Paolo Atzeni
Stefano Ceri
Piero Fraternali
Stefano Paraboschi
Riccardo Torlone

Basi di dati

Quarta edizione

McGraw-Hill

Milano • New York • San Francisco • Washington D.C. • Auckland
Bogotà • Lisboa • London • Madrid • Mexico City • Montreal
New Delhi • San Juan • Singapore • Sydney • Tokyo • Toronto

Copyright © 2014 McGraw-Hill Education (Italy) srl
via Ripamonti, 89 - 20141 Milano

I diritti di traduzione, di riproduzione, di memorizzazione elettronica e di adattamento totale e parziale con qualsiasi mezzo (compresi i microfilm e le copie fotostatiche) sono riservati per tutti i Paesi.

Le fotocopie per uso personale del lettore possono essere effettuate nei limiti del 15% di ciascun volume/fascicolo di periodico dietro pagamento alla SIAE del compenso previsto dall'art. 68, commi 4 e 5, della Legge 22 aprile 1941, n. 633.

Le riproduzioni effettuate per finalità di carattere professionale, economico o commerciale o comunque per uso diverso da quello personale possono essere effettuate a seguito di specifica autorizzazione rilasciata da CLEARedi, Corso di Porta Romana 108, Milano 20122, e-mail info@clearedi.org e sito web www.clearedi.org.

Nomi e marchi citati nel testo sono generalmente depositati o registrati dalle rispettive case produttrici.

Publisher: Paolo Roncoroni
Senior Acquisition Editor: Filippo Aroffo
Development Editor: Barbara Ferrario
Produzione: Donatella Giuliani
Realizzazione editoriale: Fotocompos, Gussago (BS)
Grafica di copertina: Feel Italia, Milano
Immagine di copertina: ©Kentoh
Stampa: LaLitotipo, Settimo Milanese (MI)

ISBN 978-88-386-6587-5

Printed in Italy
123456789LITLIT987654

Indice breve

1 Introduzione 1

Parte prima Basi di dati relazionali: modello e linguaggi 15

2 Il modello relazionale 15

3 Algebra e calcolo relazionale 43

4 SQL: concetti base 89

5 SQL: caratteristiche evolute 149

Parte seconda Progettazione di basi di dati e delle applicazioni 191

6 Metodologie e modelli per il progetto 191

7 La progettazione concettuale 235

8 La progettazione logica 277

9 La normalizzazione 323

10 Sviluppo di applicazioni per basi di dati 355

Parte terza Tecnologie delle basi di dati 401

11 Organizzazione fisica e gestione delle interrogazioni 401

12 Gestione delle transazioni 445

Parte quarta Evoluzione dei modelli e dei linguaggi per basi di dati 481

13 Basi di dati a oggetti, multimediali e NoSQL 481

14 Basi di dati per XML 523

15 Basi di dati semantiche, SPARQL e linked open data 555

Parte quinta Architetture evolute per basi di dati **579**

16 Architetture distribuite 579

17 Basi di dati parallele e architetture per il cloud 619

18 Architetture dei sistemi informativi su Web 639

19 Architetture per l'analisi dei dati 697

Indice

Prefazione XVII

Autori XXI

Ringraziamenti dell'Editore XXIII

1 Introduzione 1
1.1 Sistemi informativi, informazioni e dati 1
1.2 Basi di dati e sistemi di gestione di basi di dati 3
1.3 Modelli dei dati 6
1.3.1 Schemi e istanze 7
1.3.2 Livelli di astrazione nei DBMS 8
1.3.3 Indipendenza dei dati 9
1.4 Linguaggi e utenti delle basi di dati 10
1.4.1 Linguaggi per basi di dati 10
1.4.2 Utenti e progettisti 11
1.5 Vantaggi e svantaggi dei DBMS 13
Note bibliografiche 13

Parte prima Basi di dati relazionali: modello e linguaggi 15

2 Il modello relazionale 15
2.1 Il modello relazionale: strutture 15
2.1.1 Modelli logici nei sistemi di basi di dati 15
2.1.2 Relazioni e tabelle 16
2.1.3 Relazioni con attributi 17
2.1.4 Relazioni e basi di dati 20
2.1.5 Informazione incompleta e valori nulli 25
2.2 Vincoli di integrità 28
2.2.1 Vincoli di tupla 30
2.2.2 Chiavi 31
2.2.3 Chiavi e valori nulli 33
2.2.4 Vincoli di integrità referenziale 35
2.3 Conclusioni 38
Note bibliografiche 39
Esercizi 39

3 Algebra e calcolo relazionale **43**
3.1 Algebra relazionale 43
3.1.1 Unione, intersezione, differenza 44
3.1.2 Ridenominazione 44
3.1.3 Selezione 47
3.1.4 Proiezione 49
3.1.5 Join 51
3.1.6 Interrogazioni in algebra relazionale 58
3.1.7 Equivalenza di espressioni algebriche 62
3.1.8 Algebra con valori nulli 65
3.1.9 Viste 67
3.2 Calcolo relazionale 69
3.2.1 Calcolo relazionale su domini 70
3.2.2 Pregi e difetti del calcolo su domini 74
3.2.3 Calcolo su tuple con dichiarazioni di range 76
3.3 Datalog 79
Note bibliografiche 83
Esercizi 83

4 SQL: concetti base **89**
4.1 Il linguaggio SQL e gli standard 89
4.2 Definizione dei dati in SQL 92
4.2.1 I domini elementari 92
4.2.2 Definizione di schema 95
4.2.3 Definizione delle tabelle 96
4.2.4 Definizione dei domini 96
4.2.5 Specifica di valori di default 97
4.2.6 Vincoli intrarelazionali 98
4.2.7 Vincoli interrelazionali 100
4.2.8 Modifica degli schemi 102
4.2.9 Cataloghi relazionali 104
4.3 Interrogazioni in SQL 106
4.3.1 Dichiaratività di SQL 106
4.3.2 Interrogazioni semplici 106
4.3.3 Operatori aggregati 120
4.3.4 Interrogazioni con raggruppamento 123
4.3.5 Interrogazioni di tipo insiemistico 128
4.3.6 Interrogazioni nidificate 130
4.4 Modifica dei dati in SQL 137
4.4.1 Inserimento 137
4.4.2 Cancellazione 138
4.4.3 Modifica 139
4.5 Esempi riepilogativi 141
Note bibliografiche 143
Esercizi 143

5 SQL: caratteristiche evolute **149**
5.1 Caratteristiche evolute di definizione dei dati 149
5.1.1 Vincoli di integrità generici 149
5.1.2 Asserzioni 150
5.1.3 Viste 152
5.1.4 Le viste per la scrittura di interrogazioni 153
5.1.5 Esempi riepilogativi d'uso delle viste 155
5.1.6 Viste ricorsive in SQL-3 156
5.2 Funzioni scalari 157
5.2.1 Famiglie di funzioni 157
5.2.2 Funzioni condizionali 158
5.3 Procedure 160
5.4 Trigger e basi di dati attive 163
5.4.1 Definizione e uso dei trigger in SQL-3 164
5.4.2 Definizione e uso dei trigger in DB2 167
5.4.3 Definizione e uso dei trigger in Oracle 169
5.4.4 Caratteristiche evolute e proprietà delle regole attive 172
5.4.5 Applicazioni delle basi di dati attive 174
5.5 Controllo dell'accesso 178
5.5.1 Risorse e privilegi 179
5.5.2 Comandi per concedere e revocare privilegi 180
5.5.3 I ruoli in SQL-3 181
5.6 Transazioni 182
5.6.1 Specifica delle transazioni: commit e rollback 182
5.6.2 Proprietà acide delle transazioni 183
Note bibliografiche 185
Esercizi 186

Parte seconda Progettazione di basi di dati e delle applicazioni **191**

6 Metodologie e modelli per il progetto **191**
6.1 Introduzione alla progettazione 191
6.1.1 Il ciclo di vita dei sistemi informativi 191
6.1.2 Metodologie di progettazione e basi di dati 193
6.2 Il modello Entità-Relazione 196
6.2.1 I costrutti principali del modello 198
6.2.2 Altri costrutti del modello 204
6.2.3 Panoramica finale sul Modello E-R 212
6.3 Documentazione di schemi E-R 213
6.3.1 Regole aziendali 214
6.3.2 Tecniche di documentazione 216
6.4 Modellazione dei dati in UML 218
6.4.1 Panoramica su UML 219
6.4.2 Rappresentazione di dati con i diagrammi delle classi 220
Note bibliografiche 227
Esercizi 228

7 La progettazione concettuale **235**
7.1 La raccolta e l'analisi dei requisiti 235
7.2 Rappresentazione concettuale di dati 240
7.2.1 Criteri generali di rappresentazione 241
7.2.2 Pattern di progetto 242
7.3 Strategie di progetto 250
7.3.1 Strategia top-down 250
7.3.2 Strategia bottom-up 252
7.3.3 Strategia inside-out 253
7.3.4 Strategia mista 254
7.4 Qualità di uno schema concettuale 255
7.5 Una metodologia generale 257
7.6 Un esempio di progettazione concettuale 258
7.7 Strumenti CASE per la progettazione di basi di dati 262
Note bibliografiche 265
Esercizi 265

8 La progettazione logica **277**
8.1 Fasi della progettazione logica 277
8.2 Analisi delle prestazioni su schemi E-R 278
8.3 Ristrutturazione di schemi E-R 282
8.3.1 Analisi delle ridondanze 282
8.3.2 Eliminazione delle generalizzazioni 286
8.3.3 Partizionamento/accorpamento di concetti 289
8.3.4 Scelta degli identificatori principali 293
8.4 Traduzione verso il modello relazionale 294
8.4.1 Entità e associazioni molti a molti 295
8.4.2 Associazioni uno a molti 297
8.4.3 Entità con identificatore esterno 298
8.4.4 Associazioni uno a uno 299
8.4.5 Traduzioni di schemi complessi 301
8.4.6 Tabelle riassuntive 302
8.4.7 Documentazione di schemi logici 303
8.5 Un esempio di progettazione logica 305
8.5.1 Fase di ristrutturazione 307
8.5.2 Traduzione verso il relazionale 311
8.6 Progettazione logica con gli strumenti CASE 312
Note bibliografiche 314
Esercizi 314

9 La normalizzazione **323**
9.1 Ridondanze e anomalie 323
9.2 Dipendenze funzionali 325
9.3 Forma normale di Boyce e Codd 326
9.3.1 Definizione di forma normale di Boyce e Codd 326
9.3.2 Decomposizione in forma normale di Boyce e Codd 328
9.4 Proprietà delle decomposizioni 328
9.4.1 Decomposizione senza perdita 329
9.4.2 Conservazione delle dipendenze 332
9.4.3 Qualità delle decomposizioni 333

9.5 Terza forma normale 333
9.5.1 Limitazioni della forma normale di Boyce e Codd 333
9.5.2 Definizione di terza forma normale 334
9.5.3 Decomposizione in terza forma normale 335
9.5.4 Altre forme normali 336
9.5.5 Normalizzazione e scelta degli attributi 337
9.6 Teoria delle dipendenze e normalizzazione 338
9.6.1 Implicazione di dipendenze funzionali 338
9.6.2 Coperture di insiemi di dipendenze funzionali 341
9.6.3 Sintesi di schemi in terza forma normale 342
9.7 Progettazione di basi di dati e normalizzazione 343
9.7.1 Verifiche di normalizzazione su entità 344
9.7.2 Verifiche di normalizzazione su associazioni 346
9.7.3 Ulteriori decomposizioni di associazioni 347
9.7.4 Ulteriori decomposizioni di schemi concettuali 349
Note bibliografiche 350
Esercizi 350

10 Sviluppo di applicazioni per basi di dati 355
10.1 SQL Embedded 355
10.1.1 Cursori 358
10.1.2 SQL dinamico 361
10.2 Call Level Interface (CLI) 364
10.2.1 ODBC e soluzioni proprietarie Microsoft 364
10.2.2 Java Database Connectivity (JDBC) 370
10.3 Il controllo delle transazioni nelle applicazioni 375
10.3.1 Il controllo della concorrenza e delle transazioni in JDBC 378
10.4 Mappatura relazionale degli oggetti e sistemi ORM 379
10.5 Java Persistence API (JPA) 381
10.5.1 Mappatura tra classi e tabelle 381
10.5.2 Architettura e utilizzo di JPA 387
10.5.3 Interrogazioni in JPA 393
Note bibliografiche 397
Esercizi 398

Parte terza Tecnologie delle basi di dati 401

11 Organizzazione fisica e gestione delle interrogazioni 401
11.1 Memoria principale, memoria secondaria e gestione dei buffer 402
11.1.1 Memoria secondaria: caratteristiche 403
11.1.2 Gestione dei buffer 404
11.1.3 DBMS e file system 406
11.2 Gestione delle tuple nelle pagine 407
11.3 Strutture primarie per l'organizzazione di file 409
11.3.1 Strutture sequenziali 409
11.3.2 Strutture con accesso calcolato (hash) 411
11.4 Strutture ad albero 415
11.4.1 Indici primari e secondari 416
11.4.2 Strutture ad albero dinamiche 418

11.5 Strutture fisiche e indici nei DBMS relazionali 425
11.6 Gestore delle interrogazioni: esecuzione e ottimizzazione 427
11.6.1 Profili delle relazioni 428
11.6.2 Rappresentazione interna delle interrogazioni 430
11.6.3 Ottimizzazione basata sui costi 434
11.7 Progettazione fisica di una base di dati 436
Note bibliografiche 440
Esercizi 440

12 Gestione delle transazioni 445
12.1 Controllo di affidabilità 445
12.1.1 Architettura del controllore dell'affidabilità 446
12.1.2 Organizzazione del log 448
12.1.3 Esecuzione delle transazioni e scrittura del log 450
12.1.4 Gestione dei guasti 452
12.2 Controllo di concorrenza 455
12.2.1 Architettura 455
12.2.2 Anomalie delle transazioni concorrenti 456
12.2.3 Gestione della concorrenza in SQL e in JDBC 459
12.2.4 Teoria del controllo di concorrenza 460
12.2.5 Meccanismi per la gestione dei lock 472
12.2.6 Blocco critico 475
Note bibliografiche 477
Esercizi 477

Parte quarta Evoluzione dei modelli e dei linguaggi per basi di dati 481

13 Basi di dati a oggetti, multimediali e NoSQL 481
13.1 Basi di dati a oggetti 482
13.1.1 Tipi 482
13.1.2 Classi 486
13.1.3 Metodi 488
13.1.4 Gerarchie di generalizzazione 492
13.1.5 Persistenza 495
13.1.6 Ridefinizioni dei metodi 495
13.1.7 Ridefinizione con raffinamenti di tipo 497
13.2 Basi di dati relazionali a oggetti 499
13.2.1 Modello dei dati di SQL-3 499
13.2.2 Linguaggio di interrogazione di SQL-3 504
13.3 Estensioni tecnologiche per le basi di dati a oggetti 505
13.3.1 Rappresentazione dei dati e degli identificatori 505
13.3.2 Indici complessi 507
13.3.3 Architettura client-server 507
13.3.4 Transazioni 508
13.4 Basi di dati multimediali 509
13.4.1 Tipi di dati multimediali 509
13.4.2 Interrogazioni su dati multimediali 511
13.4.3 Ricerca di documenti 511
13.4.4 Rappresentazione dei dati spaziali 513

13.5 NoSQL 514
13.5.1 Diversificazione dei sistemi 516
13.5.2 Modelli dei dati nei sistemi NoSQL 517
13.5.3 Gestione delle transazioni 518
Note bibliografiche 518
Esercizi 519

14 Basi di dati per XML 523
14.1 HTML 524
14.2 Definizione di dati semistrutturati in XML 526
14.3 Interrogazione di dati XML 530
14.3.1 XPath 531
14.3.2 XSL 532
14.3.3 XQuery 535
14.4 Gestione di dati XML con DBMS relazionali 546
14.5 Le basi di dati XML native 549
Note bibliografiche 551
Esercizi 551

15 Basi di dati semantiche, SPARQL e linked open data 555
15.1 Il modello dei dati RDF 556
15.2 RDF Schema e OWL 558
15.3 Il linguaggio SPARQL 561
15.3.1 Triple e Graph Patterns 561
15.3.2 Clausole FILTER, UNION, OPTIONAL 564
15.3.3 Negazione e modificatori 566
15.3.4 Forme CONSTRUCT, ASK e DESCRIBE 567
15.3.5 Aggregazione, negazione e sotto-query in SPARQL 1.1 569
15.3.6 Aspetti avanzati di SPARQL 1.1 571
15.3.7 SPARQL e interoperabilità 572
15.4 Linked e open data 574
Note bibliografiche 576
Esercizi 576

Parte quinta Architetture evolute per basi di dati 579

16 Architetture distribuite 579
16.1 Architettura client-server 580
16.2 Basi di dati distribuite 582
16.2.1 Applicazioni delle basi di dati distribuite 583
16.2.2 Autonomia locale e cooperazione 584
16.2.3 Frammentazione e allocazione dei dati 584
16.2.4 Livelli di trasparenza 587
16.2.5 Classificazione delle transazioni 589
16.3 Tecnologia delle basi di dati distribuite 590
16.3.1 Ottimizzazione di interrogazioni distribuite 591
16.3.2 Controllo di concorrenza 592
16.3.3 Atomicità di transazioni distribuite 596
16.4 Protocollo di commit a due fasi 597

16.4.1 Nuovi record nel log 598
16.4.2 Protocollo in assenza di guasti 598
16.4.3 Protocolli di ripristino 601
16.4.4 Ottimizzazioni del commit a due fasi 602
16.4.5 Altri protocolli di commit 603
16.4.6 Interoperabilità del commit a due fasi: X-Open DTP 606
16.5 Basi di dati replicate 609
16.5.1 Nuove funzionalità dei gestori della replicazione 610
16.6 Cooperazione di basi di dati eterogenee e autonome 611
Note bibliografiche 615
Esercizi 615

17 Basi di dati parallele e architetture per il cloud 619
17.1 Parallelismo 619
17.1.1 Parallelismo e frammentazione dei dati 620
17.1.2 Speed-up e scale-up 621
17.1.3 Benchmark delle transazioni 622
17.2 Le architetture per il cloud computing 623
17.2.1 Programmazione distribuita per il cloud con Hadoop e MapReduce 625
17.3 Pig e Pig Latin 628
17.3.1 Caratteristiche di base 628
17.3.2 Modello dei dati 629
17.3.3 Query semplici 631
17.3.4 Query con ordinamenti e raggruppamenti 634
17.3.5 Integrazione con MapReduce 636
17.3.6 Apache Hive 637
Note bibliografiche 638
Esercizi 638

18 Architetture dei sistemi informativi su Web 639
18.1 Tecnologie di base 640
18.1.1 Internet 640
18.1.2 Il World Wide Web 641
18.1.3 Il protocollo HTTP 643
18.2 Architetture a tre livelli 645
18.2.1 Common Gateway Interface 645
18.2.2 L'architettura Java Servlet 647
18.2.3 Template di pagina e server side scripting 651
18.2.4 Model-View-Controller 654
18.3 Architetture multi-livello 658
18.3.1 Il concetto di application server 658
18.3.2 Enterprise Java Bean 660
18.4 Architetture a servizi 661
18.4.1 Il concetto di Web Service 661
18.4.2 Il protocollo SOAP 663
18.4.3 Il linguaggio WSDL 665
18.4.4 I Web Service con REST 667
18.4.5 Un confronto tra EJB e Web Service 671
18.5 Le architetture per Rich Internet Application 671

18.5.1 Tecnologie per le RIA 672
18.5.2 DOM 673
18.5.3 JavaScript 674
18.5.4 AJAX 676
18.5.5 JavaScript Object Notation (JSON) 678
18.5.6 HTML 5 e la persistenza a lato client 679
18.6 Progettazione delle applicazioni Web 682
18.6.1 Analisi dei requisiti 682
18.6.2 Progettazione concettuale della base di dati 685
18.6.3 Progettazione dell'ipertesto 687
18.6.4 Realizzazione dell'applicazione 692
Note bibliografiche 694
Esercizi 695

19 Architetture per l'analisi dei dati 697
19.1 Architettura di un data warehouse 698
19.2 Rappresentazione multidimensionale dei dati 702
19.2.1 Il modello multidimensionale 703
19.2.2 Operazioni nel modello multidimensionale 705
19.3 Realizzazione di un data warehouse 708
19.3.1 Rappresentazione relazionale di un data warehouse 708
19.3.2 Operazioni su data mart relazionali 713
19.3.3 Indici bitmap e indici di join 715
19.3.4 Materializzazione delle viste 717
19.4 Progettazione di un data warehouse 717
19.4.1 Una metodologia generale 718
19.4.2 Raccolta e analisi dei requisiti 719
19.4.3 Progettazione dei dati 720
19.4.4 Un esempio di progettazione 723
19.5 Data mining 728
19.5.1 Il processo di data mining 729
19.5.2 Problemi di data mining 730
19.6 Big data 733
Note bibliografiche 737
Esercizi 737

Bibliografia 741

Indice analitico 749

Appendici

disponibili sul sito http://www.ateneonline.it/atzeni

A Microsoft Access

B DB2 Universal Database

C DBMS open source: Postgres

Prefazione

Questo testo presenta i concetti fondamentali sulle basi di dati, sui linguaggi di interrogazione e di gestione, sulle tecniche e sui metodi di progettazione, e sulle tecnologie e architetture per la gestione dei dati. Esso nasce da una lunga esperienza di insegnamento in corsi riguardanti le basi di dati, in ambito sia universitario sia industriale e applicativo, e pertanto si rivolge al pubblico degli studenti (in particolare di Ingegneria e di Scienze dell'informazione o Informatica) e a quello dei professionisti (utenti e progettisti di applicazioni).

