19 API de SQL

Varios productos DBMS basados en SQL adoptan un planteamiento de SQL programado que es muy diferente al de SQL incorporado, adoptado por IBM y el estándar ANSI/ISO. En lugar de tratar de mezclar SQL con otro lenguaje de programación, estos productos proporcionan una biblioteca de llamadas de funciones que componen un interfaz de programación de aplicaciones (API —application programming interface) para el DBMS. Para transferir sentencias SQL al DBMS, un programa de aplicación llama a funciones de API, y llama a otras funciones para recuperar resultados de consultas e información del estado procedente del DBMS.

Para muchos programadores, un API SQL es un modo muy sencillo de utilizar SQL. La mayoría de los programadores tienen cierta experiencia en la utilización de bibliotecas de funciones para otros propósitos, como manipulación de cadenas de caracteres, funciones matemáticas, entrada/salida de archivos y gestión de formularios en pantalla. El API SQL resulta ser por tanto «sólo otra biblioteca» para que la aprienda el programador.

Este capítulo describe los conceptos generales utilizados en un interfaz API SQL y luego describe tres de los API más importantes —los proporcionados por SQL Server, Oracle y SQLBase.

Conceptos API

Cuando un DBMS soporta un interfaz de llamada de funciones, un programa de aplicación se comunica con el DBMS, exclusivamente, a través de un conjunto de llamadas que son colectivamente conocidas como un *interfaz de programación*.

ción de aplicaciones, o API (*Application Programming Interface*). La operación básica de un API DBMS se ilustra en la Figura 19.1:

- El programa comienza el acceso a la base de datos con una llamada API que conecta el programa al DBMS y, con frecuencia, a una base de datos específica.
- Para enviar una sentencia SQL al DBMS, el programa construye la sentencia como una cadena de texto en un buffer y luego hace una llamada API para transferir el contenido del buffer al DBMS.
- El programa hace llamadas API para comprobar el estado de su petición al DBMS y para manejar errores.
- Si la petición es una consulta, el programa utiliza llamadas API para recuperar los resultados en los buffers del programa. Típicamente, las llamadas devuelven datos fila a fila o columna a columna.
- El programa termina su acceso a la base de datos con una llamada API que le desconecta del DBMS.

Un API SQL se utiliza, con frecuencia, cuando el programa de aplicación y la base de datos están sobre dos sistemas diferentes en una arquitectura cliente/servidor, como se muestra en la Figura 19.2. En esta configuración, el código para las funciones API está localizado en el sistema cliente, donde se ejecuta el programa de aplicaciones. El software DBMS está localizado en el sistema servidor, donde reside la base de datos. Las llamadas desde el programa de aplicación al API tienen lugar localmente dentro del sistema cliente, pero la comunicación entre el API y el DBMS tiene lugar sobre una red. Como se explicará más tarde en este capítulo, un API SQL ofrece particularidades ventajas para una arquitectura cliente/servidor, ya que puede minimizar la cantidad de tráfico de red entre el API y el DBMS.

Los API ofrecidos por los diferentes productos DBMS difieren sustancialmente entre sí. A diferencia de la interfaz del SQL incorporado, donde DB2 proporcionó un estándar inicial y donde ahora existe un estándar oficial ANSI/ISO, no existe estándar API SQL. No obstante, todos los API SQL disponibles en productos SQL comerciales están basados en los conceptos fundamentales ilustrados en las Figuras 19.1 y 19.2.

El API de SQL Server

Uno de los productos DBMS más importantes que ofrecen un API es SQL Server. El API de SQL Server es importante porque es la *única* interfaz ofrecida

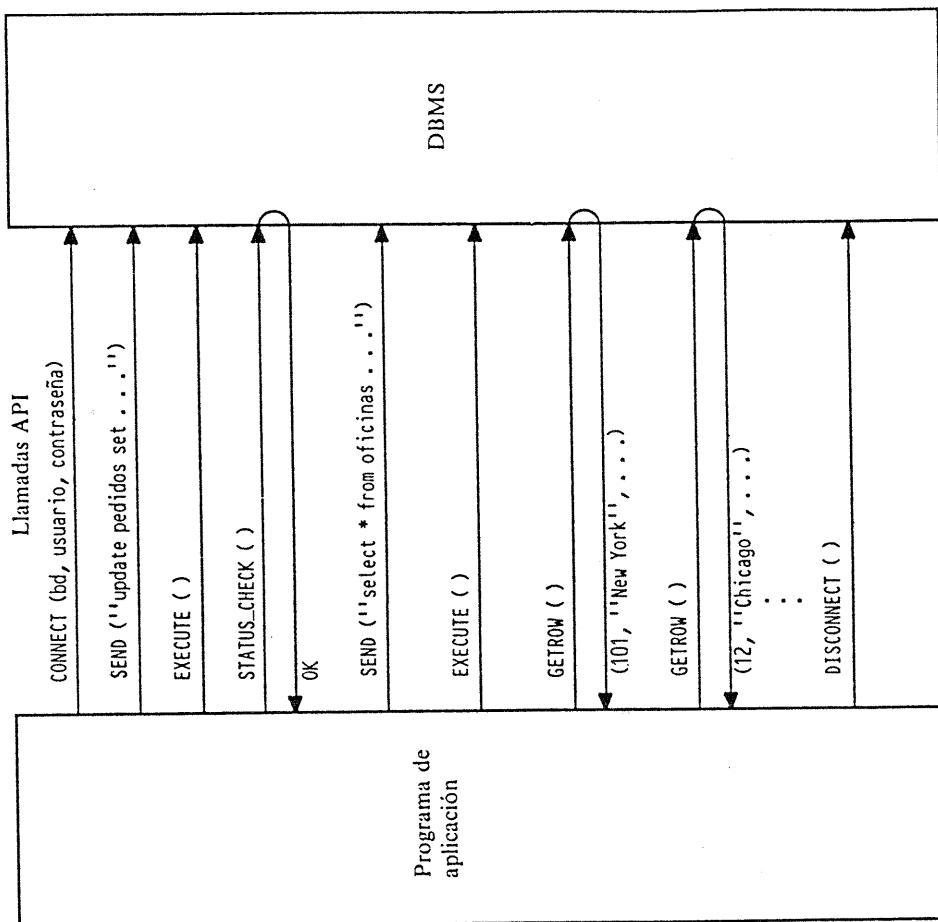


Figura 19.1. Utilización de un API SQL para acceso a DBMS.

por SQL Server que tiene un amplio soporte por parte de Microsoft, Ashton-Tate, Lotus y otros jugadores en la arena de la base de datos OS/2. SQL Server y su API son también un excelente ejemplo de un DBMS diseñado sobre el terreno en torno a una arquitectura cliente/servidor.

El API de SQL Server, al cual se le llama *biblioteca de la base de datos o dblib (database library)*, consta de unas 100 funciones disponibles para un programa de aplicación. El API es muy completo, pero un programa típico sólo utiliza alrededor de una docena de llamadas, que están resumidas en la Ta-

bla 19.1. Las otras llamadas proporcionan características avanzadas, métodos alternativos de interactuar con el DBMS, o versiones de simple llamada de características que en otro caso requerirían múltiples llamadas.

Función	Descripción
<i>Conexión/desconexión de la base de datos</i>	
dblogin()	Proporciona una estructura de datos para información de presentación.
dbopen()	Abre una conexión al SQL Server.
dbuse()	Establece la base de datos por omisión.
dbexit()	Cierra una conexión al SQL Server.
<i>Procesamiento básico de sentencias</i>	
dbcmd()	Transfiere el texto de sentencia SQL a dblib.
dbsqlexec()	Pide la ejecución de un lote de sentencias.
dbresults()	Obtiene resultados de la siguiente sentencia SQL en un lote.
dbcancel()	Cancela el resto de un lote de sentencias.
<i>Manipulación de errores</i>	
dbmsghandle()	Establece un procedimiento manipulador de mensajes escritos por el usuario.
dberrhandle()	Establece un procedimiento manipulador de errores escritos por el usuario.
<i>Procesamiento de resultados de consulta</i>	
dbbind()	Liga una columna de resultados de consulta a una variable de programa.
dbnextrow()	Accede a la siguiente fila de resultados.
dbnumcols()	Obtiene el número de columnas de los resultados de consulta.
dbcolname()	Obtiene el nombre de una columna de resultados.
dbcoltype()	Obtiene el tipo de datos de una columna de resultados.
dbcollen()	Obtiene la longitud máxima de una columna de resultados.
dbdata()	Obtiene un puntero a un valor de dato recuperado.
dbdatlen()	Obtiene la longitud efectiva de un valor de dato recuperado.
dbcquery()	Cancela una consulta antes de que se haya accedido a todas las filas.

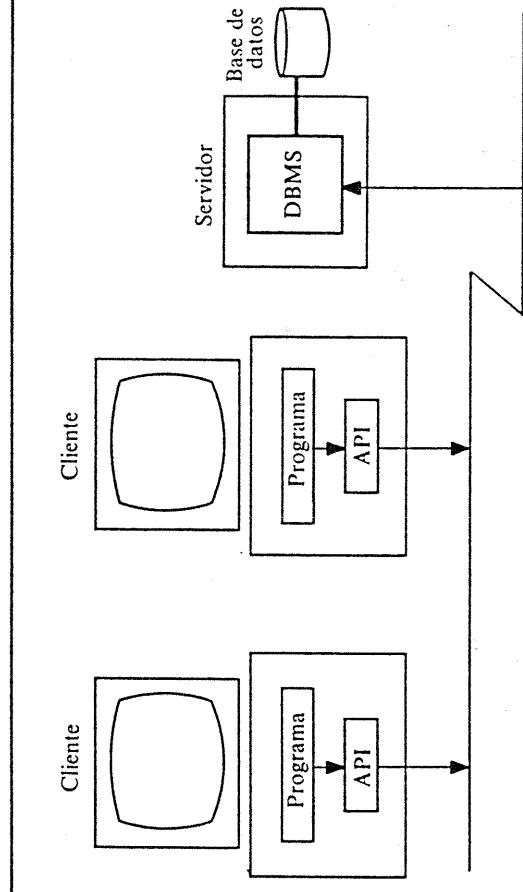


Figura 19.2. Una API SQL en una arquitectura cliente/servidor.

Técnicas básicas de SQL Server

Un programa simple de SQL Server que actualiza una base de datos puede utilizar un conjunto muy pequeño de llamadas dblib para efectuar este trabajo. El programa de la Figura 19.3 implementa una sencilla aplicación de actualización de cuotas para la tabla REVENTAS en la base de datos ejemplo. Es idéntico al programa de la Figura 17.14, pero utiliza el API de SQL Server en vez de SQL incorporado. La figura ilustra la interacción básica entre un programa y SQL Server.

- El programa prepara un «registro de presentación», relleno con el nombre de usuario, su contraseña y la información necesaria para conectarse al DBMS.
- El programa llama a dbopen() para establecer una conexión con el DBMS. La conexión debe existir antes de que el programa pueda enviar sentencias SQL a SQL Server.
- El programa construye una sentencia SQL en un buffer e invoca a dbcmd(). Para transferir el texto SQL a dblib. Llamadas sucesivas a dbcmd() se añaden al texto previamente transferido; no se exige que una sentencia SQL completa se envíe con una única llamada dbcmd().
- El programa invoca a dbsqlexec(), instruyendo a SQL Server para que ejecute la sentencia previamente transferida con dbcmd().

Tabla 19.1. Funciones API básicas de dblib.

```

main()
{
    LOGINREC *regconex;
    DBPROCESS *procdb;
    char cad_importe[31];
    int estado;

    /* Obtiene estructura de presentación y define nombre y contraseña usuario */
    regconex = dblogin();
    /* estructura de datos para conexión */
    /* importe introducido por el usuario (como cadena) */
    /* es estado devuelto por la llamada dblib */

    /* Pide al usuario que escriba la cantidad de aumento/disminución de cuotas */
    printf("Elevar/rebajar las cuotas en cuánto: ");
    gets(cad_importe);

    /* Conecta con SQL Server */
    procdb = dbopen(regconex, " ");
    /* 1 */

    /* Transfiere sentencia SQL a dblib */
    dbcmd(procdb, "update repventas set cuota = cuota + "); /* 2 */
    dbcmd(procdb, cad_importe);
    /* Manda a SQL Server ejecutar la sentencia */
    dosqlexec(procdb); /* 3 */

    /* Obtiene resultados de la ejecución de la sentencia */
    estado = dbresult(procdb); /* 4 */
    /* 5 */
    if (estado !=SUCCEED)
        printf("Error durante actualización.\n");
    else
        printf("Actualización con éxito.\n");

    /* Suspende la conexión con SQL Server */
    dbexit(procdb); /* 6 */
    exit();
}

```

Figura 19.3. Un programa SQL Server sencillo.

5. El programa llama a dbresults() para determinar el éxito o fallo de la sentencia.
 6. El programa invoca a dbexit() para dar por concluida la conexión a SQL Server.
- Es instructivo comparar los programas de las Figuras 19.3 y 17.14 para ver las diferencias entre los enfoques de SQL incorporado y de dblib:

- El programa de SQL incorporado se conecta implícitamente a la única base de datos disponible (como en DB2 o en OS/2 Extended Edition), o incluye una sentencia de SQL incorporado para conexión (tal como la sentencia CONNECT de Oracle, la sentencia DATABASE de Informix o la sentencia DECLARE DATABASE de VAX SQL). El programa dblib se conecta a un SQL Server particular como la llamada dbopen().

- La sentencia UPDATE de SQL procesada efectivamente por el DBMS es idéntica en ambos programas. Con SQL incorporado, la sentencia forma parte del código fuente del programa. Con dblib, la sentencia se transfiere al API como una secuencia de una o más cadenas de caracteres. De hecho, el planteamiento de dblib se asemeja más estrechamente a la sentencia EXECUTE IMMEDIATE de SQL dinámico que al SQL estático.

- En el programa de SQL incorporado, las variables principales proporcionan el enlace entre las sentencias SQL y los valores de las variables del programa. Con dblib, el programa transfiere valores variables al DBMS del mismo modo que transfiere texto del programa —como parte de una cadena de sentencia SQL.

- Con SQL incorporado, los errores se indican en el campo SQLCODE de la estructura SQLCA. Con dblib, la llamada dbresults() recupera el estado de cada una de las sentencias AQL.

Globalmente, el programa de SQL incorporado de la Figura 17.14 es más corto y probablemente más fácil de leer. Sin embargo, el programa no es C puro ni SQL puro, y un programador debe estar entrenado en el uso de SQL incorporado para comprenderlo. El uso de variables principales significa que las formas interactiva e incorporada de la sentencia SQL son diferentes. Adicionalmente, el programa de SQL incorporado debe ser procesado tanto por el precompilador de SQL como por el compilador de C, alargando el ciclo de compilación. En contraste, el programa de SQL Server es un programa totalmente en C, directamente aceptable por el compilador de C, y no requiere técnicas de codificación especiales.

Lotes de sentencias. El programa de la Figura 19.3 envía una única sentencia SQL a SQL Server y comprueba su estado. Si un programa de aplicación debe ejecutar varias sentencias SQL, puede repetir el ciclo dbcmd() / dbsqlexec() / dbresults() para cada sentencia. Alternativamente, el programa puede enviar varias sentencias como un único *lote de sentencias* a ejecutar por el SQL Server.

La Figura 19.4 muestra un programa que utiliza un lote de tres sentencias SQL. Como en la Figura 19.3, el programa llama a dbcmd() para transferir texto SQL a dblib. El API simplemente concatena el texto procedente de cada llamarada. Observe que es responsabilidad del programa incluir los espacios o la puntuación necesaria en el texto transferido. SQL Server no comienza a ejecutar

```

main()
{
    LOGINREC *regconex;
    DBPROCESS *procdb;           /* estructura de datos para presentación (login) */
                                /* estructura de datos para conexión */

    :

    /* Suprime vendedores con bajas ventas */
    dbcmd(procdb, "delete from repventas where ventas < 100000.00");

    /* Aumenta la cuota de vendedores con ventas moderadas */
    dbcmd(procdb, "update repventas set cuota = cuota + 10000.00");
    dbcmd(procdb, "where ventas <= 150000.00");

    /* Aumenta la cuota de vendedores con ventas elevadas */
    dbcmd(procdb, "update repventas set cuota = cuota + 20000.00");
    dbcmd(procdb, "where ventas > 150000.00");

    /* Manda a SQL Server ejecutar el lote de sentencias */
    dbsqlexec(procdb);

    /* Comprueba los resultados de las tres sentencias */
    if (dbresults(procdb) != SUCCEED) goto a_error;
    if (dbresults(procdb) != SUCCEED) goto a_error;
    if (dbresults(procdb) != SUCCEED) goto a_error;

    :
}

```

Figura 19.4. Utilización de un lote de sentencias en SQL Server.

ma debe disponer de su propia función de manejo de mensajes. El software de dblib invoca automáticamente la función de manejo de mensajes cuando SQL Server encuentra un error mientras está ejecutando sentencias de SQL. Observe que dblib invoca la función de manejo de mensajes durante el procesamiento de las llamadas de función dbsqlexec() o dbresults(), antes de volver al programa. Esto permite a la función de manejo de mensajes efectuar su propio procesamiento de errores.

La Figura 19.6 muestra un extracto de un programa SQL Server que incluye una función de manejo de mensajes llamada rtmnmens(). Cuando comienza el programa, activa la función de manejo de mensajes invocando a msghandle(). Supongamos que más tarde se produce un error, mientras SQL Server está procesando la sentencia DELETE. Cuando el programa invoca a dbsqlexec() o dbresults() y dblib recibe el mensaje de error procedente de SQL Server, «invoca» a la rutina rtmnmens() del programa, transfiriéndole cinco parámetros:

- dblproc, la conexión en la cual se produjo el error;
- nomens, el número de error de SQL Server que identifica el error;
- estadomens, que proporciona información referente al contexto del error;
- severidad, un número que indica la seriedad del error;
- textomens, un mensaje de error correspondiente a nomens.

```

/* Ejecuta sentencias previamente generadas con llamadas dbcmd()
   dbsqlexec(procdb);

/* Itera comprobando los resultados de cada sentencia del lote */
while (estado = dbresults(procdb) != NO_MORE_RESULTS) {
    if (estado == FAIL)
        goto handle_error;
    else
        printf("Sentencia con éxito.\n");
}

/* Iteración acabada; lote completado con éxito */
printf("Lote completo.\n");
exit();
}

```

Manejo de errores. El valor devuelto por la función dbresults() informa al programa de si la sentencia correspondiente en el lote ha tenido éxito o ha fallado. Para obtener información más detallada referente a un fallo, el progra-

ma debe disponer de su propia función de manejo de mensajes. El software de dblib invoca automáticamente la función de manejo de mensajes cuando SQL Server encuentra un error mientras está ejecutando sentencias de SQL. Observe que dblib invoca la función de manejo de mensajes durante el procesamiento de las llamadas de función dbsqlexec() o dbresults(), antes de volver al programa. Esto permite a la función de manejo de mensajes efectuar su propio procesamiento de errores.

La Figura 19.6 muestra un extracto de un programa SQL Server que incluye una función de manejo de mensajes llamada rtmnmens(). Cuando comienza el programa, activa la función de manejo de mensajes invocando a msghandle(). Supongamos que más tarde se produce un error, mientras SQL Server está procesando la sentencia DELETE. Cuando el programa invoca a dbsqlexec() o dbresults() y dblib recibe el mensaje de error procedente de SQL Server, «invoca» a la rutina rtmnmens() del programa, transfiriéndole cinco parámetros:

- dblproc, la conexión en la cual se produjo el error;
- nomens, el número de error de SQL Server que identifica el error;
- estadomens, que proporciona información referente al contexto del error;
- severidad, un número que indica la seriedad del error;
- textomens, un mensaje de error correspondiente a nomens.

```

/* Ejecuta sentencias previamente generadas con llamadas dbcmd(),
   dbsqlexec(procdb);

/* Itera comprobando los resultados de cada sentencia del lote */
while (estado = dbresults(procdb) != NO_MORE_RESULTS) {
    if (estado == FAIL)
        goto handle_error;
    else
        printf("Sentencia con éxito.\n");
}

/* Iteración acabada; lote completado con éxito */
printf("Lote completo.\n");
exit();
}

```

Figura 19.5. Procesamiento de los resultados de un lote de sentencias en SQL Server.

La función `rtn.mens()` de este programa maneja el mensaje imprimiéndolo y almacenando el número de error en una variable para su uso posterior en el programa. Cuando la función de manejo de mensaje vuelve a `dblib` (el cual la invocó), `dblib` completa su propio procesamiento y luego regresa al programa

```
:
/* Variables externas para contener información de error */
/* código de error salvado */
int coderror;
char menserr[256];
/* mensaje de error salvado */

/* Define una función manejo-de-mensaje propia */
int mens_rtn(procdb, nomens, estadomens, severidad, textomens)
DBPROCESS *procdb;
DBINT nomens;
int estadomens;
int severidad;
char *estadomens;
char coderror;
extern int *menserr;
extern char *menserr;

{
    /* Imprime número de error y mensaje */
    printf("**** Error %d Message: %s\n", nomens, textomens);
    /* Salva información de error para el programa de aplicación */
    coderror = nomens;
    strcpy(menserr, textomens);

    /* Retorna a dblib para completar la llamada API */
    return(0);
}

main()
{
    DBPROCESS *procdb;
    /* estructura de datos para conexión */

    /* Instala la función de manejo de error propia */
    dberrhandle(mens_rtn)

    /* Ejecuta una sentencia DELETE */
    dbcmd(procdb, "delete from repventas where cuota < 100000.00");
    dbexec(procdb);
    dbresultset(procdb);
    :
    :
}
```

con un estado FAIL (FAIL0). El programa puede detectar este valor de retorno y efectuar procesamiento de error adicional, si es adecuado.

El extracto de programa de la figura presenta realmente una visión simplificada del manejo de errores de SQL Server. Además de los errores de las sentencias SQL detectados por SQL Server, los errores también pueden producirse dentro del propio API de `dblib`. Por ejemplo, si se pierde la conexión de la red al SQL Server, una llamada `dblib` puede agotar un plazo de espera por la respuesta de SQL Server, dando lugar a un error. El API maneja estos errores invocando una *función de manejo de error* separada, la cual opera de forma semejante a la función de manejo de mensajes descrita anteriormente.

Una comparación de la Figura 19.6 con las Figuras 17.10 y 17.13 ilustra las diferencias en las técnicas de manejo de errores entre `dblib` y SQL incorporado:

- En SQL incorporado, la estructura SQLCA se utiliza para señalar errores y avisos al programa. SQL Server comunica errores y avisos al invocar funciones especiales dentro del programa de aplicación y devolver un estado de fallo para la función API que encuentra el error.
- En SQL incorporado, el procesamiento de errores es sincrónico. La sentencia de SQL incorporado falla, el control se devuelve al programa y se examina el valor de SQLCODE. El procesamiento de errores en SQL Server es asíncrono. Cuando falla una llamada API, SQL Server invoca a la función de manejo de mensaje o de manejo de error del programa de aplicación durante la llamada API. Posteriormente vuelve al programa de aplicación con un estado de error.
- SQL incorporado solamente tiene un único tipo de error y un único mecanismo para informar de él. El esquema de SQL Server tiene dos tipos de errores y dos mecanismos paralelos.

En resumen, el manejo de errores en SQL incorporado es sencillo y directo, pero hay un número limitado de respuestas que puede ofrecer el programa de aplicación cuando se produce un error. Un programa de SQL Server tiene más flexibilidad en el manejo de errores. Sin embargo, el esquema de invocaciones utilizado por `dblib` es más sofisticado, y aunque es familiar a los programadores de sistemas, puede ser poco familiar a los programadores de aplicación.

Consultas en SQL Server

La técnica de SQL Server para manejar las consultas programadas es muy similar a su técnica para manejar otras sentencias SQL. Para efectuar una consulta, un programa envía una sentencia SELECT a SQL Server y utiliza `dblib` para recuperar los resultados fila a fila. El programa de la Figura 19.7 ilustra la técnica de procesamiento de consultas en SQL Server.

Figura 19.6. Manejo de errores en un programa de SQL Server.

- El programa utiliza las llamadas `dbcmd()` y `dbsqlexec()` para transferir una sentencia `SELECT` a SQL Server y solicitar su ejecución.
- Cuando el programa llama a `dbresult()` para la sentencia `SELECT`, `dblib` devuelve el estado de terminación para la consulta y también hace que los resultados estén disponibles para su procesamiento.
- El programa invoca a `dbbind()` una vez por cada columna de resultados, diciendo a `dblib` dónde debería devolver los datos para esa columna particular. Los argumentos de `dbbind()` indican el número de columna, el buffer que va a recibir sus datos, el tamaño del buffer y el tipo esperado de datos.
- El programa entra en un bucle, invocando a `dbnextrow()` repetidamente para obtener las filas de resultados de la consulta. El API coloca los datos devueltos en las áreas de datos indicadas en las llamadas `dbbind()` anteriores.
- Cuando ya no quedan más filas de resultados disponibles, la llamada `dbnextrow()` devuelve el valor `NO_MORE_ROWS (NO_MAS_FILAS)`. Si hubiera más sentencias en el lote de sentencias a continuación de la sentencia `SELECT`, el programa podría llamar a `dbresult()` para que avanzara a la siguiente sentencia.

Dos de las llamadas de `dblib` de la Figura 19.7, `dbbind()` y `dbnextrow()`, soportan el procesamiento de resultados de consulta en SQL Server. La llamada `dbbind()` establece una correspondencia uno a uno entre cada columna de resultados y la variable del programa que va a recibir los datos recuperados. Este proceso se denomina *ligadura (binding)* de la columna. En la figura, la primera columna (`NOMBRE`) está ligada a un array de caracteres de 16 bytes y será devuelta como una cadena terminada en nulo. Las columnas segunda y tercera, `CUOTA` y `VENTAS`, están ambas ligadas a números en coma flotante. Es responsabilidad del programador asegurarse que el tipo de datos de cada columna de resultados sea compatible con el tipo de datos de la variable del programa a la cual está ligada.

Una vez más, es útil comparar el procesamiento de consultas en SQL Server de la Figura 19.7 con las consultas en SQL incorporado de las Figuras 17.17 y 17.20:

- SQL incorporado tiene dos técnicas de procesamiento de consultas diferentes —una para consultas de una fila (`SELECT singular`) y otra para consultas multifila (`cursosores`)—. SQL Server utiliza una única técnica, independiente del número de filas de resultados.
- Para especificar la consulta, SQL incorporado reemplaza la sentencia `SELECT` con la sentencia `SELECT singular o la sentencia DECLARE CURSOR`. Con SQL Server, la sentencia `SELECT` enviada por el programa es *idéntica* a la sentencia `SELECT interactivo` para la consulta.

```

main()
{
    LOGINREC      *regconex;           /* estructura de datos para información de presentación (login) */
    DBPROCESS     *procdb;             /* estructura de datos para conexión */
    char          nombrerep[16];       /* nombre de vendedor, recuperada */
    short         cuotarep;           /* cuota de vendedor, recuperada */
    float        ventasrep;          /* ventas por vendedor, recuperada */

    /* Abre una conexión a SQL Server */
    regconex = dblogin();
    DBSETUSER(regconex, "scott");
    DBSETPWD (regconex, "tigre");
    procdb = dbopen(regconex, ".\n");

    /* Transfiere consulta a dblib y manda a SQL Server ejecutarla */
    dbcmd(procdb, "select nombre, cuota, ventas from repventas");
    dbcmd(procdb, "where ventas > cuota order by nombre");
    dbexec(dbcmd(procdb);

    /* Obtiene la primera sentencia del lote */
    dbresults(procdb);               ②

    /* Liga cada columna a una variable en este programa */
    dbbind(procdb, 1, NBSTRINGIND, 16, nombrerep);
    dbbind(procdb, 2, FLT4BIND, 0, &cuotarep);
    dbbind(procdb, 3, FLT4BIND, 0, &ventasrep);           ③

    /* Itera recuperando filas de resultados de consulta */
    while (testado = dbnextrow(procdb) == SUCCEED) {        ④
        /* Imprime datos para este vendedor */
        printf("Nombre: %s\n", nombrerep);
        printf("Cuota: %f\n", cuotarep);
        printf("Ventas: %f\n", ventasrep);
    }

    /* Comprueba errores y cierra la conexión */
    if (testado == FAIL) {
        printf("SQL error.\n");
        dbexit(procdb);
        exit();
    }
}

```

Figura 19.7. Recuperación de resultados de consultas en SQL Server.

- Con SQL incorporado, las variables principales que reciben los resultados de la consulta están designadas en la cláusula `INFO` del `SELECT` singular o de la sentencia `FETCH`. Con SQL Server, las variables que reciben los resultados están especificadas en las llamadas `dbbind()`.

■ Con SQL incorporado, el acceso fila a fila a los resultados se logra mediante sentencias SQL incorporado de propósito especial (OPEN, FETCH y CLOSE). Con SQL Server, el acceso a los resultados de la consulta se hace a través de llamadas de función dblib (dbresults() y dbnextrow()), lo cual hace que el propio lenguaje SQL sea más complicado.

Debido a su relativa simplicidad y a su similitud con la interfaz de SQL interactivo, muchos programadores encuentran más fácil de utilizar la interfaz de SQL Server para el procesamiento de consultas que la interfaz de SQL incorporado.

Recuperación de valores NULL. Las llamadas dbnextrw() y dbbind(), mostradas en la Figura 19.7, proporcionan un modo sencillo de recuperar resultados de consulta, pero no soportan valores NULL. Cuando una fila recuperada por dbnextrw() incluye una columna con un valor NULL, SQL Server reemplaza el NULL con un *valor de sustitución nulo*. Por omisión, SQL Server utiliza cero como valor de sustitución para tipos de datos numéricos, una cadena de blancos para cadenas de longitud fija y una cadena vacía para cadenas de longitud variable. El programa de aplicación puede modificar el valor por omisión de cualquier tipo de datos mediante la invocación de la función API dbsetnull().

En el programa mostrado en la Figura 19.7, si una de las oficinas tuviera un valor NULL en la columna CUOTA, la llamada dbnextrw() para esa oficina recuperaría un cero en la variable valor_cuota. Observe que el programa no puede diferenciar el dato recuperado si la columna CUOTA de la fila tuviera realmente un valor cero o si es NULL. En algunas aplicaciones el uso de valores de sustitución es aceptable, pero en otros es importante poder detectar valores NULL. Estas últimas aplicaciones deben utilizar un esquema alternativo para recuperar los resultados de las consultas, descrito en la próxima sección.

Recuperación utilizando punteros. Con la técnica estándar de recuperación de datos de SQL Server, la llamada dbnextrw() copia el valor del dato para cada columna en una de las variables del programa. Si hubiera muchas filas de resultados o muchas columnas largas de texto, el copiar los datos en las áreas reservadas del programa puede crear un recurgo significativo. Además, la llamada dbnextrw() carece de un mecanismo para devolver valores NULL al programa. Para resolver estos dos problemas, dblib ofrece un método alternativo de recuperación. La Figura 19.8 muestra el extracto de programa de la Figura 19.7, reescrito para utilizar este método alternativo:

1. El programa envía la consulta a SQL Server y utiliza dbresults() para acceder a los resultados, lo mismo que hace con cualquier sentencia SQL. Sin embargo, el programa *no* invoca a dbbind() para ligar las columnas de los resultados a las variables del programa.

2. El programa invoca a dbnextrw() para avanzar, fila a fila, a través de los resultados de la consulta.
3. Por cada columna de cada fila, el programa invoca a dbdata() para obtener un *puntero* a los valores de datos para la columna. El puntero apunta a una posición dentro de los buffers internos de dblib.

4. Si una columna contiene datos de longitud variable, como por ejemplo, los elementos de datos VARCHAR, el programa invoca a dbdat(len()) para determinar la longitud del dato.
5. Si una columna tiene un valor NULL, la función dbdata() devuelve un puntero nulo (0) y dbdat(len()) devuelve 0 como longitud del elemento. Estos valores de retorno dan al programa un modo de detectar y responder a valores NULL en los resultados de la consulta.

El programa de la Figura 19.8 es más complejo que el de la Figura 19.7. En general, es más fácil utilizar la función dbbind() que la técnica dbdata(), a menos que el programa necesite manejar valores NULL o tenga que manejar un gran volumen de resultados.

Recuperación aleatoria de filas. Un programa procesa normalmente los resultados de la consulta en SQL Server recorriendolos secuencialmente mediante el uso de la llamada dbnextrw(). Para aplicaciones de inspección, dblib también proporciona acceso aleatorio limitado a las filas de resultados. El programa debe capacitar explícitamente el acceso aleatorio a filas activando una opción dblib. La llamada dbgetrow() puede utilizarse entonces para recuperar una fila por su número de fila.

Para soportar la recuperación directa de filas, dblib almacena las filas de los resultados en una memoria interna. Si los resultados de la consulta caben enteramente dentro de la memoria de dblib, dbgetrow() soporta la recuperación directa de cualquier fila. Si los resultados de la consulta exceden del tamaño de la memoria, únicamente se almacenan las filas iniciales de los resultados. El programa puede recuperar directamente estas filas, pero una llamada a dbnex-
trw() que intente recuperar una fila después del final de la memoria devuelve la condición de error especial BUF_FULL (BUFFER_LLEN0). El programa debe descartar entonces parte de las filas salvadas del buffer, utilizando la llamada dbclrbuf(), para hacer sitio a la fila siguiente. Una vez descartadas las filas, no se pueden recuperar de nuevo con la función dbgetrow(). Por tanto, dblib soporta recuperación directa de resultados dentro de una «ventana» limitada, dictada por el tamaño de la memoria de filas, tal como se muestra en la Figura 19.9. El programa puede especificar el tamaño de la memoria de filas de dblib mediante la invocación de la rutina dbsetopt().