Contenuti

Il libro si articola in cinque parti:

Basi di dati relazionali: modello e linguaggi. Vengono presentate le caratteristiche fondamentali delle basi di dati che risultano di interesse per gli utenti e i programmatori. In particolare, si illustrano il modello relazionale e i relativi linguaggi, in modo preciso e concreto, con riferimento sia alle definizioni formali (del modello, dell'algebra e del calcolo) sia ai sistemi esistenti (con riferimento soprattutto al linguaggio SQL).

Progettazione di basi di dati e delle applicazioni. Viene illustrato ed esemplificato il processo di progettazione concettuale, logica e fisica delle basi di dati relazionali, che permette, partendo dai requisiti di utente, di arrivare a produrre strutture di basi di dati di buona qualità. Vengono poi discusse le tecniche principali per l'utilizzo delle basi di dati nelle applicazioni.

Tecnologia delle basi di dati. Vengono descritte le caratteristiche interne dei sistemi di basi di dati in rapporto all'architettura hardware e software del sistema informativo, in modo da comprenderne il funzionamento e sfruttarne appieno le potenzialità.

Evoluzione dei modelli e dei linguaggi per basi di dati. Vengono illustrate le moderne varianti rispetto al modello e al linguaggio relazionale, focalizzandosi sulle basi di dati a oggetti, multimediali, NoSql, sulla gestione di dati XML (e sul linguaggio XQuery), e sulle basi di dati semantiche (e sul linguaggio Sparql).

Architetture evolute per basi di dati. Vengono illustrate le principali architetture dei sistemi informativi moderni, focalizzandosi sulla distribuzione dei dati, sul parallelismo dei dati in ambiente Cloud, sulla integrazione con il World Wide Web e sui sistemi per l'analisi dei dati e il supporto alle decisioni.

Ciascun capitolo è corredato di numerosi esempi ed esercizi, nonché di una nota bibliografica che indica le fonti per possibili approfondimenti, elencate poi globalmente alla fine del volume.

Utilizzo didattico

Nell'esperienza degli Autori, gli argomenti trattati in questo volume vengono svolti in modo completo in un tipico corso di Basi di dati, da 12 crediti (circa 60 ore di lezione e 40 di esercitazione). A essi è opportuno associare un'ampia attività pratica; in particolare, per quanto riguarda le prime due parti del testo si consiglia lo svolgimento di un progetto di un sistema informativo che includa una base di dati, mentre per quanto riguarda le successive parti sono possibili vari approfondimenti progettuali legati all'uso di un linguaggio evoluto o di una particolare tecnologia o architettura. Informazioni utili circa l'organizzazione di alcuni DBMS relazionali sono presenti nei siti in appoggio al presente volume, descritti più avanti.

Il presente testo include al suo interno, nelle prime tre parti, i contenuti del volume *Basi di dati. Modelli e linguaggi di interrogazione* [91], anch'esso alla sua quarta edizione. Nelle precedenti edizioni, tecnologie, architetture e linee di evoluzioni erano presentate in un volume separato; abbiamo optato per un volume unico rispondendo alle indicazioni in questo senso da parte di vari docenti e lettori.

Per i docenti che nel proprio corso di Basi di dati non trattano tutti gli argomenti inclusi nel testo, l'Editore offre la possibilità di realizzare un libro "personalizzato" (`http://www.ateneonline.it/docenti/create.asp`).

Esperienze e ringraziamenti

L'organizzazione di questo testo e i suoi contenuti riflettono l'esperienza didattica degli Autori, che hanno tenuto per molti anni il corso universitario di Basi di dati e hanno svolto in altri contesti corsi sugli stessi temi. In particolare, Paolo Atzeni ha svolto in passato il corso di Basi di dati presso la facoltà di Ingegneria dell'Università di Roma "La Sapienza" e prima ancora presso l'Università di Toronto, tiene corsi di Basi di dati I, Basi di dati II e Sistemi informativi presso il Dipartimento di Ingegneria dell'Università Roma Tre. Stefano Ceri tiene i corsi di Basi di dati e Data management for the Web presso la Scuola di Ingegneria Industriale e dell'Informazione del Politecnico di Milano. Ha inoltre tenuto varie edizioni del corso Principles of Distributed Databases presso l'Università di Stanford. Piero Fraternali tiene i corsi di Basi di dati e Web e Advanced Web Technologies presso la Scuola di Ingegneria Industriale e dell'Informazione del Politecnico di Milano. Stefano Paraboschi tiene corsi di Basi di dati e Web e Tecnologie per basi di dati presso il Dipartimento di Ingegneria dell'Università di Bergamo e presso la Scuola di Ingegneria Industriale e dell'Informazione del Politecnico di Milano. Riccardo Torlone tiene corsi di Basi di dati e Big Data presso il Dipartimento di Ingegneria dell'Università Roma Tre.

Alla concezione e alla revisione di questo testo hanno contribuito, direttamente o indirettamente, anche attraverso discussioni sui contenuti didattici dei corsi o suggerimenti di vario tipo, numerosi colleghi, collaboratori e lettori. Citiamo, fra gli

altri, Maristella Agosti, Giorgio Ausiello, Elena Baralis, Giovanni Barone, Carlo Batini, Giampio Bracchi, Daniele Braga, Francesca Bugiotti, Luca Cabibbo, Alessandro Campi, Sara Comai, Sabrina De Capitani, Paolo De Nictolis, Giuseppe Di Battista, Angelo Foglietta, Sara Foresti, Maurizio Lenzerini, Davide Martinenghi, Gianni Mecca, Paolo Merialdo, Barbara Pernici, Giuseppe Psaila, Silvio Salza, Pierangela Samarati, Fabio Schreiber, Giuseppe Sindoni, Elena Tabet e Letizia Tanca. A ciascuno di essi, nonché a coloro che abbiamo dimenticato, va il nostro più sincero ringraziamento.

Nota di edizione

Questa nuova edizione esce a quasi vent'anni di distanza dalla pubblicazione del volume *Basi di dati: concetti, linguaggi e architetture*, che ha visto la sua seconda edizione e un'edizione internazionale nel 1999. Da allora, abbiamo suddiviso il materiale in due volumi, le cui prime edizioni sono uscite nel 2003 e 2004, le seconde edizioni nel 2006 e 2007, e una terza e quarta edizione del primo volume nel 2009 e 2013. Abbiamo deciso un "ritorno all'antico", cioè di riproporre un volume unico, sulla spinta di indicazioni di molti docenti che ritenevano necessario per lo svolgimento di un primo corso di basi di dati alcuni materiali presenti nel secondo volume. Termina così una operazione di ristrutturazione dei contenuti che ci ha tenuti occupati per tutto il 2013 e che ci ha portato anche a spostare alcune appendici nei materiali disponibili in rete. Il costante lavoro di aggiornamento e integrazione di queste opere tiene conto del continuo progresso dei linguaggi e della tecnologia per la gestione dei dati (che, per esempio, nell'ultimo decennio ha visto un'esplosione nelle applicazioni su Internet e su Cloud), e di commenti sull'uso del testo e di suggerimenti sui nuovi requisiti didattici espressi da numerosi colleghi, docenti di corsi di Basi di dati offerti dalle sedi accademiche sia di Ingegneria sia di Scienze dell'informazione o Informatica, raccolti da McGraw-Hill e analizzati con attenzione dagli autori.

Questo volume introduce, nella sua quarta e quinta parte, molti aspetti nuovi rispetto alla precedente edizione del volume *Architetture e linee di evoluzione* (del 2007).

- Nel capitolo sulle basi di dati ad oggetti e multimediali abbiamo introdotto una breve descrizione dei sistemi NoSQL, che si stanno diffondendo soprattutto per la gestione di grandi quantità di dati su Web (big data).
- Abbiamo introdotto un nuovo capitolo sulle basi di dati semantiche, approfondendo in particolare la descrizione del linguaggio SPARQL e lo sviluppo dei dati "aperti" (open data).
- Abbiamo separato l'argomento del parallelismo da quello della distribuzione dei dati, dedicando al parallelismo un capitolo autonomo che descrive le nuove architetture per cloud computing e il linguaggio Pig Latin per la gestione dei dati su cloud.
- Abbiamo fortemente rivisto il capitolo relativo all'uso dei dati su Web, introducendo una nuova sezione relativa alle Rich Internet Application che include la descrizione di nuovi standard - tra cui AJAX, JSON e HTML 5.

- Nel capitolo relativo alla analisi dei dati è stato dato ampio spazio alla progettazione dei data warehouse, introducendo una metodologia completa e un esempio di progettazione.

Le prime tre parti del volume sono identiche al volume *Basi di dati. Modelli e linguaggi di interrogazione*, anch'esso giunto alla sua quarta edizione (del 2013). Inoltre, tutto il materiale è stato rivisto e aggiornato, e al termine di ciascun capitolo sono stati aggiunti numerosi esercizi.

Materiale aggiuntivo

Materiale didattico di supporto al libro è disponibile ai siti:

`http://www.ateneonline.it/atzeni`
`http://www.dia.uniroma3.it/librobd`

In particolare, per gli studenti sono disponibili le soluzioni di tutti gli esercizi presenti nel testo, alcuni esercizi aggiuntivi, l'Appendice A dell'edizione del 1996 (relativa al modello reticolare e omessa nelle edizioni successive) e le Appendici A, B e C della terza edizione, relative ai sistemi *Microsoft Access*, *DB2 Universal Database* e *Postgres*.

Per i docenti che utilizzano il testo sono disponibili, oltre ai materiali presenti nell'area studenti, i lucidi che coprono in modo completo i diciannove capitoli della presente edizione. Per eventuali segnalazioni di errori e altri suggerimenti, gli Autori possono essere contattati attraverso i siti stessi.

Autori

Paolo Atzeni è professore ordinario di Basi di dati all'Università Roma Tre, dove è attualmente Direttore del Dipartimento di Ingegneria. Si è laureato in Ingegneria elettronica nel 1980 all'Università di Roma "La Sapienza" ed è stato ricercatore allo IASI-CNR di Roma e poi professore associato all'Università di Napoli e professore ordinario all'Università di Roma "La Sapienza". Ha svolto attività didattica e scientifica presso la University of California, Los Angeles (UCLA), la University of Toronto e la Microsoft Research. Ha svolto ricerche su vari temi nel settore delle basi di dati, fra cui la teoria relazionale, i modelli concettuali e gli strumenti di sviluppo, le basi di dati nel mondo Web, la gestione e la traduzione di schemi eterogenei. Ha fondato e dirige il gruppo di basi di dati di Roma Tre, che comprende cinque docenti e numerosi assegnisti e dottorandi. Il gruppo collabora con numerosi gruppi qualificati in Italia e all'estero e partecipa a progetti nazionali e internazionali. È stato Presidente dell'Associazione EDBT e Vicepresidente del VLDB Endowment.

Stefano Ceri è professore ordinario di Basi di dati al Politecnico di Milano. Si è laureato in Ingegneria elettronica nel 1978 al Politecnico di Milano, è stato ricercatore al Politecnico fino al 1985, poi professore ordinario all'Università di Modena, e infine professore ordinario al Politecnico di Milano dal 1990. È stato professore visitatore del Dipartimento di Computer Science alla Stanford University, dove dal 1983 al 1992 ha insegnato il corso "Distributed Databases: Principles and Systems". Ha svolto ricerche in vari aspetti delle basi di dati e del Web, inclusi database distribuiti, attivi, deduttivi e a oggetti e metodi e modelli per la progettazione di applicazioni sul Web. Ha ottenuto nel 2013 lo Edward F. Codd Award da parte dell'ACM SIGMOD. È un Fellow dell'ACM. Ha ricevuto un IDEAS Advanced Grant ERC su "Search Computing" (2008-2013) ed è coordinatore nazionale del progetto PRIN su "Data-Centric Genomic Computing" (2013-2016). È socio fondatore della spin-off WebRatio.

Piero Fraternali è professore ordinario presso il Dipartimento di Elettronica, Informazione e Bioingegneria del Politecnico di Milano. Si occupa della specifica e progettazione del software, dell'integrazione tra Sistemi informativi e World Wide Web, e dell'uso delle reti sociali e dei giochi nella computazione. Dal 1996 si occupa di metodologie e strumenti per lo sviluppo assistito da calcolatore di applicazioni per il Web. È coautore del linguaggio WebML per la specifica di applicazioni Web (WWW Modelling Language, `http://www.webml.org`), brevettato internazionalmente, su cui si basa l'Interaction Flow Modeling Language (IFML, `http://www.webml.org`), recepito nel 2014 dall'Object Management Group come lo standard internazionale per la modellazione delle interfacce utente. Nel 2001 ha fondato la società WebRatio, che

sviluppa e commercializza un nuovo software per la costruzione di applicazioni Web di grandi dimensioni (`http://www.webratio.com`). Nel dicembre 2002 ha pubblicato presso l'editore Morgan Kaufmann, insieme ad altri ricercatori del Politecnico di Milano, il testo *Designing Data-Intensive Web Applications*, che descrive una metodologia di sviluppo per applicazioni Web basata sulla specifica concettuale dei requisiti. Ha diretto diversi progetti scientifici finanziati dall'Unione Europea.

Stefano Paraboschi è professore ordinario dal 2002 all'Università degli Studi di Bergamo, dove presiede il corso di laurea in Ingegneria Informatica. Si è laureato in Ingegneria elettronica nel 1990 al Politecnico di Milano, dove ha conseguito il titolo di dottore di ricerca e ha poi ricoperto i ruoli di ricercatore e professore associato. Ha svolto periodi di ricerca presso la Stanford University, l'IBM Almaden Research Center e la George Mason University. Nella sua ricerca ha esplorato vari temi nell'area delle basi di dati, tra cui il progetto e l'uso di regole attive, il disegno di sistemi multidimensionali, l'integrazione tra basi di dati e Web e la gestione di dati XML. Di recente i suoi interessi si sono spostati sullo studio di problemi all'intersezione tra la sicurezza informatica e la gestione dei dati. Ha coordinato progetti nazionali nell'ambito del programma PRIN. È stato responsabile per la partecipazione dell'Università di Bergamo a progetti europei finanziati dal 7th Framework Programme.

Riccardo Torlone è professore ordinario all'Università Roma Tre, dove è attualmente il coordinatore dei corsi di studio in Ingegneria informatica. Si è laureato in Ingegneria elettronica all'Università di Roma "La Sapienza", è stato ricercatore allo IASI-CNR di Roma e ha svolto attività di ricerca presso la University of California, Los Angeles (UCLA). La sua attività di ricerca ha riguardato vari argomenti nel settore delle basi di dati, tra cui: la teoria delle basi di dati, le basi di dati attive e deduttive, i data warehouse, i sistemi informativi su Web, l'integrazione e la trasformazione di dati, i sistemi adattativi e personalizzati e il Web semantico. I suoi interessi di ricerca più recenti riguardano i big data e i sistemi di gestione di dati di nuova generazione. Fa parte del gruppo di basi di dati di Roma Tre e collabora con numerosi gruppi qualificati in Italia e all'estero. È stato responsabile scientifico di progetti di ricerca finanziati dal MIUR, dalla Commissione Europea e da numerosi enti pubblici e privati.

Ringraziamenti dell'Editore

L'Editore ringrazia i docenti che hanno partecipato alla review del testo e che, con le loro preziose indicazioni, hanno contribuito alla realizzazione della quarta edizione di *Basi di dati*:

Paolo Baldan, *Università degli Studi di Padova*
Elena Baralis, *Politecnico di Torino*
Paolo Ciaccia, *Alma Mater Studiorum Università di Bologna*
Donatello Conte, *Università degli Studi di Salerno*
Claudia Diamantini, *Università Politecnica delle Marche*
Paolino Di Felice, *Università degli Studi dell'Aquila*
Eugenio Di Sciascio, *Politecnico di Bari*
Maurizio Fermeglia, *Università degli Studi di Trieste*
Umberto Ferraro Petrillo, *Sapienza Università di Roma*
Marco Ferretti, *Università degli Studi di Pavia*
Donatella Merlini, *Università degli Studi di Firenze*
Danilo Montesi, *Alma Mater Studiorum Università di Bologna*
Marina Moscarini, *Sapienza Università di Roma*
Loredana Vigliano, *Università degli Studi di Roma "Tor Vergata"*
Aaron Visaggio, *Università degli Studi del Sannio*

Introduzione

Le attività di raccolta, organizzazione e conservazione dei dati costituiscono uno dei principali compiti dei sistemi informatici. Gli elenchi di utenze telefoniche, le quotazioni delle azioni nei mercati telematici internazionali, i saldi dei conti correnti bancari o le disponibilità di spesa associate alle carte di credito, l'elenco degli iscritti a una facoltà universitaria e gli esiti dei loro esami sono esempi di dati indispensabili a gestire alcune attività umane. I sistemi informatici garantiscono che questi dati vengano conservati in modo permanente su dispositivi per la loro memorizzazione, aggiornati per riflettere rapidamente le loro variazioni e resi accessibili alle interrogazioni degli utenti, talvolta distribuiti in modo capillare sul territorio. Si pensi, per esempio, all'interrogazione sulla disponibilità di spesa sulle carte di credito, effettuata tramite semplici dispositivi disponibili presso milioni di esercizi commerciali (quali alberghi, negozi o agenzie), che consente di addebitare sulle carte di credito spese effettuate in ogni parte del mondo.

Questo libro è dedicato alla gestione dei dati tramite sistemi informatici; descrive perciò i concetti necessari per rappresentare i dati su un calcolatore, i linguaggi che consentono il loro aggiornamento e ritrovamento, e le architetture informatiche specializzate nella gestione dei dati. In questo primo capitolo vengono introdotti i concetti di sistema informativo e di base di dati, per poi soffermarsi sulle principali caratteristiche dei sistemi informatici per gestire basi di dati.

1.1 Sistemi informativi, informazioni e dati

Nello svolgimento di ogni attività, sia a livello individuale sia in organizzazioni di ogni dimensione, sono essenziali la disponibilità di informazioni e la capacità di gestirle in modo efficace; ogni organizzazione è dotata di un *sistema informativo*, che organizza e gestisce le informazioni necessarie per perseguire gli scopi dell'organizzazione stessa.

L'esistenza del sistema informativo è in parte indipendente dalla sua automatizzazione. A sostegno di questa affermazione possiamo ricordare che i sistemi informativi esistono da molto prima dell'invenzione e della diffusione dei calcolatori elettronici; per esempio, gli archivi delle banche o dei servizi anagrafici sono istituiti da vari secoli. Per indicare la porzione automatizzata del sistema informativo viene di solito utilizzato il termine *sistema informatico*. La diffusione capillare dell'informatica a quasi tutte le attività umane, che ha caratterizzato gli ultimi vent'anni, fa sì che gran parte dei sistemi informativi siano anche, in buona misura, sistemi informatici.

Nelle attività umane più semplici, le informazioni vengono rappresentate e scambiate secondo le tecniche naturali tipiche delle attività stesse: la lingua, scritta o parla-

ta, disegni, figure, numeri. In alcune attività, può addirittura non esistere una rappresentazione esplicita dell'informazione, che viene ricordata a memoria, in maniera più o meno precisa. In ogni caso, possiamo dire che, a mano a mano che le attività si sono andate sistematizzando, sono state individuate opportune forme di organizzazione e codifica delle informazioni.