El acceso directo proporcionado por dbgetrow() es similar a los cursorres de desplazamiento soportados por varios productos DBMS y propuestos por el

```

main()
{
    LOGINREC *regconex; /* estructura de datos para información de presentación (login) */
    DBPROCESS *procdb; /* estructura de datos para conexión */
    char *pnombre; /* puntero a dato de columna NOMBRE */
    int lenombre; /* longitud de dato de columna NOMBRE */
    float *pcuota; /* puntero a dato de columna CUOTA */
    float *pventas; /* puntero a dato de columna VENTAS */

    /* Abre una conexión con SQL Server */
    regconex = dblogin();
    DBSETUSER(regconex, "scott");
    DBSETPWD(regconex, "tigre");
    procdb = dbopen(regconex, "", "");

    /* Transfiere consulta a dblib y manda a SQL Server ejecutarla */
    dbcmd(procdb, "select nombre, cuota, ventas from repventas ''");
    dbcmd(procdb, "where ventas > cuota order by nombre ''");

    /* Obtiene la primera sentencia del lote */
    dbresults(procdb);

    /* Recupera la única fila de resultados de la consulta */
    while (testado = dbnextrow(procdb) == SUCCESS) { → ②

        /* Obtiene la dirección de cada dato de esta fila */
        nombre = dbdata(procdb, 1); → ③
        pcuota = dbdata(procdb, 2); → ④
        pventas = dbdata(procdb, 3); → ⑤
        lenombre = dbdatlen(procdb, 1);

        /* Copia NOMBRE en buffer propio y lo termina en nulo */
        strcpy(bufname, pnombre, lenombre);
        *bufname + lenombre) = (char) 0;

        /* Imprime datos para este vendedor */
        printf("Nombre: %s\n", bufname);
        if (pcuota == 0) → ①
            printf("La cuota es nula (NULL).\n");
        else
            printf("cuota: %6.2f\n", *pcuota);
        printf("ventas: %f\n", *pventas);

        /* Comprueba que haya terminado con éxito */
        if (testado == FAIL)
            printf("SQL error.\n");
        dbexit(procdb);
        exit();
    }
}

```

Figura 19.8. Recuperación utilizando la función dbdata() de SQL Server.

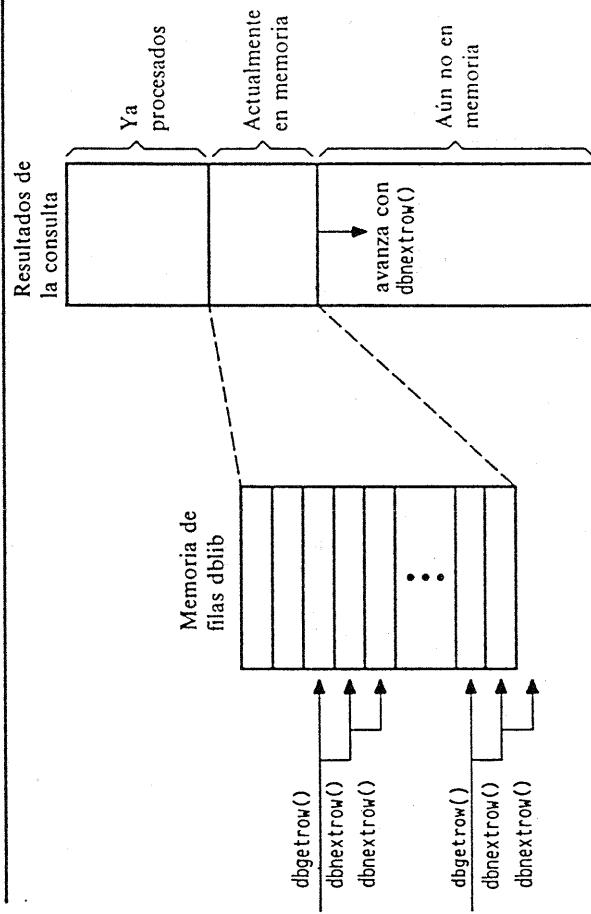


Figura 19.9. Recuperación directa de filas con SQL Server.

estándar SQL2. En ambos casos se soporta la recuperación directa por número de fila. Sin embargo, un cursor de desplazamiento es un puntero real al conjunto completo de resultados; puede ir desde la primera a la última fila, incluso si los resultados contienen miles de filas. Por contra, la función dbgetrow() proporciona acceso directo únicamente dentro de una ventana limitada. Esto es adecuado para aplicaciones de inspección limitadas, pero no puede ser extendido fácilmente a consultas grandes.

Procedimientos almacenados

Una de las características más importantes y más ampliamente promocionadas de SQL Server es su soporte de procedimientos almacenados. Un procedimiento almacenado es una secuencia de sentencias de Transact-SQL a las que se les asigna un nombre, se compila y se almacena en una base de datos SQL Server. Una vez que el procedimiento almacenado ha sido definido en la base de datos, un programa de aplicación puede llamarlo por su nombre, utilizando la interfaz dblib.

Las Figuras 19.10 y 19.11 ofrecen un sencillo ejemplo de cómo funcionan los procedimientos almacenados. La Figura 19.10 es una definición de procedimien-

to, escrita en el dialecto Transact-SQL de SQL Server. El procedimiento, denominado obt_infocliente(), acepta un nombre de cliente como argumento y ejecuta dos consultas. El argumento del procedimiento, num_clie, es una variable del lenguaje Transact-SQL, que puede ser utilizada dentro del procedimiento almacenado en otras sentencias Transact-SQL. En este ejemplo, la variable se utiliza dentro de las cláusulas WHERE de las dos sentencias SELECT. Cuando SQL Server ejecuta la sentencia CREATE PROCEDURE, compila el procedimiento y lo almacena en la base de datos.

```
create procedure obt_infocliente (@num_clie integer) as
begin
    /* Recupera el nombre de la empresa del cliente y su límite de crédito */
    select empresa, limite_credito
    from clientes
    where num_clie = @num_clie

    /* Recupera información referente a cada pedido ordenado por el cliente */
    select num_pedido, importe
    from pedidos
    where clie = @num_clie
end
```

Figura 19.10. Un procedimiento almacenado de SQL Server.

La Figura 19.11 muestra un extracto de una aplicación indagatoria sobre clientes. Pide al usuario que escriba un número de cliente y luego invoca el procedimiento almacenado de la Figura 19.10 para recuperar el nombre y el límite de crédito del cliente desde la tabla CLIENTES y para recuperar todos los pedidos actuales remitidos por ese cliente desde la tabla PEDIDOS. Observe que el programa llama dos veces a dbrsults() para procesar los resultados de las dos consultas en el procedimiento almacenado.

El ejemplo de la Figura 19.10 tan sólo esboza la capacidad de los procedimientos almacenados de SQL Server. El lenguaje Transact-SQL proporciona un conjunto completo de extensões procedurales a SQL, incluyendo comprobación de condiciones (IF/THEN/ELSE), iteración (bucles FOR y WHILE con BREAK y CONTINUE), bifurcación (GO10), llamadas a procedimiento (EXECUTE), bloques de sentencias (BEGIN/END) y variables locales y globales. Utilizando estas características, se pueden crear procedimientos almacenados muy complejos.

Los procedimientos almacenados y el rendimiento del DBMS. SQL Server y el DBMS de Sybase del cual se deriva están ambos posicionados como sistemas de gestión de bases de datos de alto rendimiento para aplicaciones

```
main()
{
    LOGINREC *regconex; /* estructura de datos para información de presentación (login) */
    DBPROCESS *procdb; /* estructura de datos para conexión */
    char numero[11]; /* número de cliente introducido por el usuario */
    char empresa[31]; /* nombre de empresa, recuperado */
    float limite; /* límite de crédito, recuperado */
    long num_pedido; /* número pedido, recuperado */
    float importe; /* importe de pedido, recuperado */

    :

    /* Pide al usuario que escriba un número de cliente */
    printf("Introduzca un número de cliente: ");
    gets(numero);

    /* Manda a SQL Server ejecutar el procedimiento almacenado */
    dbcmd(procdb, "execute obt_infocliente '" );
    dbcmd(procdb, numero );
    dbexec(procdb);

    /* Recupera resultados de la primera consulta y los visualiza */
    dbrsults(procdb);
    dbbind(procdb, 1, NTBSTRNGBND, 31, &empresa);
    dbbind(procdb, 2, FLTBIND, 0, &limite);
    estado = dbnextrw(procdb);
    if (estado != SUCCESS) {
        printf("No existe tal cliente.\n");
        exit();
    }

    printf("Cliente: %s Límite crédito: %f\n", empresa, limite);

    /* Procesa la segunda consulta, imprimiendo información de pedido */
    dbrsults(procdb);
    dbbind(procdb, 1, INTBIND, 0, &num_pedido);
    dbbind(procdb, 2, FLTBIND, 0, &importe);
    while (estado = dbnextrw(procdb) == SUCCESS) {
        /* Imprime datos para este vendedor */
        printf("Número de pedido: %d Importe: %f\n", num_pedido, importe);
    }
}
```

Figura 19.11. Utilización de un procedimiento almacenado en SQL Server.

OLTP (*online transactions processing*). Los procedimientos almacenados son una parte crítica de este reclamo de rendimiento. Para comprender por qué, es útil considerar cómo SQL Server procesa una sentencia SQL, como se muestra en la Figura 19.12. Los cinco pasos de la figura son los mismos que se introdujeron en el Capítulo 17 y se mostraron en la Figura 17.1. La Figura 18.1 mostraba

cómo se aplicaban estos cinco pasos a SQL incorporado estático y a SQL incorporado dinámico, y la Figura 19.12 muestra cómo se aplican a SQL Server. Una comparación de las Figuras 18.1 y 19.12 muestra que dblib es básicamente una interfaz *dinámica* a SQL Server. Cada sentencia que un programa envía a SQL Server recorre los cinco pasos (análisis, validación, optimización, compilación y, finalmente, ejecución) en *tiempo de ejecución*. Pero recuerde del Capítulo 18 que una de las desventajas de SQL dinámico es su rendimiento relativamente pobre comparado con SQL estático. ¿No se contradice el uso de una interfaz dinámica en SQL Server con sus proclamaciones de rendimiento?

La respuesta es no, por dos razones. Primero, las transacciones típicas de un programa OLTP (hacer una reserva, modificar un pedido, añadir un cliente, etc.) utilizan las sentencias INSERT, DELETE, UPDATE y SELECT que acceden a una única fila de datos basándose en la clave primaria de la fila. Es relativamente fácil optimizar este tipo de sentencia SQL. Por tanto, el recargo de la optimización y la compilación en tiempo de ejecución es relativamente bajo para los programas OLTP sencillos de alto uso que necesitan el rendimiento más elevado.

Segundo, y más importante, los procedimientos almacenados de SQL Server proporcionan una modo de analizar, validar, optimizar y compilar una *secuencia entera* de sentencias SQL de antemano. Para obtener el mejor rendimiento posible de un programa, el programador toma las sentencias SQL a ejecutar, define un procedimiento que las contenga y almacena el procedimiento compilado en la base de datos. En tiempo de ejecución, el programa simplemente pide a SQL Server que ejecute el procedimiento almacenado. Por tanto, los procedimientos almacenados proporcionan a SQL Server la flexibilidad de una interfaz SQL dinámico y los beneficios de rendimiento de las sentencias precompiladas encontradas en SQL estático.

Los procedimientos almacenados y el rendimiento de red. Los procedimientos almacenados también juegan un papel esencial en la mejora del rendimiento de red de la arquitectura cliente/servidor propuesta inicialmente por Sybase y SQL Server. En esta arquitectura, el programa de aplicación y el API SQL residen en un sistema *cliente* (típicamente un computador personal) mientras que el DBMS reside en un sistema *servidor* diferente, conectado a los sistemas clientes mediante una red de área local. Para obtener buenos rendimientos de la arquitectura cliente/servidor, el tráfico sobre la red debe ser minimizado, y los procedimientos almacenados ayudan a lograr este objetivo.

Las Figuras 19.13 y 19.14 comparan dos implementaciones cliente/servidor —una utilizando una interfaz de SQL incorporado y la otra utilizando el API de SQL Server y un procedimiento almacenado—. En ambas figuras, el programa de aplicación efectúa una actualización y una consulta que genera una docena de filas de resultados. Observe la diferencia de tráfico de red entre los dos esquemas.

Con el planteamiento de SQL incorporado de la Figura 19.13, cada sentencia SQL debe ser enviada a través de la red desde el cliente al servidor para su procesamiento. El DBMS debe devolver los resultados de cada sentencia al sistema cliente, para que el valor de SQLCODE pueda ser establecido correctamente en el programa de aplicación. Puesto que el procesamiento fila a fila de los resultados de la consulta se maneja por una sentencia SQL, cada fila recuperada también genera una ronda completa de información a través de la red.

Con el planteamiento de SQL Server de la Figura 19.14, una única sentencia SQL que invoca el procedimiento almacenado se envía a través de la red desde el cliente al servidor. El DBMS ejecuta el procedimiento almacenado, generando

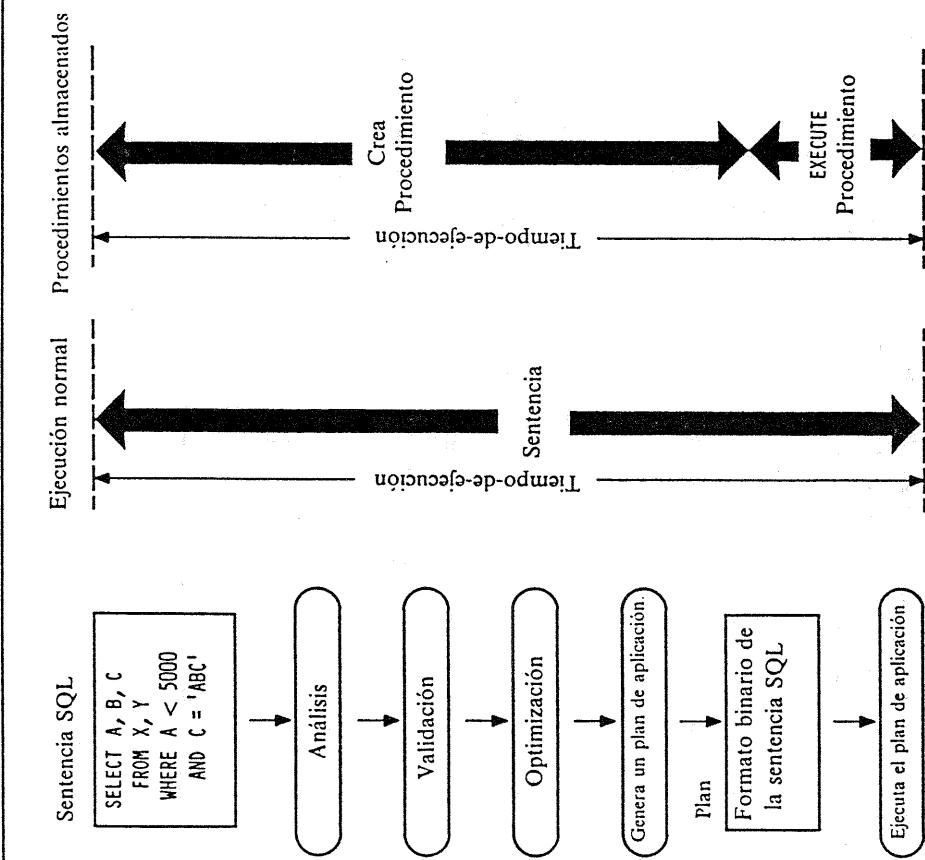


Figura 19.12. Procesamiento en SQL Server de una sentencia SQL.

un flujo de mensajes de estado y de resultados de consulta que se devuelven a través de la red al sistema cliente. En este caso los resultados son relativamente pequeños y el DBMS los envía de vuelta al sistema cliente, en donde dblib los almacena en buffers. Las llamadas `dbnextrow()` del programa dblib son entonces satisfechas a partir de los datos contenidos en los buffers. En vez de las múltiples rondas requeridas por el esquema de SQL incorporado, el esquema SQL Server genera una única ronda a través de la red.

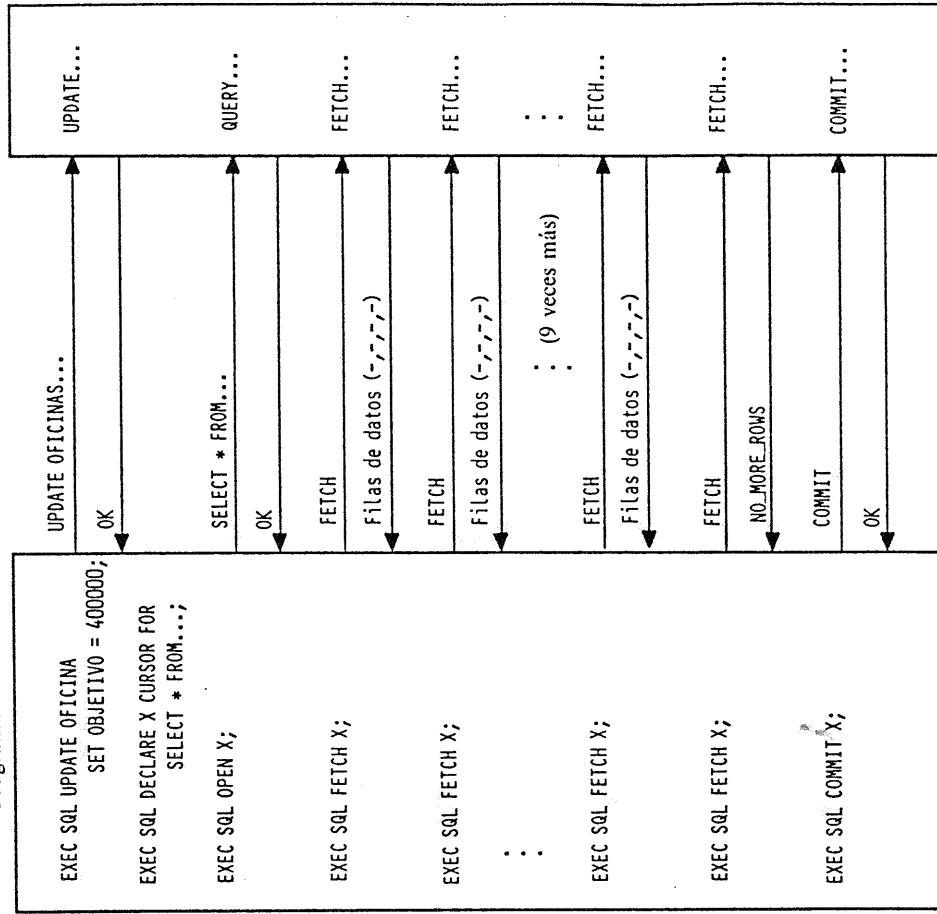
La diferencia entre las dos arquitecturas no es siempre tan importante como las de las Figuras 19.13 y 19.14. Por ejemplo, puede ser imposible incorporar todas las sentencias SQL de un programa en un único procedimiento almacenado. Además, si los resultados de la consulta tienen más de unas pocas filas, el DBMS de SQL Server debe transmitir los resultados a través de la red hasta dblib en pequeñas piezas, generando más tráfico por red. No obstante, las ventajas de la arquitectura de SQL Server en un entorno de red se demuestran claramente en las dos figuras.

Ventajas de los procedimientos almacenados. Los procedimientos almacenados de SQL Server son similares en concepto a los planes de aplicación almacenados en una base de datos DB2 (descritos en el Capítulo 17) y a los módulos de acceso almacenados en una base de datos SQL/DS (descritos en el Capítulo 18). Cada una de estas estructuras almacena la forma compilada de una o más sentencias SQL. De hecho, hay una progresión lógica de SQL estático a SQL dinámico extendido de SQL/DS y a los procedimientos almacenados de SQL Server. Con cada paso de la progresión, el objeto compilado almacenado en la base de datos resulta ser más compatible y más capaz.

- En SQL estático, cada sentencia de SQL incorporado genera su propia sentencia compilada. La sentencia compilada se utiliza exclusivamente en el único punto del programa donde aparece la sentencia incorporada. No puede ser compartida por otros programas, porque no hay modo de identificar la sentencia compilada.

- En SQL dinámico, cada sentencia SQL preparada se identifica por un nombre de sentencia. El nombre de sentencia permite a otras partes del programa reutilizar la sentencia compilada durante la misma transacción. Sin embargo, la sentencia compilada aún no puede ser compartida por otros programas, ya que el nombre de sentencia es específico para un programa particular.

- En SQL dinámico extendido, cada sentencia SQL/DS preparada es identificada mediante un número id-sentencia. El id-sentencia es visible fuera de los confines de la base de datos o de un programa particular, permitiendo que varios programas aprovechen una única sentencia compilada o un conjunto de sentencias compiladas.



Actualizaciones posicionadas

En un programa de SQL incorporado, un cursor proporciona un enlace directo e íntimo entre el programa y el procesamiento de consulta del DBMS. El programa se comunica con el DBMS fila a fila cuando utiliza la sentencia `FETCH` para recuperar resultados. Si la consulta es simple, de una sola tabla, el DBMS puede mantener una correspondencia directa entre la fila actual de resultados y la fila correspondiente dentro de la base de datos. Utilizando esta correspondencia, el programa puede utilizar las sentencias de actualización posicionada (`UPDATE/WHERE CURRENT OF` y `DELETE/WHERE CURRENT OF`) para modificar o suprimir la fila actual de resultados.

Función	Descripción
<code>dbtabcount()</code>	Informa del número de tablas fuentes para la consulta actual.
<code>dbtabname()</code>	Informa del nombre de una de las tablas fuentes.
<code>dbtabbrowse()</code>	Informa de si una tabla particular puede ser actualizada.
<code>dbcolbrowse()</code>	Informa si una columna particular puede ser actualizada.
<code>dbtbsource()</code>	Informa del nombre de la tabla especificada de resultados.
<code>dbcolsource()</code>	Informa del nombre de la columna especificada de resultados.
<code>dbqual()</code>	Produce una cláusula <code>WHERE</code> que seleccionará la fila de resultados actualmente en proceso.

Tabla 19.2. Funciones dblib de modo inspección (Browse).

El procesamiento de consultas en SQL Server utiliza una conexión mucho más separada y asíncrona entre el programa y el DBMS. En respuesta a un lote de sentencias que contiene una o más sentencias `SELECT`, SQL Server envía los resultados de la consulta de vuelta al software dblib, el cual los maneja. La recuperación fila a fila es gestionada por las llamadas API de dblib, no por las sentencias del lenguaje SQL. Como resultado, SQL Server no puede soportar actualizaciones posicionadas, ya que el propio DBMS no tiene noción de una fila «actual». El dialecto Transact-SQL solamente soporta las sentencias `UPDATE` y `DELETE` con búsqueda.

La falta de actualización posicionada puede ser una desventaja real en aplicaciones que permiten al usuario inspeccionar un conjunto de resultados y actualizar ocasionalmente el registro actualmente a la vista. Para minimizar esta desventaja, el API dblib incluye un conjunto especial de funciones en modo *inspección*, resumidas en la Tabla 19.2. Utilizando estas funciones, un programa puede crear su propia capacidad de actualización posicionada con esa técnica:

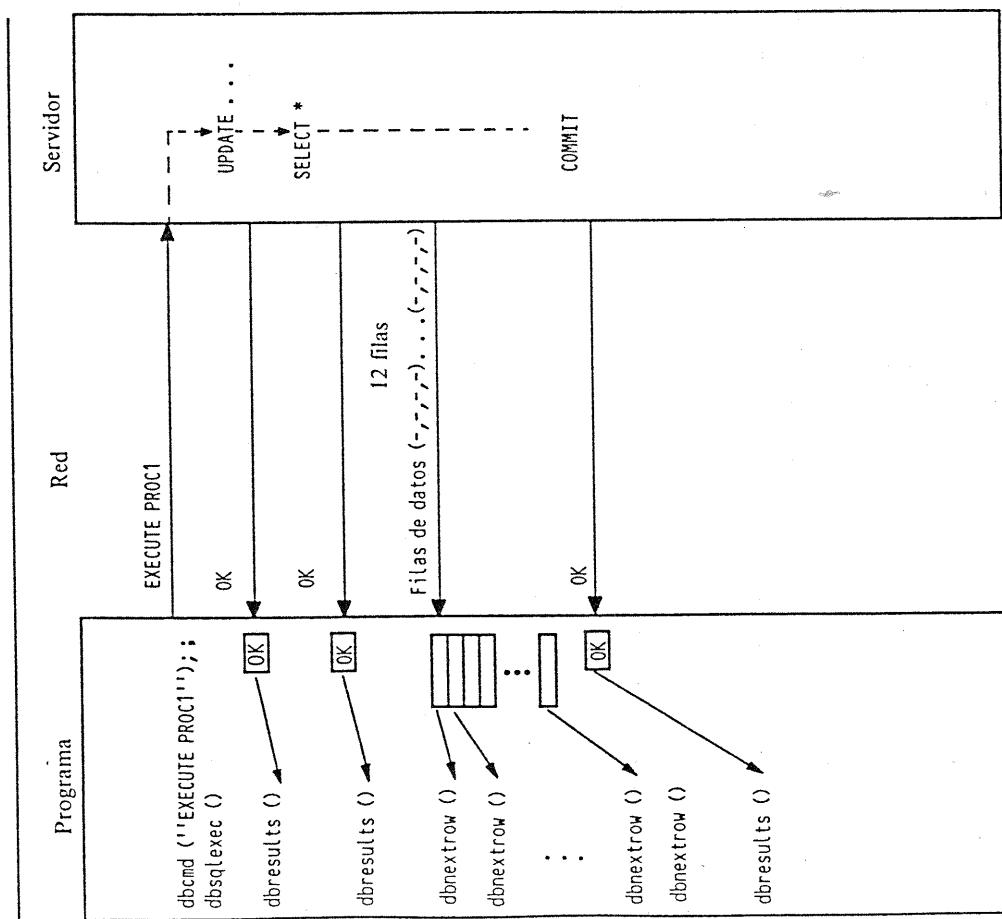


Figura 19.14. Tráfico de red para una transacción en SQL Server.

Los procedimientos almacenados de SQL Server proporcionan una ventaja adicional. Pueden incluir sentencias de Transact-SQL que interactúen con otro software en el servidor aparte del propio DBMS. Por ejemplo, un procedimiento almacenado puede enviar un mensaje por correo a otro usuario o desencadenar la ejecución de un programa escrito por el usuario sobre el sistema servidor. Con esta capacidad, un DBMS SQL Server puede ir más allá del papel de un servidor de base de datos y tomar un conjunto más amplio de responsabilidades como servidor de aplicaciones.

1. La(s) tabla(s) a consultar y actualizar deben tener un único índice definido y deben incluir una columna TIMESTAMP. El índice único asegura que cada fila individual de la(s) tabla(s) puede ser localizada con una condición de búsqueda; la columna TIMESTAMP proporciona un modo de determinar cuán recientemente han sido modificados los contenidos de una fila particular.
2. El programa debe abrir dos conexiones al SQL Server —una para procesar la consulta y la otra para manejar las actualizaciones a filas individuales de los resultados de la consulta.
3. El programa envía una sentencia SELECT a SQL Server sobre la primera conexión, añadiendo las palabras clave FOR BROWSE al final de la sentencia. Estas palabras claves hacen que SQL Server devuelva información adicional a dblib cuando envíe los resultados de vuelta.
4. El programa recupera cada fila de resultados normalmente, utilizando la llamada dbnextrow().

5. Si el usuario desea actualizar la fila actual de resultados, genera una sentencia UPDATE con búsqueda. La sentencia UPDATE debe especificar el nombre de la tabla y la(s) columnas(s) a actualizar y los nuevos valores para cada columna. Si el programador no conoce los nombres de tabla y columnas (por ejemplo, si el programa llama a un procedimiento almacenado para efectuar la consulta), el programa puede determinar los nombres mediante la invocación de dbtabsource(), dbtabname(), dbcolsource() y otras funciones de modo inspección.

6. El programa invoca a la función dblib especial dbqual() para generar una cláusula WHERE adecuada para la sentencia UPDATE. La cláusula WHERE especifica una coincidencia con el índice único (para asegurar que se actualice la fila correcta) y una coincidencia con el valor TIMESTAMP (para asegurar que ningún otro usuario haya actualizado la fila desde que fue recuperada por la consulta).

7. El programa envía la sentencia UPDATE a SQL Server sobre la segunda conexión y llama a dbresults() para asegurarse que ha tenido éxito. Si otro usuario ha modificado la fila o la ha suprimido desde que se produjo la consulta original, la sentencia UPDATE fallará, debido a su cláusula WHERE.

8. El programa repite los Pasos 4 hasta el 7 para cada fila de resultados.

Este es el *único* modo en el que las actualizaciones posicionadas pueden ser procesadas con seguridad en SQL Server. La técnica efectúa básicamente la actualización a «la fuerza» —haciendo tomar al programa de aplicación responsabilidades que son asumidas por SQL incorporado:

■ El programa de aplicación debe generar una sentencia UPDATE con una condi-

ción de búsqueda que identifique únicamente la fila actual de los resultados, en vez de poder especificar WHERE CURRENT OF.

■ La aplicación debe utilizar marcas de tiempo para poder sincronizar las actualizaciones, en vez de poder confiar en el mecanismo de transacciones SQL, debido a que se requieren dos conexiones separadas con la base de datos, con dos contextos de transacción separados.

Las actualizaciones inspeccionadas son utilizadas raramente en aplicaciones OLTP, ya que tienden a disminuir el rendimiento de la base de datos, por lo que las funciones de modo inspección no son necesarias normalmente a los programadores de aplicaciones. Esto es una suerte, ya que las actualizaciones posicionadas son un área en donde la interfaz de SQL Server es mucho más compleja que la de SQL incorporado.

Consultas dinámicas

En todos los ejemplos de programas vistos hasta ahora en este capítulo, las consultas a efectuar son conocidas de antemano. Las columnas de resultados pueden ir ligadas a variables de programa mediante llamadas explícitas dbbind() codificadas de forma fija en el programa. La mayoría de los programas que utilizan SQL Server pueden ser escritos utilizando esta técnica. (La ligazón de columnas estáticas corresponde a la lista fija de variables principales utilizadas en la sentencia FETCH de SQL estático, descrita en el Capítulo 17.)

Si la consulta a efectuar por un programa no es conocida en el momento en que se escribe el programa, éste no puede incluir las llamadas codificadas fijas a dbbind(). En vez de ello, el programa debe pedir a dblib una descripción de cada columna de resultados, utilizando funciones API especiales. El programa puede entonces ligar las columnas «al vuelo» con áreas de datos que asigna en tiempo de ejecución. (La ligadura dinámica de columnas corresponde con el uso de la sentencia DESCRIBE de SQL dinámico y con SQLDA, descritos en el Capítulo 18.)

La Figura 19.15 muestra un programa de consulta interactiva que ilustra la técnica dblib de manejo de consultas dinámicas. El programa acepta un nombre de tabla introducido por el usuario y luego pide al usuario que elija las columnas que se van a recuperar de la tabla. Cuando el usuario selecciona las columnas, el programa construye una sentencia SELECT y luego utiliza estos pasos para ejecutar la sentencia SELECT y visualizar los datos de las columnas seleccionadas.

1. El programa transfiere la sentencia SELECT generada a SQL Server utilizando la llamada dbcmd(), solicita su ejecución con la llamada dbsqlexec() y llama a dbresults() para que avance el API a los resultados de la consulta, como hace para todas las consultas.

```

main()
{
    /*Este es un sencillo programa de consulta de propósito general. Sigue el
    usuario el nombre de una tabla, y luego le pregunta qué columnas de la tabla
    va a incluir en la consulta. Una vez completas las selecciones del usuario,
    el programa efectúa la consulta solicitada y visualiza los resultados
    */

    LOGINREC *regconex;
    DBPROCESS *procdb;
    char bufsent[2001];
    char tb acceso;
    char col acceso;
    int estado;
    int primer_col = 0;
    int numcols;
    int i;
    char bufin [101];
    char *nombre_el [100];
    char *datos_el [100];
    int tipo_el [100];
    char *dirección;
    int longitud;

    /* Abre una conexión con SQL Server */
    regconex = dblogin();
    DBSETUSER(regconex, "Scott");
    DBSETPWD (regconex, "tigre");
    procd = dbopen(regconex, "", "");

    /* Pregunta al usuario qué tabla va a consultar */
    printf("**** Mini-Programa de Consulta ***\n");
    printf("Introduzca un nombre de tabla para consulta: ");
    gets(tl acceso);

    /* Inicia la sentencia SELECT en el buffer */
    strcpy(bufsent, "select ");

    /* Consulta el catálogo de sistema para obtener los nombres de las columnas de la tabla */
    dbcmd(procd, "select nombre from syscolumns ");
    docmd(procd, "where id = (select id from sysobjects '')");
    docmd(procd, "where type = 'u' and name = ''");
    docmd(procd, "tl acceso");
    dbcmd(procd, "'''");

    /* Procesa los resultados de la consulta */
    dbresults(procd);
    dbbind(procd, col acceso);
}

```

```

while (estado = dbnextrow(procd) == SUCCESS) {
    printf("Se incluye la columna %s (s/n)? , col acceso");
    gets(tl acceso);
    if (inbuf[0] == 's' {
        /* El usuario desea la columna; se añade a la lista de selección */
        if (primer_col > 0) strcat(bufsent, ", ");
        strcat(bufsent, col acceso);
    }
}

/* Acaba la sentencia SELECT con una cláusula FROM */
strcat(bufsent, "from ");
strcat(bufsent, tl acceso);

/* Ejecuta la consulta y avanza los resultados */
dbcmd(procd, bufsent);
dbqexec(procd);
dbresults(procd);

/* Pide a dblib que describa cada columna, le asigna memoria y la liga a variable */
numcols = dbnumcols(procd);
for (i = 0; i < numcols; i++) {
    nombre_el[i] = dbcolname(procd, i);
    tipo_el[i] = dbcoltype(procd, i);
    tipo = dbcoltype(procd, i);
    switch(tipo) {

        case SQLCHAR:
        case SQLTEXT:
        case SQLTIME:
            longitud + dbcollen(procd, i) + 1;
            datos_el[i] = malloc(longitud);
            datos_el[i] = dirección;
            tipo_el[i] = NIBSTRINGBND;
            dbbind(procd, i, NIBSTRINGBND, longitud, dirección);
            break;

        case SQLINT1:
        case SQLINT2:
        case SQLINT4:
            datos_el[i] = malloc(sizeof(long));
            datos_el[i] = dirección;
            tipo_el[i] = INTBND;
            dbbind(procd, i, INTBND, sizeof(long), dirección);
            break;

        case SQLFLT8:
        case SQLMONEY:
            datos_el[i] = malloc(sizeof(double));
            datos_el[i] = dirección;
            tipo_el[i] = FLT8BND;
            dbbind(procd, i, FLT8BND, sizeof(double), dirección);
            break;
    }
}

```

Figura 19.15. Utilización de SQL Server para una consulta dinámica.

Figura 19.15. Utilización de SQL Server para una consulta dinámica (continuación).

4. El programa asigna un buffer para recibir cada columna de resultados e invoca a dbbind() para ligar cada columna a su buffer.
5. Cuando todas las columnas han sido ligadas, el programa invoca a dbnextrow() repetidamente para recuperar cada fila de resultados de la consulta.

El programa de SQL Server de la Figura 19.15 realiza exactamente la misma función que el programa de SQL incorporado dinámico de la Figura 18.11. Es instructivo comparar los dos programas y las técnicas que utilizan:

- Tanto para SQL incorporado como para dblib, el programa construye una sentencia SELECT en sus buffers y la envía al DBMS para su proceso. Con SQL dinámico, la sentencia especial PREPARE se ocupa de esta tarea; con el API de SQL Server, se utilizan las funciones estándar dbcmd() y dbsqlexec().
- Para ambas interfaces, el programa debe solicitar una descripción de las columnas de resultados al DBMS. Con SQL dinámico, la sentencia especial DESCRIBE maneja esta tarea, y la descripción se devuelve en una estructura de datos SQLDA. Con dblib, la descripción se obtiene invocando funciones API. Observe que el programa de la Figura 19.15 mantiene sus propios arrays para llevar la cuenta de la información referente a columnas.
- Para ambas interfaces, el programa debe asignar buffers para recibir los resultados de la consulta y debe ligar columnas individuales a esas posiciones de buffer. Con SQL dinámico, el programa liga las columnas mediante la colocación de las direcciones del buffer en las estructuras SQLVAR en el SQLDA. Con SQL Server, el programa utiliza la función dbbind() para ligar las columnas.
- Para ambas interfaces los resultados de la consulta se devuelven a los buffers del programa, fila a fila. Con SQL dinámico, el programa recupera una fila de resultados utilizando una versión especial de la sentencia FETCH que especifica el SQLDA. Con SQL Server, el programa invoca a dbnextrow() para recuperar una fila.

Globalmente, la estrategia utilizada para manejar consultas dinámicas es muy similar para ambas interfaces. La técnica de SQL dinámico utiliza sentencias especiales y estructuras de datos que son propias de SQL dinámico; son bastante diferentes a las técnicas utilizadas para las consultas en SQL estático. En contraste, las técnicas de SQL Server para consultas dinámicas son básicamente las mismas que las utilizadas para todas las demás consultas. Las únicas características añadidas son las funciones dblib que devuelven información referente a las columnas de resultados de la consulta. Esto hace que el planteamiento de SQL Server sea más fácil de entender para los programadores en SQL menos experimentados.

Figura 19.15. Utilización de SQL Server para una consulta dinámica (continuación).

2. El programa llama a dnumcols() para determinar cuántas columnas de resultados fueron producidas por la sentencia SELECT.
3. Por cada columna, el programa llama a dbcolname() para determinar el nombre de la columna, llama a dbcoltype() para determinar su tipo de datos y llama a dbcollen() para determinar su longitud máxima.

Otras interfaces de llamada de función

Aunque el API de SQL Server ha recibido mucha atención debido al soporte que de SQL Server realizan Microsoft, Ashton-Tate y Lotus, diversos productos DBMS importantes también proporcionan una interfaz invocable. La interfaz mediante llamadas de Oracle es mucho más pequeña y más compacta que el API de SQL Server, y se asemeja estrechamente a la interfaz de SQL incorporado de Oracle en cuanto a sus capacidades. El API de SQLBase también se asemeja al SQL incorporado, pero al igual que el API de SQL Server, proporciona características que van bastante más allá de las proporcionadas por SQL incorporado. Estos dos APIs ofrecen una excelente perspectiva sobre los diferentes modos que SQL puede ser soportado mediante una interfaz de llamadas. Sus características más interesantes se describen en las dos secciones siguientes.

El API de llamadas Oracle

La principal interfaz programática a Oracle es la interfaz de SQL incorporado. Sin embargo, Oracle también proporciona una interfaz API alternativa, conocida como *Interfaz de llamada Oracle (Oracle Call Interface)*. El API incluye unas veinte llamadas resumidas en la Tabla 19.3.

La interfaz de llamada Oracle utiliza el término «cursor» para referirse a una conexión a la base de datos Oracle. Un programa utiliza la llamada `olon()` para conectarse a la base de datos Oracle, pero debe utilizar la llamada `open()` para abrir un cursor a través del cual pueden ejecutarse las sentencias SQL. Mediante la emisión de múltiples llamadas `open()`, los programas de aplicaciones pueden establecer múltiples cursosres (conexiones) y ejecutar sentencias en paralelo. Por ejemplo, un programa podría estar recuperando resultados de consulta sobre una de sus conexiones y utilizar una conexión diferente para emitir sentencias `UPDATE`.

La característica más notable de la interfaz de llamadas Oracle es que se asemeja estrechamente a la interfaz de SQL dinámico incorporado. La Figura 19.16 muestra extractos de dos programas que acceden a una base de datos Oracle, una utilizando SQL incorporado y otra utilizando la interfaz de llamadas. Observe que existe una correspondencia, uno a uno, entre las sentencias de SQL incorporado `CONNECT`, `PREPARE`, `EXECUTE`, `COMMIT` y `ROLLBACK` y las llamadas equivalentes. En el caso de la sentencia `UPDATE`, las variables principales que suministran valores de parámetros están listadas en la sentencia `EXECUTE` incorporada y están especificadas por llamadas `obndrv()` en la interfaz de llamadas. Observe también que la sentencia `UPDATE` incorporada en la figura no tiene contrapartida directa en la interfaz de llamadas; debe ser preparada y ejecutada con llamadas a `osql3()` y a `oexec()`.

Funció n	Descripción
<i>Conexión/desconexión a la base de datos</i>	
<code>olon()</code>	Conecta con una base de datos Oracle.
<code>open()</code>	Abre un cursor (conexión) para procesamiento de sentencias SQL.
<code>close()</code>	Cierra un cursor abierto (conexión).
<code>olbgof()</code>	Desconecta de una base de datos Oracle.
<i>Procesamiento básico de sentencias</i>	
<code>osql3()</code>	Prepara (compila) una cadena de sentencia SQL.
<code>oexec()</code>	Ejecuta una sentencia previamente compilada.
<code>oean()</code>	Ejecuta con un array de variables ligadas.
<code>obreak()</code>	Aborta la función actual de la interfaz.
<code>oerrmsg()</code>	Obtiene texto de mensaje de error.
<i>Parámetros de sentencia</i>	
<code>obndrv()</code>	Liga un parámetro a una variable del programa (por nombre).
<code>obndrn()</code>	Liga un parámetro a una variable del programa (por número).
<i>Procesamiento de transacciones</i>	
<code>ocom()</code>	Cumplimenta la transacción actual.
<code>orol()</code>	Vuelve atrás la transacción actual.
<code>oco()</code>	Activa el modo de autocumplimentación.
<code>ocofo()</code>	Desactiva el modo de autocumplimentación.
<i>Procesamiento de resultados de consulta</i>	
<code>odsc()</code>	Obtiene una descripción de una columna.
<code>oname()</code>	Obtiene el nombre de una columna de resultados.
<code>odefin()</code>	Liga una columna de resultados a una variable de programa.
<code>ofetch()</code>	Accede a múltiples filas de resultados en un array.
<code>open()</code>	Accede a múltiple filas de resultados en un array.
<code>ocan()</code>	Canceled una consulta antes de que se hayan accedido a todas las filas.

Tabla 19.3 Funciones de la interfaz de llamadas Oracle.

Las características de procesamiento de consultas dinámicas de las dos interfaces Oracle son también muy parecidas:

- La sentencia `DESCRIBE` incorporada se convierte en una serie de llamadas a `oname()` y a `odsc()` para recuperar los nombres de las columnas y la informa-

ión sobre el tipo de datos de los resultados. Cada llamada devuelve datos para una sola columna.

- En vez de hacer que el programa prepare los campos de SQLDA para ligar las columnas de resultados a variables principales, para ligar las columnas la interfaz de llamadas utiliza llamadas a `odefin()`. La sentencia `FETCH` incorporada se convierte en una llamada a `ofetch()`. La sentencia `CLOSE` incorporada pasa a ser una llamada a `ocean()`, que finaliza el acceso a los resultados.

Una característica útil y propia de la interfaz de llamadas Oracle es su capacidad de interrumpir una consulta de larga duración. En SQL incorporado en la mayoría de las interfaces de llamadas SQL, un programa pasa control al BMS cuando emite una sentencia OPEN incorporada o una llamada «execute» para iniciar una consulta. El programa no vuelve a obtener control hasta que se completa el procesamiento de la consulta. No existe, por tanto, mecanismo para que el programa interrumpa la consulta. La interfaz de llamadas Oracle proporciona una función `break()` que puede ser invocada asíncronamente, mientras el BMS de Oracle tiene control, para interrumpir el procesamiento Oracle. De este modo, si el programa puede volver a obtener control durante una consulta asíncronamente estableciendo un temporizador y recibiendo una interrupción cuando el tiempo), el programa puede invocar a `break()` para terminar inmediatamente la consulta.

1 API de SOL Race

El DBMS de SQLBase ofrece un API como interfaz único de SQL programado. El API de SQLBase incluye alrededor de 75 llamadas. Las llamadas más frecuentemente utilizadas están resumidas en la Tabla 19.4.

El API de SQLBase mezcla algunas de las características de los APIs de Oracle y SQL Server. Como el API de Oracle, las funciones esenciales del API de SQLBase se asemejan muy estrechamente a las sentencias de SQL incorporadas. Por ejemplo,

Interfaz de SQL incorporado

```

EXEC SQL BEGIN DECLARE SECTION
    char texto1[255]; /* texto sent */
    char texto2[255]; /* texto sent */
    int  param1;      /* parámetro */
    float param2;     /* parámetro */
    char ciudad[31];  /* recuperado */
    float ventas;    /* recuperado */
EXEC SQL END DECLARE SECTION

EXEC SQL CONNECT USING SCOTT/TIGRE;

EXEC SQL UPDATE OFICINAS
    SET CUOTA = 0;

EXEC SQL ROLLBACK TRABAJO;

EXEC SQL PREPARE sent2 USING :texto2;

EXEC SQL EXECUTE sent2
    USING &.param1, &.param2;
    INTO :ciudad, :ventas;

EXEC SQL COMMIT TRABAJO;

EXEC SQL DECLARE C1 CURSOR FOR
    SELECT CIUDAD, VENTAS
    FROM OFICINAS;
EXEC SQL OPEN C1;

EXEC SQL FETCH C1
    INTO :ciudad, :ventas;

EXEC SQL CLOSE C1;

EXEC SQL COMMIT TRABAJO RELEASE;

/* texto sent */
/* texto sent */
/* parámetro */
/* recuperado */
/* área de presentación */
/* área de cursor */

olon(lda, "SCOTT/TIGRE", . . .);

open(crs, adc, . . .);
osql3(crs, "UPDATE OFICINAS SET CUOTA = 0");
exec(crs);

orol(lida);

osql3(crs, texto2);

obndrn(crs, 1, param1, sizeof(int));
obndrn(crs, 2, param2, sizeof(float));
exec(crs);

ocom(adc);

osql3(crs, "SELECT CIUDAD, VENTAS FROM OFICINAS");
onefin(crs, 1, ciudad, 30, 5);
onefin(crs, 2, &ventas, sizeof(float), 4);
exec(crs);

ofetch(crs);

ocan(crs);

oclose(crs);
olosf(adc);

```

Figura 19-16 Comparación de interfaces SQL programado de Oracle.

- Las sentencias DESCRIBE incorporada se sustituye por una serie de llamadas a la función sql_describe(), que proporciona una descripción de una columna individual de resultados.

La Figura 19.17 muestra algunos extractos de programas que comparan API de SQLBase con la interfaz de SQL incorporado. Los extractos de SQL incorporado son los mismos mostrados en la Figura 19.16, y la comparación de

ambas figuras revela el fuerte paralelismo entre las llamadas al API de Oracle y las correspondientes llamadas al API de SQLBase. Igual que el API de SQL Server, el API de SQLBase soporta características y funciones que van más allá de las proporcionadas en el SQL incorporado estándar. Entre estas características se incluyen:

- **Cursores de desplazamiento.** Un programa puede moverse hacia adelante y hacia atrás a través de los resultados utilizando un cursor SQLBase. La llamada `sqlfb()` capacita el acceso bidireccional, `sqlfet()` se mueve hacia adelante una fila, y `sqlfbk()` se mueve atrás una fila.
- **Parámetros nominados.** Al igual que Oracle, SQLBase permite que los parámetros de las sentencias tengan nombres asignados, utilizando una notación semejante a la utilizada por SQL incorporado para variables principales. Los parámetros pueden ser entonces ligados a variables del programa bien por nombre (con `sqlbnd()`) o posicionalmente por número (con `sqlbnn()`).
- **Sentencias nominadas.** Al igual que SQL/DS, SQLBase permite compilar una sentencia SQL y salvar su forma compilada en la base de datos para uso posterior por uno o más programas. La sentencia SQL compilada recibe un nombre mediante la llamada `sqlsto()` y es reinvocada para su uso con la llamada `sqlres()`. La llamada `sqldst()` suprime una sentencia compilada de la base de datos.
- **Conjuntos de resultados.** Tras efectuar una consulta, un programa puede pedir a SQLBase que establezca los resultados de la consulta como un *grupo de resultados*. Las consultas subsiguientes pueden extraer sus datos *únicamente* de las filas del conjunto de resultados, no de las tablas completas de la base de datos. De este modo un programa puede ejecutar una secuencia de consultas, cada una de las cuales estrecha cada vez más el conjunto de resultados obtenido por la consulta previa, para centrarse gradualmente en los datos que necesita. La Tabla 19.5 muestra las llamadas API de SQLBase que soportan el procesamiento de conjunto de resultados.
- **Datos largos.** SQLBase puede almacenar objetos de datos muy largos, tales como documentos e imágenes exploradas, como columnas de una base de datos. Llamadas especiales API, mostradas en la Tabla 19.6, proporcionan accesos a estos «datos largos», permitiendo al programa recuperar y actualizar los datos pieza a pieza. Por ejemplo, un programa puede recuperar una fila que contenga un documento de 50 páginas como una de sus columnas y recuperar el documento página a página, permitiéndole procesar el documento completo mientras asigna suficiente memoria para contener solamente una única página.

ambas figuras revela el fuerte paralelismo entre las llamadas al API de Oracle y las correspondientes llamadas al API de SQLBase. Igual que el API de SQL Server, el API de SQLBase soporta características y funciones que van más allá de las proporcionadas en el SQL incorporado estándar. Entre estas características se incluyen:

Función	Descripción
<i>Conexión/desconexión de la base de datos</i>	
<code>sqlcon()</code>	Compila una sentencia SQL.
<code>sqldis()</code>	Ejecuta una sentencia previamente compilada.
<i>Ejecución básica de sentencias</i>	
<code>sqlcom()</code>	Conecta a una base de datos SQLBase por nombre.
<code>sqlexe()</code>	Desconecta de una base de datos SQLBase.
<code>sqlceo()</code>	Compila y ejecuta una sentencia en un paso.
<code>sqlcty()</code>	Obtiene el tipo de sentencia de la sentencia que acaba de compilar.
<code>sqlerr()</code>	Obtiene texto de mensaje de error.
<code>sqlepo()</code>	Obtiene el desplazamiento de error dentro del texto de sentencia.
<i>Procesamiento de transacciones</i>	
<code>sqlcmto()</code>	Complimenta una transacción.
<code>sqlrbk()</code>	Vuelve atrás una transacción.
<i>Parámetros de sentencia</i>	
<code>sqlbmv()</code>	Obtiene el número de parámetros de una sentencia.
<code>sqlbnn()</code>	Liga un parámetro a una variable de programa (por número).
<code>sqlbnd()</code>	Liga un parámetro a una variable de programa (por nombre).
<i>Procesamiento de resultados de consulta</i>	
<code>sqlnsi()</code>	Obtiene el número de columna de resultados.
<code>sqldes()</code>	Obtiene una descripción de una columna de resultados.
<code>sqlssb()</code>	Liga columna de resultados a una variable de programa.
<code>sqlfn()</code>	Obtiene el nombre totalmente cualificado de una columna de resultados.
<code>sqlffb()</code>	Habilita el acceso hacia atrás.
<code>sqlfet()</code>	Accede a la fila siguiente de resultados.
<code>sqlfbk()</code>	Accede a la fila anterior de resultados.
<code>sqlgfi()</code>	Obtiene información acerca de una columna en la fila actual.

Tabla 19.4. Funciones básicas del API de SQLBase.

20 Gestión de bases de datos distribuidas

Una de las principales tendencias de la computación a lo largo de los últimos diez años ha sido el pasar de grandes computadores centralizados a redes distribuidas de sistemas informáticos. Con la llegada de los minicomputadores, las tareas de procesamiento de datos como el control de inventarios y procesamiento de pedidos pasaron de maxicomputadores corporativos a sistemas departamentales más pequeños. El incremento explosivo en la popularidad del computador personal en los ochenta trajo la potencia del computador directamente a las mesas de despacho de millones de personas.

Conforme los computadores y las redes de computadores se extendían a través de las organizaciones, los datos ya no residían en un único sistema bajo el control de un único DBMS. En vez de ello, los datos fueron extendiéndose a través de muchos sistemas diferentes, cada uno con su propio gestor de base de datos. Con frecuencia los diferentes sistemas informáticos y los diferentes sistemas de gestión de base de datos procedían de diferentes fabricantes.

Estas tendencias han conducido a una fuerte centralización en la industria informática y en la comunidad de gestión de datos sobre los problemas de la gestión de bases de datos distribuidas. Este capítulo explica los retos de la gestión de datos distribuidos y de los productos que los principales vendedores de DBMS están comenzando a ofrecer para satisfacer estos retos.

El reto de la gestión de datos distribuidos

La Figura 20.1 muestra una parte de una red informática que podría encontrarse en una empresa fabrikante, una firma de servicios financieros o una empresa

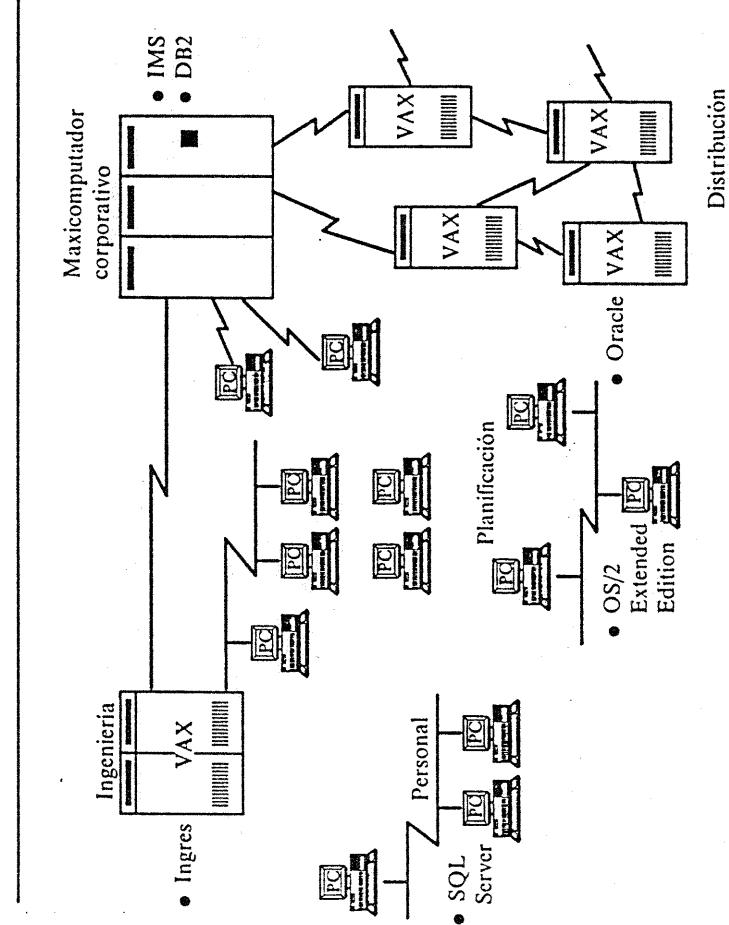


Figura 20.1. Utilización de DRMS en una red corporativa típica.

de distribución. Los datos están almacenados en una variedad de sistemas informáticos dentro de la red:

- **Maxicomputadores.** Las principales aplicaciones de procesamiento de datos de la empresa, tales como la contabilidad y la nómina, corren en un maxicomputador IBM. Desarrolladas durante los últimos veinte años, estas aplicaciones almacenan sus datos en bases de datos IMS, y la empresa está utilizando DB2 para una nueva aplicación en maxicomputador que se completará al final de este año.
- **Minicomputadores.** La organización técnica de la empresa utiliza un minicomputador VAX e Ingres para soporte ingenieril y aplicaciones de tiempo compartido. Los resultados de las pruebas de ingeniería se almacenan en la base de datos Ingres. La empresa también utiliza bases de datos Oracle en minicomputadores localizados en sus seis centros de distribución para gestionar inventarios y procesar pedidos.
- **Servidores LAN.** Algunos departamentos de la empresa han sido pioneros en

el uso de redes de área local en PC (LANs) para compartir impresoras y archivos. Recientemente el departamento de personal automatizó su sistema de personal utilizando SQL Server. En el departamento de planes financieros, una aplicación de planeamiento la está construyendo el equipo de procesamiento de datos, el cual cree que OS/2 Extended Edition debería ser el DBMS OS/2 elegido.

■ **Computadoras personales.** Muchos empleados de la empresa utilizan computadoras personales. Mantienen bases de datos personales utilizando dBASE, Paradox, e incluso hojas de cálculo Lotus 1-2-3. En pocos casos, las bases de datos están compartidas con otros usuarios, utilizando versiones LAN de estos productos.

Con los datos extendidos sobre muchos sistemas diferentes, es fácil imaginar peticiones que referencian más de una base de datos:

- Un ingeniero necesita combinar resultados de test de laboratorio (sobre un VAX) con previsiones de productos (sobre el maxicomputador) para elegir entre tres tecnologías alternativas.
- Un proyectista financiero necesita relacionar previsiones financieras (en una base de datos OS/2 Extended Edition) con datos financieros históricos (sobre el maxicomputador).
- Un director de productos necesita conocer cuántas existencias de un producto particular existen en cada centro de distribución (datos almacenados en seis VAXes) para planear la obsolescencia del producto.
- Los datos sobre precios actuales necesitan ser descargados diariamente desde el maxicomputador a los VAXes de los centros de distribución.
- Los pedidos necesitan ser extraídos diariamente desde los VAXes hasta el maxicomputador para que el plan de fabricación pueda ser ajustado.
- Los directores de toda empresa desean consultar las diferentes bases de datos compartidas utilizando los PC de sus despachos.

Como sugieren estos ejemplos, el acceso a datos distribuidos es útil y pasará a ser crítico cuando se confirme la tendencia hacia la computación distribuida. Los principales vendedores de DBMS están comprometidos en la entrega de gestión de base de datos distribuida, y algunos de ellos ya ofrecen capacidades limitadas de bases de datos distribuidas. Los expertos en la industria y otros han escrito extensamente acerca de las características que debería proporcionar un DBMS distribuido, y existe un acuerdo general sobre estas características «ideales».

- **Transparencia de ubicación.** El usuario no debería tener que preocuparse acerca de dónde están localizados físicamente los datos. El DBMS debería presentar todos los datos como si fueran locales y debería ser responsable de mantener esa ilusión.
- **Sistemas heterogéneos.** El DBMS debería soportar datos almacenados en sistemas diferentes, con diferentes arquitecturas y niveles de rendimiento, incluyendo PC, estaciones de trabajo, servidores de LAN, minicomputadores y mainframe.

■ **Transparencia de red.** Excepto por las diferencias en rendimiento, el DBMS debería trabajar de la misma manera sobre diferentes redes, desde las LANs de alta velocidad a los enlaces telefónicos de baja velocidad.

■ **Consultas distribuidas.** El usuario debería ser capaz de componer datos procedentes de cualquiera de las tablas de la base de datos (distribuida), incluso si las tablas están localizadas en sistemas físicos diferentes.

■ **Actualizaciones distribuidas.** El usuario debería ser capaz de actualizar datos en cualquier tabla para la que tenga los privilegios necesarios, tanto si la tabla está en un sistema local como si está en un sistema remoto.

■ **Transacciones distribuidas.** El DBMS debe soportar transacciones (utilizando COMMIT y ROLLBACK) a través de fronteras de sistema, manteniendo la integridad de la base de datos (distribuida) incluso en presencia de fallos de red y fallos de sistemas individuales.

■ **Seguridad.** El DBMS debe proporcionar un esquema de seguridad adecuado para proteger la base de datos (distribuida) completa frente a formas no autorizadas de acceso.

■ **Acceso universal.** El DBMS debería proporcionar acceso uniforme y universal a todos los datos de la organización.

Ningún producto DBMS distribuido actual ni tan siquiera se acerca a satisfacer este ideal. De hecho, existen formidables obstáculos que hacen difícil proporcionar incluso las formas más sencillas de gestión de una base de datos distribuida. Estos obstáculos incluyen:

■ **Rendimiento.** En una base de datos centralizada, la trayectoria desde el DBMS a los datos tiene una velocidad de acceso de unos pocos milisegundos y una velocidad de transferencia de varios millones de caracteres por segundo. Incluso en una red de área local rápida, las velocidades de acceso se alargan a décimas de segundo y las velocidades de transferencia caen hasta 100.000 caracteres por segundo o menos. En un enlace por modem a 9.600 baudios, el acceso a los datos puede tardar segundos o minutos, y 500 caracteres por segundo puede ser la máxima productividad. Esta gran diferencia en velocidad-

des puede hacer bajar drásticamente el rendimiento del acceso a datos remotos.

■ **Integridad.** Las transacciones distribuidas requieren cooperación activa de dos o más copias independientes del software DBMS ejecutándose sobre sistemas informáticos diferentes si las transacciones van a seguir siendo proporciones «todo o nada». Deben utilizarse protocolos especiales de transacción «cumplimentación en dos fases».

■ **SQL estático.** Una sentencia de SQL incorporado estático se compila y almacena en la base de datos como un plan de aplicación. Cuando una consulta combina datos de dos o más bases de datos, ¿dónde debería almacenarse su plan de aplicación? ¿Deben existir dos o más planes cooperantes? Si hay un cambio en la estructura de una base de datos, ¿cómo se notifica a los planes de aplicación de las otras bases de datos?

■ **Optimización.** Cuando los datos se acceden a través de una red, las reglas normales de optimización de SQL no son aplicables. Por ejemplo, puede ser más eficaz rastrear secuencialmente una tabla local completa que utilizar una búsqueda por índice en una tabla remota. El software de optimización debe conocer las redes y sus velocidades. En general, la optimización pasa a ser más crítica y más difícil.

■ **Compatibilidad de datos.** Diferentes sistemas informáticos soportan diferentes tipos de datos, e incluso cuando dos sistemas ofrecen los mismos tipos de datos utilizan con frecuencia formatos diferentes. Por ejemplo, un VAX y un Macintosh almacenan los enteros de 16 bits de forma diferente. Los mainframe y los minicomputadores IBM almacenan códigos de caracteres EBCDIC mientras que los PC utilizan ASCII. Un DBMS distribuido debe enmascarar estas diferencias.

■ **Catálogos de sistema.** Cuando un DBMS realiza sus tareas, efectúa accesos muy frecuentes a sus catálogos de sistema. ¿Dónde debería mantenerse el catálogo en una base de datos distribuida? Si está centralizado en un sistema, el acceso remoto a él será lento, afectando al DBMS. Si está distribuido a través de muchos sistemas diferentes, los cambios deben propagarse alrededor de la red y deben sincronizarse.

■ **Entorno de vendedores mixtos.** Es altamente improbable que todos los datos de una organización puedan ser gestionados por un único producto DBMS, por lo que el acceso a base de datos distribuida atravesará fronteras de productos DBMS. Esto requiere cooperación activa entre productos DBMS de vendedores altamente competitivos —una perspectiva improbable.

■ **Interbloqueos distribuidos.** Cuando las transacciones de dos sistemas diferentes tratan de acceder a datos cerrados en el otro sistema, puede ocurrir un interbloqueo en la base de datos distribuida, aun cuando el interbloqueo no

sea visible a ninguno de los dos sistemas por separado. El DBMS debe proporcionar la detección del interbloqueo global para una base de datos distribuida.

■ **Recuperación.** Si uno de los sistemas que ejecuta un DBMS distribuido falla, el operador de ese sistema debe ser capaz de ejecutar sus procedimientos de recuperación con independencia del resto de los sistemas en la red, y el estado recuperado de la base de datos debe ser consistente con el de los otros sistemas.

A causa de estos obstáculos, los principales vendedores de DBMS han adoptado un planteamiento de paso a paso para la gestión de datos distribuidos. Al principio el DBMS puede permitir que un usuario en un sistema consulte una base de datos localizada en algún otro sistema. En versiones posteriores las capacidades de DBMS distribuido pueden crecer, acercándose cada vez más al objetivo de proporcionar acceso de datos transparente y universal.

Etapas del acceso a datos distribuidos

En 1989, IBM anunció su proyecto de implementación paso a paso del acceso a datos distribuido en sus productos SQL. IBM no fue el primer vendedor en ofrecer acceso a datos distribuidos, y no es el vendedor con capacidad de DBMS distribuido más avanzado hoy día. Sin embargo, las cuatro etapas de IBM, mostradas en la Tabla 20.1, proporcionan un excelente esquema para comprender las capacidades de gestión de datos distribuidos de IBM y otros vendedores. Naturalmente las capacidades de productos DBMS de IBM caen netamente en las etapas descritas en la tabla.

Las cuatro etapas del modelo IBM tratan con una sencilla definición del problema de acceso a datos distribuidos: un usuario de un sistema informático necesita acceder a datos almacenados en uno o más sistemas informáticos diferentes. La sofisticación del acceso distribuido se incrementa en cada etapa. Por tanto, las capacidades proporcionadas por un DBMS determinado pueden describirse en términos de qué etapa haya alcanzado. Además, dentro de cada etapa puede efectuarse una distinción entre acceso de sólo lectura (con la sentencia SELECT) y acceso de actualización (con las sentencias INSERT, DELETE y UPDATE). Un producto DBMS proporcionará con frecuencia capacidad de sólo lectura para una etapa determinada antes de proporcionar la capacidad de actualización completa.

Peticiones remotas

La primera etapa de acceso a datos distribuidos, tal como la define IBM, es una petición remota, mostrada en la Figura 20.2. En esta etapa, el usuario de un PC

puede emitir una sentencia SQL que consulte o actualice datos en una única base de datos remota. Cada sentencia SQL individual opera como su propia transacción, similar al modo de «autocomplimentación» proporcionada por muchos programas SQL interactivos. El usuario puede emitir una secuencia de sentencias SQL para varias bases de datos, pero el DBMS no soporta transacciones multisentencia.

Etapa	Descripción
1. Petición remota.	Cada sentencia SQL accede a una única base de datos remota; cada sentencia es una transacción.
2. Transacción remota.	Cada sentencia SQL accede a una única base de datos remota; se soportan transacciones multisentencia para una única base de datos.
3. Transacción distribuida.	Cada sentencia SQL accede a una única base de datos remota; se soportan transacciones multisentencia a través de múltiples bases de datos.
4. Petición distribuida.	Las sentencias SQL pueden acceder a múltiples bases de datos; se soportan transacciones multisentencia a través de múltiples bases de datos.

Tabla 20.1. Las cuatro etapas de acceso a una base de datos distribuida propuestas por IBM.