Nei sistemi informatici, per ragioni che in parte sono tecnologiche e in parte sono legate alla semplicità dei meccanismi di gestione, il concetto di rappresentazione e codifica viene portato all'estremo: le informazioni vengono rappresentate per mezzo di *dati*, che hanno bisogno di essere interpretati per fornire informazioni. Come per molti termini fondamentali, è difficile dare una definizione precisa del concetto di dato e soprattutto delle differenze fra *dato* e *informazione*: in modo approssimativo possiamo dire che i dati da soli non hanno alcun significato, ma, una volta interpretati e correlati opportunamente, essi forniscono informazioni, che consentono di arricchire la nostra conoscenza del mondo. Come ulteriore contributo riportiamo le definizioni dei due termini contenute in un recente dizionario [81]:

informazione: notizia, dato o elemento che consente di avere conoscenza più o meno esatta di fatti, situazioni, modi di essere;

dato: ciò che è immediatamente presente alla conoscenza, prima di ogni elaborazione; (in informatica) elementi di informazione costituiti da simboli che devono essere elaborati.

Per esempio, la stringa `Ferrari`, e il numero `8`, scritti su un foglio di carta, sono due dati e da soli non significano niente. Se il foglio di carta è relativo alle ordinazioni presso il ristorante di un albergo la notte di Capodanno e sono note le regole che camerieri e cassiere devono seguire, allora si può dedurre che è stata ordinata una bottiglia di spumante della marca 'Ferrari', che va addebitata sul conto della camera numero 8: con le indicazioni aggiuntive, i dati diventano informazione e arricchiscono la conoscenza. Chiaramente, un foglietto con gli stessi dati sul taccuino di un radiocronista sportivo avrebbe un significato molto diverso.

Introdotto così il concetto di dato, possiamo passare a quello di base di dati, oggetto principale di questo testo. Varie accezioni sono possibili per questo termine; secondo la più generale di esse, una *base di dati* è una collezione di dati, utilizzati per rappresentare le informazioni di interesse per un sistema informativo. In questo libro, considereremo una accezione molto più specifica del termine. A tale scopo è dedicato il Paragrafo 1.2.

Concludiamo invece questo paragrafo con una osservazione. In molte applicazioni, i dati hanno caratteristiche più stabili rispetto a quelle delle procedure (manuali o automatizzate) che operano su di essi. Riferendoci a un esempio già citato, possiamo osservare che i dati relativi alle applicazioni bancarie hanno una struttura sostanzialmente invariata da decenni, mentre le procedure che agiscono su di essi variano con una certa frequenza, come ogni cliente può facilmente verificare. Tra l'altro, quando una procedura sostituisce un'altra, la nuova procedura "eredita" i dati della vecchia, con opportune trasformazioni, più o meno semplici. Questa caratteristica di stabilità porta ad affermare che i dati costituiscono una "risorsa" per l'organizzazione che li gestisce, un patrimonio significativo da sfruttare e proteggere.

1.2 Basi di dati e sistemi di gestione di basi di dati

L'attenzione ai dati ha caratterizzato le applicazioni dell'informatica fin dalle sue origini, ma sistemi software specificamente dedicati alla gestione dei dati sono stati realizzati solo a partire dalla fine degli anni Sessanta del secolo scorso, e tuttora alcune applicazioni non ne fanno uso. In assenza di un software specifico, la gestione dei dati è affidata ai linguaggi di programmazione tradizionali, per esempio C e Fortran, oppure, in tempi più recenti, ai linguaggi a oggetti, tra cui C++ e Java. Esistono ancora diverse applicazioni scritte in COBOL, un linguaggio di programmazione degli anni Sessanta, che può ritenersi ormai superato.

L'approccio "convenzionale" alla gestione dei dati sfrutta la presenza di archivi o *file* per memorizzare i dati in modo persistente sulla memoria di massa. Un file consente di memorizzare e ricercare dati, ma fornisce solo semplici meccanismi di accesso e di condivisione. Secondo questo approccio, le procedure scritte in un linguaggio di programmazione sono completamente autonome; ciascuna di esse definisce e utilizza uno o più file "privati". Eventuali dati di interesse per più programmi sono replicati tante volte quanti sono i programmi che li utilizzano, con evidente ridondanza e possibilità di incoerenza.

Illustriamo questo concetto con riferimento a una situazione reale, di media complessità. In un'università, le informazioni relative ai docenti possono essere utilizzate in vario modo e da diverse persone o uffici; per esempio, potremmo avere le seguenti situazioni (alcune delle quali sono state semplificate o adattate a fini di presentazione):

- l'ufficio del personale è di solito responsabile delle informazioni relative alla "carriera" (per esempio, alla distinzione fra ricercatore, professore associato e professore ordinario e alla anzianità);
- le presidenze delle facoltà (sulla base delle delibere dei consigli delle facoltà stesse) mantengono le informazioni sugli incarichi di insegnamento dei docenti;
- l'ufficio Web di Ateneo, interessato a pubblicare le informazioni sui corsi e i relativi docenti, utilizza parte di quelle illustrate ai punti precedenti, ma necessita anche di altre, per esempio i recapiti dei docenti e i programmi dei corsi;
- l'ufficio stipendi (quasi sempre diverso dall'ufficio del personale) utilizza le informazioni sulla carriera e quelle sugli incarichi didattici per calcolare le retribuzioni, sulla base delle regole fissate dalla legge e delle eventuali integrazioni locali.

Se ciascuno di questi soggetti gestisse separatamente le informazioni di proprio interesse, avremmo molte duplicazioni e, come spesso succede in presenza di duplicazioni, a lungo andare ci sarebbero molti dati con le varie copie non aggiornate nello stesso modo. Per esempio, potremmo trovare sul sito Web una qualifica diversa da quella corretta, riportata nell'archivio dell'ufficio del personale.

Le basi di dati sono state concepite in buona misura per superare questo tipo di inconvenienti, gestendo in modo integrato e flessibile le informazioni di interesse per diversi soggetti, limitando i rischi di ridondanza e incoerenza. In generale, potremmo dire che una base di dati è semplicemente una collezione di dati, di interesse per una o più applicazioni. Preferiamo però dare una definizione che abbia un contenuto anche tecnologico.

Un *sistema di gestione di basi di dati* (in inglese *Data Base Management System*, abbreviato con DBMS) è un sistema software in grado di gestire collezioni di dati che siano *grandi*, *condivise* e *persistenti*, assicurando la loro *affidabilità* e *privatezza*. Come ogni prodotto informatico, un DBMS deve essere *efficiente* ed *efficace*. Una *base di dati* è una collezione di dati gestita da un DBMS.

Precisiamo le caratteristiche dei DBMS e delle basi di dati su cui si fondano le definizioni date in precedenza.

- Le basi di dati sono *grandi*, nel senso che possono avere anche dimensioni enormi e comunque in generale molto maggiori della memoria centrale disponibile. Alla data di redazione di questo testo, le più grandi basi di dati hanno dimensioni dell'ordine delle centinaia (o forse migliaia) di terabyte e contengono migliaia di miliardi di record. Di conseguenza, i DBMS devono prevedere una gestione dei dati in memoria secondaria. Ovviamente, possono esistere anche basi di dati "piccole", ma i sistemi devono poter gestire i dati senza porre limiti alle dimensioni, a parte quelle fisiche dei dispositivi.
- Le basi di dati sono *condivise*, nel senso che applicazioni e utenti diversi devono poter accedere, secondo opportune modalità, a dati comuni. È importante notare che in questo modo si riduce la *ridondanza* dei dati, poiché si evitano ripetizioni, e conseguentemente si riduce anche la possibilità di *inconsistenze*: se esistono varie copie degli stessi dati, è possibile che esse, in qualche momento, non siano uguali; viceversa, se ogni dato è memorizzato nel sistema in modo univoco, non è possibile incorrere in disallineamenti. Per garantire l'accesso condiviso ai dati da parte di molti utenti che operano contemporaneamente, il DBMS dispone di un meccanismo apposito, detto *controllo di concorrenza*. Esistono al giorno d'oggi basi di dati che devono gestire più di dieci milioni di operazioni (per esempio, lettura o scrittura di record) al secondo.
- Le basi di dati sono *persistenti*, cioè hanno un tempo di vita che non è limitato a quello delle singole esecuzioni dei programmi che le utilizzano. In contrasto, ricordiamo che i dati gestiti da un programma in memoria centrale hanno una vita che inizia e termina con l'esecuzione del programma; tali dati, quindi, non sono persistenti.
- I DBMS garantiscono la *affidabilità*, cioè la capacità del sistema di conservare sostanzialmente intatto il contenuto della base di dati (o almeno di permetterne la ricostruzione) in caso di malfunzionamenti hardware e software. Questo aspetto è essenziale se si pensa che in molte applicazioni (per esempio quelle finanziarie) ogni dato ha un valore enorme, che deve essere preservato nel tempo e a fronte di qualsiasi guasto del sistema, errore umano, o anche evento catastrofico. A questo scopo i DBMS forniscono specifiche funzionalità di *salvataggio* e *ripristino* (*backup* e *recovery*). In alcuni casi, i DBMS gestiscono, in modo controllato, versioni replicate dei dati, collocate su dispositivi fisici diversi e talvolta su server che sono disposti a distanza, così da garantire maggiore affidabilità complessiva.
- I DBMS garantiscono la *privatezza* dei dati. Ciascun utente, opportunamente riconosciuto (per esempio in base a un nome d'utente specificato all'atto di interagire con il DBMS), viene abilitato a svolgere solo determinate azioni sui dati, attraverso meccanismi di *autorizzazione*.

- I DBMS sono *efficienti*, cioè capaci di svolgere le operazioni utilizzando un insieme di risorse (tempo e spazio) che sia accettabile per gli utenti. Questa caratteristica dipende dalle tecniche utilizzate nell'implementazione del DBMS e dalla bontà della realizzazione della base di dati da parte dei suoi progettisti. Va sottolineato che i DBMS forniscono un insieme piuttosto ampio di funzionalità che richiedono molte risorse, e quindi possono garantire efficienza solo a condizione che il sistema informatico su cui sono installati sia adeguatamente dimensionato.
- I DBMS sono *efficaci* in quanto sono capaci di rendere produttive, in ogni senso, le attività dei loro utenti. Questa definizione è chiaramente generica e non corrisponde a un aspetto specifico. L'attività di progettazione della base di dati e delle applicazioni che la utilizzano mira essenzialmente a garantire una buona efficacia complessiva del sistema.

È importante sottolineare che la gestione di collezioni di dati grandi e persistenti è possibile anche per mezzo di strumenti meno sofisticati dei DBMS, a cominciare dai file già citati, presenti in tutti i sistemi operativi. I file sono stati introdotti per gestire insiemi di dati "localmente" a una specifica procedura o applicazione. I DBMS sono stati concepiti e realizzati per estendere le funzioni dei file system, fornendo la possibilità di accesso condiviso agli stessi dati da parte di più utenti e applicazioni, e garantendo anche molti altri servizi in maniera integrata. Precisiamo inoltre che i DBMS, a loro volta, utilizzano file per la memorizzazione dei dati; i file gestiti dal DBMS ammettono però organizzazioni dei dati più sofisticate.

Ritornando all'esempio brevemente illustrato all'inizio di questo paragrafo, possiamo dire che un modello ideale che adotti fino in fondo le idee appena discusse dovrebbe prevedere l'utilizzo di una sola base di dati, con tutte le informazioni di interesse (e quindi non solo quelle sui docenti, ma per esempio anche quelle sui corsi e sulle varie strutture dell'ateneo, per esempio facoltà, scuole, dipartimenti e corsi di laurea, nonché sugli aspetti contabili e amministrativi e tanti altri che non abbiamo per niente citato). Tale base di dati verrebbe poi a essere utilizzata dai vari uffici (e persone), ciascuno per le proprie competenze, attraverso programmi diversi. In effetti, in molti casi, non è possibile o non è conveniente, per varie ragioni, avere un'unica base di dati, ma è importante avere presente che l'obiettivo della integrazione e condivisione è comunque importante, e che i flussi di informazione devono essere coordinati e governati: nell'esempio, l'utilizzo della stessa base di dati da parte dell'ufficio stipendi e dell'ufficio del personale è senz'altro un obiettivo importante. Viceversa, può essere più difficile integrare i dati di questa base di dati con quelli del sito Web, in quanto quest'ultimo deve essere sempre disponibile per la consultazione, oltre che accessibile dall'esterno, mentre buona parte dei dati sul personale, non rilevanti per il sito, sono riservati e delicati. È quindi ragionevole pensare a soluzioni che prevedano l'uso di una base di dati per mantenere i dati di interesse per il sito Web, diversa da quella relativa a personale e stipendi. È però opportuno pensare che la base di dati per il sito Web sia periodicamente aggiornata, in modo predefinito e automatico, con le informazioni sulle carriere dei docenti, al fine di evitare incoerenze.

Più in generale, possiamo pensare che una organizzazione complessa (azienda o ente pubblico) utilizzi un insieme di basi dati, che siano ciascuna dedicata a un insieme di applicazioni strettamente correlate (quindi con un certo grado di integrazione e

condivisione) e che siano al tempo stesso coinvolte in operazioni di interscambio di informazioni finalizzate a evitare duplicazioni di dati e ripetizioni di attività.

Abbiamo detto che un DBMS mette a disposizione una base di dati in generale a più utenti e ai loro programmi. Torneremo presto, nel Paragrafo 1.4, a commentare le caratteristiche dei programmi (e linguaggi utilizzati per scriverli) e degli utenti. Qui invece facciamo un'altra osservazione, relativa peraltro a un argomento che non viene sviluppato in questo volume, ma nel secondo [6]. Esistono varie organizzazioni architetturali per l'accesso a basi di dati: per esempio, i programmi possono essere eseguiti sullo stesso calcolatore che gestisce i dati oppure su un altro; gli utenti possono utilizzare i cosiddetti "terminali stupidi", cioè privi di capacità elaborativa, oppure personal computer che svolgono parte delle attività, in una architettura di tipo "client/server". Negli ultimi anni, si è poi molto diffuso l'accesso a basi di dati attraverso il World Wide Web: in effetti, la maggior parte dei siti Web contengono pagine che sono generate dinamicamente (cioè al momento della richiesta) a partire da dati contenuti in basi di dati.

1.3 Modelli dei dati

Un *modello dei dati* è un insieme di concetti utilizzati per organizzare i dati di interesse e descriverne la struttura in modo che essa risulti comprensibile a un elaboratore. Ogni modello dei dati fornisce *meccanismi di strutturazione*, analoghi ai costruttori di tipo dei linguaggi di programmazione, che permettono di definire nuovi tipi sulla base di tipi predefiniti (elementari) e costruttori di tipo. Per esempio, il C permette di costruire tipi per mezzo dei costruttori *struct, union, enum, * (pointer)*.

Il *modello relazionale* dei dati, attualmente il più diffuso, permette di definire tipi per mezzo del costruttore *relazione*, che consente di organizzare i dati in insiemi di record a struttura fissa. Una relazione viene spesso rappresentata per mezzo di una tabella, le cui righe rappresentano specifici record e le cui colonne corrispondono ai campi dei record; l'ordine delle righe e delle colonne è sostanzialmente irrilevante. Per esempio, i dati relativi ai corsi universitari e ai loro docenti e all'inserimento dei corsi nel manifesto degli studi dei vari corsi di laurea possono essere organizzati per mezzo di due relazioni, DOCENZA e MANIFESTO, rappresentabili con le tabelle in Figura 1.1. Come si vede nella figura, in una base di dati relazionale ci sono in generale più relazioni.

Il modello relazionale, definito formalmente agli inizi degli anni Settanta del secolo scorso, e affermatosi nel decennio successivo, è, come abbiamo detto, il più diffuso e viene utilizzato in questo libro come modello di riferimento. Oltre al modello relazionale sono stati definiti altri quattro tipi di modelli:

- il *modello gerarchico*, basato sull'uso di strutture ad albero (e quindi gerarchie, da cui il nome), definito durante la prima fase di sviluppo dei DBMS (anni Sessanta), ma tuttora utilizzato;
- il *modello reticolare* (detto anche modello CODASYL, dal comitato di standardizzazione che lo definì con precisione), basato sull'uso di grafi, sviluppato successivamente al modello gerarchico (inizio anni Settanta);

DOCENZA

Corso	NomeDocente
Basi di dati	Rossi
Reti	Neri
Linguaggi	Verdi

MANIFESTO

CdL	Materia	Anno
IngInf	Basi di dati	2
IngInf	Reti	3
IngInf	Linguaggi	2
IngEl	Basi di dati	3
IngEl	Reti	3

Figura 1.1 Esempio di base di dati relazionale.

- il *modello a oggetti*, sviluppato negli anni Ottanta come evoluzione del modello relazionale, che estende alle basi di dati il paradigma di programmazione a oggetti; al modello a oggetti è dedicato un capitolo nel secondo volume [6];
- il *modello XML*, sviluppato negli anni Novanta come rivisitazione del modello gerarchico, in cui però i dati vengono presentati assieme alla loro descrizione e non devono sottostare rigidamente a un'unica struttura logica (tenuto conto di queste due caratteristiche, si dice che il modello XML è auto-descrittivo e semi-strutturato); anche questo modello è approfondito nel secondo volume [6];
- modelli semistrutturati e flessibili, sviluppati nel contesto dei cosiddetti sistemi *NoSQL*, che cercano di superare, in specifici contesti applicativi, alcune delle limitazioni dei sistemi relazionali, in termini tanto di prestazioni quanto di rigidità dell'organizzazione dei dati.

I modelli dei dati precedentemente elencati sono effettivamente disponibili su DBMS commerciali; essi vengono detti *logici*, per sottolineare il fatto che le strutture utilizzate da questi modelli, pur essendo astratte, riflettono una particolare organizzazione (ad alberi, a grafi, a tabelle o a oggetti). Più recentemente rispetto ai primi modelli logici, sono stati introdotti altri modelli dei dati, detti *concettuali*, utilizzati per descrivere i dati in maniera completamente indipendente dalla scelta del modello logico. Questi modelli non sono disponibili su DBMS commerciali. Il loro nome deriva dal fatto che essi tendono a descrivere i *concetti* del mondo reale, piuttosto che i dati utili a rappresentarli. Essi vengono utilizzati nella fase preliminare del processo di progettazione di basi di dati, per analizzare nel modo migliore la realtà di interesse, senza "contaminazioni" di tipo realizzativo. Nella Parte Seconda di questo libro, dedicata al progetto delle basi di dati, vedremo in dettaglio un modello concettuale, il modello *Entità-Relazione*.

1.3.1 Schemi e istanze

Nelle basi di dati esiste una parte sostanzialmente invariante nel tempo, detta *schema* della base di dati, costituita dalle caratteristiche dei dati, e una parte variabile nel tempo, detta *istanza* o *stato* della base di dati, costituita dai valori effettivi. Nell'esempio della Figura 1.1, le relazioni hanno una struttura fissa: la relazione DOCENZA ha due

colonne (dette *attributi*), che si riferiscono rispettivamente a corsi e docenti. Lo *schema di una relazione* è costituito dalla sua intestazione, cioè dal nome della relazione seguito dai nomi dei suoi attributi; per esempio:

DOCENZA(Corso,NomeDocente)

Viceversa, le righe della tabella variano nel tempo, e corrispondono ai corsi attualmente offerti e ai relativi docenti. Durante la vita della base di dati, docenti e corsi vengono aggiunti, tolti o modificati; in modo analogo, il manifesto degli studi viene modificato di anno in anno. L'*istanza di una relazione* è costituita dall'insieme, variante nel tempo, delle sue righe (che sono coerenti con lo schema); nell'esempio abbiamo le tre coppie:

Basi di dati	Rossi
Reti	Neri
Linguaggi	Verdi

Le tre righe fanno chiaramente riferimento allo schema e solo attraverso di esso possono essere interpretate.

Si dice anche che lo schema è la componente *intensionale* della base di dati e l'istanza la componente *estensionale*. Queste definizioni verranno riprese e approfondite nel Capitolo 2.