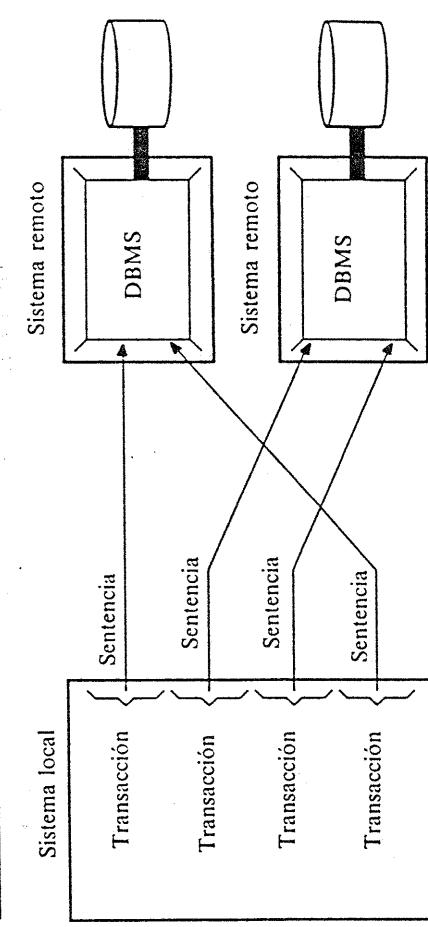


Figura 20.2. Acceso a datos distribuidos: peticiones remotas.

Las peticiones remotas son muy útiles cuando un usuario PC necesita consultar datos corporativos. Generalmente los datos requeridos estarán localizados dentro de una única base de datos, como por ejemplo una base de datos de procesamiento de pedidos o de datos de fabricación. Utilizando una petición remota, el programa del PC puede recuperar los datos remotos para su procesamiento por parte de una hoja de cálculo, un programa gráfico o un paquete de audioedición en el PC.

La capacidad de petición remota no es lo suficientemente potente para la mayoría de las aplicaciones de procesamiento de transacciones. Por ejemplo, consideremos una aplicación de entrada de pedidos basada en PC que acceda a una base de datos corporativa. Para procesar un nuevo pedido, el programa del PC debe comprobar niveles de inventario, añadir el pedido a la base de datos, disminuir los totales de inventario y ajustar los totales de cliente y ventas, implicando quizás media docena de sentencias SQL diferentes. Como se explicó en el Capítulo 11, la integridad de la base de datos puede corromperse si estas sentencias no se ejecutan como una única transacción. Sin embargo, la etapa de petición remota no soporta transacciones multisentencia, por lo que no puede soportar esta aplicación.

La capacidad de petición remota no requiere un DBMS en el sistema solicitante, y algunos vendedores proporcionan productos que utilizan esta configuración. En la familia de productos IBM, Host Data Base View (HDBV) proporciona una interfaz de usuario sobre el PC de IBM para consultar una base de datos DB2 sobre un mainframe, y Extended Connectivity Facility (ECF) permite que un programa PC consulte una base de datos DB2 o SQL/DS. Ingres ofrece su producto, Ingres/PCLink, que proporciona una interfaz de usuario de estilo Lotus para consultar una base de datos Ingres. Tanto Informix como Oracle ofrecen productos adicionales para Lotus 1-2-3 que proporcionan acceso a bases de datos remotas desde dentro de una hoja de cálculo.

La segunda etapa de acceso a datos distribuidos, tal como la define IBM, es una *transacción remota* (llamada «unidad de trabajo remota» por IBM), mostrada en la Figura 20.3. Las transacciones remotas extienden la etapa de petición remota para incluir soporte de transacción multisentencia. El usuario del PC puede emitir una serie de sentencias SQL que consultan o actualizan datos en una base de datos remota y luego cumplimentar o volver atrás la serie entera de sentencias como una única transacción. El DBMS garantiza que la transacción completa tendrá éxito o fallará como una unidad, como ocurre con las transacciones sobre una base de datos local. Sin embargo, todas las sentencias SQL que forman la transacción deben referenciar una única base de datos remota.

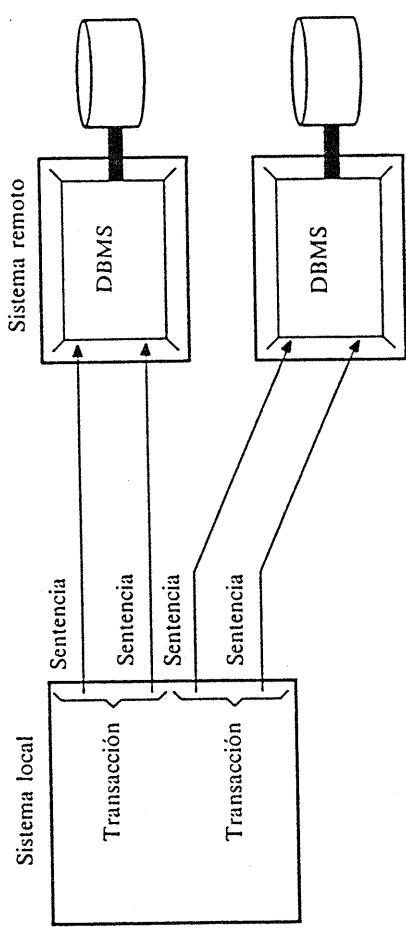


Figura 20.3. Acceso a datos distribuidos: transacciones remotas.

Las transacciones remotas abren la puerta a las aplicaciones de procesamiento de transacciones distribuido. Por ejemplo, en el ejemplo de procesamiento de pedidos descrito en la sección anterior, el programa de PC puede ahora efectuar una secuencia de consultas, actualizaciones e inserciones en la base de datos de inventario para procesar un nuevo pedido. El programa finaliza la secuencia de sentencias con un COMMIT o un ROLLBACK para la transacción.

La capacidad de transacción remota requiere típicamente un DBMS (o al menos una lógica de procesamiento de transacciones) sobre el PC además del sistema donde está localizada la base de datos. La lógica de transacción del DBMS debe extenderse a través de la red para asegurar que los sistemas local y remoto tengan siempre la misma opinión referente a si la transacción ha sido cumplimentada. Sin embargo, la responsabilidad efectiva de mantener la integridad de la base de datos sigue estando en el DBMS remoto.

Algunos vendedores de DBMS y diversas compañías independientes ofrecen productos con soporte de transacción distribuida. El producto SQL Services de Digital Equipment Corporation ofrece acceso programado a las bases de datos Rdb/VMS desde un VAX remoto, PC, Macintosh o estación de trabajo basada en RISC. El CL/1 de Apple Computer permite que un programa de aplicación Macintosh acceda a diversas bases de datos anfitrionas, incluyendo Oracle, Ingres, Sybase, Rdb/VMS, Informix, DB2 y SQL/DS, sobre sistemas VAX de Digital y mainframes de IBM. La arquitectura cliente/servidor de SQL Server se basa en una capacidad de transacción remota, con el usuario localizado en un computador personal y la base de datos localizada a través de un LAN sobre un sistema servidor.

La capacidad de transacción remota también es con frecuencia proporcionada por pasarelas de bases de datos que enlazan un DBMS de un vendedor con otros productos DBMS. Por ejemplo, los productos SQL*Net de Oracle incluyen pasarelas de Oracle a DB2 y a SQL/DS. Ingres proporciona pasarelas similares a DB2 y otros productos de DBMS. Sybase ha anunciado el Open Server, una interfaz de programación de aplicaciones para SQL Server que permite a vendedores de software independientes desarrollar pasarelas desde SQL Server a otros productos DBMS.

Algunos productos pasarelas van más allá de los límites de las transacciones remotas, permitiendo a un usuario componer, en una sola consulta, tablas de una base de datos local con tablas de una base de datos remota gestionada por un producto diferente de DBMS. Sin embargo, estas pasarelas no proporcionan (y no pueden proporcionar) la lógica de transacción subyacente necesaria para soportar las etapas superiores de acceso distribuido tal como las define IBM. La pasarela puede asegurar individualmente la integridad de las bases de datos local y remota, pero no puede garantizar que una transacción no pueda ser cumplimentada en uno y vuelta a atrás en el otro.

Transacciones distribuidas

La tercera etapa de un acceso de datos distribuidos, tal como las define IBM, es una *transacción distribuida* (una «unidad distribuida de trabajo» en la jerga de IBM), mostrada en la Figura 20.4. En esta etapa, cada sentencia SQL individual aún consulta o actualiza una única base de datos sobre un único sistema

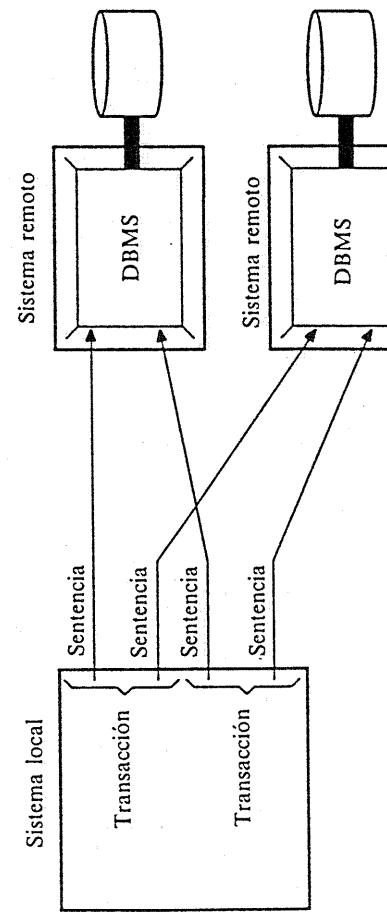


Figura 20.4. Acceso a datos distribuidos: transacciones distribuidas.

informático remoto. Sin embargo, la secuencia de sentencias SQL dentro de una transacción puede acceder a dos o más bases de datos localizadas sobre sistemas diferentes. Cuando la transacción se cumplimenta o se vuelve atrás, el DBMS garantiza que todas las partes de la transacción, sobre todos los sistemas implicados en la transacción, serán cumplimentados o vuelvidos atrás. El DBMS garantiza específicamente que no habrá una «transacción parcial», en donde la transacción se cumplimente en un sistema y vuelta atrás en otro.

Las transacciones distribuidas soportan el desarrollo de aplicaciones de procesamiento de transacciones muy sofisticados. Por ejemplo, en la red corporativa de la Figura 20.1, una aplicación de procesamiento de pedidos en PC puede consultar las bases de datos de inventario en dos o tres sistemas VAX diferentes para comprobar las existencias de un producto escaso y luego actualizar las bases de datos para cumplimentar el inventario desde múltiples posiciones en un pedido de cliente. El DBMS asegura que otros pedidos concurrentes no interferirán con el acceso remoto de la primera transacción.

Las transacciones distribuidas son mucho más difíciles de proporcionar que las dos primeras etapas de acceso a datos distribuidos. Es imposible proporcionar transacciones distribuidas sin la cooperación activa de los sistemas DBMS individuales implicados en la transacción. Un protocolo de transacción especial, llamado protocolo de cumplimentación en *dos fases*, es utilizado generalmente para forzar esta cooperación. Los detalles de este protocolo se describirán posteriormente en este capítulo.

Debido a que las transacciones distribuidas son un paso importante, la mayoría de los vendedores de DBMS han decidido soportarlo en etapas. Tanto Ingres (con Ingres/Star) como Oracle (con SQL*Net) han adoptado este planteamiento, anunciando transacciones distribuidas de sólo lectura para las versiones actuales o próximas a aparecer de su software, y anunciando planes para soportar transacciones distribuidas de lectura/escritura en el futuro. IBM anunció soporte de transacción distribuida limitada en DB2 Versión 2 Edición 2, sacada al mercado en 1989. El DB2 proporciona soporte de transacciones distribuidas de sólo lectura a través de múltiples maincomputadores que ejecutan DB2. También soporta actualizaciones distribuidas, pero restringe las actualizaciones a un único sistema por transacción. Efectivamente, esto significa que DB2 proporciona transacciones distribuidas para consultas, pero solamente transacciones remotas para actualizaciones. El soporte de transacción distribuida es proporcionado únicamente para conexiones DB2-a-DB2, no a través de productos SQL diferentes de IBM.

Sybase y SQL Server también proporcionan transacciones distribuidas, pero ponen el énfasis en la transacción distribuida sobre el programa de aplicación. El programa debe conectar explícitamente dos SQL Server y debe enviar las sentencias SQL adecuadas sobre cada conexión. Cuando sea tiempo de cumplimentar o volver atrás una transacción, las sentencias COMMIT y ROLLBACK estándar no se utilizan. En su lugar, el programa utiliza un conjunto especial de llamadas

de cumplimiento en dos fases en el API de SQL Server para sincronizar las cumplimentaciones de las dos copias de SQL Server. Aunque el esquema proporciona soporte completo para transacciones distribuidas (incluyendo actualizaciones distribuidas), ha sido criticado por su falta de transparencia.

Peticiones distribuidas

La etapa final de acceso a datos distribuidos en el modelo IBM es una petición distribuida, mostrada en la Figura 20.5. En esta etapa, una sola sentencia SQL puede referenciar tablas de dos o más bases de datos localizadas en diferentes sistemas informáticos. El DBMS es responsable de llevar a cabo automáticamente la sentencia a través de la red. Una secuencia de sentencias de petición distribuida pueden agruparse para formar una transacción. Como en la etapa previa de transacción distribuida, el DBMS debe garantizar la integridad de la transacción distribuida sobre todos los sistemas que estén implicados. La etapa de petición distribuida no efectúa ninguna nueva demanda sobre la lógica de procesamiento de transacción DBMS, ya que el DBMS tenía que soportar transacciones a través de fronteras de sistema en la etapa previa de transacción distribuida. Sin embargo, las peticiones distribuidas presentan nuevos retos importantes para la lógica de optimización de DBMS. El optimizador debe considerar ahora la velocidad de red cuando evalúe métodos alternativos para llevar a cabo una sentencia SQL. Si el DBMS local debe acceder repetidamente a parte de una tabla remota (por ejemplo, cuando realiza una composición), puede ser más rápido copiar parte de la tabla a través de la red en una

larga transferencia de bloques en lugar de recuperar repetidamente filas individuales a través de la red.

El optimizar también debe decidir qué copia del DBMS debería manejar la ejecución de la sentencia. Si la mayoría de las tablas están en un sistema remoto, puede ser una buena idea que el DBMS remoto en ese sistema ejecute la sentencia. Sin embargo, puede ser una mala elección si el sistema remoto tiene una fuerte carga de trabajo. Por tanto, la tarea del optimizador es más compleja y mucho más importante en una petición distribuida.

Finalmente, el objetivo de la etapa de petición distribuida es hacer que la base de datos distribuida total se asemeje a una gran base de datos desde el punto de vista del usuario. Idealmente, el usuario debería tener acceso completo a cualquier tabla de la base de datos distribuida y debería poder utilizar transacciones SQL sin necesidad de conocer la ubicación física de los datos. Desgraciadamente, este escenario «ideal» demostraría rápidamente ser poco práctico en redes reales. En una red de cualquier tamaño, el número de tablas de la base de datos distribuida pasaría a ser rápidamente muy grande, y los usuarios encontrarían imposible hallar datos de interés. Los id-usuarios de todas las bases de datos de la organización tendrían que estar coordinados para asegurarse que un id-usuario determinado identificase unívocamente a un usuario en *todas* las bases de datos. La administración de la base de datos también sería muy difícil.

En la práctica, por tanto, las peticiones distribuidas deben ser implementadas selectivamente. Los administradores de bases de datos deben decidir qué tablas remotas van a hacerse visibles a usuarios locales y cuáles permanecerán ocultas. Las copias de DBMS cooperativos deben traducir los id-usuarios de un sistema a otro, permitiendo que cada base de datos sea administrada automáticamente mientras siga proporcionando seguridad para acceso a datos remotos. Las peticiones distribuidas que pudieran consumir demasiados recursos DBMS o de red deben ser detectadas y prohibidas antes que afecten al rendimiento global del DBMS.

Debido a su complejidad, las peticiones distribuidas no son solamente soportadas por ningún DBMS comercial basado en SQL hoy día, y pasará algún tiempo antes de que ni siquiera una mayoría de sus características estén disponibles. Los productos de bases de datos distribuidas de Oracle e Ingres proporcionan capacidad de petición distribuida limitada actualmente. Con ambos sistemas, una tabla en una base de datos remota puede aparecer como si fuera una tabla local, y el usuario puede ejecutar consultas multitable que combinan datos de tablas locales y remotas. Sin embargo, el soporte de transacción completa requerido por la definición IBM de una petición distribuida no es proporcionado, y el acceso multitable está restringido a solamente consultas de bases de

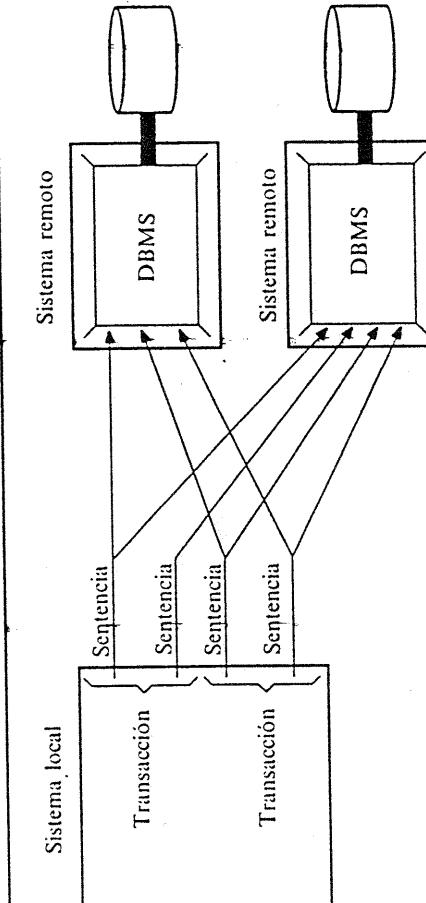


Figura 20.5. Acceso a datos distribuidos: peticiones distribuidas.

Tablas distribuidas

Las etapas definidas por IBM de acceso a datos distribuidos tratan a una tabla de base de datos como una unidad indivisible. Suponen que una tabla está localizada en un único sistema, en una única base de datos, bajo el control de una única copia del DBMS. Cierta investigación sobre bases de datos distribuidas ha relajado esta restricción, permitiendo que una tabla individual esté distribuida a través de dos o más sistemas. Los productos DBMS comerciales no proporcionan aún tablas distribuidas, pero varios de los vendedores de DBMS han afirmado públicamente que proporcionarán esta capacidad en el futuro.

Al principio, la idea de dividir una tabla a través de múltiples sistemas informáticos puede parecer necia o académica. Sin embargo, existen ciertos tipos de aplicaciones donde tiene mucho sentido dividir una tabla. Dos tipos diferentes de división de tabla —horizontal y vertical— son adecuados en situaciones diferentes.

Divisiones horizontales de tabla

Un modo de distinguir una tabla a través de una red es dividir la tabla horizontalmente, colocando diferentes filas de la tabla en diferentes sistemas. La Figura 20.6 muestra un sencillo ejemplo donde es útil una división horizontal de tabla. En esta aplicación, una empresa opera con tres centros de distribución, cada uno con su propio sistema informático y su propio DBMS para gestionar una base de datos de inventario. La mayor parte de la actividad de cada centro de distribución afecta a datos almacenados localmente, pero la empresa tiene una política de satisfacer un pedido de cliente desde *cualquier* centro de distribución si el centro local no dispone de un producto particular.

Para implementar esta política, la tabla PRODUCTOS está dividida horizontalmente en tres partes y se expande para incluir una columna POBLACION que dice dónde está localizado el inventario. Las filas de la tabla que describen el inventario de cada centro de distribución están almacenadas localmente y gestionadas por el DBMS de ese centro. Sin embargo, para propósitos de procesamiento de pedidos, la tabla PRODUCTOS es una gran tabla, que puede ser consultada y actualizada como una unidad.

Las tablas horizontalmente divididas requieren cambios importantes en el DBMS y en su lógica de optimización. Consideremos estas tres consultas de la tabla PRODUCTOS que son ejecutadas en el sistema localizado en New York:

```
SELECT * FROM PRODUCTOS WHERE LOCALIDAD = 'New York'
AND ID_FAB = 'ACI'
```

Halla todos los productos construidos por ACI en el centro de distribución de Chicago.

```
SELECT * FROM PRODUCTOS WHERE LOCALIDAD = 'Chicago'
AND ID_FAB = 'ACI'
```

Halla todos los productos fabricados por ACI.

```
SELECT * FROM PRODUCTOS WHERE ID_FAB = 'ACI'
```

Las tres consultas parecen ser muy similares. Sin embargo, la primera puede ser manejada localmente, la segunda requiere una petición remota y la tercera requiere una petición distribuida que se extiende a los tres sistemas. El DBMS debe manejar las tres consultas de forma muy diferente.

Las actualizaciones de la base de datos también requieren un manejo especial por parte del DBMS. Por ejemplo, cuando el DBMS procesa una sentencia INSERT tal como ésta:

```
INSERT INTO PRODUCTOS (ID_FAB, ID_PRODUCTO, LOCALIDAD, ...)
```

```
VALUES ('ACI', '41004', 'New York', ...)
```

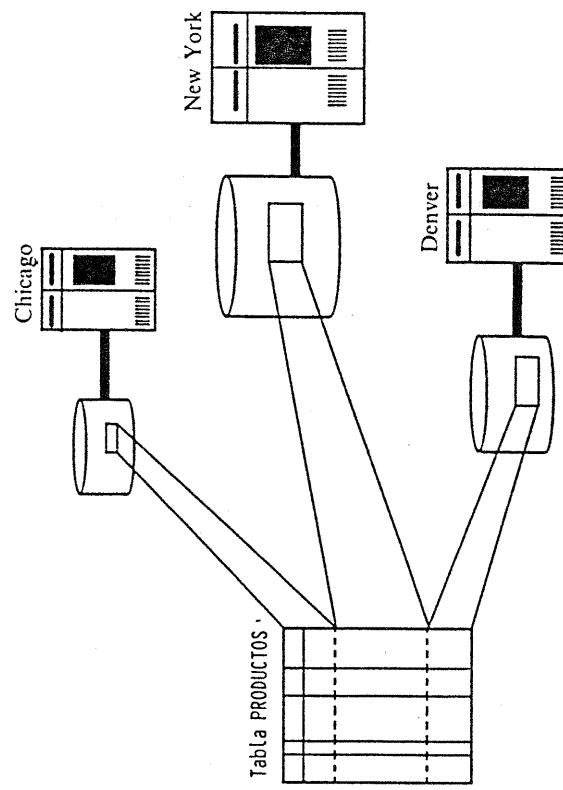


Figura 20.6. Una tabla distribuida dividida horizontalmente.

debe decidir en qué sistema debería almacenarse la nueva fila. Además, una actualización de la fila tal como ésta:

```
UPDATE PRODUCTOS SET LOGICO = 'Población'
SET OFICINA = 'Chicago'
WHERE FAB = 'ACI'
AND PRODUCTO = '41004'
```

provocaría realmente que el DBMS suprimiera una fila en un sistema y la insertara en otro. Como muestran estas sentencias, el DBMS debe tener conocimiento detallado de la tabla dividida y sus contenidos para manejar eficientemente las peticiones de SQL.

Divisiones verticales de tabla

Otro modo de distribuir una tabla a través de una red es dividir la tabla verticalmente, colocando diferentes columnas de la tabla en diferentes sistemas. La Figura 20.7 muestra un sencillo ejemplo de una división vertical de tabla. La tabla REPVENTAS ha sido ampliada para incluir nuevas columnas de información del personal (número de teléfono, estado civil, etc.) y ahora la utiliza tanto el departamento de procesamiento de pedidos como el departamento de personal,

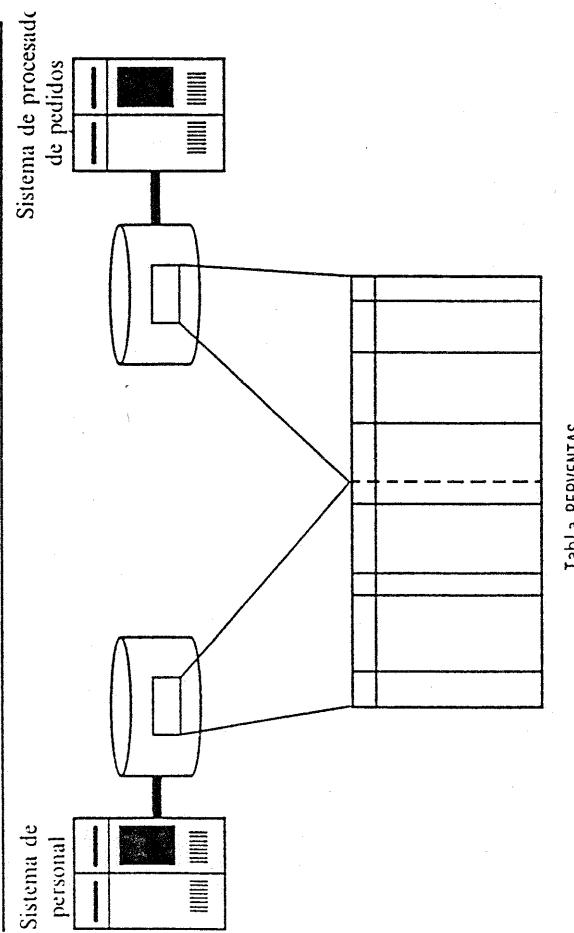


Figura 20.7. Una tabla dividida verticalmente.

cada uno de los cuales tiene su propio sistema informático y su propio DBMS. La mayor parte de la actividad de cada departamento se centra en una o dos columnas de la tabla, pero hay muchas consultas e informes que utilizan las columnas relativas al personal y a los pedidos.

Para almacenar los datos de los vendedores, la tabla REPVENTAS está dividida verticalmente en dos partes. Las columnas de la tabla que almacenan datos del personal (NOMBRE, EDAD, CONTRATO, TELÉFONO, CASADO) están almacenadas en el sistema de personal y gestionadas por su DBMS. Las otras columnas (NUM_EMPL, CUOTA, VENTAS, OFICINAREP) están almacenadas en el sistema de procesamiento de pedidos y gestionadas por su DBMS. Sin embargo, la tabla REPVENTAS es tratada como una tabla grande, que puede ser consultada y actualizada como una unidad.

Observe que estas dos consultas de la tabla REPVENTAS parecen muy similares, pero una implica acceso local, y la otra acceso remoto:

Halla los nombres de todos los vendedores mayores de 30 años.

```
SELECT NOMBRE, EDAD
FROM REPVENTAS
WHERE EDAD > 30
```

Muestra el rendimiento de cuotas de todos los vendedores de Chicago.

```
SELECT NUM_EMPL, VENTAS, CUOTA
FROM REPVENTAS
WHERE OFICINAREP = 12
```

La siguiente consulta requiere que el DBMS recoja partes de cada fila de los dos sistemas diferentes:

```
Muestra el rendimiento de cuota de todos los vendedores de Chicago.
SELECT NOMBRE, VENTAS, CUOTA
FROM REPVENTAS
WHERE OFICINAREP = 12
```

Con una división vertical de tabla, el DBMS no tiene que preocuparse de los contenidos de las filas para determinar dónde están localizados los datos. Sin embargo, tiene el problema igualmente difícil de mantener una relación uno a uno entre las filas parciales localizadas en dos o más sistemas diferentes. Además, las tablas verticalmente divididas pueden requerir cambios importantes en la lógica de cierre del DBMS. Si un programa necesita cerrar una fila, el DBMS debe ahora cerrar partes de la fila en más de un sistema, ¿Debería el DBMS cerrar todas las partes de la fila, o solamente las que están siendo realmente actualizadas? Dependiendo de la decisión, el DBMS puede necesitar implementar cierre a nivel de elemento.

Tablas reflejadas

El diseño de una base de datos distribuida debería seguir idealmente una *regla 80/20*: el 80 por 100 (o más) de las peticiones de acceso a la base de datos debería satisfacerse localmente, y sólo el 20 por 100 (o menos) debería requerir acceso remoto. Sin embargo, si una o dos de las tablas de una base de datos distribuida son fuertemente utilizadas por muchos usuarios en sistemas diferentes, puede ser difícil lograr este equilibrio 80/20. No importa dónde estén localizadas esas tablas, el acceso a ellas por parte de usuarios de otros sistemas generará tráfico de red y retardos.

Una solución a este problema es el uso de una *tabla reflejada*, donde copias duplicadas de una tabla se almacenan en varios sistemas, tal como muestra la Figura 20.8. Cuando un usuario ejecuta una consulta que referencia a la tabla, puede utilizarse la copia local de la tabla, reduciendo el tráfico de red y elevando la productividad de la base de datos. Sin embargo, las actualizaciones de la tabla reflejada deben estar sincronizadas de modo que todas las copias de la tabla se actualicen al mismo tiempo, para prevenir que los usuarios de diferentes sistemas utilicen versiones inconsistentes de la tabla. Por esta razón, las tablas reflejadas son prácticas únicamente para tablas que tienen acceso preferente de lectura y que son actualizadas con poca frecuencia. En la base de datos ejemplo,

la tabla CLIENTES podría ser un buen candidato para ser reflejada, pero la tabla PEDIDOS no sería un buen candidato.

El protocolo de cumplimentación en dos fases*

Un DBMS distribuido debe preservar la calidad «todo o nada» de una transacción SQL si pretende proporcionar transacciones distribuidas. El usuario del DBMS distribuido espera que una transacción cumplimentada esté cumplimentada en todos los sistemas en donde residen los datos, y que una transacción vuelta atrás sea vuelta atrás en todos los sistemas también. Además, los fallos en una conexión de red o en uno de los sistemas debería provocar que el DBMS abortara una transacción y la volviera atrás, en lugar de dejar la transacción en un estado parcialmente cumplimentado.

Todos los sistemas DBMS comerciales que soportan o planean soportar transacciones distribuidas utilizan una técnica llamada *cumplimentación en dos fases* para proporcionar este soporte. No es preciso entender el esquema de cumplimentación en dos fases para utilizar transacciones distribuidas. De hecho, lo que el esquema pretende es soportar transacciones distribuidas sin necesidad de su conocimiento. Sin embargo, la comprensión de la mecánica de una cumplimentación en dos fases puede ayudar a planear un acceso eficiente a la base de datos.

Para entender por qué es necesario un protocolo especial de cumplimentación en dos fases, consideremos la base de datos de la Figura 20.9. El usuario, localizado en el Sistema A, ha actualizado una tabla en el Sistema B y una tabla en el Sistema C, y ahora desea cumplimentar la transacción. Supongamos que el software DBMS del Sistema A tratase de cumplimentar la transacción enviando simplemente un mensaje COMMIT a los Sistemas B y C, y luego esperando respuestas afirmativas. Esta estrategia funciona siempre que los Sistemas B y C puedan ambos cumplimentar con éxito su parte de la transacción. Pero ¿qué sucede si un problema como el fallo de un disco o una condición de interbloqueo impide al Sistema C cumplimentar lo que se le solicita? El Sistema B cumplimentaría su parte de la transacción y enviaría un reconocimiento de vuelta, el Sistema C volvería atrás su parte de la transacción debido al error y devolvería un mensaje de error y el usuario acabaría con una transacción parcialmente cumplimentada y parcialmente vuelta atrás. Observe que el Sistema A no puede «cambiar de idea» en este momento y pedir al Sistema B que vuelva atrás la transacción. La transacción en el Sistema B ha sido cumplimentada, y otros usuarios pueden ya haber modificado los datos del Sistema B basándose en los cambios efectuados por la transacción.

El protocolo de cumplimentación en dos fases elimina los problemas de la sencilla estrategia mostrada en la Figura 20.9. La Figura 20.10 ilustra los pasos implicados en una cumplimentación en dos fases:

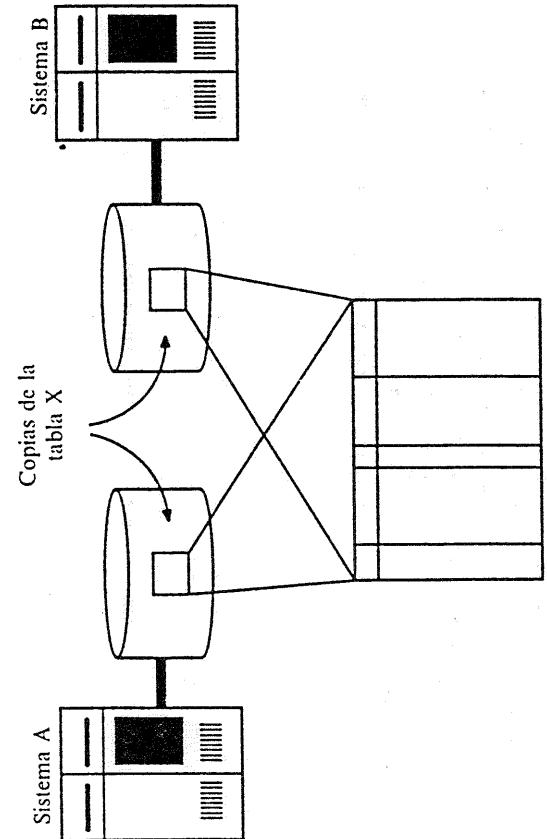


Figura 20.8. Una tabla reflejada.

Figura 20.9. Una tabla reflejada.