1.3.2 Livelli di astrazione nei DBMS

La nozione di modello e di schema descritta in precedenza può essere ulteriormente sviluppata tenendo presenti altre dimensioni nella descrizione dei dati. In particolare, esiste una proposta di architettura standardizzata per DBMS articolata su tre livelli, detti rispettivamente *esterno*, *logico*[1] e *interno*; per ciascun livello esiste uno schema.

- Lo *schema logico* costituisce una descrizione dell'intera base di dati per mezzo del modello logico adottato dal DBMS (cioè tramite uno dei modelli descritti in precedenza, per esempio relazionale o a oggetti).
- Lo *schema interno* costituisce la rappresentazione dello schema logico per mezzo di strutture fisiche di memorizzazione. Per esempio, una relazione può essere realizzata fisicamente per mezzo di un file sequenziale, o di un file hash, o di un file sequenziale con uno o più indici.
- Uno *schema esterno* costituisce la descrizione di una porzione della base di dati di interesse, per mezzo del modello logico. Uno schema esterno può prevedere organizzazioni dei dati diverse rispetto a quelle utilizzate nello schema logico, che riflettono il punto di vista di un particolare utente o insieme di utenti. Pertanto, è possibile associare a uno schema logico vari schemi esterni.

[1] Questo livello viene da alcuni autori chiamato *concettuale*, seguendo la terminologia utilizzata originariamente nella proposta. Noi preferiamo il termine "logico", in quanto, come abbiamo visto, usiamo il termine "concettuale" per altri scopi.

ELETTRONICA

CdL	Materia	Anno
IngEl	Basi di dati	3
IngEl	Reti	3

Figura 1.2 Una "vista" relazionale.

Nei sistemi più moderni il livello esterno non è esplicitamente presente, ma è possibile definire relazioni derivate (o *viste*, dall'inglese *views*). Per esempio, relativamente alla base di dati di Figura 1.1, uno studente del corso di laurea in Ingegneria Elettronica (IngEl) potrebbe essere interessato solo ai corsi offerti dal manifesto del suo corso di laurea; questa informazione è presente nella relazione ELETTRONICA, mostrata in Figura 1.2, ottenuta come vista a partire dalla relazione MANIFESTO. Inoltre, tramite il meccanismo delle *autorizzazioni di accesso*, è possibile disciplinare gli accessi degli utenti alla base di dati.

1.3.3 Indipendenza dei dati

L'architettura a livelli così definita garantisce *l'indipendenza dei dati*, la principale proprietà dei DBMS. In generale, questa proprietà permette a utenti e programmi applicativi che utilizzano una base di dati di interagire a un elevato livello di astrazione, che prescinde dai dettagli realizzativi utilizzati nella costruzione della base di dati. In particolare, l'indipendenza dei dati può essere caratterizzata ulteriormente come indipendenza fisica e logica.

- L'*indipendenza fisica* consente di interagire con il DBMS in modo indipendente dalla struttura fisica dei dati. In base a questa proprietà è possibile modificare le strutture fisiche (per esempio le modalità di organizzazione dei file gestiti dal DBMS o la allocazione fisica dei file sui dispositivi di memorizzazione) senza influire sulle descrizioni dei dati ad alto livello e quindi sui programmi che utilizzano i dati stessi.
- L'*indipendenza logica* consente di interagire con il livello esterno della base di dati in modo indipendente dal livello logico. Per esempio, è possibile aggiungere uno schema esterno in base alle esigenze di un nuovo utente oppure modificare uno schema esterno senza dover modificare lo schema logico e perciò la sottostante organizzazione fisica dei dati. Dualmente, è possibile modificare il livello logico, mantenendo inalterate le strutture esterne (modificandone ovviamente la definizione in termini delle strutture logiche) di interesse per l'utente.

È importante sottolineare che gli accessi alla base di dati avvengono solo attraverso il livello esterno (che può coincidere con quello logico); è il DBMS che traduce le operazioni in termini dei livelli sottostanti. L'architettura a livelli è quindi il meccanismo fondamentale attraverso cui i DBMS realizzano l'indipendenza dei dati.

1.4 Linguaggi e utenti delle basi di dati

I DBMS sono caratterizzati, da un lato, dalla presenza di molteplici linguaggi per la gestione dei dati, dall'altro dalla presenza di molteplici tipologie di utenti.

1.4.1 Linguaggi per basi di dati

Su un DBMS è possibile specificare operazioni di vario tipo, in particolare quelle relative agli schemi e alle istanze. Al riguardo, i linguaggi per basi di dati si distinguono in due categorie:

- *linguaggi di definizione dei dati* o *Data Definition Language* (abbreviato con DDL), utilizzati per definire gli schemi logici, esterni e fisici e le autorizzazioni per l'accesso;
- *linguaggi di manipolazione dei dati* o *Data Manipulation Language* (abbreviato con DML), utilizzati per l'interrogazione e l'aggiornamento delle istanze di basi di dati.

Va sottolineato che alcuni linguaggi, come per esempio il linguaggio SQL, che vedremo approfonditamente nei Capitoli 4 e 5, presentano in forma integrata le funzionalità di entrambe le categorie.

L'accesso ai dati può essere effettuato con varie modalità:

- tramite linguaggi testuali interattivi, soprattutto il linguaggio SQL, nelle sue varie versioni; per esempio, possiamo definire la struttura della relazione DOCENZA, già mostrata in Figura 1.1, sottoponendo a un interprete SQL (disponibile in tutti i DBMS) la seguente istruzione[2] (che appartiene quindi al DDL):

```
create table Docenza(
 Corso        character(20),
 NomeDocente character(30)
)
```

  La seguente istruzione, invece, permette di visualizzare, sempre in un ambiente interattivo, i corsi di ingegneria informatica del secondo anno con i relativi docenti:

```
select Corso, NomeDocente
from Docenza, Manifesto
where Corso=Materia
  and Anno=2
  and CdL='IngInf'
```

[2]Per non appesantire, mostriamo un'istruzione che, pur formalmente corretta, è incompleta, in quanto, come vedremo più avanti, non contiene la specifica della chiave primaria.

- tramite comandi simili a quelli interattivi immersi in linguaggi di programmazione, quali C, C++, Java, COBOL, come vedremo nel Capitolo 10. Questi linguaggi di programmazione si dicono *linguaggi ospite* perché "ospitano" comandi scritti nel linguaggio per basi di dati. Una caratteristica importante di questo approccio è la possibilità di realizzare applicazioni parametriche e di farle eseguire anche a un utente che non conosca SQL: con riferimento al semplice esempio precedente, si potrà scrivere un programma che permette di ricercare corsi e docenti per un certo corso di laurea e un certo anno, specificati dall'utente solo al momento dell'esecuzione del programma (ma non noti al momento della scrittura dello stesso);
- tramite comandi simili a quelli interattivi immersi in linguaggi di sviluppo *ad hoc*, spesso con funzionalità specifiche (per esempio per la generazione di grafici, di stampe complesse, oppure di maschere su video). Questi linguaggi variano purtroppo molto da sistema a sistema e possiamo quindi solo accennare ad alcuni aspetti nelle appendici (disponibili sui siti di supporto al testo), che sono appunto dedicate a specifici sistemi;
- tramite interfacce amichevoli che permettono di sintetizzare interrogazioni senza usare un linguaggio testuale. Anche queste interfacce differiscono molto le une dalle altre e sono in continua evoluzione. Pure per esse, accenniamo nelle appendici agli aspetti più semplici e importanti. Vediamo qui un esempio, relativo alla stessa interrogazione vista sopra in SQL: la Figura 1.3 mostra la sua realizzazione in Postgres, un DBMS open source molto diffuso.

Una gran parte dei programmi per la immissione dei dati, la loro gestione e la stampa hanno una struttura regolare; conseguentemente, la presenza di linguaggi di sviluppo e di interfacce amichevoli può semplificare significativamente la produzione di applicazioni, facilitando il compito del programmatore e riducendo i tempi e i costi di sviluppo.

1.4.2 Utenti e progettisti

Varie categorie di persone possono interagire con una base di dati o con un DBMS. Descriviamo brevemente le più importanti.

- L'*amministratore della base di dati* (o *Database Administrator*, abbreviato con DBA) è la persona (o gruppo di persone) responsabile della progettazione, controllo e amministrazione della base di dati. Il DBA ha il compito di mediare le varie esigenze espresse dagli utenti, in genere contrastanti fra loro, garantendo un controllo centralizzato sui dati. L'amministratore delle basi di dati è responsabile, in particolare, di garantire sufficienti prestazioni, di assicurare l'affidabilità del sistema, e di gestire le autorizzazioni di accesso ai dati. Al progetto delle basi di dati è dedicata la Parte Seconda del libro, alle tematiche tecnologiche, da cui dipendono le prestazioni, è dedicata la Parte Terza.
- I *progettisti e programmatori di applicazioni* definiscono e realizzano i programmi che accedono alla base di dati. Essi utilizzano il linguaggio di manipolazione dei dati (DML) oppure i vari strumenti di supporto alla generazione di interfacce verso

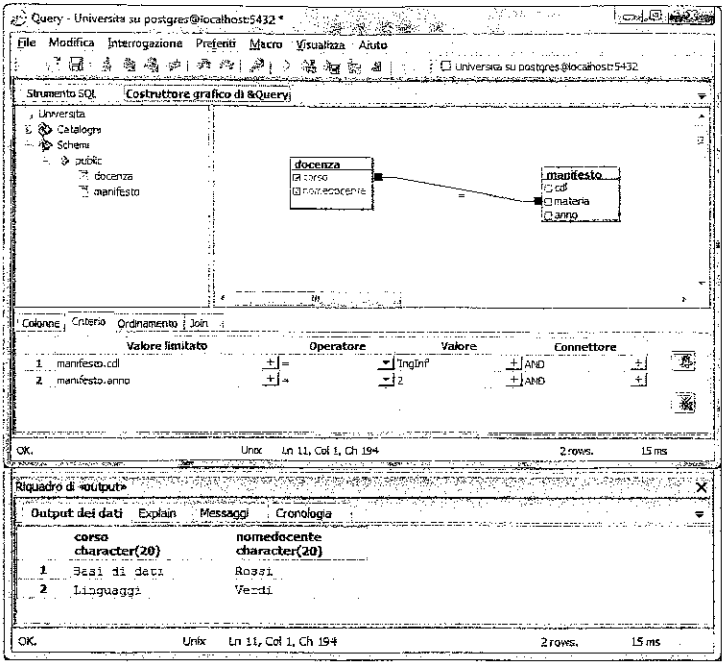

Figura 1.3 Un'interrogazione formulata con un'interfaccia amichevole in Postgres e il suo risultato.

la base di dati descritti in precedenza. Alla programmazione di applicazioni sulle basi di dati sono dedicati i capitoli relativi al linguaggio SQL, nella Parte Prima del testo e il Capitolo 10, sullo sviluppo di applicazioni, nella Parte Seconda.

- Gli *utenti* utilizzano la base di dati per le proprie attività. Essi possono a loro volta essere divisi in due categorie:
  - *utenti finali* (o *terminalisti*), che utilizzano *transazioni*, cioè programmi che realizzano attività predefinite e di frequenza elevata, con poche eccezioni previste a priori;
  - *utenti casuali*, in grado di impiegare i linguaggi interattivi per l'accesso alla base di dati, formulando interrogazioni (o aggiornamenti) di tipo vario. Essi possono essere specializzati (rispetto al linguaggio che utilizzano) e interagire frequentemente con la base di dati. Si noti che il termine "casuale" sta a indicare il fatto che le interrogazioni specificate da utenti casuali non sono predefinite.

1.5 Vantaggi e svantaggi dei DBMS

Concludiamo questo capitolo riassumendo le caratteristiche essenziali delle basi di dati e dei DBMS, e i relativi vantaggi e svantaggi. Possiamo elencare i seguenti vantaggi.

- I DBMS permettono di considerare i dati come una risorsa comune di una organizzazione, a disposizione (con opportune forme di controllo) di tutte le sue componenti.
- La base di dati fornisce un modello unificato e preciso della parte del mondo reale di interesse per l'organizzazione, utilizzabile nelle applicazioni attuali e, con possibili estensioni, in applicazioni future.
- Con l'uso di un DBMS è possibile un controllo centralizzato dei dati, che può essere arricchito da forme di standardizzazione e beneficiare di "economie di scala".
- La condivisione permette di ridurre ridondanze e inconsistenze.
- L'indipendenza dei dati, caratteristica fondamentale dei DBMS, favorisce lo sviluppo di applicazioni più flessibili e facilmente modificabili.

L'uso dei DBMS comporta anche alcuni aspetti negativi, o almeno delicati, fra i quali i seguenti.

- I DBMS sono prodotti spesso costosi, complessi e abbastanza diversi da molti altri strumenti informatici. La loro introduzione comporta quindi notevoli investimenti, diretti (acquisto del prodotto) e indiretti (acquisizione delle risorse hardware e software necessarie, conversione delle applicazioni, formazione del personale).
- I DBMS forniscono, in forma integrata, una serie di servizi, che sono necessariamente associati a un costo. Nei casi in cui alcuni di questi servizi non siano necessari, è difficile scorporare dagli altri quelli effettivamente richiesti, e ciò può comportare una riduzione di prestazioni.

Concludendo, possiamo dire che si incontrano situazioni nelle quali l'adozione di un DBMS può risultare sconveniente: applicazioni con uno o pochi utenti, senza necessità di accessi concorrenti e relativamente stabili nel tempo possono essere realizzate più proficuamente con file ordinari piuttosto che con DBMS. Tuttavia, la tecnologia dei DBMS si è notevolmente evoluta negli ultimi anni, traducendosi in sistemi sempre più efficienti e affidabili su architetture sempre più diffuse e poco costose, aumentando la convenienza di sviluppare applicazioni con un DBMS.

Note bibliografiche

Esistono molti testi di tipo generale sulle basi di dati, quasi tutti in lingua inglese ma poi tradotti in italiano: segnaliamo in particolare quelli di ElMasri e Navathe [41], Silberschatz, Korth e Sudarshan [71], Ramakhrishnan e Gehrke [66] e Garcia-Molina, Ullman e Widom [46], che coprono in modo equilibrato gli aspetti metodologici e quelli tecnologici. Quello di Garcia-Molina, Ullman e Widom [46] tratta in modo integrato tecnologia esistente e aspetti di natura metodologica. Per dettagli relativi ai singoli aspetti citati in questo capitolo, rimandiamo ai capitoli successivi, in cui verranno approfonditi, e alle relative note bibliografiche.

2 Il modello relazionale

La maggior parte dei sistemi di basi di dati oggi sul mercato si fonda sul modello relazionale, che fu proposto in una pubblicazione scientifica (di E.F. Codd [30]), nel 1970, al fine di superare le limitazioni dei modelli all'epoca utilizzati a livello logico, che non permettevano di realizzare efficacemente la proprietà di indipendenza dei dati, già riconosciuta fondamentale. L'affermazione del modello relazionale è stata abbastanza lenta, a causa dell'alto livello di astrazione: non è stato immediato individuare realizzazioni efficienti per strutture significativamente diverse da quelle utilizzate allora. Infatti, nonostante i primi prototipi di sistemi relazionali siano stati realizzati già nei primi anni Settanta, i primi sistemi relazionali sono apparsi sul mercato nel 1981, acquisendone una frazione significativa solo a metà degli anni Ottanta.

La presentazione del modello relazionale è articolata in quattro capitoli, questo e i tre successivi. Nel presente capitolo sono illustrate le caratteristiche strutturali del modello, cioè le modalità secondo cui esso permette di organizzare i dati. Dopo una breve discussione sui vari modelli logici, si mostra come il concetto di relazione possa essere mutuato dalla teoria degli insiemi e utilizzato, con alcune varianti, per rappresentare le informazioni di interesse in una base di dati. Viene in particolare approfondito il fatto che le corrispondenze fra dati in strutture diverse sono rappresentate per mezzo dei dati stessi. Poi, dopo una breve discussione delle modalità per la rappresentazione di informazione incompleta, l'attenzione viene volta ai vincoli di integrità, che permettono di specificare proprietà aggiuntive che devono essere soddisfatte dalle basi di dati.

La presentazione del modello relazionale è completata nei tre capitoli successivi, il primo dedicato alla specifica delle operazioni di interrogazione di basi di dati relazionali e gli altri al linguaggio SQL, che, nei DBMS oggi esistenti, permette di definire, aggiornare e interrogare basi di dati.

2.1 Il modello relazionale: strutture

2.1.1 Modelli logici nei sistemi di basi di dati

Il *modello relazionale* si basa su due concetti, *relazione* e *tabella*, di natura diversa ma facilmente riconducibili l'uno all'altro. La nozione di *relazione* proviene dalla matematica, in particolare dalla teoria degli insiemi, mentre il concetto di *tabella* è semplice e intuitivo. La presenza contemporanea di questi due concetti, l'uno formale e l'altro intuitivo, è responsabile del grande successo ottenuto dal modello relazionale. Infatti, le tabelle risultano naturali e comprensibili anche per gli utenti finali (che spesso le utilizzano in tanti contesti per scopi diversi, senza riferimento alle basi di dati). D'altra parte, la disponibilità di una formalizzazione semplice e precisa ha permesso anche uno sviluppo teorico a supporto del modello con risultati di interesse concreto.

Il modello relazionale risponde al requisito dell'indipendenza dei dati, che, come abbiamo visto nel Capitolo 1, prevede una distinzione, nella descrizione dei dati, fra il livello *fisico* e il livello *logico*; gli utenti che accedono ai dati e i programmatori che sviluppano le applicazioni fanno riferimento solo al livello logico; i dati descritti al livello logico sono poi concretamente organizzati per mezzo di opportune strutture fisiche, ma per accedere ai dati non è necessario conoscere le strutture fisiche stesse. Al contrario, i modelli proposti precedentemente al modello relazionale, quello reticolare e quello gerarchico, includevano espliciti riferimenti alla sottostante struttura realizzativa, attraverso l'uso di puntatori e l'ordinamento fisico dei dati.

Una precisazione è utile prima di passare all'introduzione del modello relazionale. Il termine *relazione* viene utilizzato in questo testo (e in generale con riferimento alle basi di dati) in tre accezioni che, nei dettagli, differiscono in modo importante:

- *relazione matematica*, secondo la definizione normalmente data nella teoria degli insiemi elementare. Da questa nozione, che verrà richiamata nel prossimo paragrafo, derivano le altre due;
- *relazione* secondo la definizione del modello relazionale che, come vedremo nel Paragrafo 2.1.3, presenta alcune differenze rispetto a quella della teoria degli insiemi;
- *relazione*, come traduzione di *relationship*,[1] costrutto del modello concettuale *Entità-Relazione* (in inglese *Entity-Relationship*) utilizzato, come vedremo nel Capitolo 6, per descrivere legami tra entità del mondo reale.

2.1.2 Relazioni e tabelle

Ricordiamo che, in matematica, dati due insiemi D_1 e D_2, si chiama *prodotto cartesiano* di D_1 e D_2, in simboli $D_1 \times D_2$, l'insieme delle coppie ordinate (v_1, v_2), tali che v_1 è un elemento di D_1 e v_2 è un elemento di D_2. Per esempio, dati gli insiemi $A = \{1, 2, 4\}$ e $B = \{a, b\}$, il prodotto cartesiano $A \times B$ è costituito dall'insieme di tutte le possibili coppie in cui il primo elemento appartiene ad A e il secondo a B. Poiché A ha tre elementi e B due, si tratta quindi di sei coppie:

$$\{(1, a), (1, b), (2, a), (2, b), (4, a), (4, b)\}$$

Una *relazione matematica* sugli insiemi D_1 e D_2 (chiamati *domini* della relazione) è un sottoinsieme di $D_1 \times D_2$. Dati gli insiemi A e B di cui sopra, una possibile relazione matematica su A e B è costituita dall'insieme di coppie $\{(1, a), (1, b), (4, b)\}$.

Le relazioni possono essere rappresentate graficamente, in maniera utilmente espressiva, sotto forma tabellare. Le due tabelle riportate nella Figura 2.1 descrivono il prodotto cartesiano $A \times B$ e la relazione matematica su A e B sopra discusse.

Vale la pena fare una annotazione, importante dal punto di vista formale (anche se pressoché ovvia da quello pratico). Finora non abbiamo detto niente riguardo alla finitezza degli insiemi che consideriamo, e abbiamo quindi implicitamente ammesso

[1] Nei due casi precedenti si usa in inglese il termine *relation*, che quindi non è ambiguo rispetto a *relationship*. Talvolta sono usate altre traduzioni per *relationship*, come *associazione* o *correlazione*.