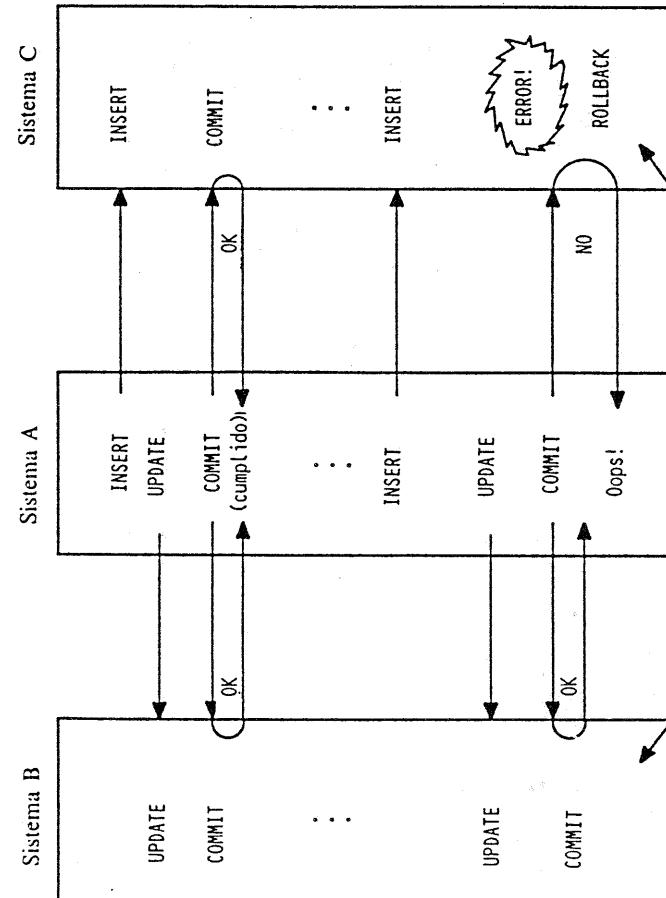


Figura 20.9. Problemas con un esquema de cumplimentación «dijinidio».

- El programa del Sistema A emite un COMMIT para la transacción (distribuida) actual, que ha actualizado tablas en el Sistema B y el Sistema C. El sistema A actuará como *coordinador* del proceso de cumplimentación, coordinando las actividades del software DBMS en los Sistemas B y C.
- El Sistema A envía un mensaje GET READY a los Sistemas B y C, y anota el mensaje en su propio registro de transacciones.
- Cuando el DBMS del Sistema B o C recibe el mensaje GET READY, debe prepararse para cumplimentar o para volver atrás la transacción actual. Si el DBMS puede pasar a este estado de «listo para cumplimentar», replica YES al Sistema A y anota este hecho en su registro de transacciones locales; si no puede pasar a este estado, replica NO.

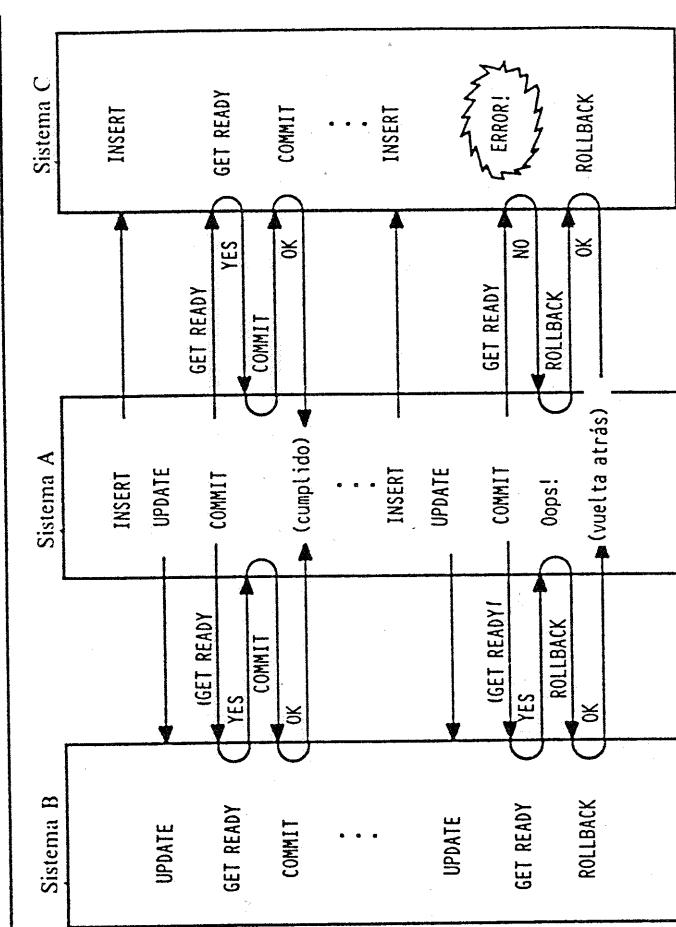


Figura 20.10. El protocolo de cumplimentación en dos fases.

- El Sistema A espera las contestaciones a su mensaje GET READY. Si todas las réplicas son YES, el Sistema A envía un mensaje COMMIT a los Sistemas B y C, y anota la decisión en su registro de transacciones. Si alguna de las réplicas es NO, o si todas las réplicas no se reciben antes de que se cumpla un cierto plazo de tiempo, el Sistema A envía un mensaje ROLLBACK a ambos sistemas y anota esa decisión en su registro de transacciones.
- Cuando el DBMS de los Sistemas B o C recibe el mensaje COMMIT o ROLLBACK, debe hacer lo que se le dice. El DBMS cede la capacidad de decidir el destino de la transacción autónomamente cuando replicó YES al mensaje GET READY del Paso 3. El DBMS cumplimenta o vuelve atrás su parte de la transacción según se le indica, escribe el mensaje COMMIT o ROLLBACK en su registro de transacciones, y devuelve un mensaje OK al Sistema A.
- Cuando el Sistema A ha recibido todos los mensajes OK, sabe que la transacción ha sido cumplimentada o vuelta atrás y devuelve el valor SQLCODE adecuado al programa.

Resumen

El protocolo protege la transacción distribuida frente a cualquier fallo simple en el Sistema B, el Sistema C o la red de comunicaciones. Estos dos ejemplos ilustran cómo el protocolo permite recuperación de fallos:

- Supongamos que un fallo ocurre en el Sistema C antes de que envíe un mensaje YES en el Paso 3. El Sistema A no recibirá una réplica YES y difundirá un mensaje ROLLBACK, haciendo que el Sistema B vuelva atrás la transacción. El programa de recuperación en el Sistema C no encontrará el mensaje YES o un mensaje COMMIT en el registro de transacción local, y volverá atrás la transacción en el Sistema C como parte del proceso de recuperación. Todas las partes de la transacción habrán dado marcha atrás en este punto.
- Supongamos que ocurre un fallo en el Sistema C después que envíe un mensaje YES en el Paso 3. El Sistema A decidirá si cumplimentar o volver atrás la transacción distribuida basándose en la réplica del Sistema B. El programa de recuperación del Sistema C encontrará el mensaje YES en el registro de transacciones local, pero no encontrará un mensaje COMMIT o ROLLBACK para marcar el final de la transacción. El programa de recuperación pide entonces al coordinador (el Sistema A) cuál fue la disposición final de la transacción y actúa adecuadamente. Observe que el Sistema A debe mantener un registro de sus decisiones de cumplimentar o volver atrás la transacción hasta que reciba el OK final de todos los participantes, de modo que pueda responder al programa de recuperación en caso de fallo.

El protocolo de cumplimentación en dos fases garantiza la integridad de las transacciones distribuidas, pero genera una gran cantidad de tráfico de red. Si hubiera n sistemas implicados en la transacción, el coordinador debería enviar y recibir un total de $(4 * n)$ mensajes para cumplimentar con éxito la transacción. Observe que estos mensajes son *además* de los mensajes que realmente transmiten las sentencias SQL y los resultados de las consultas entre los sistemas. Sin embargo, no hay modo de evitar el tráfico de mensajes si una transacción distribuida debe proporcionar integridad de la base de datos frente a fallos de sistema.

Debido a su fuerte recargo de red, las transacciones distribuidas pueden tener un efecto negativo serio sobre el rendimiento de la base de datos. Por esta razón, las bases de datos cuidadosamente diseñadas para que los datos frecuentemente accedidos (o al menos frecuentemente actualizados) se encuentren en un sistema local o en un único sistema remoto. Si es posible, las transacciones que actualizan dos o más sistemas remotos deberían ser una ocurrencia relativamente rara.

Este capítulo ha descrito las capacidades de gestión de datos distribuidos ofrecidas por varios productos DBMS y los compromisos implicados al proporcionar acceso a datos remotos:

- Una base de datos distribuida se implementa típicamente mediante una red de sistemas informáticos, cada uno ejecutando su propia copia del software DBMS y operando autónomamente para acceso a datos locales. Las copias del DBMS cooperan para proporcionar acceso a datos remotos cuando es preciso.
- La base de datos distribuida «ideal» es aquélla en la que el usuario no conoce ni se preocupa de que los datos estén distribuidos; para el usuario, todos los datos relevantes aparecen como si estuvieran en el sistema local.
- Debido a que este DBMS distribuido ideal es muy difícil (y quizás imposible) de lograr, los productos DBMS comerciales proporcionan capacidad de base de datos distribuida en fases.
- IBM ha definido cuatro etapas de acceso a datos remotos: peticiones remotas, transacciones remotas, transacciones distribuidas y peticiones distribuidas, proporcionando cada etapa capacidad más amplia que la anterior. Las definiciones de IBM son muy rigurosas, y ninguno de los productos DBMS de IBM ha alcanzado totalmente la tercera etapa.
- Otros vendedores DBMS, tales como Oracle e Ingres, han proporcionado acceso remoto a múltiples bases de datos dentro de una sola sentencia, pero o han limitado las capacidades de actualización de la sentencia o no han proporcionado soporte total de transacciones para transacciones distribuidas.
- La capacidad de redistribuir tablas individuales a través de múltiples sistemas en una red puede ser útil en ciertas aplicaciones, pero esta capacidad está aún en etapa de investigación para la mayoría de los vendedores de DBMS.
- Las bases de datos distribuidas presentan serios retos de implementación para un vendedor de DBMS, especialmente en las áreas de optimización, cierre y gestión de transacciones. En estas áreas, el DBMS debe reconocer los diferentes sistemas y redes, y debe utilizar técnicas especiales tales como las cumplimentaciones en dos fases para soportar el acceso a datos remotos.

21 *El futuro de SQL*

SQL es una de las tecnologías más importantes que conducen y conforman el mercado informático hoy día. Desde sus primeras implementaciones comerciales hace poco más de una década, SQL ha crecido hasta convertirse en el lenguaje estándar de bases de datos. Con el respaldo de IBM, la bendición de las corporaciones de estándares y el soporte entusiasta de los vendedores de DBMS, la evidencia muestra claramente la importancia de SQL:

- Las bases de datos insignia de IBM, en sistemas de procesadores mainframe hasta computadores personales, están todas basadas en SQL.
- Los principales vendedores de DBMS independientes y la mayoría de los fabricantes de hardware de computador ofrecen o planean ofrecer productos basados en SQL.
- Las bases de datos basadas en SQL están comenzando a dominar en virtualmente todos los segmentos de mercado, desde los ordenadores personales a los sistemas OLTP.
- El más importante vendedor de bases de datos para PC, Ashton-Tate, se ha inclinado ante la presión del mercado y ha injertado SQL en su producto dBASE IV.

Este capítulo describe las tendencias actuales en el mercado de las bases de datos y el impacto que tendrán en SQL y en el mercado informático de la década de los noventa.

Tendencia del mercado de bases de datos

Al comienzo de la década de los noventa, el tamaño del mercado para los productos de gestión de bases de datos traspasó la marca de los tres mil millones de dólares, y las previsiones dicen que continuará este crecimiento a lo largo de la década. Al mismo tiempo, las divisiones netas que habían caracterizado el mercado de bases de datos de los ochenta fueron desapareciendo rápidamente. Los productos DBMS no podían seguir categorizados claramente como «bases de datos para PC», «bases de datos para minicomputadores», «bases de datos para OLTP», «bases de datos para maxicomputadores», etc., cada una participando en su propio segmento de mercado. En vez de ello, la normalización de SQL y del modelo de base de datos relacional creó un nuevo mercado de bases de datos más unificado.

Estos cambios en el mercado han producido alteraciones dramáticas en las fortunas y en las estrategias de mercado de los vendedores de DBMS. Para tener éxito, un vendedor de DBMS debe proporcionar soporte SQL y luego proporcionar valor añadido adicional encima de SQL. Los vendedores establecidos de DBMS cuyos productos no están basados en SQL están barajando si proporcionar soporte SQL o mantener su base instalada de sistema. Las experiencias de Ashton-Tate en el mercado de PC y de Cullinet en el mercado de maxicomputadores muestran cómo este fenómeno ha atravesado todos los segmentos del mercado de la base de datos. Mientras tanto, los vendedores de DBMS que ya disponen de productos basados en SQL están aprovechando la oportunidad, atacando a los vendedores establecidos y atacándose entre sí.

Las líneas de batalla están particularmente bien trazadas en dos segmentos del mercado informático. En el segmento de procesamiento de transacciones, los principales vendedores de hardware (como IBM, Digital y Tandem) han introducido todos sistemas OLTP basados en bases de datos relacionales (DB2, Rdb/VMS y NonStop SQL, respectivamente) compitiendo con vendedores de DBMS independientes (como Oracle, Ingres y Sybase) que elogian las versiones de procesamiento de transacciones de sus productos. En el segmento del computador personal, la llegada de OS/2 ha producido un choque entre el gigante hardware (IBM, con OS/2 Extended Edition), una colección de gigantes de los computadores personales (Microsoft, Ashton-Tate y Lotus con SQL Server) y el más importante vendedor de DBMS independiente (Oracle con Oracle Server para OS/2) por el dominio de la sobremesa.

Como muestra la experiencia de estos dos segmentos de mercado, la elevación de SQL ha producido una dramática reorientación del mercado de base de datos, que aún está en progreso. La competición basada en el segmento de mercado está dando lugar a la competición basada en otros factores, incluyendo:

- características DBMS y extensiones SQL;

■ rendimiento de procesamiento de transacciones;

- portabilidad a través de múltiples plataformas hardware;
- reputación del vendedor de DBMS;
- canales de distribución;
- soporte y servicio.

Con el cambiante panorama competitivo, parece muy probable que algunos de los principales vendedores de DBMS al comienzo de la década no continúan siendo líderes al final de la década.

Tendencias del rendimiento hardware

Uno de los contribuidores más importantes a la elevación de SQL durante los ochenta fue un incremento dramático en el rendimiento de las bases de datos relacionales. Parte de este incremento de rendimiento fue debido a avances en la tecnología de las bases de datos y la optimización de las consultas. Sin embargo, la mayoría de la mejora en el rendimiento del DBMS provino de ganancias en la potencia de procesamiento puro de los sistemas informáticos subyacentes. Por ejemplo, con la introducción de DB2 Versión 2, IBM proclamó un rendimiento de 270 transacciones por segundo (tps), una elevación del 120 por 100 con respecto a las 123 tps de la revisión anterior de DB2. Sin embargo, un examen más detallado de los resultados de los programas de prueba mostraron que el rendimiento del software DB2 se mejoró únicamente un 51 por 100, desde 123 tps a 186 tps. El 69 por 100 restante de la elevación del rendimiento fue debido al mainframe más potente sobre el cual se habían conducido los tests del nuevo software DB2.

Las mejoras de rendimiento en sistemas informáticos fueron seguidas en el mercado de minicomputadores, donde el VAX 11/780 fue reemplazado por VAXes con 10 ó 20 veces más potencia, y en el mercado de computadores personales, donde el Intel 8086 dio paso al 80286, 80386 y 80486. Estos avances en el hardware vencieron el recargo mayor del modelo de datos relacional y proporcionaron rendimiento comparable a las bases de datos no relacionales de una década antes.

Las mejoras en el precio/rendimiento del hardware informático deberían continuar sin disminución a través de los noventa, proporcionando continuas mejoras de rendimiento para los productos de DBMS relacional. Algunos de los avances más importantes provienen de los sistemas multiprocesadores, donde dos, cuatro, ocho o más procesadores operan en paralelo, compartiendo la carga de trabajo de procesamiento. Una arquitectura multiprocesador es especialmente adecuada para aplicaciones OLTP, donde la carga de trabajo consiste en

muchas pequeñas transacciones paralelas de bases de datos. Los vendedores de OLTP tradicionales, como Tandem, han utilizado siempre una arquitectura multiprocesadora, pero los sistemas multiprocesadores están ahora pasando a formar parte del flujo principal de procesamiento de datos. Los mayores sistemas mainframe ya utilizan un diseño multiprocesador, y los sistemas multiprocesadores se están convirtiendo en una parte importante de la línea de productos de los minicomputadores VAX. Estos desarrollos harán descender el coste de los sistemas multiprocesadores, mejorando aún más la relación precio/rendimiento de los sistemas de bases de datos relacionales.

Las arquitecturas multiprocesadoras presentan un dilema especial para los vendedores de DBMS que evitan el sistema operativo en un intento de mejorar el rendimiento de la base de datos. El DBMS Sybase, por ejemplo, opera como único proceso y toma la responsabilidad de su propia gestión de tareas, manejo de eventos y entrada/salida —funciones que son generalmente gestionadas por un sistema operativo tal como UNIX o VMS—. Aunque esto mejora el rendimiento en un sistema monoprocesador, impide al sistema operativo extender automáticamente la carga de trabajo de Sybase a través de múltiples procesadores. Para cosechar las ventajas de un sistema multiprocesador, el software DBMS Sybase debe ser reescrito, duplicando la lógica que ya está en el sistema operativo. En contraste, los productos DBMS que no evitan el sistema operativo pueden beneficiarse automáticamente del entorno multiprocesador. Oracle e Ingres utilizan ambos este argumento para defender sus propias arquitecturas, y apuntan a resultados de bancos de prueba sobre sistemas multiprocesadores como prueba de su programa.

Varias empresas combinan los microprocesadores de alto rendimiento, las unidades de disco rápidas y las arquitecturas multiprocesadoras para construir sistemas dedicados que están optimizados como servidores de base de datos. Estos vendedores arguyen que pueden entregar mucho mejor rendimiento de base de datos con una máquina especialmente diseñada que con un sistema informático de propósito general. En algunos casos, sus sistemas incluyen circuitos integrados de aplicación específica (ASIC: application specific integrated circuits) que implementa a parte de la lógica DBMS en hardware para maximizar la velocidad.

Sistemas de base de datos dedicados procedentes de empresas tales como Teradata y Sharebase (antes Britton-Lee) han encontrado cierta aceptación en aplicaciones que implican consultas complejas sobre bases de datos muy grandes. Sin embargo, no han resultado una parte importante de la corriente principal del mercado de base de datos. La historia demuestra que otros sistemas específicos de aplicación (como los procesadores de texto) han tenido momentos muy difíciles guardando el paso de los avances técnicos y las economías de escala en el mercado informático de propósito general. Los sistemas de bases de datos dedicadas tendrán que vencer estos formidables obstáculos si pretenden jugar algo más que un papel residual.

La guerra de los bancos de prueba

Las bases de datos relacionales y SQL han pasado a ser parte del flujo principal del procesamiento de datos, y están siendo utilizados para implementar serias aplicaciones de procesamiento de datos. Como resultado, los usuarios están prestando más atención al rendimiento de los productos SQL para asegurarse que los DBMS elegidos no «se quedan sin gas» cuando se amplíen sus aplicaciones. Esta preocupación del usuario, junto con el interés de los vendedores de DBMS por el lucrativo mercado de procesamiento de transacciones, ha producido «guerras de bancos de pruebas» entre vendedores DBMS. Virtualmente todos los vendedores DBMS se han unido a la disputa, cada uno proclamando bancos de prueba que muestran el superior rendimiento de sus productos mientras a la vez tratan de desacreditar los bancos de prueba de los otros vendedores.

Desgraciadamente, la mayoría de los bancos de prueba y los datos resultantes están diseñados para soportar las proclamas de marketing del vendedor en vez de iluminar a los usuarios acerca del rendimiento del DBMS en aplicaciones reales. La mayoría de los bancos de prueba son propiedad de los vendedores, pero han aparecido dos bancos de prueba independientes de los vendedores. El banco de prueba Debit/Credit simula simples transacciones de contabilidad. El banco de prueba TP1, definido inicialmente por Tandem en 1985, mide una mezcla de transacciones de OLTP típicas.

En una serie de resultados de bancos de prueba en 1989, un minicomputador Tandem multiprocesador especializado en aplicaciones OLTP conseguía de 40 a 50 tps en un banco de prueba TP1. El rendimiento TP1 de Oracle en sistemas VAX de medio rango iba de 30 a 40 tps. El rendimiento de Sybase en sistemas VAX de medio rango estaba también entre 30 y 40 tps. SQL Server ejecutado sobre un PC Compaq 386/33 conseguía de 8 a 12 tps, ligeramente mejor que el rendimiento de 5 a 10 tps de Sybase en un MicroVAX II, un minicomputador VAX de la gama baja. Este último resultado da cierta credibilidad a la proclama de que las redes de área local PC se han convertido en una alternativa viable a los minicomputadores para aplicaciones de procesamiento de datos.

Todas las indicaciones apuntan a una escalada continua de las guerras de banco de pruebas durante los próximos años. Los sistemas informáticos con múltiples procesadores, servidores fuertemente acoplados, mejoras en las técnicas de «caching» de base de datos y otras tendencias continuarán empujando las proclamas de rendimiento de los vendedores de DBMS cada vez más alto. Por ejemplo, Oracle ya ha proclamado un rendimiento de hasta miles de transacciones por segundo en un NCUBE, un sistema paralelo masivo que conecta hasta 8192 procesadores en paralelo.

En un intento de traer más estabilidad y significado a los datos de los bancos de prueba, varios vendedores y consultores de bases de datos se han agrupado para producir un banco de pruebas estándar para base de datos relacional que pudiera permitir comparaciones significativas entre varios productos DBMS.

Sin embargo, hasta la fecha, este grupo ha tenido poco impacto real en las pruebas de las bases de datos. Otro fenómeno actualmente en alza es el uso de consultores independientes para «validar» los datos de los bancos de prueba, proporcionando un aura de independencia y respetabilidad a las proclamas de los vendedores de DBMS. Desgraciadamente, el vendedor de DBMS paga generalmente al consultor de base de datos por este servicio, y muchos de los consultores ganan tasas sustanciales por sus trabajos para los vendedores de DBMS, creando un conflicto de intereses.

A pesar de las proclamas de los vendedores acerca del rendimiento de OLTP, hay dudas persistentes acerca de la adecuación de SQL para aplicaciones de procesamiento de transacciones de elevado volumen. El mejor consejo para los compradores de un DBMS basado en SQL es ser escéptico respecto a las proclamas de bancos de pruebas de vendedores y hacer preguntas respecto a cómo fue efectuado el banco de pruebas, en qué tipo de hardware, y por quién. Para información real sobre cómo actuará un DBMS en una aplicación, el mejor dato que puede ser obtenido es mediante creación de prototipos de aplicación y su sometimiento por banco de pruebas a datos propios. Una discusión con otro usuario que haya implementado una aplicación análoga utilizando el DBMS propuesto puede también proporcionar información útil.

Productos DBMS en grupos

Con el desvelamiento de su estrategia SAA, IBM mostró que considera una base de datos relacional como un producto estratégico que es lógicamente una extensión de un sistema operativo del computador. Los ofrecimientos de productos IBM soportan este punto de vista. DB2 opera como un subsistema del sistema operativo MVS en mainframe computadores IBM, y el sistema operativo OS/400 para los sistemas AS/400 de gama media incluye un DBMS relacional integrado. Con OS/2, IBM afirmó este punto aún más explícitamente combinando un DMBS relacional con funciones de comunicaciones claves y llamado al resultado OS/2 Extended Edition.

Otros vendedores de sistemas informáticos, independientemente y en respuesta a IBM, han concluido también que las bases de datos relacionales son una tecnología estratégica, y ofrecen ahora productos DBMS propietarios del vendedor para sus sistemas:

- Digital Equipment Corporation ha gastado millones de dólares en el desarrollo de Rdb/VMS y VAX SQL, sus productos DBMS relacional para la familia de sistemas VAX/VMS.
- Hewlett-Packard ha invertido fuertemente en Allbase, un DBMS híbrido que soporta HPSQL, una interfaz relacional, y también proporciona compatibilidad con el producto SQL actual es totalmente compatible con el estándar ANSI/ISO, aunque muchos de los productos están lentamente moviéndose hacia la

dad retroactiva con Image, un DBMS de HP popular basado en el modelo de datos en red.

- Tandem ha desarrollado el NonStop SQL, un DBMS relacional que soporta reflejo de disco y otras características especiales para su familia tolerante a fallos de sistema OLTP.
- Data General ofrece DG/SQL, una implementación de SQL para su familia MV de sistemas minicomputadores.

Digital ha adoptado la postura más agresiva de todos estos vendedores de computadores agrupando los módulos de ejecución de Rdb/VMS con cada copia del sistema operativo VAX/VMS. Este paso ha generado una fuerte crítica por parte de los vendedores de DBMS independientes, que acusan a Digital de utilizar su posición como vendedor de hardware para obtener una ventaja competitiva injusta para lo que ellos proclaman que es un producto mediocre. De hecho, los vendedores de DBMS independientes han continuado prosperando sobre plataformas VAX a pesar de la decisión de Digital de adjuntar Rdb/VMS.

El paso de agrupamiento de Digital y el empaquetamiento de SQL de IBM a través de su línea de productos indica la nueva importancia que los vendedores de computadores están asociando a la tecnología de bases de datos relacionales y a SQL. En el futuro, estos vendedores pueden mejorar aún más el rendimiento de sus productos SQL propios mediante la construcción del soporte de base de datos directamente en su hardware y en los sistemas operativos. En respuesta, los vendedores de DBMS independientes continuarán ofreciendo nuevas características innovadoras, portabilidad a través de múltiples plataformas hardware e independencia de cualquier vendedor particular de computadores. Parece por tanto probable que la relación simbiótica entre vendedores de computadores y vendedores de DBMS independientes que prevaleció en los ochenta será reemplazada por un creciente conflicto en los noventa.

Estándares SQL

La adopción de un estándar SQL ANSI/ISO oficial fue uno de los factores importantes que aseguraron el lugar de SQL como el lenguaje estándar de base de datos relacional en los ochenta. La conformidad con el estándar ANSI/ISO se ha convertido en una exigencia en la evaluación de productos DBMS, de modo que cada vendedor de DBMS proclama que su producto es «compatible con» o «basado en» el estándar ANSI/ISO. A pesar de estas afirmaciones,

compatibilidad. Por ejemplo, las más recientes versiones de DB2 y Oracle contienen mínimos cambios que se efectuaron para satisfacer la conformidad con el estándar. Esta tendencia debería continuar durante los primeros años de los noventa, creando una lenta convergencia hacia el estándar ANSI/ISO.

Como se discutió en el Capítulo 3, el estándar ANSI/ISO es relativamente débil, con muchas omisiones y áreas que se dejan como opciones de implementación. Durante los últimos pocos años, el comité de estándares ha estado trabajando en un estándar SQL2 ampliado que remedia esta debilidad y amplia significativamente el lenguaje SQL. A diferencia del primer estándar SQL, que especificaba características que ya estaban disponibles en la mayoría de los productos SQL, el estándar SQL2 es un intento de liderar más que seguir al mercado. Especifica características y funciones que aún no están ampliamente implementadas en los productos DBMS actuales, tales como cursor de deslizamiento, catálogos de sistema normalizados, un uso mucho más amplio de subconsultas y un nuevo esquema de mensajes de error. Queda por ver si los vendedores de DBMS verán ventajas más competitivas en adoptar estas nuevas características o en continuar extendiendo SQL en sus propias direcciones.

Además del estándar SQL ANSI/ISO, el estándar SQL SAA de IBM tendrá una poderosa influencia sobre SQL en los noventa. Como creador de SQL y como el mayor vendedor de software DBMS del mundo, las decisiones de IBM respecto a SQL han tenido siempre un fuerte impacto sobre los restantes vendedores de productos SQL. Cuando los dialectos SQL IBM y ANSI han diferido en el pasado, la mayoría de los vendedores de DBMS independientes han optado por seguir el estándar IBM. Es muy probable que se tome la misma decisión en el futuro.

Mientras que los estándares ANSI/ISO y SAA de IBM se centran en el propio lenguaje SQL, otros grupos han estado trabajando en estandarizar la interfaz de programación de aplicaciones a SQL. El estándar Remote Data Access (RDA) desarrollado por ISO entra dentro de esta categoría; es un intento de estandarizar el acceso a la base de datos a través de una red. El trabajo de RDA ha progresado muy lentamente, con considerable definición del estándar propuesto durante los últimos años. En 1989, un grupo de vendedores de bases de datos, que se llaman a sí mismos el SQL Access Group, se reunieron para desarrollar su propio estándar para una interfaz del programa de aplicaciones SQL. Aunque el SQL Access Group incluyan a varios de los más importantes vendedores de DBMS, tanto IBM como Sybase declinaron participar, arrojando la duda sobre la adopción amplia de las recomendaciones del SQL Access Group.

En resumen, los diferentes estándares SQL continuarán creciendo y evolucionando durante los noventa. La existencia de varios estándares diferentes, junto con la introducción de nuevas extensiones y características propietarias de los vendedores de DBMS, asegura que no existirá una única versión portable de SQL en el futuro cercano. En su lugar, los estándares de SQL continuarán

siendo tratados como normas y objetivos en lugar de como reglas irrefutables. Los vendedores de DBMS seguirán generalmente el estándar IBM siempre que haya uno, y buscarán oportunidades para diferenciar sus productos en áreas que IBM aún no haya abordado.

Extensiones del lenguaje SQL

SQL tiene unos diez años como producto comercial, y aún está activamente creciendo y cambiando en su adolescencia. Los vendedores de DBMS están invirtiendo fuertemente en extensiones y nuevas características de SQL, incluyendo:

- **Sentencias procedurales.** La adición de sentencias tradicionales de lenguajes de programación (IF/THEN/ELSE, GOTO, FOR y bucles WHILE, etc.) para que SQL permita a los usuarios «unir» las sentencias DML y DDL de SQL en guiones SQL completos. Esta capacidad es especialmente útil en una arquitectura cliente/servidor, ya que el servidor puede ejecutar un guion SQL entero con interacción sentencia a sentencia con el programa solicitante, minimizando el tráfico de red.
- **Procedimientos almacenados.** Los procedimientos almacenados permiten que una secuencia de sentencias SQL sea precompilada, nominada y almacenada en la base de datos para ejecución posterior. Combinan la flexibilidad de SQL dinámico (el programa de aplicación puede decidir qué procedimiento ejecutar en tiempo de ejecución) con el rendimiento de SQL estático (el código del procedimiento está precompilado y el DBMS simplemente lo ejecuta en tiempo de ejecución).
- **Disparadores.** Los disparadores permiten a un DBMS ejecutar automáticamente una sentencia o secuencia de sentencias en respuesta a un evento externo, tal como un intento de actualizar, insertar o suprimir datos en una tabla. Con los disparadores, el DBMS puede tomar un papel más activo en el forzamiento de la integridad de la base de datos y las reglas comerciales.
- **Reglas.** Varios vendedores de DBMS han comenzado a incorporar reglas y otras características prestadas de los sistemas basados en conocimientos en sus bases de datos relacionales. Las reglas permiten a una base de datos incluir las «reglas de sentido común» y los procedimientos de operación normales de una organización, almacenando no solamente datos en bruto sino también el conocimiento acerca de cómo funciona una organización.
- **Cursos de desplazamiento.** Los cursos de desplazamiento soportan el desarrollo de programas de inspección de base de datos que permiten a un

usuario moverse libremente hacia adelante y hacia atrás a través de un conjunto de resultados de consulta. Este tipo de interfaz de usuario puede ser muy valioso en soporte de decisiones y en aplicaciones de consultas de base de datos.

El trabajo más activo en la extensión del lenguaje SQL está siendo efectuado por los vendedores de DBMS independientes, tales como Sybase, Oracle, Ingres e Informix. Sybase soporta ya sentencias procedurales, procedimientos almacenados y disparadores en su dialecto Transact-SQL. Ingres Versión 6.3 ha añadido soporte para procedimientos almacenados y reglas. Oracle soporta sentencias procedurales en la opción de procesamiento de transacciones de Oracle Versión 6, y ha anunciado soporte de integridad referencial y procedimientos almacenados para Oracle Versión 7. Informix Versión 7. Informix-4GL, un lenguaje de cuarta generación que combina SQL y sentencias de lenguaje procedural.

En esta área la década de los noventa es probable que continúen la historia de los ochenta, donde cada vendedor introduce nuevas características de SQL que esperan que proporcionen una razón de peso para que los clientes comprren sus productos DBMS. El resultado es un juego continuo de saltos técnicos, donde cada vendedor se halla por delante en ciertas áreas mientras juega a alcanzar a sus competidores en otras áreas. En el proceso, todas aquellas características que el mercado considera importantes (las sentencias procedurales y los procedimientos almacenados son los actuales favoritos) encontrarán su acomodo en todos los productos de los principales vendedores de DBMS independientes.

Tipos de datos complejos

SQL fue concebido y diseñado originalmente para manejar datos de procesamiento típicos tales como enteros, números decimales y en coma flotante y cadenas de caracteres cortas. Sin embargo, conforme las bases de datos relacionales se han ido popularizando, los usuarios han deseado almacenar y organizar otros tipos de datos, tales como imágenes exploradas, dibujos de diseño asistido por computador, hojas de cálculo, documentos en jeroglíficos, etc. Los tipos de datos complejos, como éstos, son con frecuencia conocidos como *objetos binarios grandes*.