1	a
1	b
2	a
2	b
4	a
4	b

1	a
1	b
4	b

Figura 2.1 Rappresentazione tabellare di un prodotto cartesiano e di una relazione.

la possibilità di insiemi infiniti (e perciò di relazioni infinite). In pratica, poiché le nostre basi di dati devono essere memorizzate in sistemi di calcolo di dimensione finita, le relazioni sono necessariamente finite. Peraltro, in talune trattazioni teoriche, che comunque esulano dagli interessi di questo testo, vengono talvolta ammesse relazioni infinite. Al tempo stesso, è comodo a volte che i domini abbiano dimensione infinita (in modo che sia sempre possibile assumere l'esistenza di un valore non presente nella base di dati). Pertanto, assumeremo, ove necessario, che le nostre basi di dati siano costituite da relazioni finite su domini eventualmente infiniti.

Le definizioni precedenti di prodotto cartesiano e relazione matematica fanno riferimento a due insiemi, ma possono essere generalizzate rispetto al numero di insiemi. Dati $n > 0$ insiemi $D_1, D_2, \ldots, D_n$, non necessariamente distinti, il prodotto cartesiano di $D_1, D_2, \ldots, D_n$, indicato con $D_1 \times D_2 \times \ldots \times D_n$, è costituito dall'insieme delle n-uple $(v_1, v_2, \ldots, v_n)$ tali che v_i appartiene a D_i, per $1 \leq i \leq n$. Una relazione matematica sui domini $D_1, D_2, \ldots, D_n$ è un sottoinsieme del prodotto cartesiano $D_1 \times D_2 \times \ldots \times D_n$. Il numero n delle componenti del prodotto cartesiano (e quindi di ogni n-upla) viene detto *grado* del prodotto cartesiano e della relazione. Il numero di elementi (cioè di n-uple) della relazione viene chiamato, come di solito nella teoria degli insiemi, *cardinalità* della relazione. Nella Figura 2.2 sono mostrate le rappresentazioni tabellari del prodotto cartesiano e di una relazione di grado tre sui domini $C = \{x, y\}$, $D = \{a, b, c\}$ ed $E = \{3, 5\}$. La relazione ha cardinalità pari a sei.

Le relazioni (e le corrispondenti tabelle) possono essere utilizzate per rappresentare i dati di interesse per qualche applicazione. Per esempio, la relazione nella Figura 2.3 contiene i dati relativi ai risultati di un insieme di partite di calcio.

Essa è definita con riferimento a due domini *intero* e *stringa*, ognuno dei quali compare due volte. La relazione è infatti un sottoinsieme del prodotto cartesiano:

Stringa × Stringa × Intero × Intero

2.1.3 Relazioni con attributi

Sulle relazioni e sulle loro rappresentazioni tabellari possiamo fare varie osservazioni. In base alla definizione, una relazione matematica è un *insieme* di n-uple *ordinate* $(v_1, v_2, \ldots, v_n)$, con $v_1 \in D_1$, $v_2 \in D_2$, ..., $v_n \in D_n$. Con riferimento all'uso che

x	a	3
x	a	5
x	b	3
x	b	5
x	c	3
x	c	5
y	a	3
y	a	5
y	b	3
y	b	5
y	c	3
y	c	5

x	a	3
x	a	5
x	c	5
y	a	3
y	c	3
y	c	5

Figura 2.2 Rappresentazione tabellare di un prodotto cartesiano e di una relazione ternari.

facciamo delle relazioni per organizzare i dati nelle nostre basi di dati, possiamo dire che ciascuna n-upla contiene dati fra loro collegati, anzi stabilisce un legame fra loro: per esempio la prima n-upla della relazione nella Figura 2.3 stabilisce un legame fra i valori "Juventus," "Lazio," "3," "1", a indicare che il risultato della partita fra Juventus e Lazio è 3 a 1. Possiamo poi ricordare che una relazione è un *insieme*, quindi:

- non è definito alcun ordinamento fra le n-uple; nelle tabelle che le rappresentano c'è, per necessità, un ordine, ma è "occasionale": due tabelle con le stesse righe, ma in ordine diverso, rappresentano la stessa relazione;
- le n-uple di una relazione sono distinte l'una dall'altra, in quanto tra gli elementi di un insieme non ne possono essere presenti due uguali fra loro; quindi una tabella rappresenta una relazione solo se le sue righe sono l'una diversa dall'altra.

Al tempo stesso, ciascuna n-upla è, al proprio interno, *ordinata*: l'i-esimo valore di ciascuna proviene dall'i-esimo dominio. È cioè definito un ordinamento fra i domini, che è significativo ai fini dell'interpretazione dei dati nelle relazioni: se nella relazione nella Figura 2.3 scambiassimo il terzo dominio con il quarto, cambieremmo completamente il significato della nostra relazione, in quanto i risultati delle partite verrebbero invertiti. In effetti, questo accade perché nella relazione ciascuno dei due

Juventus	Lazio	3	1
Lazio	Milan	2	0
Juventus	Roma	1	2
Roma	Milan	0	1

Figura 2.3 Una relazione con i risultati di partite di calcio.

domini *intero* e *stringa* compare due volte, e le due occorrenze sono distinte attraverso la posizione: la prima occorrenza del dominio *stringa* fa riferimento alla squadra di casa e la seconda a quella ospitata; analogamente le due occorrenze del dominio *intero*.

L'ordinamento che abbiamo appena evidenziato fra i domini di una relazione corrisponde in effetti a una caratteristica insoddisfacente del concetto di relazione matematica rispetto alla possibilità di organizzare e utilizzare i dati. Infatti, in vari contesti dell'informatica si tende a privilegiare notazioni *non posizionali* (quali quelle che permettono, in un linguaggio ad alto livello, di far riferimento ai campi di un record per mezzo di nomi simbolici) rispetto a quelle posizionali (utilizzate per esempio negli array per indicare il primo elemento, il secondo e così via): si tende a utilizzare le notazioni posizionali solo quando l'ordinamento corrisponde a una esigenza intrinseca, come accade per esempio nei problemi di natura matematica, in cui gli array permettono di rappresentare vettori e matrici in modo ovvio e diretto. Risulta evidente come le informazioni che siamo interessati a organizzare nelle relazioni delle nostre basi di dati abbiano una struttura che si può naturalmente ricondurre a quella dei record: una relazione è sostanzialmente un insieme di record omogenei, cioè definiti sugli stessi campi. Nel caso dei record, a ogni campo è associato un nome: associamo a ciascuna occorrenza di dominio nella relazione un nome, detto *attributo*, che descrive il "ruolo" giocato dal dominio stesso. Per esempio, per la relazione relativa alle partite, possiamo usare nomi quali SquadraDiCasa, SquadraOspitata, RetiCasa, RetiOspitata; nella rappresentazione tabellare, utilizziamo gli attributi come intestazioni per le colonne (Figura 2.4); sottolineiamo che, dovendo identificare univocamente le componenti, gli attributi di una relazione (e quindi le intestazioni delle colonne delle tabelle) devono essere diversi l'uno dall'altro.

Modificando la definizione di relazione con l'introduzione degli attributi, e prima ancora di dare la definizione formale, possiamo vedere che l'ordinamento degli attributi (e delle colonne nella rappresentazione tabellare) risulta irrilevante: non è più necessario parlare di primo dominio, secondo dominio e così via, è sufficiente far riferimento agli attributi. La Figura 2.5 mostra un'altra rappresentazione tabellare della relazione nella Figura 2.4, con gli attributi (e quindi le colonne) in ordine diverso (secondo lo stile americano in cui la squadra di casa viene indicata dopo quella ospitata).

Per formalizzare i concetti, indichiamo con $\mathcal{D}$ l'insieme dei domini e specifichiamo la corrispondenza fra attributi e domini, nell'ambito di una relazione, per mezzo

SquadraDiCasa	SquadraOspitata	RetiCasa	RetiOspitata
Juventus	Lazio	3	1
Lazio	Milan	2	0
Juventus	Roma	1	2
Roma	Milan	0	1

Figura 2.4 Una relazione con attributi.

SquadraOspitata	SquadraDiCasa	RetiOspitata	RetiCasa
Lazio	Juventus	1	3
Milan	Lazio	0	2
Roma	Juventus	2	1
Milan	Roma	1	0

Figura 2.5 Un'altra rappresentazione per la relazione nella Figura 2.4.

di una funzione $dom : X \rightarrow \mathcal{D}$, che associa a ciascun attributo $A \in X$ un dominio $dom(A) \in \mathcal{D}$. Poi, diciamo che una *tupla*[2] su un insieme di attributi X è una funzione t che associa a ciascun attributo $A \in X$ un valore del dominio $dom(A)$. Possiamo quindi dare la nuova definizione di relazione: una *relazione* su X è un insieme di tuple su X. La differenza fra questa definizione e quella tradizionale di relazione matematica risiede solo nella definizione di tupla: nella relazione matematica abbiamo n-uple i cui elementi sono individuati per posizione, mentre nelle tuple della nuova definizione gli elementi sono individuati per mezzo degli attributi, cioè con una tecnica non posizionale.

Introduciamo una notazione che utilizzeremo molto in seguito. Se t è una tupla su X e $A \in X$, allora $t[A]$ (o $t.A$) indica il valore di t su A. Per esempio, se t è la prima tupla della relazione[3] nella Figura 2.5, possiamo dire che:

$$t[\text{SquadraOspitata}] = \text{Lazio}$$

La notazione si usa anche per insiemi di attributi, nel qual caso denota tuple:

$$t[\text{SquadraOspitata,RetiOspitata}]$$

è una tupla su due attributi.

2.1.4 Relazioni e basi di dati

Come già notato, una relazione può essere utilizzata per organizzare dati rilevanti nell'ambito di un'applicazione di interesse. Peraltro, di solito non è sufficiente allo scopo una singola relazione: una base di dati è in generale costituita da più relazioni, le cui tuple contengono valori comuni, ove necessario per stabilire corrispondenze. Approfondiamo questo concetto commentando la base di dati nella Figura 2.6:

- la prima relazione contiene informazioni relative a un insieme di studenti, con numero di matricola, cognome, nome e data di nascita;

[2]Traslitterazione dell'inglese *tuple*. In italiano sarebbe forse più corretto usare il termine *ennupla*, ma è ormai diffuso il termine *tupla*, che permette di sottolineare la differenza con l'usuale concetto di n-upla ordinata visto in precedenza.

[3]Più precisamente, dovremmo dire "la tupla rappresentata dalla prima riga della tabella...", ma sarebbe un inutile appesantimento.

STUDENTI

Matricola	Cognome	Nome	Data di nascita
276545	Rossi	Maria	25/11/1991
485745	Neri	Anna	23/04/1992
200768	Verdi	Fabio	12/02/1992
587614	Rossi	Luca	10/10/1991
937653	Bruni	Mario	01/12/1991

ESAMI

Studente	Voto	Corso
276545	28	01
276545	27	04
937653	25	01
200768	24	04

CORSI

Codice	Titolo	Docente
01	Analisi	Giani
03	Chimica	Melli
04	Chimica	Belli

Figura 2.6 Una base di dati relazionale.

- la terza relazione contiene informazioni su alcuni corsi, con codice, titolo e docente;
- la seconda relazione contiene informazioni relative a esami: il numero di matricola dello studente, il codice del corso e il voto; questa relazione fa riferimento ai dati contenuti nelle altre due: agli studenti, attraverso i numeri di matricola, e ai corsi, attraverso i relativi codici.

La base di dati nella Figura 2.6 mostra una delle caratteristiche fondamentali del modello relazionale, che viene spesso indicata dicendo che esso è "basato su valori": i riferimenti fra dati in relazioni diverse sono rappresentati per mezzo di valori dei domini che compaiono nelle tuple. Va notato che altri modelli logici, per esempio reticolare e gerarchico (ormai superati, ma molto utilizzati prima della diffusione del modello relazionale), realizzano le corrispondenze in modo esplicito attraverso puntatori e vengono pertanto detti modelli "basati su record e puntatori". In questo testo non presentiamo in modo articolato tali modelli, ma vogliamo qui brevemente evidenziare la caratteristica fondamentale di un modello ideale basato su record e puntatori. La Figura 2.7 rappresenta la stessa base di dati nella Figura 2.6 con puntatori al posto dei riferimenti realizzati tramite valori (i numeri di matricola degli studenti e i codici dei corsi).

Rispetto a un modello basato su record e puntatori, il modello relazionale, basato su valori, presenta diversi vantaggi:

- esso richiede di rappresentare solo ciò che è rilevante dal punto di vista dell'applicazione (e quindi dell'utente); i puntatori sono qualcosa di aggiuntivo, legato ad aspetti realizzativi; nei modelli con puntatori, il programmatore delle applicazioni fa riferimento a dati che non sono significativi per l'applicazione;

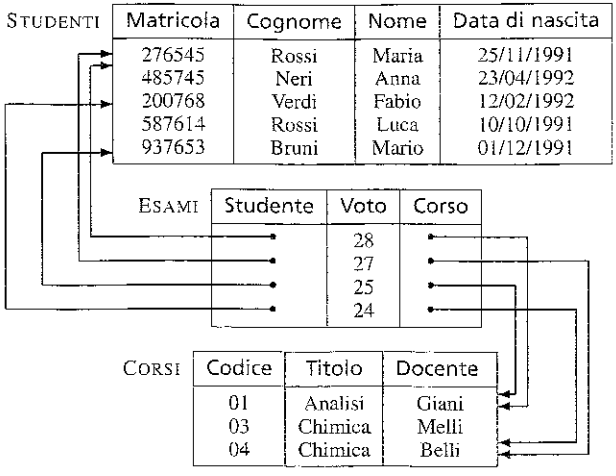

Figura 2.7 Una base di dati con puntatori.

- la rappresentazione logica dei dati (costituita dai soli valori) non fa alcun riferimento a quella fisica, che può anche cambiare nel tempo: il modello relazionale permette quindi di ottenere l'indipendenza fisica dei dati;
- essendo tutta l'informazione contenuta nei valori, è relativamente semplice trasferire i dati da un contesto a un altro (per esempio se si deve trasferire una base di dati da un calcolatore a un altro); in presenza di puntatori, l'operazione è più complessa, perché i puntatori hanno un significato locale al singolo sistema, che non sempre è immediato esportare.

A titolo di inciso, vale la pena notare che anche in una base di dati relazionale, a livello fisico, i dati possono essere rappresentati secondo modalità che prevedono l'uso di puntatori. La differenza, rispetto ai modelli basati su puntatore, è nel fatto che qui i puntatori non sono visibili a livello logico. Inoltre, sottolineiamo come nei sistemi di basi di dati a oggetti, che rappresentano una delle direzioni di evoluzione delle basi di dati (come discusso nel secondo volume [6]), vengano introdotti gli identificatori di oggetto, che, pur a un livello di astrazione più alto, presentano alcune delle caratteristiche dei puntatori.

Possiamo a questo punto riassumere le definizioni relative al modello relazionale, con un po' di precisione, distinguendo il livello degli schemi da quello delle istanze.

- Uno *schema di relazione* è costituito da un simbolo R, detto *nome della relazione*, e da un insieme di (nomi di) *attributi* $X = \{A_1, A_2, \ldots, A_n\}$, il tutto di solito

indicato con $R(X)$. A ciascun attributo è associato un dominio, come visto in precedenza.

- Uno *schema di base di dati* è un insieme di schemi di relazione con nomi diversi:

$$\mathbf{R} = \{R_1(X_1), R_2(X_2), \ldots, R_n(X_n)\}$$

  I nomi di relazione hanno come scopo principale quello di distinguere le varie relazioni nella base di dati.
- Una *istanza di relazione* (o semplicemente *relazione*) su uno schema $R(X)$ è un insieme r di tuple su X. Talvolta si usa la notazione $r(X)$ per indicare una relazione sull'insieme di attributi X, descrivendo così al tempo stesso lo schema e l'istanza.
- Una *istanza di base di dati* (o semplicemente *base di dati*) su uno schema $\mathbf{R} = \{R_1(X_1), R_2(X_2), \ldots, R_n(X_n)\}$ è un insieme di relazioni $\mathbf{r} = \{r_1, r_2, \ldots, r_n\}$, dove ogni r_i, per $1 \leq i \leq n$, è una relazione sullo schema $R_i(X_i)$.

Per esemplificare, possiamo dire che lo schema della base di dati nella Figura 2.6 è così definito (con opportune definizioni per i domini):

$\mathbf{R}$ = {STUDENTI (Matricola, Cognome, Nome, Data di nascita),
ESAMI (Studente, Voto, Corso),
CORSI (Codice, Titolo, Docente)}

Per comodità, accenniamo brevemente alle convenzioni che adotteremo nel seguito (e che abbiamo già usato nelle definizioni e negli esempi), allo scopo di favorire la sinteticità della notazione senza compromettere la comprensione:

- gli attributi (quando non si utilizzeranno nomi significativi dal punto di vista dell'applicazione) verranno indicati con lettere iniziali dell'alfabeto, maiuscole, eventualmente con indici e/o pedici: A, B, C, A', $A_1, \ldots$
- insiemi di attributi verranno indicati con lettere finali dell'alfabeto, maiuscole: X, Y, Z, X', $X_1, \ldots$; un insieme in cui si vogliano evidenziare gli attributi componenti verrà denotato dalla giustapposizione dei nomi degli attributi stessi: scriveremo cioè $X = ABC$ anziché $X = \{A, B, C\}$; analogamente, l'unione di insiemi verrà denotata dalla giustapposizione dei relativi nomi: scriveremo XY anziché $X \cup Y$; riunendo le due convenzioni, scriveremo XA anziché $X \cup \{A\}$;
- per i nomi di relazione (sempre nel caso in cui si possa o debba fare a meno di nomi significativi) utilizzeremo la R e le lettere contigue, maiuscole: R_1, S, $S', \ldots$; per le relazioni, gli stessi simboli dei corrispondenti nomi di relazione, ma in lettere minuscole.

Per approfondire ancora i concetti fondamentali del modello relazionale, discutiamo un altro paio di esempi.

In primo luogo notiamo come, secondo la definizione, siano ammissibili relazioni su un solo attributo. Ciò può avere senso in particolare in basi di dati su più relazioni, in cui la relazione su singolo attributo contiene valori che appaiono come valori di un attributo di un'altra relazione. Per esempio, in una base di dati in cui sia presente la relazione STUDENTI della Figura 2.6, si può utilizzare un'altra relazione sul solo attributo

STUDENTI

Matr.	Cognome	Nome	Data di nascita
276545	Rossi	Maria	25/11/1991
485745	Neri	Anna	23/04/1992
200768	Verdi	Fabio	12/02/1992
587614	Rossi	Luca	10/10/1991
937653	Bruni	Mario	01/12/1991

LAVORATORI

Matr.
276545
485745
937653

Figura 2.8 Una relazione su un solo attributo.

Matricola per indicare gli studenti lavoratori (attraverso i relativi numeri di matricola, che devono apparire nella relazione STUDENTI, come si può notare nella Figura 2.8.

Discutiamo ora un esempio un po' più complesso, che mostra come, sia pur indirettamente, il modello relazionale permetta di rappresentare informazione strutturata in modo articolato. Nella Figura 2.9 sono schematizzate tre ricevute fiscali emesse da un ristorante. Esse hanno una struttura che prevede (a parte le frasi prestampate che abbiamo riportato in grassetto) alcune informazioni fisse (numero, data e totale) e un numero di righe variabile, ognuna relativa a un insieme di portate omogenee (con quantità, descrizione e importo complessivo). Poiché le nostre relazioni hanno una struttura fissa, non è possibile rappresentare l'insieme delle ricevute con un'unica relazione, in quanto le ricevute non hanno un numero di righe prefissato.

Possiamo però rappresentare le relative informazioni per mezzo di due relazioni, come mostrato nella Figura 2.10: la relazione RICEVUTE contiene i dati presenti una sola volta in ciascuna ricevuta (numero, data e totale) e la relazione DETTAGLIO contiene le varie righe di ciascuna ricevuta (con quantità, descrizione e importo

"DA MARIO"		
Ricevuta n. 1357		
del 5/5/2012		
3	coperti	6,00
2	antipasti	12,00
3	primi	27,00
2	bistecche	36,00
	Totale	81,00

"DA MARIO"		
Ricevuta n. 2334		
del 4/7/2012		
2	coperti	4,00
1	antipasti	6,00
2	primi	15,00
2	orate	50,00
2	caffè	3,00
	Totale	78,00

"DA MARIO"		
Ricevuta n. 3002		
del 4/8/2012		
3	coperti	6,00
2	antipasti	14,00
3	primi	20,00
1	orate	25,00
1	capresc	8,00
2	caffè	3,00
	Totale	76,00

Figura 2.9 Alcune ricevute fiscali.

RICEVUTE

Num	Data	Totale
1357	5/5/2012	81,00
2334	4/7/2012	78,00
3002	4/8/2012	76,00

DETTAGLIO

Num	Q.tà	Descr	Costo
1357	3	coperti	6,00
1357	2	antipasti	12,00
1357	3	primi	27,00
1357	2	bistecche	36,00
2334	2	coperti	4,00
2334	1	antipasti	6,00
2334	2	primi	15,00
2334	2	orate	50,00
2334	2	caffé	3,00
3002	3	coperti	6,00
3002	2	antipasti	14,00
3002	3	primi	20,00
3002	1	orate	25,00
3002	1	caprese	8,00
3002	2	caffé	3,00

Figura 2.10 Una base di dati per le ricevute fiscali nella Figura 2.9.

complessivo), associate alla ricevuta stessa tramite il relativo numero.

È opportuno notare che la base di dati nella Figura 2.10 rappresenta correttamente le ricevute solo se sono vere le due condizioni seguenti:

- non interessa mantenere traccia dell'ordine con cui le righe compaiono in ciascuna ricevuta: infatti, poiché nessun ordinamento è definito fra le tuple di una relazione, le tuple di DETTAGLIO non sono in alcun modo ordinate;
- in una ricevuta non compaiono due righe uguali (il che potrebbe accadere in presenza di ordinazioni diverse relative alle stesse pietanze con le stesse quantità).

Altrimenti, si può risolvere il problema aggiungendo un attributo, che indica la posizione della riga sulla ricevuta (Figura 2.11): in questo modo è sempre possibile ricostruire perfettamente il contenuto di tutte le ricevute. In generale, possiamo dire che la soluzione di Figura 2.10 è da preferirsi quando le informazioni sulla ricevuta interessano solo in quanto tali (e nelle ricevute non vi sono righe ripetute), mentre quella di Figura 2.11 permette di tenere traccia dell'effettiva impaginazione di ciascuna ricevuta. L'esempio ci permette quindi di notare che, anche in una stessa situazione, i dati da rappresentare nella base di dati possono essere diversi a seconda degli specifici obiettivi che ci si prefigge.

2.1.5 Informazione incompleta e valori nulli

La struttura del modello relazionale, come discussa nei paragrafi precedenti, è indubbiamente molto semplice e potente. Al tempo stesso, essa impone però un certo grado

RICEVUTE

Num	Data	Totale
1357	5/5/2012	81,00
2334	4/7/2012	78,00
3002	4/8/2012	76,00

DETTAGLIO

Num	Riga	Q.tà	Descr	Costo
1357	1	3	coperti	6,00
1357	2	2	antipasti	12,00
1357	3	3	primi	27,00
1357	4	2	bistecche	36,00
2334	1	2	coperti	4,00
2334	2	1	antipasti	6,00
2334	3	2	primi	15,00
2334	4	2	orate	50,00
2334	5	2	caffè	3,00
3002	1	3	coperti	6,00
3002	2	2	antipasti	14,00
3002	3	3	primi	20,00
3002	4	1	orate	25,00
3002	5	1	caprese	8,00
3002	6	2	caffè	3,00

Figura 2.11 Un'altra base di dati per le ricevute fiscali.

di rigidità, in quanto le informazioni devono essere rappresentate per mezzo di tuple di dati omogenee: in particolare, in ogni relazione possiamo rappresentare solo tuple corrispondenti allo schema della relazione stessa. In effetti, in molti casi, i dati disponibili possono non corrispondere esattamente al formato previsto. Per esempio, in una relazione sullo schema:

PERSONE(Cognome,Nome,Indirizzo,Telefono)

il valore dell'attributo Telefono potrebbe non essere disponibile per tutte le tuple. Vale la pena notare che non sarebbe corretto utilizzare un valore del dominio per rappresentare l'assenza di informazione, in quanto si potrebbe in tal modo generare confusione. In questo caso, supponendo i numeri telefonici rappresentati per mezzo di interi, potremmo per esempio utilizzare lo zero per indicare l'assenza di un valore significativo. In generale, però, questa scelta non risulta soddisfacente, per due motivi. In primo luogo, essa richiede l'esistenza di un valore del dominio mai utilizzato per valori significativi: nel caso dei numeri di telefono, lo zero è chiaramente distinguibile, ma in altri casi non esiste un valore disponibile allo scopo; per esempio, in un attributo che rappresenti la data di nascita e che utilizzi come dominio un tipo Data correttamente

definito, non esistono elementi "non utilizzati" per valori significativi e quindi utilizzabili per denotare assenza di informazione. In secondo luogo, l'uso di valori del dominio può generare confusione: la distinzione fra valori "veri" e valori fittizi è nascosta, e quindi i programmi che accedono alla base di dati devono tenerne conto, distinguendo opportunamente (e tenendo conto di quali sono, in ciascun caso, i valori fittizi).

Per rappresentare in modo semplice, ma al tempo stesso comodo, la non disponibilità di valori, il concetto di relazione viene di solito esteso prevedendo che una tupla possa assumere, su ciascun attributo, o un valore del dominio, come visto finora, oppure un valore speciale, detto *valore nullo*, che denota appunto l'assenza di informazione, ma è un valore aggiuntivo rispetto a quelli del dominio, e ben distinto da essi. Nelle rappresentazioni tabellari, utilizzeremo per il valore nullo il simbolo NULL, come nella Figura 2.12. Con riferimento alla tabella in figura, possiamo notare come in effetti i tre valori nulli che compaiono in essa siano dovuti a motivazioni diverse, come segue.

- Firenze è capoluogo di provincia e quindi ha certamente una prefettura. Al momento, non disponiamo del suo indirizzo. Il valore nullo sostituisce un valore ordinario, non noto alla base di dati: per questo diciamo che si tratta di un valore *sconosciuto*.
- Tivoli non è capoluogo di provincia e perciò non ha una prefettura. Quindi l'attributo IndirizzoPrefettura non può avere un valore per questa tupla. Il valore nullo denota l'inapplicabilità dell'attributo o l'inesistenza del valore: il valore è *inesistente*.
- La provincia di Olbia-Tempio, nel momento in cui scriviamo, è stata istituita da alcuni anni (e probabilmente sarà presto abolita) e non sappiamo se la prefettura sia stata costituita, né conosciamo il suo indirizzo (già operativo o previsto). In sostanza, non sappiamo se il valore esista e, in caso affermativo, non lo conosciamo. Sostanzialmente, ci troviamo in una situazione che corrisponde alla disgiunzione logica (l'"or") delle due precedenti: il valore è inesistente oppure sconosciuto. Questo tipo di valore nullo viene di solito chiamato *senza informazione*, perché non ci dice assolutamente niente: il valore può esistere o non esistere, e se esiste non sappiamo quale sia.

Nei sistemi di basi di dati relazionali, è di solito prevista una gestione molto semplice, ma tutto sommato efficace, del valore nullo, sul quale non viene fatta alcuna ipotesi e quindi in pratica ci si trova nell'ultimo caso, quello del valore senza informazione.

Città	IndirizzoPrefettura
Roma	Via Quattro Novembre
Firenze	NULL
Tivoli	NULL
Olbia-Tempio	NULL

Figura 2.12 Una relazione con valori nulli.

STUDENTI

Matricola	Cognome	Nome	Data di nascita
276545	Rossi	Maria	NULL
NULL	Neri	Anna	23/04/1992
NULL	Verdi	Fabio	12/02/1992

ESAMI

Studente	Voto	Corso
276545	28	01
NULL	27	NULL
200768	24	NULL

CORSI

Codice	Titolo	Docente
01	Analisi	Giani
03	Chimica	NULL
NULL	Chimica	Belli

Figura 2.13 Una base di dati con molti valori nulli.

Per un'ulteriore riflessione sui valori nulli, consideriamo la base di dati nella Figura 2.13, che è definita sullo stesso schema della base di dati di Figura 2.6.

Il valore nullo sulla data di nascita nella prima tupla della relazione STUDENTI è tutto sommato ammissibile, perché si può pensare che l'informazione non sia in questo contesto essenziale. Viceversa, un valore nullo sul numero di matricola o sul codice di un corso genera problemi maggiori, in quanto questi valori, come abbiamo discusso con riferimento alla Figura 2.6, sono utilizzati per stabilire correlazioni fra tuple di relazioni diverse. Poi, la presenza di valori nulli nella relazione ESAMI rende addirittura inutilizzabili le informazioni: per esempio, la seconda tupla, con il solo voto e due valori nulli, non fornisce alcuna informazione utile. Infine, la presenza di molteplici valori nulli in una relazione può addirittura generare dubbi sull'effettiva significatività e identità delle tuple: le ultime due tuple della relazione CORSI possono essere diverse o addirittura coincidere! È evidente quindi come sia necessario moderare opportunamente la presenza dei valori nulli nelle nostre relazioni: solo alcune configurazioni devono essere ammesse. In genere, quando si definisce una relazione, è possibile specificare che i nulli sono ammessi solo su alcuni attributi e non su altri. Alla fine del Paragrafo 2.2 vedremo quale possa essere un criterio fondamentale per individuare alcuni attributi su cui è essenziale evitare la presenza di valori nulli.

2.2 Vincoli di integrità

Le strutture del modello relazionale ci permettono di organizzare le informazioni di interesse per le nostre applicazioni. In molti casi, però, non è vero che qualsiasi insieme di tuple sullo schema rappresenti informazioni corrette per l'applicazione. Abbiamo già discusso in breve il problema relativamente alla presenza di valori nulli. Ora, approfondiamo il problema con riferimento anche a relazioni prive di valori nulli. Consideriamo per esempio la base di dati nella Figura 2.14 e notiamo in essa varie situazioni che in pratica non si dovrebbero presentare.

STUDENTI

Matricola	Cognome	Nome	Data di nascita
200768	Verdi	Fabio	12/02/1992
937653	Rossi	Luca	10/10/1991
937653	Bruni	Mario	01/12/1991

ESAMI

Studente	Voto	Lode	Corso
200768	36		05
937653	28	lode	01
937653	30	lode	04
276545	25		01

CORSI

Codice	Titolo	Docente
01	Analisi	Giani
03	Chimica	Melli
04	Chimica	Belli

Figura 2.14 Una base di dati con informazioni scorrette.

- Nella prima tupla della relazione ESAMI abbiamo un voto pari a 36 che, nel sistema italiano, non è ammissibile, in quanto i voti devono essere compresi fra 0 e 30 (o, con riferimento al superamento dell'esame, fra 18 e 30).
- Nella seconda tupla ancora della relazione ESAMI viene indicato che è stata attribuita la lode in un esame in cui il voto è 28, il che è impossibile: la lode può essere attribuita solo se il voto è 30.
- Le ultime due tuple della relazione STUDENTI contengono informazioni su due studenti diversi con lo stesso numero di matricola: ancora una situazione impossibile, in quanto il numero di matricola serve appunto a identificare univocamente gli studenti.
- La quarta tupla della relazione ESAMI presenta, per l'attributo Studente, un valore che non compare fra i numeri di matricola nella relazione STUDENTI: anche questa è una situazione indesiderabile, poiché i numeri di matricola ci forniscono informazioni solo come tramite verso le corrispondenti tuple della relazione STUDENTI. Analogamente, la prima tupla presenta un codice di corso che non compare nella relazione CORSI.

In una base di dati, è opportuno evitare situazioni come quelle appena descritte. Allo scopo, è stato introdotto il concetto di *vincolo di integrità*, come proprietà che deve essere soddisfatta dalle istanze che rappresentano informazioni corrette per l'applicazione. Ogni vincolo può essere visto come un *predicato* che associa a ogni istanza il valore *vero* o *falso*. Se il predicato assume il valore *vero*, allora diciamo che l'istanza

soddisfa il vincolo. In generale, a uno schema di base di dati associamo un insieme di vincoli e consideriamo *corrette* (o *lecite*, o *ammissibili*) le istanze che soddisfano tutti i vincoli. In ciascuno dei quattro casi sopra discussi potrebbe essere introdotto un vincolo che vieti la situazione indesiderata.

È possibile classificare i vincoli a seconda degli elementi di una base di dati che ne sono coinvolti. Distinguiamo due categorie, la prima delle quali ha alcuni casi particolari.

- Un vincolo è *intrarelazionale* se il suo soddisfacimento è definito rispetto a singole relazioni della base di dati; i primi tre casi sopra discussi corrispondono a vincoli intrarelazionali; talvolta, il coinvolgimento riguarda le tuple (o addirittura i valori) separatamente le une dalle altre:
  - un *vincolo di tupla* è un vincolo che può essere valutato su ciascuna tupla indipendentemente dalle altre: i vincoli relativi ai primi due casi rientrano in questa categoria;
  - come caso ancora più specifico, un vincolo definito con riferimento a singoli valori (è il caso del primo esempio, in cui sono ammessi solo valori dell'attributo Voto compresi fra 18 e 30) viene detto *vincolo su valori* o *vincolo di dominio*, in quanto impone una restrizione sul dominio dell'attributo.
- Un vincolo è *interrelazionale* se coinvolge più relazioni; è questo il caso del quarto esempio, in cui la situazione indesiderata può essere vietata richiedendo che un numero di matricola compaia nella relazione ESAMI solo se compare nella relazione STUDENTI.

Nei paragrafi seguenti esaminiamo con un certo dettaglio tre classi di vincoli molto importanti:

- una classe interessante di vincoli di tupla;
- i vincoli di chiave, che sono i più importanti vincoli intrarelazionali;
- i vincoli di integrità referenziale, che sono i vincoli interrelazionali di maggiore interesse.

2.2.1 Vincoli di tupla

Come abbiamo detto, i vincoli di tupla esprimono condizioni sui valori di ciascuna tupla, indipendentemente dalle altre tuple.

Una possibile sintassi per esprimere vincoli di questo tipo è quella che permette di definire espressioni booleane (cioè con connettivi AND, OR e NOT) con atomi che confrontano (con gli operatori di uguaglianza, disuguaglianza e ordinamento) valori di attributo o espressioni aritmetiche su valori di attributo. I vincoli violati individuati nei primi due esempi della lista del Paragrafo 2.2 potrebbero essere descritti con le seguenti espressioni:

$$(\text{Voto} \geq 18) \text{ AND } (\text{Voto} \leq 30)$$

$$(\text{NOT } (\text{Lode} = \text{'lode'})) \text{ OR } (\text{Voto} = 30)$$

In particolare, il secondo vincolo indica che è ammissibile la lode solo se il voto è pari a 30 (dicendo che c'è o non c'è la lode, oppure il voto è pari a 30). Il primo vincolo è in effetti un vincolo di dominio in quanto coinvolge un solo attributo.

La definizione che abbiamo dato ammette anche espressioni più complesse, purché definite sui valori delle singole tuple. Per esempio, su una relazione sullo schema:

PAGAMENTI(Data,Importo,Ritenute,Netto)

è possibile definire il vincolo che impone, come naturale, che il netto sia pari alla differenza fra l'importo originario e le ritenute, nel modo seguente:

Netto = Importo − Ritenute

2.2.2 Chiavi

In questo paragrafo discutiamo i vincoli di chiave, che sono senz'altro i più importanti del modello relazionale; potremmo addirittura affermare che senza di essi il modello stesso non avrebbe senso. Cominciamo con un esempio. Nella relazione della Figura 2.15 possiamo notare che i valori delle varie tuple sull'attributo Matricola sono tutti diversi l'uno dall'altro: come già notato più volte, ribadiamo che il valore della matricola *identifica univocamente* gli studenti; a suo tempo il concetto stesso di numero di matricola fu introdotto nelle università proprio per avere uno strumento semplice ed efficace per far riferimento agli studenti in modo non ambiguo. Analogamente, possiamo notare che nella relazione non vi sono coppie di tuple con gli stessi valori su ciascuno dei tre attributi Cognome, Nome e Nascita: anche i dati anagrafici identificano univocamente le persone.[4] Anche altri insiemi di attributi identificano univocamente le tuple della relazione nella Figura 2.15: per esempio, Matricola e Corso, come è in effetti ovvio, visto che Matricola è da solo sufficiente.

Intuitivamente, una chiave è un insieme di attributi utilizzato per identificare univocamente le tuple di una relazione. Per formalizzare la definizione, procediamo in due passi:

Matricola	Cognome	Nome	Nascita	Corso
4328	Rossi	Luigi	29/04/1994	Ing. Informatica
6328	Rossi	Dario	29/04/1994	Ing. Informatica
4766	Rossi	Luca	01/05/1995	Ing. Civile
4856	Neri	Luca	01/05/1995	Ing. Meccanica
5536	Neri	Luca	05/03/1993	Ing. Meccanica

Figura 2.15 Una relazione per la discussione delle chiavi.

[4]Per non appesantire l'esempio, assumiamo qui che cognome, nome e data di nascita siano sufficienti a identificare univocamente le persone, il che in generale non è vero.

- un insieme K di attributi è *superchiave* di una relazione r se r non contiene due tuple distinte t_1 e t_2 con $t_1[K] = t_2[K]$;
- K è *chiave* di r se è una superchiave minimale di r (cioè non esiste un'altra superchiave K' di r che sia contenuta in K come sottoinsieme proprio).

Nell'esempio:

- l'insieme {Matricola} è superchiave; è anche una superchiave minimale, in quanto contiene un solo attributo (l'insieme vuoto non è in grado di identificare tuple), quindi {Matricola} è una chiave;
- l'insieme {Cognome, Nome, Nascita} è superchiave; inoltre, nessuno dei suoi sottoinsiemi è superchiave: infatti esistono due tuple (la prima e la seconda) uguali su Cognome e Nascita, due (le ultime) uguali su Cognome e Nome e due (la terza e la quarta) uguali su Nome e Nascita; quindi {Cognome, Nome, Nascita} è un'altra chiave;
- l'insieme {Matricola, Corso} è superchiave, come già discusso, ma non è una superchiave minimale, perché esiste un suo sottoinsieme proprio, {Matricola}, esso stesso superchiave minimale e quindi {Matricola, Corso} non è una chiave;
- l'insieme {Nome, Corso} non è superchiave, perché nella relazione compaiono due tuple, le ultime due, fra loro uguali sia su Nome sia su Corso.

Per approfondire la discussione, esaminiamo un'altra relazione, quella nella Figura 2.16. Essa non contiene tuple fra loro uguali sia su Cognome sia su Corso. Quindi, per questa relazione, l'insieme {Cognome, Corso} è superchiave. Poiché vi sono tuple uguali su Cognome (le prime due) e su Corso (la seconda e la quarta), tale insieme è superchiave minimale e cioè chiave. Ora, in questa relazione, Cognome e Corso identificano univocamente le tuple; ma possiamo dire che questo è vero in generale? Certamente no, in quanto possono benissimo esistere studenti con uguale cognome iscritti allo stesso corso di studio.

In un certo senso, possiamo quindi dire che {Cognome, Corso} è "casualmente" una chiave per la relazione nella Figura 2.16, mentre a noi interessano le chiavi corrispondenti a vincoli di integrità, soddisfatti da tutte le relazioni lecite su un certo schema. In effetti, definendo uno schema, noi associamo a esso i vincoli di interesse, corrispondenti a proprietà del mondo reale le cui informazioni vengono rappresentate per mezzo della nostra base di dati. I vincoli sono cioè definiti a livello di schema, con riferimento a tutte le istanze: consideriamo corrette solo le istanze che soddisfano

Matricola	Cognome	Nome	Nascita	Corso
6328	Rossi	Dario	29/04/1994	Ing. Informatica
4766	Rossi	Luca	01/05/1995	Ing. Civile
4856	Neri	Luca	01/05/1995	Ing. Meccanica
5536	Neri	Luca	05/03/1993	Ing. Civile

Figura 2.16 Un'altra relazione per la discussione delle chiavi.

tutti i vincoli. Un'istanza corretta può poi soddisfare altri vincoli oltre a quelli definiti sullo schema. Per esempio, a uno schema:

STUDENTI(Matricola,Cognome,Nome,Nascita,Corso)

vanno associati i vincoli che impongono come chiavi i due insiemi di attributi sopra discussi:

{Matricola}
{Cognome, Nome, Nascita}

Entrambe le relazioni nelle Figure 2.15 e 2.16 soddisfano tutti e due i vincoli; la seconda soddisfa anche ("per caso," come abbiamo detto) il vincolo secondo cui {Cognome,Corso} è un'altra chiave.