Con el lenguaje SQL y la tecnología DBMS de los ochenta, el único modo de manejar los «blobs» fue almacenarlos en archivos, aparte de la base de datos. La propia base de datos almacenaba información descriptiva referente al objeto (como el autor, el título, la fecha y las palabras clave del documento) y el nombre del archivo donde el objeto estuviese almacenado. El programa de

aplicación podía entonces utilizar SQL para buscar la base de datos, pero estaba forzado a utilizar entrada/salida convencional de archivos para procesar el propio «blob». Sin embargo, han aparecido fuertes presiones para extender SQL y soportar blobs directamente dentro de la base de datos.

Al principio puede parecer sencillo soportar un blob directamente como un tipo de datos SQL, almacenado en una columna de una tabla relacional. Sin embargo, los blobs dan lugar a varias cuestiones serias para SQL y el modelo relacional:

- **Datos no atómicos.** El modelo relacional asume que cada columna de cada fila contiene un único dato atómico, pero los blobs con frecuencia tienen partes y subpartes y sub-subpartes. Esta estructura no puede representarse sin violar el modelo relacional.
- **Recuperación directa.** Si un blob contiene una subestructura, el programa de aplicación puede desear recuperar partes individuales del blob (por ejemplo, secciones de un documento o capas de un esquema de una placa de circuitos) sin recuperar todo el blob. SQL no proporciona ningún mecanismo para seleccionar y recuperar partes de un elemento de datos.
- **LIMITACIONES DE MEMORIA.** Con SQL convencional, un programa de aplicación accede a una fila entera de datos en sus buffers para proceso. Sin embargo, un blob grande puede fácilmente exceder la memoria principal disponible de un computador personal. Para procesar tales blobs, un programa debe ser capaz de almacenar y recuperar pieza a pieza, del mismo modo que procesa archivos de datos.

- **Búsqueda.** Si una columna de blob contiene datos de texto, un usuario puede desear seleccionar filas basándose en la búsqueda del texto completo del contenido del blob. Otras búsquedas pueden ser adecuadas a otros tipos de blobs. Estas búsquedas no estánndar requieren extensiones del lenguaje SQL.

A pesar de estos obstáculos, el soporte de los blobs se ha convertido en una cuestión importante, especialmente para productos DBMS que corren en computadores personales y en estaciones de trabajo que manipulan objetos de datos complejos. Informix, Ingres y Sybase han anunciado planes para soportar blobs en nuevas versiones de sus productos DBMS, y otros vendedores de DBMS cabe esperar que les sigan. Sin embargo, el desarrollo de técnicas para almacenar, recuperar, buscar y procesar blobs está en su infancia y continuará siendo un área activa de desarrollo y cambios en SQL hasta bien adentrada la década de los noventa.

Bases de datos orientadas a objetos

Gran parte de la investigación académica en el área de las bases de datos durante los últimos años se ha concentrado en nuevos modelos de datos «orientacionales». Al igual que el modelo relacional proporcionó claras ventajas sobre los primitivos modelos jerárquicos y de red, el objetivo de esta investigación es desarrollar nuevos modelos de datos que puedan superar algunas de las desventajas del modelo relacional. Hoy día, la mayoría de la investigación sobre nuevos modelos de datos está centrada en las llamadas bases de datos «orientadas a objetos». Los entusiastas defensores de bases de datos orientadas a objetos proclaman que son la generación siguiente a los sistemas de bases de datos y predicen que supondrán un serio reto a las bases de datos relacionales hacia el final del siglo.

A diferencia del modelo de datos relacional, donde el artículo de Codd de 1970 proporcionó una clara definición matemática de una base de datos relacional, no existe acuerdo sobre qué es una base de datos orientada a objetos. Cuando los investigadores utilizan el término, están generalmente describiendo una base de datos que utiliza los mismos principios organizativos que la programación orientada a objetos. Estos principios incluyen:

- **Objetos.** En una base de datos orientada a objetos cualquier cosa es un objeto y se manipula como un objeto. La organización tabular de fila/columna de una base de datos relacional es sustituida por la noción de colecciones de objetos. En general, una colección de objetos es ella misma un objeto y puede ser manipulada del mismo modo que se manipulan los restantes objetos.

- **Clases.** Las bases de datos orientadas a objetos sustituyen la noción relacional de tipos de datos atómicos con una noción jerárquica de clases y subclases. Por ejemplo, VEHICULOS podría ser una clase de objeto, y los miembros individuales («instancias») de esa clase incluirían un coche, una bicicleta, un tren o un bote. La clase VEHICULOS podría incluir subclases denominadas COCHES y BOTES, representando una forma más especializada de vehículo. Análogamente, la clase COCHES podría incluir una subclase denominada CONVERTIBLES, etc.

- **Herencia.** Los objetos heredan las características de su clase y de todas las clases de nivel superior a la que pertenezcan. Por ejemplo, una de las características de un vehículo podría ser el «número de pasajeros». Todos los miembros de las clases COCHES, BOTES y CONVERTIBLES tendrían este atributo, ya que son subclases de VEHICULOS. La clase COCHES podría también tener el atributo «número de puertas», y la clase CONVERTIBLES heredaría este atributo. Sin embargo, la clase BOTES no heredaría el atributo.

- **Mensajes y métodos.** Los objetos se comunican unos con otros mediante el

envío y recepción de *mensajes*. Cuando recibe un mensaje, un objeto responde ejecutando un *método*, un programa almacenado dentro del objeto que determina cómo se procesa el mensaje. Por tanto un objeto incluye un conjunto de comportamientos descritos por sus métodos. Generalmente un objeto comparte muchos de los mismos métodos con otros objetos de su clase.

Estos principios y técnicas hacen que las bases de datos orientadas a objetos estén bien adecuadas a aplicaciones que implican tipos de datos complejos, tales como documentos compuestos o de diseños asistidos por computador que combinan texto, gráficos y hojas de cálculo. La base de datos proporciona un modo natural de representar las jerarquías que aparecen en los datos complejos. Por ejemplo, un documento entero puede presentarse como un único objeto, compuesto de objetos más pequeños (secciones), compuestos de objetos aún más pequeños (párrafos, gráficos, etc.). La jerarquía de clases permite a la base de datos seguir la pista del «tipo» de cada objeto en el documento (párrafos, gráficos, ilustraciones, títulos, pies de página, etc.). Finalmente, el mecanismo de mensajes ofrece soporte natural para una interfaz de usuario gráfica. El programa de aplicación puede enviar un mensaje «extráigalo usted mismo» a cada parte del documento, pidiendo que se extraiga de la pantalla. Si el usuario cambia la forma de la ventana que visualiza el documento, el programa de aplicación puede responder enviando un mensaje «modifica tu tamaño» a cada parte del documento, etc. Cada objeto del documento tiene la responsabilidad de su propia visualización, por lo que los nuevos objetos pueden ser añadidos fácilmente a la arquitectura del documento.

Existen varias bases de datos comerciales orientadas a objetos actualmente disponibles. Sin embargo, las bases de datos orientadas a objetos ya han provocado una tormenta de controversia en la comunidad de las bases de datos. Los proponentes proclaman que las bases de datos orientadas a objetos reemplazarán en el futuro al modelo de datos relacional. Los críticos arguyen que las bases de datos orientadas a objetos son simplemente una reformulación de las bases de datos jerárquicas de los sesenta, con una nueva terminología y un nuevo giro en conceptos antiguos. Los vendedores de bases de datos relacionales proclaman que la mayoría, si no todos, los beneficios de las bases de datos orientadas a objetos pueden lograrse añadiendo soporte de blobs a una base de datos relacional.

Las bases de datos orientadas a objetos jugarán probablemente un papel creciente en segmentos de mercado especializados, tales como diseño teórico, procesamiento de documentos compuestos e interfaces de usuario gráficas. Sin embargo, es poco probable que puedan tener un impacto importante en la corriente principal de aplicaciones de procesamiento de datos durante los próximos años. Con el soporte de un estándar oficial, el apoyo al mercado de IBM y el soporte de varios vendedores DBMS independientes importantes, el vagón de

SQL es simplemente demasiado potente para que descarrile a un puñado de vendedores de DBMS orientado a objetos en cualquier momento en el futuro cercano. Sin embargo, hace tan solo una década, la misma observación podría haberse efectuado referente a las bases de datos relacionales y a SQL. De todas las tendencias del mercado de las bases de datos, el desarrollo de bases de datos orientadas a objetos es la única que amenaza el dominio continuo de SQL en las próximas décadas.

A *La base de datos ejemplo*

La mayor parte de los ejemplos de este libro están basados en la base de datos que se describe en este apéndice. Esta base de datos ejemplo contiene datos correspondientes a una sencilla aplicación de procesamiento de pedidos para una pequeña empresa de distribución. Consiste de cinco tablas:

- **CLIENTES**, que contiene una fila por cada uno de los clientes de la empresa.
- **REPVENTAS**, que contiene una fila por cada uno de los diez vendedores de la empresa.
- **OFICINAS**, que contiene una fila por cada una de las cinco oficinas en las que trabajan los vendedores.
- **PRODUCTOS**, que contiene una fila por cada tipo de producto disponible para la venta.
- **PEDIDOS**, que contiene una fila por cada pedido ordenado por un cliente. Por simplicidad, se supone que cada pedido se refiere a un solo producto.

La Figura A.1 muestra gráficamente las cinco tablas, las columnas que contienen y las relaciones padre/hijo entre ellas. La clave primaria de cada tabla está sombreada. Las cinco tablas de esta base de datos pueden crearse utilizando las siguientes sentencias CREATE TABLE:

```
CREATE TABLE CLIENTES  
  (NUM_CLIE INTEGER NOT NULL,  
   EMPRESA VARCHAR(20) NOT NULL,  
   REP_CLIE INTEGER,
```

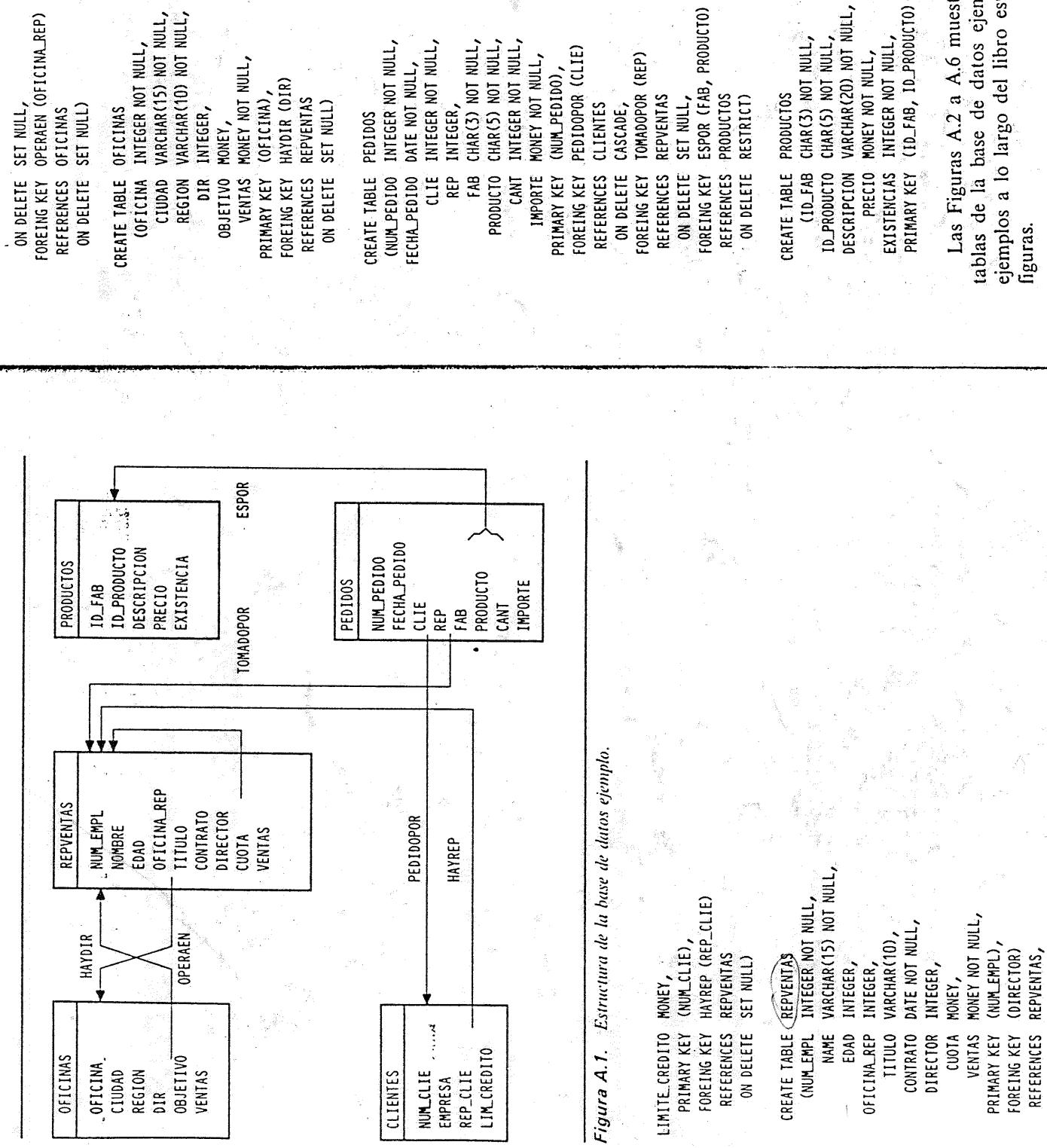


Figura A.1. Estructura de la base de datos ejemplo.

```

CREATE TABLE REPVENTAS
  (NUM_EMPL INTEGER NOT NULL,
  NAME VARCHAR(15) NOT NULL,
  EDAD INTEGER,
  OFICINA REP INTEGER,
  TITULO VARCHAR(10),
  CONTRATO DATE NOT NULL,
  DIRECTOR INTEGER,
  CUOTA MONEY,
  VENTAS MONEY NOT NULL,
  PRIMARY KEY (NUM_EMPL),
  FOREIGN KEY (DIRECTOR)
  REFERENCES REPVENTAS,
  ON DELETE SET NULL,
  FOREIGN KEY OPERAEN (OFICINA REP)
  REFERENCES OFICINAS
  ON DELETE SET NULL)
  
```

Las Figuras A.2 a A.6 muestran los contenidos de cada una de las cinco tablas de la base de datos ejemplo. Los resultados de las consultas de los ejemplos a lo largo del libro están basados en los datos mostrados en estas figuras.

NUM_CLIE	EMPRESA	REP_CLIE	LIMITE_CREDITO
2111	JCP Inc.	103	\$50,000.00
2102	Firts Corp.	101	\$65,000.00
2103	Acme Mfg.	105	\$50,000.00
2123	Carter & Sons	102	\$40,000.00
2107	Ace International	110	\$35,000.00
2115	Smithson Corp.	101	\$20,000.00
2101	Jones Mfg.	106	\$65,000.00
2112	Zetacorp	108	\$50,000.00
2121	QMA Assoc.	103	\$45,000.00
2114	Orion Corp	102	\$20,000.00
2124	Peter Brothers	107	\$40,000.00
2108	Holm & Landis	109	\$55,000.00
2117	J.P. Sinclair	106	\$35,000.00
2122	Three Way Lines	105	\$30,000.00
2120	Rico Enterprises	102	\$50,000.00
2106	Fred Lewis Corp.	102	\$65,000.00
2119	Solomon Inc.	109	\$25,000.00
2118	Midwest Systems	108	\$60,000.00
2113	Ian & Schmidt	104	\$20,000.00
2109	Chen Associates	103	\$25,000.00
2105	AAA Investments	101	\$45,000.00

Figura A.2. La tabla CLIENTES.

OFICINA	CIUDAD	REGION	DIR	OBJETIVO	VENTAS
22	Denver	Oeste	108	\$300,000.00	\$186,042.00
11	New York	Este	106	\$575,000.00	\$692,637.00
12	Chicago	Este	104	\$800,000.00	\$735,042.00
13	Atlanta	Este	105	\$350,000.00	\$367,911.00
21	Los Angeles	Oeste	108	\$725,000.00	\$835,915.00

Figura A.4. La tabla OFICINAS.

NUM_PEDIDO	FECHA_PEDIDO	CLT	REP	FAB	PRODUCTO	CANT	IMPORTE
112961	17-DIC-89	2117	106	REI	2A44L	7	\$31,500.00
113012	11-ENE-90	2111	105	ACI	41003	35	\$1,475.00
112989	03-ENE-90	2101	106	FEA	114	6	\$1,458.00
113051	10-FEB-90	2118	108	QSA	K47	4	\$1,420.00
112968	12-OCT-89	2102	101	ACI	41004	34	\$3,978.00
110036	30-ENE-90	2107	110	ACI	41002	9	\$22,500.00
113045	02-FEB-90	2112	108	REI	2A44R	10	\$45,000.00
112963	17-DIC-89	2103	105	ACI	41004	28	\$3,276.00
113013	14-ENE-90	2118	108	BIC	41003	1	\$652.00
113058	23-FEB-90	2108	109	FEA	112	10	\$1,480.00
112997	08-ENE-90	2124	107	BLC	41003	1	\$652.00
112983	27-DIC-89	2103	105	ACI	41004	6	\$702.00
113024	20-FEB-90	2114	108	QSA	XK47	20	\$7,100.00
113062	24-FEB-90	2124	107	FEA	114	10	\$2,430.00
112979	12-OCT-89	2114	102	ACI	41002	6	\$15,000.00
113027	22-ENE-90	2103	105	ACI	41002	54	\$4,104.00
113007	08-FEB-90	2112	108	IMM	773C	3	\$2,925.00
113069	02-MAR-90	2109	107	IMM	773C	22	\$31,350.00
113034	04-ABR-90	2107	110	REI	2A45C	8	\$632.00
112992	04-NOV-89	2118	108	ACI	41002	10	\$760.00
112975	12-OCT-89	2111	103	REI	2A44G	6	\$2,100.00
113055	15-FEB-90	2108	101	ACI	4100X	6	\$150.00
113048	10-FEB-90	2120	102	IMM	779C	2	\$3,750.00
112993	04-ENE-89	2106	102	REI	2A45C	24	\$1,896.00
113065	27-FEB-90	2106	102	QSA	XK47	6	\$2,130.00
113003	25-FEB-90	2108	109	IMM	779C	3	\$5,625.00
113049	10-FEB-90	2118	108	QSA	XK47	2	\$776.00
112987	31-OCT-89	2103	105	ACI	4100Y	11	\$27,500.00
113057	18-FEB-90	2111	103	ACI	4100X	24	\$600.00
113042	02-FEB-90	2113	101	REI	2A44R	5	\$22,500.00

Figura A.3. La tabla REVENTAS.

Figura A.5. La tabla PEDIDOS.

B Profiles de vendedores DBMS

ID_FAB	ID_PRODUCTO	DESCRIPCION	PRECIO	EXISTENCIAS
REI	2A45C	V Stago Trinquete Extractor	\$79.00	210
ACI	4100Y	Reducutor	\$2,750.00	25
QSA	XK47	Plate	\$555.00	38
BIC	41672	Riostra 2-Tm	\$1,875.00	9
IMM	779C	Articulo Tipo 3	\$107.00	207
ACI	41003	Articulo tipo 4	\$117.00	139
ACI	41004	Manivela	\$652.00	3
BIC	41003	Perno Riostra	\$250.00	24
IMM	887P	Reducutor	\$134.00	203
QSA	XK48	Bisagra Izqda.	\$4,200.00	12
REI	2A44L	Cubierta	\$148.00	115
FEA	112	Soporte Riostra	\$54.00	223
IMM	887H	Retn	\$225.00	78
BIC	41089	Articulo Tipo 1	\$55.00	277
ACI	41001	Riostra 1-Tm	\$1,425.00	5
IMM	775C	Montador	\$2,500.00	28
ACI	41002	Reducutor	\$117.00	37
QSA	XK48A	Articulo Tipo 2	\$76.00	167
ACI	41002	Bisagra Dcha.	\$4,500.00	12
REI	2A44R	Riostra 1/2-Tm	\$975.00	28
IMM	773C	Ajustador	\$25.00	37
ACI	4100X	Bancada Motor	\$243.00	15
FEA	114	Retenedor Riostra	\$475.00	32
IMM	887X	Pasador Bisagra	\$350.00	14
REI	2A44G			

Los vendedores perfilados en este apéndice son líderes en sus respectivos segmentos de mercado en términos bien de la tecnología o bien de la cuota de mercado, y colectivamente son responsables del 75 por 100 o más de las ventas relacionadas con SQL. Los vendedores y sus productos son:

- Ashton-Tate (dBASE IV, SQL Server).
- Digital Equipment (Rdb/VMS, VAX SQL).
- Gupta Technologies (SQLBase, SQLWindows).
- Hewlett-Packard (HPSQL).
- Ingres (Ingres).
- Microsoft (SQL Server).
- Oracle (Oracle).
- Sybase (Sybase SQL Server).
- Tandem (NonStop SQL).
- Unify (Unify, Accell).

Figura A.6. La tabla PRODUCTOS.

Ashton-Tate Corporation

Ashton-Tate es el principal vendedor de productos de bases de datos para computadores personales. Sus productos dBASE II y dBASE III han vendido más de dos millones de copias y han mantenido consistentemente una cuota del 70 por 100 o más del mercado de DBMS para PCs durante los ochenta. Tanto dBASE II como dBASE III eran básicamente gestores de archivos planos con una capacidad limitada para componer datos a través de archivos múltiples. Aun cuando estas bases de datos eran limitadas, el lenguaje de programación dBASE fue utilizado para desarrollar una amplia gama de aplicaciones comerciales sobre PCs y de hecho se ha convertido en un estándar.

A finales de la década de los ochenta, los vendedores de DBMS sobre minicomputadores habían transportado sus productos a los computadores personales y comenzaban a ejercer una presión competitiva sobre la envejecida línea de productos dBASE. Además, los competidores en el mercado del computador personal ofrecían productos más avanzados. Ashton-Tate respondió con dBASE IV, que fue diseñado para proporcionar compatibilidad con versiones anteriores de dBASE mientras proporcionaba las características más avanzadas que los usuarios estaban demandando. Además, dBASE IV disponía del soporte SQL «injetado en» el lenguaje dBASE. Finalmente, dBASE IV iba a proporcionar un frontal para SQL Server, la versión OS/2 del DBMS de Sybase.

Desgraciadamente, la versión inicial de dBASE IV tuvo algunos defectos serios, y los ingenieros de Ashton-Tate descubrieron que no era posible lograr todos los objetivos de dBASE IV dentro de la limitación de 640KB de memoria del MS-DOS. El producto fue revisado y ha sido reintroducido en dos versiones. El producto dBASE IV autónomo corre en un PC IBM estándar y soporta dBASE y sentencias de SQL que acceden a una base de datos local. El producto dBASE IV de red requiere un extensor de la memoria del MS-DOS. Soporta las mismas sentencias de lenguaje que la versión autónoma y proporciona acceso a las bases de datos de SQL Server a través de una red de área local OS/2.

Ashton-Tate y Microsoft comparten la responsabilidad de marketing para el DBMS de SQL Server en el mercado de OS/2. Ashton-Tate tenía inicialmente derechos exclusivos de distribución para el producto y era responsable de venderlo al por mayor a través de tiendas de informática junto con sus otros productos de bases de datos. Sin embargo, con las ventas inicialmente muy bajas de OS/2, el producto ha tenido un éxito menos que espectacular, y Microsoft ha buscado otros canales de distribución para el producto.

Ashton-Tate continúa manteniendo la cuota mayor del mercado de DBMS para PCs gracias a su base instalada de usuarios de dBASE, pero encara un reto importante para mantener esa posición a la vista de las presiones competitivas de IBM, otros vendedores de DBMS para PCs tales como Borland, y los vendedores de DBMS de línea completa tales como Oracle Corporation.

Digital Equipment Corporation

Digital Equipment Corporation, el mayor vendedor de minicomputadores del mundo, es también un importante participante en el mercado de las bases de datos sobre minicomputadores. La familia de sistemas VAX de Digital se extiende en precio y rendimiento desde las estaciones de trabajo personales VAX que cuestan menos de \$10,000 hasta la serie VAX 9000, un sistema de clase maxi-computador que cuesta millones de dólares. La línea de productos soporta dos sistemas operativos diferentes. VAX/VMS, el sistema operativo insignia de la compañía, es propietario de Digital. Ultrix es un sistema operativo alternativo basado en UNIX que es popular en entornos científicos y técnicos. Además de sus productos de arquitectura VAX, Digital comercializa una familia de estaciones de trabajo RISC y una línea de computadores personales compatibles IBM.

El DBMS relacional insignia de Digital es Rdb/VMS, que corre únicamente bajo el sistema operativo VAX/VMS. Internamente la máquina de base de datos Rdb/VMS soporta el Digital Standard Relational Interface (DSRI), una interfaz binaria de bajo nivel. Por encima del nivel DSRI, Rdb proporciona dos interfaces basadas en lenguaje —VAX SQL, que soporta SQL incorporado y acceso SQL interactivo, y RDO, que soporta un lenguaje de base de datos propio de Digital—. El lenguaje RDO está lentamente en desfase en favor de VAX SQL. El DBMS Rdb/VMS proporciona sólo un conjunto básico de herramientas de bases de datos para administración del sistema. Para reforzar su oferta, Digital compró la licencia de las herramientas Ingres a Ingres Corporation y las modificó para soportar Rdb/VMS. Digital también ha comprado la licencia del propio DBMS de Ingres y lo ofrece como el RDBMS soportado por Digital bajo el sistema operativo UNIX.

En 1989, Digital ofreció conjuntamente el sistema de tiempo de ejecución Rdb/VMS junto con el sistema operativo VAX/VMS, haciendo de éste una parte fundamental del software operativo VMS. Digital también indicó su intención de construir otros productos VMS en capas por encima de la base Rdb/VMS. El acceso remoto a datos Rdb/VMS es proporcionado por VAX SQL Services, un API basado en SQL que utiliza los protocolos de red DECnet. El acceso SQL Services está disponible desde estaciones de trabajo VMS y Ultrix, desde computadores personales basados en MS-DOS, y Ultrix, desde sistemas Macintosh. Un producto adicional de Rdb/VMS, llamado VIDAS, proporciona acceso a datos maxicomputadores almacenados en IDMS/SQL o bases de datos DB2.

Gupta Technologies, Inc.

Gupta Technologies es uno de los vendedores de DBMS relativamente más pequeños, pero tiene una impresionante lista de «primicias» técnicas a su cargo.

La empresa fue fundada por un anterior director de la división de microcomputadores Oracle Corporation. La línea de productos Gupta incluye un DBMS relacional y herramientas de desarrollo de base de datos, todos destinados a PC y redes de área local con PC.

SQIBase, la máquina DBMS basada en SQL de la compañía, corre en computadores personales basados en MS-DOS u OS/2. SQIBase corre sobre un PC autónomo o sobre una red de área local de PC. En esta última configuración, el servidor de SQIBase fue el primero y único servidor importante basado en SQL que corre sobre un PC basado en MS-DOS. El servidor soporta PC clientes que corren en una red de área local basada en NETBIOS. SQLBase también proporciona una pasarela a DB2, enlazando los productos PC de Gupta a datos mainframe. Otras características únicas de SQLBase incluyen el soporte de tipos de datos largos (blobs), cursorres de desplazamiento actualizables, control programado sobre niveles de aislamiento del cursor y conjuntos de resultados, lo cual permite que los resultados de una consulta sean utilizados como fuentes de datos para consultas posteriores. La interfaz programada a SQIBase se realiza a través de un API de llamadas.

Además de su máquina de base de datos, Gupta ofrece SQLWindows, una colección de herramientas de desarrollo que proporcionan una interfaz de usuario gráfica para procesamiento de base de datos. SQLWindows corre bajo Microsoft Windows y soporta el desarrollo de aplicaciones de base de datos del estilo «apunta y pulsa». SQLWindows trabaja con el DBMS SQLBase de Gupta y también con otros DBMS basados en SQL, incluyendo OS/2 Extended Edition, SQL Server y Oracle Server. Una versión de SQLWindows que corre bajo OS/2 Presentation Manager está planificada, y proporcionará una interfaz gráfica similar a los PCs basados en OS/2 para un rango de máquinas de base de datos de soporte.

Hewlett-Packard Company es un importante vendedor de sistemas minicomputadores, y la gestión de base de datos ha jugado siempre un papel importante en la línea de productos de HP. En los setenta la compañía fue pionera de la gestión de base de datos sobre minicomputadores con su DBMS Image/1000, que corría sobre minicomputadores HP 1000 en aplicaciones ingenieriles y técnicas, y su DBMS Image/3000, que corría sobre minicomputadores HP 3000 en aplicaciones comerciales. El DBMS Image estaba basado en el modelo de datos en red.

A mediados de los ochenta HP convirtió toda su línea de productos de minicomputadores a una arquitectura RISC de 32 bits y ofreció dos nuevos sistemas operativos para la nueva línea de productos. El primero, MPE/XL, proporcionaba compatibilidad hacia atrás con la serie HP 3000. El segundo,

HP/UX, era una implementación HP de UNIX y representaba el compromiso de HP con el vagón de UNIX.

Con la ayuda de la tecnología comprada a un vendedor de software DBMS externo, HP ha implementado un nuevo DBMS, llamado Allbase, bajo ambos sistemas operativos MPE/XL y HP/UX. HP ofrece dos interfaces alternativas a la máquina de base de datos Allbase. Una interfaz Image proporciona compatibilidad hacia atrás para aplicaciones existentes y presenta una visión de modelo de datos en red de todos los datos Allbase. HPSQL proporciona una interfaz relacional para nuevas aplicaciones y ofrece un SQL incorporado y una interfaz de SQL interactivo a los datos de Allbase. Este enfoque dual del acceso a bases de datos es único y representa un logro técnico significativo.

IBM Corporation

IBM es el mayor vendedor de bases de datos del mundo, con la mayoría de sus beneficios de bases de datos provenientes de dos productos de bases de datos, en maxicomputadores, antiguas y no jerárquico y DL/1 (Data Language/1), un método de acceso a bases de datos jerárquico—. Sin embargo, la parte más rápidamente creciente del negocio de bases de datos de IBM se halla en las bases de datos relacionales basadas en SQL, y SQL es la interfaz de base de datos estándar para la estrategia SAA de IBM. IBM tiene una oferta de producto basado en SQL para cada una de sus cuatro principales familias de sistemas:

■ DB2 es el producto de DBMS relacional insignia de IBM. Es un producto software grande y complejo que corre en sistemas mainframe de IBM bajo el sistema operativo MVS. En sus versiones más recientes, el rendimiento de DB2 ha mejorado dramáticamente, e IBM lo comercializa ahora como una solución OLTP. Sin embargo, DB2 supone un consumo sustancial de recursos maxicomputadores y requiere un sistema mainframe de buen tamaño para soportarlo.

■ SQL/DS es un sistema de gestión de base de datos de IBM para aplicaciones de centros de información. Corre bajo el sistema operativo VM/CMS sobre maxicomputadores de IBM. Desciende casi directamente del prototipo de base de datos relacional de IBM System/R, SQL/DS ha sido pionero de algunas características innovadoras de las bases de datos relacionales, entre las que se incluye el SQL dinámico extendido. Sin embargo, desde la perspectiva de marketing de IBM, permanece a la sombra de DB2.

■ SQL/400 es la implementación SQL de IBM para la serie AS/400 de sistemas. Sucesor del Sistema/38 de IBM, el AS/400 tiene un sistema de base de datos

HP/UX, era una implementación HP de UNIX y representaba el compromiso de HP con el vagón de UNIX.

Con la ayuda de la tecnología comprada a un vendedor de software DBMS externo, HP ha implementado un nuevo DBMS, llamado Allbase, bajo ambos sistemas operativos MPE/XL y HP/UX. HP ofrece dos interfaces alternativas a la máquina de base de datos Allbase. Una interfaz Image proporciona compatibilidad hacia atrás para aplicaciones existentes y presenta una visión de modelo de datos en red de todos los datos Allbase. HPSQL proporciona una interfaz relacional para nuevas aplicaciones y ofrece un SQL incorporado y una interfaz de SQL interactivo a los datos de Allbase. Este enfoque dual del acceso a bases de datos es único y representa un logro técnico significativo.

IBM Corporation

IBM es el mayor vendedor de bases de datos del mundo, con la mayoría de sus beneficios de bases de datos provenientes de dos productos de bases de datos, en maxicomputadores, antiguas y no jerárquico y DL/1 (Data Language/1), un método de acceso a bases de datos jerárquico—. Sin embargo, la parte más rápidamente creciente del negocio de bases de datos de IBM se halla en las bases de datos relacionales basadas en SQL, y SQL es la interfaz de base de datos estándar para la estrategia SAA de IBM. IBM tiene una oferta de producto basado en SQL para cada una de sus cuatro principales familias de sistemas:

■ DB2 es el producto de DBMS relacional insignia de IBM. Es un producto software grande y complejo que corre en sistemas mainframe de IBM bajo el sistema operativo MVS. En sus versiones más recientes, el rendimiento de DB2 ha mejorado dramáticamente, e IBM lo comercializa ahora como una solución OLTP. Sin embargo, DB2 supone un consumo sustancial de recursos maxicomputadores y requiere un sistema mainframe de buen tamaño para soportarlo.

■ SQL/DS es un sistema de gestión de base de datos de IBM para aplicaciones de centros de información. Corre bajo el sistema operativo VM/CMS sobre maxicomputadores de IBM. Desciende casi directamente del prototipo de base de datos relacional de IBM System/R, SQL/DS ha sido pionero de algunas características innovadoras de las bases de datos relacionales, entre las que se incluye el SQL dinámico extendido. Sin embargo, desde la perspectiva de marketing de IBM, permanece a la sombra de DB2.