Possiamo ora fare alcune riflessioni sul concetto di chiave, che giustificano l'importanza a esso attribuita. In primo luogo, possiamo notare come ciascuna relazione e ciascuno schema di relazione abbiano sempre una chiave. Una relazione è un insieme e quindi, come più volte ribadito, è costituita di elementi fra loro diversi; di conseguenza, per ogni relazione $r(X)$, l'insieme X di tutti gli attributi su cui è definita è senz'altro una superchiave per essa. Ora, i casi sono due: o tale insieme è anche chiave, nel qual caso confermiamo l'esistenza della chiave stessa, oppure non è chiave, perché esiste un'altra superchiave in esso contenuta: allora possiamo procedere ricorsivamente, ripetendo il ragionamento su quest'ultimo insieme, e così via; poiché l'insieme di attributi su cui è definita una relazione è finito, il processo termina in un numero finito di passi con una superchiave minimale. Quindi, possiamo affermare con certezza che ogni relazione ha una chiave. Lo stesso ragionamento può essere svolto a livello di schema di relazione: l'insieme di tutti gli attributi è superchiave per ciascuna relazione, quindi lo è per ciascuna relazione lecita: la ricerca di superchiavi minimali procede poi come sopra.

Il fatto che su ciascuno schema di relazione possa essere definita almeno una chiave garantisce l'accessibilità a tutti i valori di una base di dati e la loro univoca identificabilità. Inoltre, permette di stabilire efficacemente quelle corrispondenze fra dati contenuti in relazioni diverse che caratterizzano il modello relazionale come "modello basato su valori." Riconsideriamo l'esempio nella Figura 2.6. Nella relazione ESAMI, si fa riferimento agli studenti attraverso i numeri di matricola e ai corsi attraverso i relativi codici: in effetti, Matricola è la chiave della relazione STUDENTI e Codice è la chiave della relazione CORSI. I valori attraverso cui vengono stabilite le corrispondenze fra tuple di relazioni diverse sono valori delle chiavi delle relazioni cui si fa riferimento dall'esterno.

2.2.3 Chiavi e valori nulli

Possiamo ora riprendere il discorso avviato alla fine del Paragrafo 2.1.5 sulla necessità di evitare la proliferazione di valori nulli nelle nostre relazioni. In particolare, notiamo che, in presenza di valori nulli, non è più vero che, come abbiamo invece affermato più volte, i valori delle chiavi permettono di identificare univocamente le tuple delle relazioni e di stabilire riferimenti fra tuple di relazioni diverse. Esaminando allo scopo la relazione nella Figura 2.17, definita sullo stesso schema della relazione

Matricola	Cognome	Nome	Nascita	Corso
NULL	Rossi	Mario	NULL	Ing. Informatica
4766	Rossi	Luca	01/05/1995	Ing. Civile
4856	Neri	Luca	NULL	NULL
NULL	Neri	Luca	05/03/1993	Ing. Civile

Figura 2.17 Una relazione con valori nulli su tutte le chiavi.

nella Figura 2.16 (quindi con due chiavi, una composta dal solo attributo Matricola e l'altra dagli attributi Cognome, Nome e Nascita), notiamo problemi di due tipi. La prima tupla ha valori nulli su Matricola e Nascita e perciò su almeno un attributo di ciascuna chiave: questa tupla non è identificabile in alcun modo; se vogliamo inserire nella base di dati un'altra tupla relativa a uno studente di nome Mario Rossi, non possiamo sapere se ci stiamo in effetti riferendo allo stesso studente oppure a un altro. Inoltre, non è possibile, in altre relazioni della base di dati, fare riferimento a questa tupla, visto che ciò andrebbe fatto attraverso il valore di una chiave. Anche le ultime due tuple nella figura presentano un problema: nonostante ciascuna abbia una chiave completamente specificata (grazie al valore su Matricola nella terza tupla e ai valori su Cognome, Nome e Nascita nell'ultima), la presenza di valori nulli rende impossibile capire se le due tuple facciano riferimento allo stesso studente Luca Neri oppure a due studenti omonimi.

L'esempio ci suggerisce quindi la necessità di porre dei limiti alla presenza di valori nulli nelle chiavi delle relazioni. In pratica, si adotta una soluzione semplice, che permette di garantire l'identificazione univoca di tutte le tuple e la possibilità di far riferimento a esse da parte di altre relazioni: su una delle chiavi (detta la *chiave primaria*) si vieta la presenza di valori nulli; sulle altre, i valori nulli sono in genere (salvo necessità specifiche) ammessi. Gli attributi che costituiscono la chiave primaria vengono spesso evidenziati attraverso la sottolineatura, come mostrato nella Figura 2.18 con la sottolineatura dell'attributo Matricola. La maggior parte dei riferimenti tra relazioni vengono realizzati attraverso i valori della chiave primaria.

È opportuno notare che in quasi tutti i casi reali è possibile trovare attributi i cui valori sono identificanti e sempre disponibili. Quando ciò non accada, è necessario introdurre un attributo aggiuntivo, un codice, probabilmente non significativo dal punto

Matricola	Cognome	Nome	Nascita	Corso
3976	Rossi	Mario	NULL	Ing. Informatica
4766	Rossi	Luca	01/05/1995	Ing. Civile
4856	Neri	Luca	NULL	NULL
5591	Neri	Luca	05/03/1993	Ing. Civile

Figura 2.18 Una relazione con chiave primaria.

di vista dell'applicazione, che viene in qualche modo generato e attribuito a ciascuna tupla all'atto dell'inserimento. Tra l'altro, si può dire che molti codici identificativi (quali per esempio il numero di matricola, il codice fiscale, il numero di targa) siano stati introdotti in passato, prima dell'invenzione o della diffusione delle basi di dati, proprio per garantire l'identificazione univoca dei soggetti di un dominio (rispettivamente gli studenti, i contribuenti, le automobili) e per favorire il riferimento a essi: esattamente gli obiettivi delle chiavi.

2.2.4 Vincoli di integrità referenziale

Per discutere la più importante classe di vincoli interrelazionali, consideriamo la base di dati in Figura 2.19. In essa, la prima relazione contiene informazioni relative a un insieme di infrazioni al codice della strada, la seconda agli agenti di polizia che le hanno rilevate e la terza a un insieme di autoveicoli.[5] Le informazioni della relazione INFRAZIONI sono rese significative e complete attraverso il riferimento alle altre due relazioni: alla relazione AGENTI, per il tramite dell'attributo Agente, che contiene numeri di matricola di agenti corrispondenti alla chiave primaria della relazione AGENTI, e alla relazione AUTO, per mezzo degli attributi Prov e Numero, che contengono valori degli omonimi attributi che formano la chiave primaria della relazione AUTO. I riferimenti sono significativi in quanto i valori nella relazione INFRAZIONI sono uguali a valori effettivamente presenti nelle altre due: se un valore di Agente in INFRAZIONI non compare come valore della chiave di AGENTI, allora il riferimento non è efficace. Nell'esempio, tutti i riferimenti sono in effetti utilizzabili.

Un *vincolo di integrità referenziale* (chiamato nella letteratura in inglese *foreign key* o *referential integrity constraint*) fra un insieme di attributi X di una relazione R_1 e un'altra relazione R_2 è soddisfatto se i valori su X di ciascuna tupla dell'istanza di R_1 compaiono come valori della chiave (primaria) dell'istanza di R_2. La definizione precisa richiede un po' di attenzione, in particolare nel caso in cui la chiave della relazione riferita sia composta di più attributi e nel caso in cui vi siano più chiavi. Procediamo gradualmente, vedendo prima il caso in cui la chiave di R_2 è unica e composta di un solo attributo B (e quindi l'insieme X è a sua volta costituito da un solo attributo A): allora, il vincolo di integrità referenziale fra l'attributo A di R_1 e la relazione R_2 è soddisfatto se, per ogni tupla t_1 in R_1 per cui $t_1[A]$ non è nullo, esiste una tupla t_2 in R_2 tale che $t_1[A] = t_2[B]$. Nel caso più generale, dobbiamo fare attenzione al fatto che ciascuno degli attributi in X deve corrispondere a un preciso attributo della chiave primaria K di R_2. Allo scopo, è necessario specificare un ordinamento sia nell'insieme X sia in K. Indicando gli attributi in ordine, $X = A_1A_2 \ldots A_p$ e $K = B_1B_2 \ldots B_p$, il vincolo è soddisfatto se per ogni tupla t_1 in R_1 senza nulli su X esiste una tupla t_2 in R_2 con $t_1[A_i] = t_2[B_i]$, per ogni i compreso fra 1 e p.

Sullo schema della base di dati nella Figura 2.19 ha senso definire i vincoli di integrità referenziale:

[5] Utilizziamo la vecchia struttura delle targhe automobilistiche, utilizzata fino al 1994, con la sigla della provincia, perché ci permette di svolgere considerazioni interessanti sulle chiavi composte da più attributi.

INFRAZIONI

Codice	Data	Agente	Articolo	Prov	Numero
143256	25/10/2012	567	44	RM	4E5432
987554	26/10/2012	456	34	RM	4E5432
987557	26/10/2012	456	34	RM	2F7643
630876	15/10/2012	456	53	MI	2F7643
539856	12/10/2012	567	44	MI	2F7643

AGENTI

Matricola	CF	Cognome	Nome
567	RSSM...	Rossi	Mario
456	NREL...	Neri	Luigi
638	NREP...	Neri	Piero

AUTO

Prov	Numero	Proprietario	Indirizzo
RM	2F7643	Verdi Piero	Via Tigli
RM	1A2396	Verdi Piero	Via Tigli
RM	4E5432	Bini Luca	Via Aceri
MI	2F7643	Luci Gino	Via Aceri

Figura 2.19 Una base di dati con vincoli di integrità referenziale.

- fra l'attributo Agente della relazione INFRAZIONI e la relazione AGENTI;
- fra gli attributi Prov e Numero di INFRAZIONI e la relazione AUTO, in cui l'ordine degli attributi nella chiave preveda prima Prov e poi Numero.

La base di dati nella Figura 2.19 soddisfa entrambi i vincoli, mentre la base di dati nella Figura 2.20 li viola entrambi: il primo perché AGENTI non contiene nessuna tupla con valore di Matricola pari a 456, e il secondo perché AUTO non contiene nessuna tupla con valore 'RM' su Prov e '2F7643' su Numero (notiamo che esistono una tupla con valore 'RM' su Prov e un'altra con valore '2F7643' su Numero, ma questo non è sufficiente, perché è richiesto che ci sia una tupla con entrambi i valori: solo in questo modo, infatti, i due valori possono far riferimento a una tupla della relazione AUTO).

Relativamente al secondo vincolo, notiamo come il ragionamento sull'ordine degli attributi possa apparire pesante, visto che la corrispondenza può, almeno in questo caso, essere realizzata per mezzo dei nomi degli attributi stessi. In generale, però, questo può non accadere, quindi l'ordinamento è essenziale. Consideriamo per esempio una base di dati contenente informazioni sui veicoli coinvolti in incidenti stradali. In particolare, supponiamo di voler includere in una relazione, insieme ad altre informazioni, i numeri di targa dei due veicoli coinvolti.[6] Ovviamente, dovremo

[6] Supponiamo per semplicità che i veicoli coinvolti siano solo due.

INFRAZIONI

Codice	Data	Agente	Articolo	Prov	Numero
987554	26/10/2012	456	34	RM	2F7643
630876	15/10/2012	456	53	FI	4E5432

AGENTI

Matricola	CF	Cognome	Nome
567	RSSM...	Rossi	Mario
638	NREP...	Neri	Piero

AUTO

Prov	Numero	Proprietario	Indirizzo
RM	1A2396	Verdi Piero	Via Tigli
FI	4E5432	Bini Luca	Via Aceri
MI	2F7643	Luci Gino	Via Noci

Figura 2.20 Una base di dati che viola vincoli di integrità referenziale.

avere due coppie di attributi, che non potranno avere gli stessi nomi. Per esempio, lo schema potrebbe essere:

INCIDENTI(Codice, Prov1, Numero1, Prov2, Numero2, ...)

In questo caso, non sarà ovviamente possibile stabilire la corrispondenza nel vincolo di integrità referenziale verso la relazione AUTO per mezzo dei nomi degli attributi, in quanto essi sono diversi da quelli della chiave primaria di AUTO. Solo attraverso l'ordinamento diventa possibile specificare che il riferimento associa Prov1 (attributo di INCIDENTI) a Prov (attributo nella chiave di AUTO) e Numero1 a Numero e, analogamente, Prov2 a Prov e Numero2 a Numero. La base di dati in Figura 2.21 soddisfa i due vincoli in questione, mentre quella in Figura 2.22 soddisfa quello rela-

INCIDENTI

Codice	Prov1	Numero1	Prov2	Numero2	...
6207	RM	2F7643	FI	T39275	...
6974	FI	4E5432	FI	T39275	...

AUTO

Prov	Numero	Proprietario	Indirizzo
RM	1A2396	Verdi Piero	Via Tigli
FI	4E5432	Bini Luca	Via Aceri
FI	T39275	Bitti Piero	Via Pini
RM	2F7643	Luci Gino	Via Noci

Figura 2.21 Una base di dati con due vincoli di integrità referenziale simili.

tivo a Prov1 e Numero1 e viola l'altro, perché nella relazione AUTO non c'è nessun veicolo targato FI T39275.

Un'ultima considerazione può essere utile riguardo a relazioni con più chiavi. In questo caso, come abbiamo detto, è opportuno che una delle chiavi sia evidenziata come primaria, ed è ragionevole che i riferimenti siano diretti verso di essa: per questo motivo, nella specifica dei vincoli di integrità referenziale, abbiamo potuto omettere la citazione esplicita degli attributi che compongono la chiave primaria. Peraltro, va notato che non tutti i sistemi di gestione di basi di dati oggi sul mercato permettono di indicare esplicitamente la chiave primaria: alcuni permettono anche di specificare più chiavi, ma non di evidenziarne una come primaria. In questi casi, il vincolo di integrità referenziale deve indicare esplicitamente gli attributi che compongono la chiave cui si fa riferimento. Per esempio, consideriamo una base di dati sullo schema

IMPIEGATI(Matricola,Cognome,Nome,Dipartimento)
DIPARTIMENTI(Codice,Nome,Sede)

in cui la relazione DIPARTIMENTI sia identificata dall'attributo Codice e, separatamente, dall'attributo Nome (non esistono due dipartimenti con lo stesso codice e non esistono due dipartimenti con lo stesso nome). Metodologicamente, abbiamo detto che è opportuno che una delle due chiavi, per esempio Codice, sia individuata come primaria e utilizzata per stabilire i riferimenti. Se però il sistema non prevede il concetto di chiave primaria, il vincolo deve essere espresso indicando esplicitamente gli attributi; dovremo quindi dire che esiste un vincolo di integrità referenziale fra l'attributo Dipartimento della relazione IMPIEGATI e la chiave Codice della relazione DIPARTIMENTI.

Questo è il motivo per cui, come vedremo nel Capitolo 4, nei sistemi relazionali è prevista una specifica più articolata per i vincoli di integrità referenziale.

2.3 Conclusioni

Abbiamo visto in questo capitolo la definizione delle strutture e dei vincoli del modello relazionale. Abbiamo in primo luogo discusso il concetto di relazione, con gli

INCIDENTI

Codice	Prov1	Numero1	Prov2	Numero2	...
6207	RM	2F7643	FI	T39275	...
6974	FI	4E5432	FI	T39275	...

AUTO

Prov	Numero	Proprietario	Indirizzo
RM	1A2396	Verdi Piero	Via Tigli
FI	4E5432	Bini Luca	Via Aceri
RM	2F7643	Luci Gino	Via Noci

Figura 2.22 Una base di dati che viola un vincolo di integrità referenziale.

adattamenti necessari per meglio sfruttare i concetti della teoria degli insiemi nel contesto delle basi di dati. Poi, abbiamo mostrato come le relazioni possano essere utilizzate per organizzare insiemi di dati anche complessi, utilizzando i dati stessi per realizzare riferimenti fra le diverse componenti (senza alcun utilizzo di puntatori espliciti). Quindi, dopo aver brevemente accennato alla necessità di utilizzare valori nulli per denotare l'assenza di informazioni, abbiamo discusso il concetto di vincolo di integrità, attraverso tre classi fondamentali: i vincoli di tupla, le chiavi e i vincoli di integrità referenziale.

Nei prossimi capitoli completeremo la presentazione del modello relazionale da due punti di vista:

- nel Capitolo 3 illustreremo i principi su cui si basano le operazioni di interrogazione di una base di dati;
- nel Capitolo 4 mostreremo come tutti i concetti, quelli relativi alle strutture e ai vincoli, discussi in questo capitolo, e quelli relativi alle operazioni (Capitolo 3), siano effettivamente realizzati nei DBMS reali, attraverso il linguaggio SQL, la cui presentazione sarà poi approfondita nel Capitolo 5.

Note bibliografiche

Presentazioni del modello relazionale analoghe a quella di questo capitolo sono reperibili su tutti i moderni testi sulle basi di dati, a cominciare da quelli di Elmasri e Navathe [41], Silberschatz, Korth e Sudarshan [71] e Ramakhrishnan e Gehrke [66], nonché quello di Garcia-Molina, Ullman, e Widom [46]. Formalizzazioni ulteriori e approfondimenti teorici (molto più dettagliati di quelli che presenteremo nei prossimi capitoli) si trovano nei libri di teoria delle basi di dati, in italiano quello di Atzeni, Batini e De Antonellis [5] e in inglese quelli di Ullman [79], Maier [56], Atzeni e De Antonellis [7], Abiteboul, Hull e Vianu [1]. Può essere interessante consultare l'articolo di Codd [30] che contiene la proposta originaria del modello, per constatarne tuttora l'attualità.

Esercizi

Soluzioni sul sito http://www.ateneonline.it/atzeni

2.1 Considerare le informazioni per la gestione dei prestiti di una biblioteca personale. Il proprietario presta libri ai propri amici, che indica semplicemente attraverso i rispettivi nomi o soprannomi (così da evitare omonimie) e fa riferimento ai libri attraverso i titoli (non possiede due libri con lo stesso titolo). Quando presta un libro, prende nota della data prevista di restituzione. Definire uno schema di relazione per rappresentare queste informazioni, individuando opportuni domini per i vari attributi e mostrarne un'istanza in forma tabellare. Indicare la chiave (o le chiavi) della relazione.

2.2 Rappresentare per mezzo di una o più relazioni le informazioni contenute nell'orario delle partenze da una stazione ferroviaria: numero, orario, destinazione finale, categoria, fermate intermedie, di tutti i treni in partenza.

2.3 Definire uno schema di base di dati per organizzare le informazioni di un'azienda che ha impiegati (ognuno con codice fiscale, cognome, nome e data di

nascita), filiali (con codice, sede e direttore, che è un impiegato). Ogni impiegato lavora presso una filiale. Indicare le chiavi e i vincoli di integrità referenziale dello schema. Mostrare un'istanza della base di dati e verificare che soddisfi i vincoli.

2.4 Un albero genealogico rappresenta, in forma grafica, la struttura di una famiglia (o più famiglie, quando è ben articolato). Mostrare come si possa rappresentare, in una base di dati relazionale, un albero genealogico, cominciando eventualmente da una struttura semplificata, in cui si rappresentano solo le discendenze in linea maschile (cioè i figli vengono rappresentati solo per i componenti di sesso maschile) oppure solo quelle in linea femminile.

2.5 Per ciascuno degli Esercizi 2.1-2.4, valutare le eventuali esigenze di rappresentazione di valori nulli, con i benefici e le difficoltà connesse.

2.6 Descrivere in linguaggio naturale le informazioni organizzate nella base di dati in Figura 2.23.

2.7 Individuare le chiavi e i vincoli di integrità referenziale che sussistono nella base di dati di Figura 2.23 e che è ragionevole assumere siano soddisfatti da tutte le basi di dati sullo stesso schema. Individuare anche gli attributi sui quali possa essere sensato ammettere valori nulli.