■ SQL/400 es la implementación SQL de IBM para la serie AS/400 de sistemas. Sucesor del Sistema/38 de IBM, el AS/400 tiene un sistema de base de datos

relacional con soporte hardware como parte de su sistema operativo. SQL/400 core como una capa adicional, proporcionando soporte SQL al software DBMS subyacente. Hasta la fecha, la mayoría de los AS/400 se han vendido como mejoras de los sistemas Sistema/38 y Sistema/36 de IBM, la interfaz SQL no ha sido crítica para estos clientes. Cuando el AS/400 comienza a jugar su propio papel independiente como alternativa de gama media a VAXes y otros minicomputadores, el soporte de SQL pasará a ser más crítico.

■ OS/2 Extended Edition es la versión extendida de IBM del sistema operativo OS/2 para computadores personales. Incluye una base de datos relacional escrita por IBM con una interfaz SQL. Cuando fue introducido inicialmente, el OS/2 Extended Edition solamente corría como un producto de estación de trabajo autónoma. Fue mejorado para soportar una configuración de servidor de base de datos en 1989, con PCs basados en DOS u OS/2 actuando como clientes de un servidor de base de datos basado en OS/2.

La arquitectura SAA incluye una especificación de un SQL estándar de IBM, e IBM ha establecido que evolucionaran sus productos SQL hacia ese estándar para proporcionar portabilidad de aplicaciones. Sin embargo, en las implementaciones actuales permanecen significativas diferencias entre los productos. IBM ha anunciado también su intención de proporcionar acceso de base de datos distribuida a través de sus productos basados en SQL y ha revelado un plan de cuatro pasos para proporcionar ese acceso. No obstante, no han sido anunciantos planes temporales detallados para acceso distribuido dentro de cada producto y entre productos.

La principal ventaja competitiva de los productos de IBM es, naturalmente, el hecho de que provienen de IBM. Cada uno de los productos es una implementación de SQL sólida y básica, y cada producto está siendo continuamente mejorado con el tiempo. En algunos casos (tal como el soporte de integridad referencial), IBM ha marcado la pauta de la innovación de SQL, pero generalmente IBM ha sido quien ha establecido el estándar más fijo, dejando a los vendedores de DBMS independientes innovar en nuevas áreas de tecnología DBMS.

Informix Software, Inc.

Informix Software es un importante vendedor de sistemas DBMS relacionales basados en UNIX. El primer DBMS relacional de la empresa, Informix, fue implementado en sistemas microcomputadores basados en UNIX a principios de los ochenta, y fue conocido por su eficiencia y compactidad comparado con los sistemas DBMS basados en VAX/VMS que fueron introducidos aproximadamente al mismo tiempo. En 1985, Informix fue reescrito como un DBMS

basado en SQL e introducido como Informix-SQL. Desde entonces Informix-SQL ha sido transportado a varias docenas de sistemas basados en UNIX diferentes, yendo desde PCs basados en UNIX hasta maxicomputadores Am-dahl con soporte UNIX. Informix también ha sido transportado a computadoras personales bajo MS-DOS y al sistema operativo VMS de Digital, pero la mayoría de las ventas de la compañía siguen estando en el mercado UNIX. Además de la propia máquina de DBMS Informix, la empresa ofrece Informix-4GL, un lenguaje de cuarta generación basado en SQL que soporta el desarrollo de aplicaciones interactivas basadas en formularios. El escritor de informes ACE, otra parte de la línea de productos Informix, soporta la creación de diferentes tipos de informes sin programación. La interfaz programada con Informix-SQL se realiza a través de SQL incorporado, y soporta diversos lenguajes de programación anfisfriones. Informix-Turbo es una versión de alto rendimiento de la máquina de base de datos Informix, que está posicionado para competir con otras versiones OLTP de los productos DBMS de otros vendedores.

En 1988, Informix se unió a Innovative Software y añadió una línea de software de automatización de oficinas basado en PC a su línea de productos. Más recientemente, la empresa ha adquirido Wingz, una hoja de cálculo gráfica para el Macintosh, con una versión futura de Wingz prometida para PC de IBM bajo OS/2 también. Con estos productos, Informix se posiciona como un vendedor de línea completa de productos de automatización de oficinas, todos construidos sobre la base de datos Informix subyacente. Para mejor soporte de aplicaciones de automatización de oficinas, la empresa ha anunciado su intención de añadir soporte para documentos y otros objetos grandes a Informix.

Ingres Corporation

Ingres Corporation tuvo sus orígenes en el prototipo de base de datos relacional construida en la Universidad de California en Berkeley. Durante el comienzo y mediados de los ochenta, el DBMS de Ingres y su lenguaje de base de datos QUEL fue un rival importante de SQL, y hubo una rivalidad competitiva particularmente fuerte entre Ingres y Oracle en el mercado de DBMS de gama media. Cuando SQL se impuso como lenguaje de base de datos estándar, la empresa convirtió Ingres a un DBMS basado en SQL. El producto goza por tanto de algunas de las ventajas de ser una implementación SQL más reciente, junto con los beneficios de más de una década de experiencia en bases de datos relacionales.

Ingres fue implementada por primera vez sobre minicomputadores Digital y la mayoría de las ventas de la empresa provienen aún de la plataforma hardware

VAX. Ingres ha sido también transportada a varias docenas de sistemas diferentes basados en UNIX, e Ingres para PCs corren sobre computadores personales basados en DOS. Una versión de Ingres ha sido también introducida para maincomputadores IBM que corren VM/CMS. Utilizando Ingres/Star, una instalación Ingres en un sistema puede acceder remotamente a bases de datos Ingres en otro sistema en donde Ingres esté instalado.

Con sus orígenes académicos, Ingres ha tenido siempre una reputación como líder tecnológico del mercado de DBMS. Por ejemplo, sus ofertas de bases de datos distribuidas han estado disponibles típicamente con antelación, y con características más completas, que las de sus competidores. En el campo de los PC, la compañía comprimió su implementación de Ingres para PC en un sistema MS-DOS estándar de 640K, a diferencia de los sistemas de memoria extendida requerida por alguno de sus competidores. La mayoría de los analistas están de acuerdo en que los esfuerzos de ventas y marketing de la compañía han sido más débiles que su tecnología. La empresa vende sus productos directamente y a través de relaciones OEM con fabricantes de sistemas informáticos. Por ejemplo, Ingres se vende por Digital Equipment Corporation como DBMS relacional de DEC para Ultrix, el sistema operativo basado en UNIX para la línea de productos VAX.

La actividad de desarrollo y comercial de la empresa aparece actualmente centrada en dos áreas:

- Máquinas de bases de datos, donde la empresa está de nuevo tratando de «lidiar con el paquete» en tecnología añadiendo extensiones orientadas a objetos y características de sistemas basados en conocimientos a su DBMS basado en SQL.
- Herramientas de bases de datos, que se utilizan como frontales al DBMS de Ingres y que han sido vendidos en licencia a otros. Las herramientas de Ingres también han sido vendidas en licencia a Digital para su uso como frontales del propio DBMS relacional Rdb/VMS de Digital para sistemas VAX/VMS.

Microsoft Corporation

Microsoft es el mayor vendedor de software de computadores personales del mundo, con beneficios anuales que se aproximan a mil millones de dólares. Como creador de MS-DOS, Microsoft Windows y OS/2, la empresa es el principal suministrador de software a los sistemas de computadores personales. Es también un importante suministrador de software de aplicaciones en PC, con una línea completa de productos que incluye hojas de cálculo, procesadores de texto, paquetes gráficos, programas de gestión de proyectos y lenguajes de

computadores (C, Pascal, BASIC, Ensamblador). Microsoft es también el mayor suministrador de software de aplicaciones para el computador personal Macintosh de Apple Computer.

Hasta 1987, la línea de productos de Microsoft no incluía un sistema de gestión de base de datos. Con el anuncio de OS/2 Extended Edition en 1987, IBM extendió efectivamente su definición de un sistema operativo de computador personal para incluir un DBMS integrado y comunicaciones de datos. En 1988, Microsoft respondió con SQL Server, una versión del DBMS Sybase transportado a OS/2. Microsoft comercializa SQL Server como su solución DBMS OS/2 para sus clientes OEM, y Ashton-Tate tiene la responsabilidad para la distribución comercial de SQL Server.

SQL Server es un producto de la división de software de sistemas de Microsoft, que desarrolla y comercializa los productos de sistemas operativos de Microsoft. Microsoft ha anunciado que sus productos de aplicaciones, tales como su hoja de cálculo Excel, soportarán acceso basado en SQL a SQL Server con el tiempo. La empresa está además reclutando a otros creadores de software de computador personal para construir frontales a SQL Server.

Oracle Corporation

Oracle Corporation fue el primer vendedor de DBMS en ofrecer un producto SQL comercial, adelantándose al propio anuncio de IBM en casi dos años. Durante los ochenta, Oracle creció hasta convertirse en el mayor vendedor de DBMS independiente. Hoy día es un importante competidor de DBMS, vendiendo sus productos a través de una agresiva estrategia de ventas directas y a través de acuerdos de reventa con varios fabricantes de computadoras de gama media.

El DBMS Oracle fue implementado originalmente sobre minicomputadores Digital, y las ventas de Oracle sobre VAX continúan marcando el paso del crecimiento de la empresa. Sin embargo, una de las principales ventajas de Oracle es su portabilidad. Oracle está actualmente implementado en casi 100 tipos diferentes de sistemas informáticos, dándole la disponibilidad más amplia de todos los productos DBMS. Las implementaciones de Oracle están disponibles para MS-DOS, OS/2 y el Macintosh en el mercado de computadores personales; para Sun, MIPS y muchas otras estaciones de trabajo basadas en UNIX; para una gran variedad de sistemas minicomputadores, y para maincomputadores IBM. Utilizando el software de red SQL*Net de Oracle, muchas de estas implementaciones Oracle pueden participar en una red distribuida de sistemas Oracle, proporcionando acceso remoto de un sistema a otro. La empresa ha diseñado un plan para proporcionar capacidades totales de DBMS distribuido en etapas.

El DBMS Oracle estaba basado originalmente en el prototipo System/R de IBM, y ha permanecido compatible en términos generales con los productos basados en SQL de IBM. En años recientes, Oracle ha planteado comercialmente el rendimiento OLTP de su DBMS, utilizando resultados de bancos de prueba de sistemas multiprocesadores para sustanciar su proclamación como líder de rendimiento OLTP. Oracle Versión 6 incluye una opción especial de procesamiento de transacciones para el cual la empresa carga una cantidad adicional. La opción de procesamiento de transacciones también incluye PL/SQL, una extensión de SQL con sentencias de lenguaje de programación que los analistas consideran generalmente una respuesta al dialecto Transact-SQL de Sybase.

Oracle Corporation ha combinado una buena tecnología con campañas de ventas agresivas y marketing de alto perfil. La empresa está intentando explorar estas ventajas a la vez que trata de extender su dominio en el campo de los computadores personales con sus productos basados en OS/2. En otros intentos de mejorar su posición competitiva, Oracle ha lanzado varios subsidiarios nuevos en la integración de sistemas, software de fabricación y los mercados de software financiero. Cada subsidiario ofrece productos y servicios basados en el DBMS Oracle, pero consigue oportunidades de beneficios que caen fuera del mercado DBMS. Los subsidiarios también contribuyen al mantenimiento, soporte y beneficios de «consulting» que se han convertido en una parte importante de las ventas de la compañía. Por tanto, Oracle está posicionándose no sólo como un vendedor de DBMS, sino como un importante vendedor de software independiente a través de un amplio espectro de sistemas.

Sybase, Inc.

Sybase es una compañía DBMS iniciada a mediados de los ochenta, fundada con decenas de millones de dólares en capital inicial. El equipo fundador de la empresa y muchos de sus primeros empleados fueron de otros vendedores de DBMS, y para la mayoría de ellos, Sybase representaba el segundo o tercer DBMS relacional que habían construido. Sybase posicionó su producto como «el DBMS relacional para aplicaciones en línea» y enfatizaban las características técnicas y arquitectónicas que les distingüían de otros productos DBMS basados en SQL. Estas características incluían:

- Una arquitectura cliente/servidor, con software cliente ejecutándose en estaciones de trabajo Sun y VAX y en PC de IBM y el servidor ejecutándose en sistemas VAX/VMS o Sun.
- Un servidor multihilo que manejaba su propia gestión de tareas y entrada/salida para máxima eficiencia.

■ Un API programado, en vez de la interfaz SQL incorporado utilizada por la mayoría de los vendedores de DBMS.

■ Procedimientos almacenados, disparadores y un dialecto Transact-SQL que extendía SQL hasta convertirlo en un lenguaje de programación servidor completo.

Su agresivo «marketing» y un catálogo de primera clase de inversores financieros obtuvieron para Sybase la atención de los analistas de la industria, pero fue el posterior acuerdo OEM con Microsoft y Ashton-Tate el que posicionó a la compañía como un vendedor de DBMS en alza. El renombrado SQL Server, el DBMS de Sybase fue transportado a OS/2, y comercializado por Microsoft para vendedores de sistemas informáticos (junto con OS/2), y por Ashton-Tate a través de sus canales de ventas. En 1989, Lotus Development se unió a Microsoft y Ashton-Tate como inversores en Sybase. Aunque bastante lejos de la pretensión inicial de Sybase sobre aplicaciones OLTP basadas en VAX, estos eventos en el mercado de PCs reforzaron el «apel prominente de Sybase».

Las innovaciones en el producto Sybase le hicieron el DBMS técnicamente más «rápido» de finales de los ochenta, y el comienzo tardío de la empresa le dio un valor competitivo técnico. Para 1990, estas innovaciones habían provocado una respuesta competitiva por parte de los otros vendedores de DBMS independientes importantes, muchos de los cuales habían anunciado planes para su propio soporte de arquitecturas cliente/servidor, dialectos de SQL procedural, procedimientos almacenados, disparadores, etc. El desarrollo de productos Sybase ha tendido a concentrarse en las características esenciales del propio DBMS, con relativamente menos atención a las herramientas y utilidades asociadas. La línea de productos incluye SQL Workbench, un entorno de desarrollo y comprobación de aplicaciones con una interfaz de usuario gráfica, pero carece del conjunto extenso de paquetes de formularios, escritores de informes, paquetes gráficos y otras herramientas ofrecidas por otros vendedores de DBMS.

Tandem Computers, Inc.

Tandem es un vendedor importante de sistemas informáticos tolerantes a fallos. Muchos sistemas Tandem se venden a servicios financieros y compañías de transporte para uso en aplicaciones de procesamiento de transacciones en línea. Los sistemas de Tandem corren el sistema operativo propietario TXP, y las aplicaciones tolerantes a fallos se escriben generalmente en el Language Tandem Application Proprietary (TAL). Sin embargo, la gestión de base de datos es proporcionada por un DBMS relacional basado en SQL denominado NonStop SQL.

A causa del énfasis OLTP de Tandem, NonStop SQL ha sido pionero de varias técnicas especiales, tales como reflejo de disco, que son únicas entre las implementaciones SQL. NonStop SQL también ha aprovechado la arquitectura multiprocesadora de Tandem y proporciona capacidades de bases de datos distribuidas. La interfaz programada con NonStop SQL se realiza a través de SQL incorporado.

Unify Corporation

Unify Corporation es un suministrador de productos DBMS relacionales basados en UNIX y de herramientas de bases de datos. A mediados de los ochenta Unify, Informix, Ingres y Oracle eran los cuatro principales competidores de DBMS basado en UNIX, con Oracle e Ingres batallando por el dominio como DBMS «de alto extremo» mientras que Informix y Unify competían en el extremo bajo del mercado. El DBMS Unify, desde que creció hasta convertirse en Unify 2000, se basó en el modelo de datos en red, con una apariencia SQL para acceso relacional. Los punteros incorporados y otras características del modelo de datos en red dan a Unify excelente rendimiento, pero la falta de un fundamento relacional cierto ha restringido el nivel de soporte SQL que puede proporcionar.

Mientras el DBMS Unify sigue siendo parte importante de la línea de productos de la empresa, durante los últimos años el énfasis de Unify se ha desplazado a las herramientas de bases de datos. El entorno de desarrollo de aplicaciones Accell de la empresa es un lenguaje de cuarta generación avanzado diseñado para acelerar el desarrollo de aplicaciones de procesamiento de datos interactivas y orientadas a formularios. Inicialmente disponible sólo para uso con el DBMS Unify, Accell soporta ahora una gama de productos DBMS basados en SQL. Es un excelente ejemplo de una nueva clase de herramientas de desarrollo de aplicaciones de base de datos «portables» que han sido adaptadas para trabajar con una variedad de máquinas SQL.

C *Lista de empresas y productos*

Este apéndice contiene una lista de empresas y productos en el mercado de DBMS, la mayoría de los cuales se mencionan en este libro. La mayoría de los productos listados son sistemas de gestión de bases de datos basadas en SQL o herramientas de bases de datos. Las empresas aparecen en orden alfabético. Los principales productos de cada empresa aparecen en cursiva.

Acius
10351 Bubb Road
Cupertino, CA 95014
408-252-4444
4th DIMENSION
CL/I, HyperCard

Apple Computer, Inc.
20525 Mariani Avenue
Cupertino, CA 95014
408-996-1010
dBASE IV, SQL Server

Ashton-Tate Corporation
20101 Hamilton Avenue
Torrance, CA 90502
213-329-8000
dBASE IV, SQL Server

Borland International, Inc.
1800 Green Hills Road
Scotts Valley, CA 95066
408-438-8400
Paradox

Cincom Systems, Inc.
2300 Montan Avenue
Cincinnati, OH 45211
513-662-2300
Supra Version 2, TOTAL

Cognos, Inc.
3755 Riverside Drive
Ottawa, Ontario
Canada K1G 3Z4
613-738-1338
PowerHouse, PowerHouse Star-Base

CompuServe Data Technologies
1000 Massachusetts Avenue
Cambridge, MA 02138
617-661-9440
System 1032

Computer Associates International, Inc.
711 Stewart Avenue
Garden City, NY 11530
516-227-3300
CA-Universe, CA-Datacom/DB

Computer Corporation of America
4 Cambridge Center
Cambridge, MA 02142
617-492-8860
Model 204

Cullinet Software, Inc.
200 University Avenue
Westwood, MA 02090
617-329-7700
Enterprise:DB

DataEase International, Inc.
12 Cambridge Drive
Trumbull, CT 06611
203-374-8000
DataEase

Data General Corporation
4400 Computer Drive
Westboro, MA 01580
508-366-8911
DG/SQL

Digital Equipment Corporation
146 Main Street
Maynard, MA 01754
508-493-5111
VAX, Rdb/VMS, VAX SQL

Empress Software, Inc.
250 Vloor Street East
Toronto, Ontario
Canada M4W 1E6
416-922-1743
Empress

Gupta Technologies, Inc.
1040 Marsh Road
Menlo Park, CA 94025
415-321-9500
SQLWindow's

Hewlett-Packard Company
3000 Hanover Street
Palo Alto, CA 94304
415-857-1501
Allbase, HPSQL

IBM Corporation
Old Orchard Road
Armonk, NY 10504
914-765-1900
DB2, SQL/DS, SQL/400, OS/2 Extended Edition
Information Builders, Inc.
1250 Broadway
New York, NY 10001
212-736-4433
Focus

Informix Software, Inc.
4100 Bohannon Drive
Menlo Park, CA 94025
415-926-6300
Informix

Ingres Corporation
1080 Marina Village Parkway
Alameda, CA 94501
415-769-1400
INGRES

Interbase Software Corporation
209 Burlington Road
Bedford, MA 01730
617-275-3222
Interbase

Lotus Development Corporation
55 Cambridge Parkway
Cambridge, MA 02142
617-577-8500
Lotus 1-2-3, Lotus/DBMS

Microrim, Inc.
3925 159th Avenue N.E.
Redmond, WA 98073
206-885-2000
R:BASE

Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
206-882-8080
SQL Server

Novell, Inc.
6034 W. Courtyard Drive
Austin, TX 78730
512-346-8380
NetWare SQL, XQL

Oracle Corporation
20 Davis Drive
Belmont, CA 94002
415-598-8000
Oracle

Progress Software Corporation
5 Oak Park
Bedford, MA 01730
617-275-4500
415-598-8000
PROGRESS

Revelation Technologies, Inc.
3633 136th Place S.E.
Bellevue, WA 98006
206-643-9898
Advanced Revelation

ShareBase Corp.
14600 Winchester Boulevard
Los Gatos, CA 95030
408-378-7000
ShareBase

Software AG
11190 Sunrise Valley Drive
Reston, VA 22091
703-860-5050
Adabas

Software Publishing Corporation
1901 Landings Drive
Mountain View, CA 94039
415-962-8910
PFS:File

Sybase, Inc.

6475 Christie Avenue
Emeryville, CA 94608
415-596-3500
Sybase SQL Server

Tandem Computers, Inc.
19333 Vallico Parkway
Cupertino, CA 95014
408-725-6000
NonStop SQL

Unify Corporation
3870 Rosin Court
Sacramento, CA 32256
916-920-9092
UNIFY, ACCELL

XDB Systems, Inc.
7309 Baltimore Avenue
College Park, MD 20740
301-779-6030
XDB-SQL, XDB-Server

D Sintaxis de SQL ANSI/ISO

El estándar SQL ANSI/ISO especifica la sintaxis del lenguaje SQL utilizando una notación BNF formal. Desgraciadamente, el estándar es difícil de leer y entender por varias razones. En primer lugar, el estándar especifica el lenguaje de abajo a arriba en vez de arriba a abajo, haciendo difícil obtener una «idea general» de una sentencia SQL. En segundo lugar, el estándar utiliza términos poco familiares (tales como *expresión-de-tabla y predicado*). Finalmente, la BNF del estándar tiene muchas capas de profundidad, lo cual proporciona una especificación muy precisa, pero enmascara la estructura relativamente sencilla del lenguaje SQL.

Este apéndice presenta una BNF completa y simplificada del SQL estándar ANSI/ISO tal como es utilizado habitualmente. Específicamente:

- Las sentencias de definición de datos se tratan como sentencias autónomas en vez de como parte de una definición de esquema.
- El lenguaje de módulos se omite, ya que es reemplazado en todas las implementaciones actuales de SQL por SQL incorporado o por una API de SQL.
- Los componentes del lenguaje se designan por sus nombres comunes, en vez de por los nombres técnicos usados en el estándar.

La BNF de este apéndice utiliza estas convenciones:

- Las palabras clave de SQL aparecen en letras todas **MAYUSCULAS**.
- Los elementos sintácticos se especifican en *cursivas*.
- La notación *lista-de-elementos* indica un *elemento* o una lista de *elementos* separados por comas.

- Las barras verticales () indican una elección entre dos o más elementos sintácticos alternativos.
- Los corchetes [] indican un elemento sintáctico opcional.
- Las llaves {} indican una elección entre elementos sintácticos requeridos.

Sentencias de definición de datos

Estas sentencias definen la estructura y seguridad de una base de datos.

```
CREATE TABLE tabla ( lista-de-items-def-tabla )
CREATE VIEW vista [ ( lista-de-columnas ) ]
AS expresión-de-consulta
[ WITH CHECK OPTION ]
GRANT { ALL | PRIVILEGES } | lista-de-privilegios
ON tabla
TO { PUBLIC | lista-de-usuarios }
[ WITH GRANT OPTION ]
```

Sentencias básicas de manipulación de datos

Estas sentencias recuperan o modifican datos de una base de datos.

```
SELECT [ ALL | DISTINCT ] { lista-de-items-de-selección | * }
INTO lista-de-ref-tablas
FROM lista-de-ref-tablas
[ WHERE condición-de-búsqueda ]
INSERT INTO tabla [ ( lista-de-columnas ) ]
VALUES { ( lista-de-ítems-de-inserción | expresión-de-consulta ) }
DELETE FROM tabla [ WHERE condición-de-búsqueda ]
UPDATE tabla SET lista-de-assignaciones [ WHERE condición-de-búsqueda ]
```

Sentencias de procesamiento de transacciones

Estas sentencias señalan el fin de una transacción SQL.

```
COMMIT WORK
ROLLBACK WORK
```

Sentencias basadas en cursor

Estas sentencias de SQL programado soportan recuperación de datos y actualización posicionada de datos.

```
DECLARE cursor CURSOR FOR expresión-de-consulta [ ORDER BY lista-de-items-de-ordenación ]
OPEN cursor
CLOSE cursor
FETCH cursor INTO lista-de-variables-principales
DELETE FROM tabla WHERE CURRENT OF cursor
UPDATE tabla SET lista-de-assignaciones WHERE CURRENT OF cursor
```

Expresiones de consulta

Estas expresiones aparecen en sentencias SQL para especificar los datos que van a ser afectados por la sentencia.

```
expresión-de-consulta: ítem-de-consulta | expresión-de-consulta UNION [ ALL ] ítem-de-consulta
ítem-de-consulta: expresión-de-consulta | (expresión-de-consulta)
expresión-de-consulta: | SELECT [ ALL | DISTINCT ] { lista-de-items-de-selección | * }
FROM lista-de-ref-tablas
[ WHERE condición-de-búsqueda ]
[ GROUP BY lista-de-ref-columnas ]
[ HAVING condición-de-búsqueda ]
```

subconsulta:

```
SELECT [ ALL | DISTINCT ] { lista-de-items-de-selección | * }
```

condiciones de búsqueda:

as expresiones seleccionan filas de la base de datos para su procesamiento.

condición-de-búsqueda: item-de-búsqueda | item-de-búsqueda { AND | OR } item-de-búsqueda

item-de-búsqueda: [NOT] { test-de-búsqueda | condición-de-búsqueda } | test-comparación | test-entre | test-como | test-nulo | test-conjunto | test-cuantificado | test-existencia

test-comparación: expr { ' | < | > | < | < = | > | > = } { expr | subconsulta }

test-entre: expr [NOT] BETWEEN expr AND expr

test-como: ref-columna [NOT] LIKE valor [ESCAPE] valor

test-nulo: ref-columna IS [NOT] NULL

test-conjunto: expr [NOT] IN { lista-de-valores | subconsulta }

test-cuantificado: expr{ = | < > | < | < = | > | > = } [ALL | ANY | SOME] subconsulta

test-existencia: EXISTS subconsulta

Condiciones de búsqueda

<p>condiciones de búsqueda</p> <hr/> <p>as expresiones seleccionan filas de la base de datos para su procesamiento.</p>	<pre> FROM lista-de-ref-tablas [WHERE condición-de-búsqueda] [GROUP BY lista-de-ref-columnas] [HAVING condición-de-búsqueda] </pre>	<p>condición-de-búsqueda: item-de-búsqueda item-de-búsqueda { AND OR } item-de-búsqueda</p> <p>item-de-búsqueda: [NOT] { test-de-búsqueda (condición-de-búsqueda) }</p> <p>test-comparación: test-entre test-como test-nulo test-existencia</p> <p>expr: ' < > < = > > = { expr subconsulta }</p> <p>test-entre: expr [NOT] BETWEEN expr AND expr</p> <p>ref-columna: ref-columna [NOT] LIKE valor [ESCAPE] valor</p> <p>ref-columna IS: ref-columna IS [NOT] NULL</p> <p>expr: [NOT] IN { lista-de-valores subconsulta }</p> <p>expr{: = < > < = > = ALL ANY SOME }</p> <p>subconsulta: subconsulta</p> <p>test-existencia: EXISTS subconsulta</p>
--	---	---

Elementos de sentencias

<p>Estos elementos aparecen en varias sentencias SQL</p>	<table border="0"> <tr> <td style="vertical-align: top; padding-right: 20px;"> <i>asignación:</i> <i>item-de-ordenación:</i> <i>item-de-inserción:</i> <i>item-de-selección:</i> <i>ref-tabla:</i> <i>ref-columna:</i> <i>item-def-tabla:</i> <i>def-columna:</i> <i>restricción-de-columna:</i> <i>privilegio:</i> </td><td style="vertical-align: top;"> <pre> columna = { expr NULL } { ref-columna entero } [ASC DESC] { valor NULL } expr tabla [alias-de-tabla] [{ tabla alias } .] columna def-columna UNIQUE (lista-de-columnas) PRIMARY KEY (lista-de-columnas) FOREIGN KEY (lista-de-columnas) REFERENCES tabla [(lista-de-columnas)] columna tipo-de-datos [DEFAULT { literal USER NULL }] restrictión-de-columna: [restricción-de-columna] NOT NULL [PRIMARY KEY] CHECK (condición-de-búsqueda) REFERENCES tabla [(lista-de-columnas)] SELECT INSERT DELETE UPDATE [(lista-de-columnas)] </pre> </td></tr> </table>	<i>asignación:</i> <i>item-de-ordenación:</i> <i>item-de-inserción:</i> <i>item-de-selección:</i> <i>ref-tabla:</i> <i>ref-columna:</i> <i>item-def-tabla:</i> <i>def-columna:</i> <i>restricción-de-columna:</i> <i>privilegio:</i>	<pre> columna = { expr NULL } { ref-columna entero } [ASC DESC] { valor NULL } expr tabla [alias-de-tabla] [{ tabla alias } .] columna def-columna UNIQUE (lista-de-columnas) PRIMARY KEY (lista-de-columnas) FOREIGN KEY (lista-de-columnas) REFERENCES tabla [(lista-de-columnas)] columna tipo-de-datos [DEFAULT { literal USER NULL }] restrictión-de-columna: [restricción-de-columna] NOT NULL [PRIMARY KEY] CHECK (condición-de-búsqueda) REFERENCES tabla [(lista-de-columnas)] SELECT INSERT DELETE UPDATE [(lista-de-columnas)] </pre>
<i>asignación:</i> <i>item-de-ordenación:</i> <i>item-de-inserción:</i> <i>item-de-selección:</i> <i>ref-tabla:</i> <i>ref-columna:</i> <i>item-def-tabla:</i> <i>def-columna:</i> <i>restricción-de-columna:</i> <i>privilegio:</i>	<pre> columna = { expr NULL } { ref-columna entero } [ASC DESC] { valor NULL } expr tabla [alias-de-tabla] [{ tabla alias } .] columna def-columna UNIQUE (lista-de-columnas) PRIMARY KEY (lista-de-columnas) FOREIGN KEY (lista-de-columnas) REFERENCES tabla [(lista-de-columnas)] columna tipo-de-datos [DEFAULT { literal USER NULL }] restrictión-de-columna: [restricción-de-columna] NOT NULL [PRIMARY KEY] CHECK (condición-de-búsqueda) REFERENCES tabla [(lista-de-columnas)] SELECT INSERT DELETE UPDATE [(lista-de-columnas)] </pre>		

Elementos simples

Estos elementos son los nombres y constantes básicas que aparecen en sentencias de SQL.

Expressiveness

<i>as</i>	as expresiones se utilizan en SQL en listas de selección y en condiciones de búsqueda.
<i>tabla:</i>	un nombre de tabla;
<i>columnas:</i>	un nombre de columnas;
<i>usuario:</i>	un nombre de usuario de base de datos;
<i>variable:</i>	un nombre de variable del lenguaje anfitrión;
<i>literal:</i>	un literal numérico o de caracteres entrecomillado;
<i>entero:</i>	un número entero;
<i>tipo-de-datos:</i>	un tipo de datos SQL;
<i>alias:</i>	un identificador SQL;
<i>cursor:</i>	un identificador SQL.
<i>expr:</i>	
<i>item-de-expr:</i>	<i>item-de-expr</i> <i>item-de-expr</i> { + - * / } <i>item-de-expr</i>
<i>valor:</i>	<i>literal</i> <i>USER</i> <i>variable-principal</i>
<i>variable-principal:</i>	<i>variable</i> [[INDICADOR] <i>variable</i>]
<i>función:</i>	COUNT (*) <i>func-divisivas</i> <i>func-todos</i>
<i>func-divisivas:</i>	{ AVG MAX MIN SUM COUNT } (DISTINCT <i>ref-columna</i>)
<i>func-todos:</i>	{ AVG MAX MIN SUM COUNT } ([ALL] <i>expr</i>)

Marcas registradas

1-2-3 ® Lotus Development Corp.
4th DIMENSION ® Acius
ACCELL ® Unify Corp.
Adabas ™ Software AG
Advanced Revelation ™ Revelation Technologies, Inc.
Allbase ™ Hewlett-Packard Co.
AS/400 ™ International Business
Machines Corp.
Computer Associates International, Inc.
CA-Datocom/DB ® Computer Associates International, Inc.
CA-Universe ™ International, Inc.
COMPAQ ® COMPAQ Computer Corp.
DataEase ™ DataEase International, Inc.
DataLens ™ Lotus Development Corp.
DB2 ™ International Business
Machines Corp.
dBASE II ® Ashton-Tate Corp.
dBASE III ® Ashton-Tate Corp.
dBASE IV ® Ashton-Tate Corp.
Empress ® Empress Software, Inc.
Focus ® Information Builders, Inc.
HP ® Hewlett-Packard Co.
HyperCard ® Apple Computer, Inc.
IDMS ™ International Business
Machines Corp.

Image ® Hewlett-Packard Co.
 Informix ® Inforinx Software, Inc.
 INGRES ™ Ingres Corp.
 INGRES/PCLink ™ Ingres Corp.
 INGRES/STAR ™ Ingres Corp.
 Intel ® Intel Corp.
 Interbase ™ Interbase Software Corp.
 Lotus/DBMS ™ Lotus Development Corp.
 Macintosh ® Apple Computer, Inc.
 MPE/XL ™ Hewlett-Packard Co.
 MS-DOS ® Microsoft Corp.
 NETBIOS ™ International Business Machines Corp.
 Novell, Inc.
 NetWare ® Tandem-Computers, Inc.
 NonStop SQL ™ Sybase, Inc.
 Open Server ™ AT&T
 OpenLook ™ Oracle Corp.
 Oracle ® Oracle Corp.
 Oracle Server ™ Oracle Corp.
 OS/2 ® International Business Machines Corp.
 OS/400 ™ International Business Machines Corp.
 Paradox ® Borland International, Inc.
 PFS: ® Software Publishing Corp.
 PL/SQL ™ Oracle Corp.
 PowerHouse ® Cognos, Inc.
 PowerHouse StarBase ™ Cognos, Inc.
 Professional Oracle ™ Oracle Corp.
 PROGRESS ® Progress Software Corp.
 PS/2 ® International Business Machines Corp.
 R:BASE ® Microm, Inc.
 ShareBase ™ ShareBase Corp.
 SQL/400 ™ International Business Machines Corp.
 SQL/DS ™ International Business Machines Corp.
 SQL*Net ® Oracle Corp.
 SQLBase ® Gupta Technologies, Inc.
 SQLPLUS ™ Oracle Corp.
 Supra ® Cincom Systems, Inc.
 Sybase ® Sybase, Inc.

System 370 ™ International Business Machines Corp.
 System 1032 ® CompuServe, Inc.
 Systems Application Architecture ™ (SAA) International Business Machines Corp.
 TOTAL ® Cincom System, Inc.
 Transact-SQL ™ Digital Equipment Corp.
 ULTRIX ® Unify Corp.
 UNIFY ® AT&T
 UNIX ® Digital Equipment Corp.
 VAX ® Digital Equipment Corp.
 VMS ® Informix Software, Inc.
 Wingz ® XDB Systems, Inc.
 XDB ® XDB Systems, Inc.
 XDB-Server ™ XDB Systems, Inc.
 XDB-SQL ™ XDB Systems, Inc.
 XQL ® Novell, Inc.

International Business Machines Corp.
 CompuServe, Inc.
 International Business Machines Corp.
 Cincom System, Inc.
 Sybase, Inc.
 Digital Equipment Corp.
 Unify Corp.
 AT&T
 Digital Equipment Corp.
 Digital Equipment Corp.
 Informix Software, Inc.
 XDB Systems, Inc.
 XDB Systems, Inc.
 XDB Systems, Inc.
 Novell, Inc.

Indice

- <, test de comparación, 100-103, 196-198
- <=, test de comparación, 100-103, 196-198
- <>, test de comparación, 100-103, 196-198
- =, test de comparación, 100-103, 196-198
- >, test de comparación, 100-103, 196-198
- >=, test de comparación, 100-103, 196-198
- ?, como marcador de parámetro, 454
 - * , usado en este libro para marcar secciones avanzadas, xxiii
 - * , usado en sentencia SELECT, 95
- Accell, 590, 596
- Acceso a datos distribuidos:
 - estapas IBM de, 536-543
 - operaciones, 542-543
 - operaciones remotas, 536-538
 - tablas, 544-549
- acciones, 540-542
- transacciones remotas, 538-540
- Acceso interactivo, 10
- Acceso programado a base de datos, 10
- Acins, 591
- Actualización de la base de datos, 19
- Actualización de todas las filas (UPDATERE), 238
- Actualizaciones de base de datos, 221-240
 - basadas en cursor, 439-443
 - posicionadas, 513-515
- Actualizaciones de vistas, 344-349
- comprobación (CHECK OPTION), 347-349
- en productos SQL comerciales, 346-347
- y en estándar ANSI/ISO, 346
- Actualizaciones perdidas, 276-277
- Actualizaciones posicionadas, 513-515
- Adabas, 596
- Advanced Revelation, 595
- Alias de tablas, 148-149
- ALL, test (en subconsultas), 202-208
- Allbase, 583, 594
- Almacenamiento físico, definición, 307-308
- ALTER, sentencias, 318
- ALTER TABLE, sentencia, 309-313
 - diagrama sintáctico, 310
- Análisis de sentencias SQL, 397
- AND, condición de búsqueda, 112-115
- tabla de verdad, 114

- ANSI/ISO, definiciones, 9
 ANSI/ISO, estándares, 28-29, 67, 71
 modelo de transacción, 269-271
 sintaxis SQL, 597-601
 tabla de palabras clave SQL, 68
 tabla de tipos de datos SQL, 70
 y actualizaciones de vistas, 346
 y DDL, 326-329
 y de catálogo de sistema, 378
 y REVOKE, 371-372
 ANY, test (en subconsultas), 202-208
 API (interfaz para programación de aplicaciones), 489-528
 conceptos, 489-490
 definición dc, 489-490
 Oracle Call Interface, 520-522
 SQL Server, 490-519
 SQL-Base, 522-528
 tabla de funciones dblib, 492
 API de SQL Base, 522-528
 funciones de conjunto de resultados, 527
 funciones de datos extensos, 527
 funciones de utilidad, 528
 tabla de funciones, 525
 tabla de rango, 103-106
 BETWEEN, test de rango, 103-106
 Biblioteca de base de datos (*dblib*), 491, 492
 Blobs, 564, 582
 Borland International, Inc., 580, 592
 Base de datos utilizada en este libro, 573-578
 BEGIN TRANSACTION, sentencia, 272
 BETWEEN, test de rango, 103-106
 Biblioteca de base de datos (*dblib*), 491, 492
 Blobs, 564, 582
 Borland International, Inc., 580, 592
 Área de comunicaciones SQL/SQLCA, 411, 412
 Arquitectura cliente/servidor, 11, 34-35
 Arquitectura de base de datos, 32-33.
 multibase de datos, 322-324
 multiubicación, 324-326
 simple, 320-322
 Arquitectura de sistema, 11, 32-35, 509
 Arquitectura monobase de datos, 320-322
 Arquitectura servidor, 33-34
 Ashton-Tate Corporation, 580, 591
 Asterisco (*), usado en sentencia SELECT, 95-96
 Autocomposiciones, 145-148
 Autorización de usuario, 354-356
 Autorización, identificador (id), 326
 AVG, función de columna, 166-167
 Bancos de prueba, 559-560
 Bases de datos, 4
 adición de datos a, 18, 222-229
 actualización, 19
 acceso programado, 10
 creación, 20-21, 297-329
 ejemplo utilizado en este libro, 573-578
 en red, 47-49
 estructura, 318-326
 herramientas, 568
 jerárquicas, 44-47
 modificación de datos en, 235-239
 orientadas a objeto, 570-572
 protección, 19-20
 supresión de datos de, 19, 230-235
 Base de datos dinámica, 11
 Base de datos ejemplo utilizada en este libro, 573-578
 Base de datos en red, 47-49
 Base de datos orientada a objetos, 570-572
 Base de datos utilizada en este libro, 573-578
 BEGIN TRANSACTION, sentencia, 272
 BETWEEN, test de rango, 103-106
 Biblioteca de base de datos (*dblib*), 491, 492
 Blobs, 564, 582
 Borland International, Inc., 580, 592
 CA-Datacom/DB, 592
 CA-Universe, 592
 CASCADE, regla de supresión (DB2), 250-251, 253
 Caracteres de escape y comparación con patrones, 110
 izquierda y derecha, 158-159
 multibase de datos, 322-324
 multiubicación, 324-326
 simple, 320-322
 Arquitectura de sistema, 11, 32-35, 509
 Arquitectura monobase de datos, 320-322
 Arquitectura servidor, 33-34
 Ashton-Tate Corporation, 580, 591
 Asterisco (*), usado en sentencia SELECT, 95-96
 Autocomposiciones, 145-148
 Autorización de usuario, 354-356
 Autorización, identificador (id), 326
 AVG, función de columna, 166-167
 información de tablas, 380-389
 información de usuarios, 389-390
 información de vistas, 384-385
 y el estándar ANSI/ISO, 378
 y herramientas de consulta, 376-378
 Cerramiento (*locking*):
 explícito, 289-290
 interbloqueos, 285-288
 modo de estabilidad de cursor, 291
 niveles de, 282-284
 niveles de aislamiento, 290-292
 parámetros para, 292
 técnicas avanzadas, 289-292
 y transacciones, 282-292
 Ciclos referenciales, 254-258
 Cierre (*lock*) explícito, 289-290
 Cierres (*lock*) exclusivos, 285
 Cierres (*lock*s) compartidos, 285
 Cierres (*lock*s), compartidos y exclusivos, 285
 Comisión Systems, Inc., 592
 CL/1, 591
 Cláusulas en sentencias SQL, 67
 Claves foráneas, 57-58
 definición, 304-306
 modificación, 312-313
 y valores NULL, 258-259
 Claves primarias, 54-55
 definición, 304-306
 modificación, 312-313
 Cliente/servidor:
 aplicaciones, 566
 arquitectura, 11, 34-35, 509
 Cliente, sistema, 509
 CLOSE, sentencia, 436-437
 diagrama sintáctico, 437
 CLOSE, sentencia (dinámica), 478-479
 CODASYL, base de datos para procesamiento de pedidos, 49
 Codd, las doce reglas de, 58-61
 tabla de, 60-61
 Codd, Ted, 58
 Códigos de tipos de datos (DB2), 473
 Cognos, Inc., 592
 Columnas:
 adición a una tabla existente, 311
 agrupación múltiple, 178-181

- Condiciones de búsqueda compuestas, 112-115
 Consultas de dos tablas, ejemplo, 127-129
 Consultas de búsqueda de grupo (HAVING), 185-190
 restricciones en, 188-189
 y valores NULL, 189-190
 Condiciones de búsqueda para consultas, 100-115
 compuestas, 112-115
 de grupo (HAVING), 185-190
 para subconsultas, 196-208
 sintaxis de, 600
 Conexión en red, 32-35
 Constantes, 76-80
 Constantes de:
 cadenas, 77-78
 fecha, 78-79
 hora, 78-79
 Constantes numéricas, 77
 Constantes simbólicas, 79-80
 Consultas:
 agrupadas (GROUP BY), 175-185
 condiciones de búsqueda para simples, 100-115
 con tres o más tablas, 135-138
 dinámicas, 467-479, 515-519
 ejemplo con dos tablas, 127-129
 multitable, 127-161
 padre/hijo, 132-134
 para datos sumarios, 163-190
 reglas de procesamiento con GROUP BY, 179
 reglas para funciones de columnas, 171
 reglas para procesamiento multitable, 152-153
 simples, 85-125
 SQL Server, 499-505
 Consultas agrupadas (GROUP BY), 175-185
 basadas en múltiples columnas, 178-181
 restricciones en, 182-183
 valores NULL en, 183-185
 Consultas almacenadas (*Véase* Vistas)
 Consultas dinámicas, 467-479, 515-519
- Consultas de una fila, en SQL incorporado, 423-430
 Consultas multifila (*Véase* Composiciones)
 Consultas multifila, en SQL incorporado, 430-438
 cursores de desplazamiento, 437-438
 procesamiento, 432
 sentencia CLOSE, 436-437
 sentencia DECLARE CURSOR, 433-435
 sentencia FETCH, 436-438
 sentencia OPEN, 435
 Consultas sumarias, 163-190
 diagramas de, 165
 Conversiones de tipos de datos (Oracle), 486
 COUNT, función de columna, 169
 COUNT (*) función de fila, 170
 Creación de una base de datos, 20-21, 297-329
 CREATE/DROP INDEX, sentencias, 314-317
 CREATE/DROP SYNONYM, sentencias, 313-314
 CREATE INDEX, sentencia, 317
 CREATE, sentencias, 318
 CREATE TABLE, sentencia, 300-308
 diagrama sintáctico, 301
 CREATE VIEW, sentencia, 335-344
 diagrama sintáctico, 335
 4th Dimension, 591
 Cullinet Software, Inc., 593
 Cursor:
 desplazamiento, 437-438
 en sentencias de SQL incorporado, 433-435
 posicionamiento con OPEN, FETCH y CLOSE, 437
 y procesamiento de transacciones, 443-444
 CURSOR, 437-438
 Cursos de desplazamiento, 437-438
 sentencia FETCH para, 438
 CHECK OPTION, sentencia, 347-349
- Data Definition Language (DDL), 298
 tabla de sentencias, 319
 y el estándar ANSI/ISO, 326-329
 DataEase, 593
 DataEase International, Inc., 593
 Data General Corporation, 593
 Data Manipulation Language (DML), 297
 Datos, falta de, 83
 dBASE IV, 580, 591
 DB2, 583, 594
 códigos de tipo de datos para, 473
 tablas que implementan permisos, 391
 dblib (database library), 491
 tabla de funciones, API, 492
 tabla de funciones de modo inspección, 513
 DBMS, 489-528, 579-590
 DECLARE CURSOR, sentencia, 433-435
 con cláusula FOR UPDATE, 443
 diagrama sintáctico, 433
 (dinámica) diagrama sintáctico, 475
 DECLARE STATEMENT, sentencia, 466
 diagrama sintáctico, 466
 DECLARE TABLE, sentencia, 410
 diagrama sintáctico, 410
 Definición de datos, sentencias, 598
 DELETE posiciónada, sentencia, 439-443
 diagrama sintáctico, 439
 DELETE, sentencia, 230-232
 con subconsulta, 232-235
 diagrama sintáctico, 230
 posiciónada, 439-443
 (posiciónada) diagrama sintáctico, 439
 DESCRIBE, sentencia, 472-474
 diagrama sintáctico, 472
 Oracle, 483
 DG/SQL, 593
 Diagrama (*Véase* Diagramas sintácticos)
 Diagramas sintácticos:
 cláusula FROM, 149
 cláusula ORDER BY, 115
 cláusula WHERE, 112
 ejemplo, 68

- funciones columna, 166
 sentencia ALTER TABLE, 310
 sentencia COMMENT, 387
 sentencia COMMIT, 269
 sentencia CREATE INDEX, 317
 sentencia CREATE TABLE, 301
 sentencia CREATE VIEW, 335
 sentencia DECLARE CURSOR, 433
 sentencia DECLARE CURSOR (dinámica), 475
 sentencia DECLARE STATEMENT, 466
 sentencia DECLARE TABLE, 410
 sentencia DELETE, 230
 sentencia DELETE posiciónada, 439
 sentencia DESCRIBE, 472
 sentencia DROP TABLE, 309
 sentencia EXECUTE IMMEDIATE, 451
 sentencia EXECUTE, 458
 sentencia FETCH, 435
 sentencia FETCH (dinámica), 478
 sentencia GRANT, 363
 sentencia LABEL, 387
 sentencia LOCK TABLE, 290
 sentencia INSERT de una fila, 223
 sentencia INSERT multifila, 227
 sentencia OPEN, 435
 sentencia OPEN (dinámica), 476
 sentencia PREPARE, 457
 sentencia REVOKE, 369
 sentencia ROLLBACK, 269
 sentencia SELECT singular, 424
 sentencia UPDATE, 235
 sentencia UPDATE posiciónada, 440
 sentencia WHENEVER, 414
 subconsulta, 193
 test de comparación, 101
 test de comparación en subconsulta, 197
 test de comparación cuantificada, 203
 test de correspondencia con patrón, 108
 test de existencia (EXISTS), 200
 test de pertenencia a conjunto, 106
 test de pertenencia a conjunto en subconsulta (IN), 198

Digital Equipment Corporation, 581, 593	con variables principales, 458
Difusión, protocolo de cumplimiento	diagrama sintáctico, 458
Por, 550	Expresiones, 80-81
Disparadores:	sintaxis de, 600
concepto de, 261	Expresiones de consulta, sintaxis de, 599-
e integridad referencial, 262-263	600
futuro de, 263-264	
DISTINCT, palabra clave, 174-175	FETCH, sentencia, 436
en sentencias SELECT, 97	diagrama sintáctico, 435
Divisiones de tablas:	para cursoros de desplazamiento, 437-
horizontal, 544-546	438
vertical, 546-547	FETCH, sentencia (dinámica), 477-478
Divisiones horizontales de tabla, 544-546	diagrama sintáctico, 478
Divisiones verticales de tablas, 546-547	Filas:
División de una columna, 53	actualización (UPDATE) de todas, 238
DROP, sentencias, 313-317	selección con la cláusula WHERE, 98-
DROP TABLE, sentencia, 308-309	100
DSRI, 581	supresión de todas, 232-235
	(Véase <i>también Filas duplicadas</i>)
Eliminación de tablas, 308-309	Focus, 594
Ellipsis verticales utilizados en este texto,	Foráneas, claves (Véase Claves foráneas)
xxv	FROM, cláusula, 88
Emparejamiento múltiple, columnas, 135	diagrama sintáctico, 149
Empresas, lista de (Apéndice C), 591-596	Formatos (SQL IBM) de fecha y hora,
Empress, 593	tabla de, 79
EnterpriseDB, 593	Filas duplicadas:
Equicomposiciones (composiciones sim-	eliminación de, 96-97, 174-175
ples), 129-141	y UNION, 121-122
ESCAPE, cláusula, 110	Funciones:
Esquema de base de datos, 326	de columna, 163-175
Establecimiento de cursor, modo de, 444	internas, 82
Estdárdares:	Funciones de columna, 163-175
IBM, 30	cálculo de promedios, 166-167
SQL, 9, 28-32	cálculo de totales, 165-166
X/Open, 30	diagrama sintáctico, 166
(Véase <i>también Estándares ANSI/ISO</i>)	en una lista de selección, 170-172
Estructura de base de datos, 318-326	recuento de valores, 169-170
Estructuras de datos, 427	reglas de procesamiento de consultas
uso como variables principales, 429	con, 171
Etiqueta (catálogo de sistema), 386	y valores NULL, 172-174
EXECUTE IMMEDIATE, sentencia,	
451-453	
diagrama sintáctico, 451	Gestión de base de datos, 23-24
EXECUTE, sentencia, 457-465	Gestión de base de datos distribuidos,
con SQLDA, 458-465	531-553, 569-570

desafíos de la, 531-536	diagrama sintáctico, 223
GRANT, sentencia (privilegios), 360-368	INSERT, sentencia:
diagrama sintáctico, 363	inscripción de todas las columnas, 226
Y REVOKE, 371-372	inscripción de valores NULL, 225-226
GROUP BY, 175-185	monofila, 222-226
Grupos de usuarios, 356-357	multifila, 226-229
Gupta Technologies, Inc., 581-582, 593	Inspección, tabla de funciones dblib de modo, 513
	Integridad:
	de entidades, 245-247
	referencial, 247-259
	Integridad de entidad, 245-247
	Integridad de datos, 241-264
	comprobación de validez, 244-245
	restrictiones sobre, 241
	Integridad referencial, 247-259
	problemas, 248-250
	y disparadores, 262-263
	Interbase, 594
	Interbase Software Corporation, 594
	Interbloqueos, 285-288
	Interfaz para programación de aplicaciones (Véase API)
	Interfaces de llamada de función, 520-528
	Introductores a sentencias de SQL incorporado, 407
	IS NULL, 110-112
	diagrama sintáctico, 111
	Jerárquica, base de datos, 44-47
	LABEL, sentencia, 387
	Lenguaje de consulta interactivo, SQL como, 6
	Indicadoras, variables, 421-422
	Índices (CREATE/DROP INDEX), 314-317
	Información de usuarios (catálogo de sistema), 389-390
	Information Builders, Inc., 594
	Informix, 594
	Informix Software, Inc., 584-585, 594
	Ingres, 585-586, 594
	Ingresa Corporation, 581, 585-586, 594
	INSERT de una fila, sentencia, 222-226

	Diagrama sintáctico, 223
	INSERT, sentencia:
	inscripción de todas las columnas, 226
	inscripción de valores NULL, 225-226
	monofila, 222-226
	multifila, 226-229
	Inspección, tabla de funciones dblib de modo, 513
	Integridad:
	de entidades, 245-247
	referencial, 247-259
	Integridad de entidad, 245-247
	Integridad de datos, 241-264
	comprobación de validez, 244-245
	restrictiones sobre, 241
	Integridad referencial, 247-259
	problemas, 248-250
	y disparadores, 262-263
	Interbase, 594
	Interbase Software Corporation, 594
	Interbloqueos, 285-288
	Interfaz para programación de aplicaciones (Véase API)
	Interfaces de llamada de función, 520-528
	Introductores a sentencias de SQL incorporado, 407
	IS NULL, 110-112
	diagrama sintáctico, 111
	Jerárquica, base de datos, 44-47
	LABEL, sentencia, 387
	Lenguaje de consulta interactivo, SQL como, 6
	Indicadoras, variables, 421-422
	Índices (CREATE/DROP INDEX), 314-317
	Información de usuarios (catálogo de sistema), 389-390
	Information Builders, Inc., 594
	Informix, 594
	Informix Software, Inc., 584-585, 594
	Ingres, 585-586, 594
	Ingresa Corporation, 581, 585-586, 594
	INSERT de una fila, sentencia, 222-226

LOCK TABLE, sentencia, 290	Nombres de sentencias, 457
diagrama sintáctico, 290	Nombres de tablas, 53
Lotes de sentencias (SQL Server), 495-496	en sentencia SQL, 69-70
Lokus 1-2-3, 594	No equicomposiciones, 141-142
Lotus/DBMS, 594	NonStop SQL, 589, 590, 596
Lotus Development Corporation, 594	NOT, condición de búsqueda, 112-115
Manipulación de datos, sentencias, 598	NOT FOUND, condición en sentencia
Máquina de base de datos, 6	SELECT singular, 426
MAX, determinación de valores extremos, 167-169	NULL, valores, 82-84
Mercado informático, impacto de SQL en el, 35-41	en consultas agrupadas, 183-185
Mercado, impacto de SQL en el, 35-41	en test de comparación, 103
Microrim, Inc., 595	inserción (INSERT), 225-226
Microsoft Corporation, 586-587, 595	y claves foráneas, 258-259
MIN, determinación de valores extremos, 167-169	y condiciones de búsqueda de grupo, 189
Minicomputadores y SQL, 37	y el servidor de SQL, 502
Model 204, 592	y función de columna, 172-174
Modelo de transacción, 269	y sentencias SELECT singulares, 426-428
Modelos de datos, 43-49	y variables principales, 421-422
Módulo de acceso, 480	y variables principales, 421-422
Módulo de petición de base de datos (DBRM), 401	Objeto binario grande, 564
Montador, 403	Objetos de seguridad, 357
Multifila, sentencia INSERT, 226-229	Objetos, nominación en base de datos, 69-70
diagrama sintáctico, 227	OPEN, sentencia, 435
Multibase de datos, arquitectura, 322-324	con parámetro variable principal, 476
Multib ubicación, arquitectura, 324-326	con paso de parámetro SQLDA, 477
Múltiple agrupación, columnas, 178-181	diagrama sintáctico, 435
Múltiples uniones, 123-125	diagrama sintáctico (dinámica), 476
Multiplicación de tablas, 150-152	OR, condición de búsqueda, 112-115
Netware SQL, 595	tabla de verdad, 114
Niveles de aislamiento, 290-292	Oracle, 580, 587-588, 595
Nominación de objetos de base de datos, 69-70	SQL dinámico en, 482-486
Nombre de tabla cualificado, 69	Oracle Call Interface, 520, 522
Names de columna, 53	tabla de funciones, 521
qualificados, 142-144	Oracle Corporation, 587-588, 595
en sentencias SQL, 69-70	Ordenación de resultados de consultas, 115-117
Nombres de columna cualificados, 70, 142-144	usando UNION, 122
Nombre de tabla cualificado, 69	ORDER BY, cláusula, 115-117
Nombres de columna cualificados, 70,	diagrama sintáctico, 115
142-144	OS/2 Extended Edition, 40, 584, 594

Palabras clave (SQL ANSI/ISO), tabla de, 68	y cursores, 443-444
Paradox, 592	y SQL, 38
Parámetros nominados en Oracle, 482-483	Procesamiento multusuario y transacciones, 276-281
Paso de parámetros con la sentencia OPEN, 477	Producto cartesiano, 150
Paso de privilegios (GRANT OPTION), 365-368	Productos DBMS empaquetados, 560-561
PC y SQL, 38-39	Programación, interfaces para productos SQL, 397
Permisos, implementación DB2, 391	Programación técnica de, 395-406
Peticiones:	PROGRESS, 595
distribuidas, 542-543	Progress Software Corporation, 595
remotas, 536-538	Protección de la base de datos, 19-20
PFS:File, 596	Protocolo de cumplimentación en dos fases, 549-552
Portabilidad de aplicaciones, 8, 30-32	Punteros, recuperación de datos usando, 502-503
PowerHouse, 592	R:BASE, 595
Precompilador, 400	Recuperación directa de filas (SQL Server), 503-505
predicados (condiciones de búsqueda), 100	Rdb/VMS, 556, 581, 593
PREPARE, sentencia, 454-457	RDBMS, 581
diagrama sintáctico, 457	Recorrido de través de registros, 46
Primarias, claves, 54-55	Recuperación de datos, 15-17
Privilejos (seguridad), 357-359	en SQL dinámico, 469-471
columna, 364-365	en SQL incorporado, 422-438
concesión (GRANT), 360-368	Recuperación de una fila (test de comunicación), 102
revocación (GRANT), 368-373	Registros (<i>log</i>) de transacción, 274-276
transferencia (GRAN OPTION), 365-368	Registros, recorrido a través de, 46
y el catálogo de sistema, 390-391	Reglas comerciales, forzamiento en programmas, 260-264
Privilegios de columna, 364-365	Reglas de supresión en DB2, 250-252
Privilegios de propiedad, 358	Relacional, modelo de base de datos, 43, 49-58
Problemas de transacción:	Relacionales, bases de datos, 4, 9, 43-62
actualizaciones perdidas, 276-277	procesamiento de pedidos, 50
datos inconsistentes, 279-280	relaciones en, 55-57
datos incumplimentados, 277-279	simples, 13-15
Procedimientos almacenados (SQL Server), 505-512	Relaciones padre/hijo de registros de base de datos, 45
Ventajas de, 510-512	múltiples, 47
y rendimiento del DBMS, 506-509	y consultas, 132-134
y rendimiento de red, 509-510	Rendimiento de red y procedimientos almacenados, 509-510
Procesamiento de consulta de una tabla, 117-118	
Procesamiento de transacciones, 265-293	
sintaxis de sentencias, 599	

RERICT, regla de supresión (DB2), 250-256
 Resultados de consulta, 89-91
 combinación (UNION), 118-124
 estructura tabular, 89
 ordenación (cláusula ORDER BY), 115-117
 Revelation Technologies, Inc., 595
 REVOKE, sentencia (privilegios), 368-373
 diagrama sintáctico, 369
 y el estándar ANSI/ISO, 371-373
 y la opción GRANT, 371
 ROLLBACK, sentencia, 267-269
 ROLLBACK TRANSACTION, sentencia, 272
 SET NULL, regla de supresión DB2, 251-256
 SharePoint, 595
 ShareBase Corporation, 595
 Sinónimos, 313-314
 Sistemas de gestión de base de datos (DBMS), 4
 componentes de, 6
 procesamiento de sentencias, 397-399, 450
 productos empaquetados, 560-561
 rendimiento, 506-509
 sinopsis de vendedores, 579-590
 tendencias del mercado, 556-557
 y vistas, 333-334
 Sistema de gestión de base de datos relacional (RDBMS), 24
 Software AG, 596
 Software, lista de empresas, (Apéndice C), 591-596
 Software Publishing Corporation, 596
 SQLBase, 582, 593
 SQLCODE, variable, 411-412
 en un programa C, 413
 en un programa COBOL, 413
 SQL/400, 583-584, 594
 SQLDA:
 códigos de tipo de datos para DB2, 473
 Oracle, 484-485
 paso de parámetros, 477
 para bases de datos IBM, 459

RESTRICT, regla de supresión (DB2), ejecución, 451
 ejecución en dos pasos, 453-465
 EXECUTE IMMEDIATE, 451-453
 FETCH, 477-478
 OPEN, 475-477
 PREPARE, 454-457
 Sentencias de SQL incorporado, 406-422
 cursor en, 430, 433-435
 declaración de tablas, 409, 410
 declaración de variables principales, 417-418
 introductores, 407
 terminadores, 407
 tratamiento de errores, 410-414
 uso de variables principales, 414-422
 SET NULL, regla de supresión DB2, 251-256
 y minicomputadores, 37
 y OS/2, 40-41
 y UNIX, 37
 SQLWindows, 582, 593
 Subconsultas, 191-218
 andadas, 210-211
 cláusulas HAVING, 214-216
 cláusulas WHERE, 194
 condiciones de búsqueda, 196-208
 correlacionadas, 211-214
 diagrama sintáctico, 193
 estructura de, 193-194
 referencias externas, 194-196
 test de comparación (=, < >, < <=, > > =), 196-198
 test de pertenencia a conjunto (IN), 199-200
 uso de, 191-196
 y composiciones, 208-210
 y DELETE, 232-235
 y UPDATE, 238-239
 Subconsultas anidadas, 210-211
 Subconsultas correlacionadas, 211-214
 SUM función de columna, 166-167
 Sumarización de datos, 17-18
 Supra Versión 2, 592
 Supresión:
 de datos, 19, 230-235
 de filas, 232
 Supresiones y actualizaciones basadas en cursor, 439-443

- Sybase, Inc., 588-589
 SYBASE, modelo de transacción, 273
 SYSCOLUMNS, tabla (OS/2 Extended Edition), 382
 SYSCOLUMNS, tabla (SQL Server), 384
 SYSFOREIGNKEYS, tabla (DB2), 388
 SYSOBJECTS, tabla (SQL Server), 382
 SYSPROTECTS, tabla (SQL Server), 391
 SYSRELS, tabla (DB2), 387
 SYSTEMTABLES, tabla (OS/2 Extended Edition), 380
 System 1032, 592
 Systems Application Architecture (SAA), 3, 36
 SYSUSERAUTH, tabla (DB2), 390
 SYSUSERS, tabla (SQL Server), 390
 SYSVIEWDEP, tabla (OS/2 Extended Edition), 385
 SYSVIEWS, tabla (OS/2 Extended Edition), 385
 Sybase SQL Server, 588-589, 596
- Tabla de productos, 151
 Tabla vacía, 54
 Tablas:
 creación, 300-308
 definición, 300-313
 definición de almacenamiento físico, 307-308
 definición de columnas, 301-303
 definición de claves primarias y foráneas, 304-306
 distribuidas, 544-547
 eliminación, 308-309
 estructura de, 52-54
 modificación de una definición, 309-313
 reflejadas, 548-549
 restricciones de unicidad, 306-307
 valores por omisión, 303-304
- Tablas de sistema, 59, 376, 379, 380-382
 Tablas fuente (de consulta), 88
 Tablas reflejadas, 548-549
 Tablas virtuales, 59
- Tandem Computers, Inc., 589-590
 Tendencias en rendimiento hardware, 557-558
 Terminadores de sentencia de SQL incorporado, 407
 Test cuantificados (ANY y ALL), 202-208
 Test de comparación:
 consideración de valores NULL, 102-103
 diagrama sintáctico, 101
 recuperación de una sola fila, 102
 subconsulta, 196-198
 Test de comparación con patrón (LIKE), 108-110
 diagrama sintáctico, 108
 y caracteres de escape, 110
 Test de existencia (EXISTS):
 diagrama sintáctico, 200
 en subconsultas, 200-202
 Test de pertenencia a conjunto (IN), 106-107
 diagrama sintáctico, 106
 en subconsultas, 199-200
 Test de rango (BETWEEN), 103-106
 diagrama sintáctico, 104
 Tipos de datos, 71-76
 complejos, 564-565
 diferencias, 74-76
 en SQL y lenguajes anfítriones, 420
 extendidos, 71, 74
 soportados por los productos SQL
 más populares, 72-73
 y variables principales, 418-421
 Tipos de datos complejos, 564-565
 TOTAL, 592
- Transacciones:
 actualizaciones perdidas, 276-277
 concurrentes, 280-281
 distribuidas, 540-542
 ejemplos de, 266
 modelo ANSI/ISO, 269-271
 modelo SYBASE, 273
 otros modelos, 271-274
 remotas, 538-540
 y cierres (*locking*), 282-292
 y procesamiento multiusuario, 276-281

- uso de estructuras de datos como, 429
 tipos de datos, 418-421
 y valores NULL, 421-422
 VAX, 593
 Vendedores, independencia de los, 8
 lista de, 591-596
 síntesis de los DBMS, 579-590
 Verbos en sentencias SQL, 67
 VIDA, 581
 Visión general de SQL, 13-22
 Vistas, 331-350
 agrupadas, 340-342
 almacenadas en el catálogo de sistema, 384-385
 compuestos, 342-344
 creación (CREATE VIEW), 335-344
 desventajas de, 335
 horizontales, 336-338
 subconjunto fila/columna, 340
 ventajas de, 334
 verticales, 338-339
 y el DBMS, 333
 y seguridad, 359-360, 361, 362
 Vistas agrupadas, 340-342
 Vistas compuestas, 342-344
 Vistas de subconjunto fila/columna, 340
 Vistas horizontales, 336-338
 Vistas verticales, 338-339
 Utilidades, carga masiva, 229
 WHILEVER, sentencia, 412-414
 diagrama sintáctico, 414
 WHERE, cláusula, 98-100
 diagrama sintáctico, 112
 subconsulta en, 194, 195
 Wingz, 585
- XDB-Server, 596
 XDB-SQL, 596
 XDB Systems, Inc., 596
 X/Open, estándares, 30
 XML, 595