2.8 Definire uno schema di base di dati che organizzi i dati necessari a generare la pagina dei programmi radiofonici di un quotidiano, con stazioni, ore e titoli dei programmi; per ogni stazione sono memorizzati, oltre al nome, anche la frequenza di trasmissione e la sede.

2.9 Indicare quali tra le seguenti affermazioni sono vere in una definizione rigorosa del modello relazionale:

1. ogni relazione ha almeno una chiave;

PAZIENTI

Cod.	Cognome	Nome
A102	Necchi	Luca
B372	Rossini	Piero
B543	Missoni	Nadia
B444	Missoni	Luigi
S555	Rossetti	Gino

RICOVERI

Paz.	Inizio	Fine	Rep.
A102	2/05/11	9/05/11	A
A102	2/12/11	2/01/12	A
S555	25/04/11	3/05/11	B
B444	1/12/11	2/01/12	B
S555	5/10/11	1/11/11	A

MEDICI

Matr.	Cogn.	Nome	Rep.
203	Neri	Piero	A
574	Bisi	Mario	B
461	Bargio	Sergio	B
530	Belli	Nicola	C
405	Mizzi	Nicola	A
501	Monti	Mario	A

REPARTI

Cod.	Nome	Primario
A	Chirurgia	203
B	Pediatria	574
C	Medicina	530

Figura 2.23 Una base di dati per l'Esercizio 2.6 e 2.7.

2. ogni relazione ha esattamente una chiave;
3. ogni attributo appartiene al massimo a una chiave;
4. possono esistere attributi che non appartengono a nessuna chiave;
5. una chiave può essere sottoinsieme di un'altra chiave;
6. può esistere una chiave che coinvolge tutti gli attributi;
7. può succedere che esistano più chiavi e che una di esse coinvolga tutti gli attributi;
8. ogni relazione ha almeno una superchiave;
9. ogni relazione ha esattamente una superchiave;
10. può succedere che esistano più superchiavi e che una di esse coinvolga tutti gli attributi.

2.10 Considerare la base di dati relazionale in Figura 2.24, relativa a impiegati, progetti e partecipazioni di impiegati a progetti.
Indicare quali possano essere, per questa base di dati, ragionevoli chiavi primarie e vincoli di integrità referenziale. Giustificare brevemente la risposta, con riferimento alla realtà di interesse (cioè perché si può immaginare che tali vincoli sussistano) e all'istanza mostrata (verificando che sono soddisfatti).

2.11 Si supponga di voler rappresentare in una base di dati relazionale le informazioni relative al calendario d'esami di una facoltà universitaria, che vengono pubblicate con avvisi con la struttura mostrata in Figura 2.25. Mostrare gli schemi delle relazioni da utilizzare (con attributi e vincoli di chiave e di integrità referenziale) e l'istanza corrispondente ai dati sopra mostrati.

2.12 Si considerino le seguenti relazioni utilizzate per tenere traccia degli studenti di un'università, dei loro esami superati e verbalizzati attraverso gli esoneri e dei loro esami superati e verbalizzati attraverso i comuni appelli.

- ESAMIESONERI(Studente, Materia, VotoEson1, VotoEson2, VotoFinale)
- ESAMIAPPELLI(Studente, Materia, Voto)
- STUDENTI(Matricola, Nome, Cognome)

Indicare i vincoli di integrità che è ragionevole pensare debbano essere soddisfatti da tutte le basi di dati definite su questo schema.

IMPIEGATI

Matricola	Cognome	Nome	Età
101	Rossi	Mario	35
102	Rossi	Anna	42
103	Gialli	Mario	34
104	Neri	Gino	45

PROGETTI

ID	Titolo	Costo
A	Luna	70
B	Marte	60
C	Giove	90

PARTECIPAZIONE

Impiegato	Progetto
101	A
101	B
103	A
102	B

Figura 2.24 Una base di dati per l'Esercizio 2.10.

Calendario esami

Codice	Titolo	Prof	Appello	Data
1	Fisica	Neri	1	01/06/2012
			2	05/07/2012
			3	04/09/2012
			4	30/09/2012
2	Chimica	Rossi	1	06/06/2012
			2	05/07/2012
3	Algebra	Bruni		da definire

Figura 2.25 Un avviso con il calendario d'esami (Esercizio 2.11).

Algebra e calcolo relazionale

Poiché le basi di dati vengono utilizzate per rappresentare le informazioni di interesse per applicazioni che gestiscono dati, è evidente che i linguaggi per la specifica delle operazioni (di interrogazione e aggiornamento) sui dati stessi costituiscono a loro volta una componente essenziale delle basi di dati e quindi di ciascun modello dei dati. Un aggiornamento può essere visto come una funzione che, data un'istanza di base di dati, produce un'altra istanza di base di dati, sullo stesso schema. Un'interrogazione, invece, è essenzialmente una funzione che, data un'istanza di base di dati, produce una relazione, su un dato schema. Quindi, esistono aspetti comuni a interrogazioni e aggiornamenti, che consistono nell'esigenza di esprimere funzioni definite sull'insieme delle istanze di una base di dati. In particolare, risulta utile studiare i fondamenti dei linguaggi di interrogazione, per poi vedere come i concetti stessi sono realizzati in pratica nei sistemi relazionali e introdurre infine le operazioni di aggiornamento, che mutuano alcuni concetti delle operazioni di interrogazione.

Questo capitolo è appunto dedicato alla presentazione di linguaggi che, pur essendo diversi da quelli utilizzati dagli utenti nei sistemi, permettono però di esaminare varie questioni interessanti. In particolare, dedicheremo dapprima molto spazio all'algebra relazionale, un linguaggio *procedurale* (in cui cioè le operazioni complesse vengono specificate descrivendo il procedimento da seguire per ottenere la soluzione), illustrando i vari operatori, le espressioni e il modo in cui le espressioni possano essere trasformate per migliorarne l'efficienza. Accenneremo brevemente, sempre con riferimento all'algebra, a due argomenti: l'influenza che i valori nulli hanno sui linguaggi di interrogazione e le modalità secondo le quali possono essere definite relazioni virtuali (dette anche *viste*), non memorizzate nella base di dati.

Poi, presenteremo più sinteticamente il calcolo relazionale, che è viceversa un linguaggio *dichiarativo*, in cui le espressioni descrivono le proprietà del risultato, piuttosto che la procedura per ottenerlo. Questo linguaggio è basato sul calcolo dei predicati del primo ordine e ne presenteremo due versioni, la prima derivata direttamente dal calcolo dei predicati e la seconda che cerca di superare alcune limitazioni della prima. Concluderemo il capitolo con la breve trattazione del linguaggio Datalog, un interessante contributo della ricerca recente, che permette di formulare interrogazioni non esprimibili negli altri linguaggi. I paragrafi relativi al calcolo e al Datalog possono essere tralasciati senza alcun pregiudizio per la comprensione dei capitoli successivi del testo.

Nel Capitolo 4, dedicato all'SQL, vedremo come, dal punto di vista pratico, possa risultare utile combinare gli aspetti dichiarativi del calcolo e quelli procedurali dell'algebra. Vedremo anche come, in pratica, le operazioni di aggiornamento utilizzino gli stessi principi di quelle di interrogazione.

3.1 Algebra relazionale

Come abbiamo detto, l'algebra relazionale è un linguaggio procedurale, basato su concetti di tipo algebrico. Sostanzialmente, esso è costituito da un insieme di opera-

tori, definiti su relazioni e che producono ancora relazioni come risultati. In questo modo, è possibile costruire espressioni che coinvolgono più operatori, allo scopo di formulare interrogazioni anche complesse. Esaminiamo nei prossimi paragrafi i vari operatori:

- prima quelli insiemistici tradizionali, *unione*, *intersezione*, *differenza*, che, con piccole avvertenze, possono essere definiti anche sulle relazioni;
- poi quelli più specifici, *ridenominazione*, *selezione*, *proiezione*;
- infine il più importante, quello di *join*, in varie forme, *join naturale*, *prodotto cartesiano* e *theta-join*.

3.1.1 Unione, intersezione, differenza

Per iniziare, notiamo che le relazioni sono insiemi, e quindi ha senso definire su di esse gli operatori insiemistici tradizionali di unione, differenza e intersezione (peraltro quest'ultima esprimibile per mezzo della differenza, in quanto è sempre vero che $r \cap s = r - (r - s)$). Però, dobbiamo prestare attenzione al fatto che una relazione non è genericamente un insieme di tuple, ma un insieme di tuple *omogenee*, cioè definite sugli stessi attributi. Pertanto, pur potendosi, in linea di principio, definire gli operatori in questione su qualunque coppia di relazioni, non ha senso, dal punto di vista del modello relazionale, definirli con riferimento a relazioni su attributi diversi. Per esempio, l'unione di due relazioni su schemi diversi sarebbe un insieme di tuple disomogenee, alcune definite sugli attributi della prima e le altre su quelli della seconda. Ciò risulta insoddisfacente, perché un insieme di tuple disomogenee non è una relazione e noi, al fine di combinare gli operatori per formare espressioni complesse, vogliamo che i risultati siano relazioni. Pertanto, consideriamo ammissibili, nell'algebra relazionale, solo applicazioni degli operatori di unione, intersezione e differenza a coppie di operandi definite sugli stessi attributi. La Figura 3.1 mostra esempi di applicazioni dei tre operatori, che confermano le usuali definizioni, adattate al nostro contesto:

- l'*unione* di due relazioni r_1 e r_2 definite sullo stesso insieme di attributi X è indicata con $r_1 \cup r_2$ ed è una relazione ancora su X contenente le tuple che appartengono a r_1 oppure a r_2, oppure a entrambe;
- l'*intersezione* di $r_1(X)$ e $r_2(X)$ è indicata con $r_1 \cap r_2$ ed è una relazione su X contenente le tuple che appartengono sia a r_1 sia a r_2;
- la *differenza* di $r_1(X)$ e $r_2(X)$ è indicata con $r_1 - r_2$ ed è una relazione su X contenente le tuple che appartengono a r_1 e non appartengono a r_2.

3.1.2 Ridenominazione

La limitazione che abbiamo dovuto imporre agli operatori insiemistici, pur giustificata, risulta però particolarmente pesante. Consideriamo per esempio le due relazioni nella Figura 3.2: sarebbe sensato eseguire su di esse una sorta di unione, al fine di

LAUREATI

Matricola	Cognome	Età
7274	Rossi	37
7432	Neri	39
9824	Verdi	38

DIRIGENTI

Matricola	Cognome	Età
9297	Neri	56
7432	Neri	39
9824	Verdi	38

LAUREATI ∪ DIRIGENTI

Matricola	Cognome	Età
7274	Rossi	37
7432	Neri	39
9824	Verdi	38
9297	Neri	56

LAUREATI ∩ DIRIGENTI

Matricola	Cognome	Età
7432	Neri	39
9824	Verdi	38

LAUREATI − DIRIGENTI

Matricola	Cognome	Età
7274	Rossi	37

Figura 3.1 Operazioni di unione, intersezione e differenza.

PATERNITÀ

Padre	Figlio
Adamo	Caino
Adamo	Abele
Abramo	Isacco
Abramo	Ismaele

MATERNITÀ

Madre	Figlio
Eva	Caino
Eva	Set
Sara	Isacco
Agar	Ismaele

PATERNITÀ ∪ MATERNITÀ ??

Figura 3.2 Un'unione sensata ma scorretta.

ottenere tutte le coppie "genitore-figlio" note alla base di dati, ma ciò non è possibile, perché l'attributo che intuitivamente abbiamo indicato con Genitore si chiama in effetti Padre in una relazione e Madre nell'altra.

Per risolvere il problema, introduciamo uno specifico operatore, che ha come unico obiettivo proprio quello di adeguare i nomi degli attributi, a seconda delle necessità, in particolare al fine di facilitare le operazioni insiemistiche. L'operatore è detto di *ridenominazione*, perché appunto "cambia il nome degli attributi", lasciando inalterato il contenuto delle relazioni. Consideriamo un esempio nella Figura 3.3: nella rappresentazione tabellare vediamo bene come, fra operando e risultato, cambi solo l'intestazione, mentre il corpo rimane invariato. Infatti, la ridenominazione agisce solo sullo schema, nell'esempio cambiando il nome dell'attributo Padre in Genitore, come indicato dalla notazione Genitore ← Padre posta a pedice del simbolo

PATERNITÀ

Padre	Figlio
Adamo	Caino
Adamo	Abele
Abramo	Isacco
Isacco	Giacobbe

$\rho_{\text{Genitore}\leftarrow\text{Padre}}$(PATERNITÀ)

Genitore	Figlio
Adamo	Caino
Adamo	Abele
Abramo	Isacco
Isacco	Giacobbe

Figura 3.3 Una ridenominazione.

ρ che denota l'operatore di ridenominazione. La Figura 3.4 mostra l'applicazione dell'unione al risultato di due ridenominazioni sulle relazioni della Figura 3.2.

Definiamo l'operatore di ridenominazione in forma generale. Sia r una relazione definita sull'insieme di attributi X e sia Y un (altro) insieme di attributi con la stessa cardinalità. Inoltre, siano $A_1A_2 \ldots A_k$ e $B_1B_2 \ldots B_k$ rispettivamente un ordinamento per gli attributi in X e un ordinamento per quelli in Y. Allora la ridenominazione:

$$\rho_{B_1B_2\ldots B_k \leftarrow A_1A_2\ldots A_k}(r)$$

contiene una tupla t' per ciascuna tupla t in r, definita come segue: t' è una tupla su Y e $t'[B_i] = t[A_i]$, per $i = 1, \ldots, k$. La definizione conferma che ciò che cambia sono i nomi degli attributi, mentre i valori rimangono inalterati e vengono associati ai nuovi attributi. In pratica, nelle due liste $A_1A_2 \ldots A_k$ e $B_1B_2 \ldots B_k$ noi indicheremo solo gli attributi che vengono ridenominati (cioè quelli per cui $A_i \neq B_i$). Questo è il motivo per cui nell'esempio della Figura 3.3 abbiamo scritto:

$$\rho_{\text{Genitore}\leftarrow\text{Padre}}(\text{PATERNITÀ})$$

$\rho_{\text{Genitore}\leftarrow\text{Padre}}$(PATERNITÀ) $\cup$ $\rho_{\text{Genitore}\leftarrow\text{Madre}}$(MATERNITÀ)

Genitore	Figlio
Adamo	Caino
Adamo	Abele
Abramo	Isacco
Abramo	Ismaele
Eva	Caino
Eva	Set
Sara	Isacco
Agar	Ismaele

Figura 3.4 Un'unione preceduta da due ridenominazioni.

IMPIEGATI

Cognome	Agenzia	Stipendio
Rossi	Roma	45
Neri	Milano	53

OPERAI

Cognome	Fabbrica	Salario
Verdi	Latina	33
Bruni	Monza	32

$$\rho_{\text{Sede,Retribuzione} \leftarrow \text{Agenzia,Stipendio}}(\text{IMPIEGATI}) \cup \rho_{\text{Sede,Retribuzione} \leftarrow \text{Fabbrica,Salario}}(\text{OPERAI})$$

Cognome	Sede	Retribuzione
Rossi	Roma	45
Neri	Milano	53
Verdi	Latina	33
Bruni	Monza	32

Figura 3.5 Un'altra unione preceduta da ridenominazioni.

e non:

$$\rho_{\text{Genitore,Figlio} \leftarrow \text{Padre,Figlio}}(\text{PATERNITÀ})$$

La Figura 3.5 mostra un altro esempio di unione preceduta da ridenominazioni. In questo caso, in ciascuna relazione sono due gli attributi che vengono ridenominati, quindi l'ordinamento delle coppie (Sede, Retribuzione e così via) è significativo.

3.1.3 Selezione

Passiamo ora a esaminare gli operatori più tipici dell'algebra relazionale, che permettono effettivamente di manipolare le relazioni. Si tratta di tre operatori: selezione, proiezione e join (quest'ultimo con diverse varianti). Prima di entrare nel dettaglio, facciamo una considerazione sui primi due: selezione e proiezione svolgono funzioni che potremmo definire complementari (od ortogonali). Sono entrambe definite su un operando e producono come risultato una porzione dell'operando. Più precisamente, la selezione produce un sottoinsieme delle tuple, su tutti gli attributi, mentre la proiezione dà un risultato cui contribuiscono tutte le tuple, ma su un sottoinsieme degli attributi. Pertanto, come schematizzato nella Figura 3.6, possiamo dire che la selezione genera "decomposizioni orizzontali" e la proiezione "decomposizioni verticali".

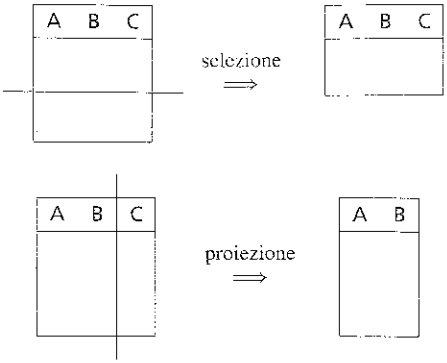

Figura 3.6 Una schematizzazione degli operatori di selezione e proiezione.

Le Figure 3.7 e 3.8 mostrano due esempi di selezioni, che illustrano le caratteristiche fondamentali dell'operatore, che è denotato dal simbolo σ a pedice del quale viene indicata la "condizione di selezione" opportuna. Il risultato contiene le tuple dell'operando che soddisfano la condizione di selezione. Come mostrato dagli esempi, le condizioni di selezione possono prevedere confronti fra attributi e confronti fra attributi e costanti, e possono essere complesse, ottenute combinando condizioni semplici con i connettivi logici $\vee$ (*or*), $\wedge$ (*and*) e $\neg$ (*not*).

In termini più precisi, data una relazione $r(X)$, una *formula proposizionale* F su X è una formula ottenuta combinando, con i connettivi $\vee$, $\wedge$ e $\neg$, condizioni atomiche del tipo $A\theta B$ o $A\theta c$, dove:

IMPIEGATI

Cognome	Nome	Età	Stipendio
Rossi	Mario	25	2.000,00
Neri	Luca	40	3.000,00
Verdi	Nico	36	4.500,00
Rossi	Marco	40	3.900,00

$\sigma_{\text{Età}>30 \wedge \text{Stipendio}>4.000,00}(\text{IMPIEGATI})$

Cognome	Nome	Età	Stipendio
Verdi	Nico	36	4.500,00

Figura 3.7 Una selezione.

CITTADINI

Cognome	Nome	CittàDiNascita	Residenza
Rossi	Mario	Roma	Milano
Neri	Luca	Roma	Roma
Verdi	Nico	Firenze	Firenze
Rossi	Marco	Napoli	Firenze

$\sigma_{\text{CittàDiNascita=Residenza}}(\text{CITTADINI})$

Cognome	Nome	CittàDiNascita	Residenza
Neri	Luca	Roma	Roma
Verdi	Nico	Firenze	Firenze

Figura 3.8 Un'altra selezione.

- θ è un operatore di confronto ($=, \neq, >, <, \geq, \leq$);
- A e B sono attributi in X sui cui valori il confronto θ abbia senso;
- c è una costante "compatibile" con il dominio di A (cioè tale che il confronto θ sia definito).

Date una formula F e una tupla t, è definito un valore di verità per F su t:

- $A\theta B$ è vera su t se $t[A]$ è in relazione θ con $t[B]$, altrimenti è falsa (per esempio, $A = B$ è vera su t se e solo se $t[A] = t[B]$);
- $A\theta c$ è vera su t se $t[A]$ è in relazione θ con c, altrimenti è falsa;
- $F_1 \vee F_2$, $F_1 \wedge F_2$ e $\neg F_1$ hanno l'usuale significato.

Possiamo a questo punto completare la definizione: la *selezione* $\sigma_F(r)$, in cui r è una relazione e F una formula proposizionale, produce una relazione sugli stessi attributi di r che contiene le tuple di r su cui F è vera.

3.1.4 Proiezione

La definizione dell'operatore di proiezione è ancora più semplice: dati una relazione $r(X)$ e un sottoinsieme Y di X, la *proiezione* di r su Y (indicata con $\pi_Y(r)$) è l'insieme di tuple su Y ottenute dalle tuple di r considerando solo i valori su Y:

$$\pi_Y(r) = \{ t[Y] \mid t \in r \}$$

La Figura 3.9 mostra un primo esempio di proiezione, che illustra chiaramente il concetto già citato secondo il quale la proiezione permette di decomporre verticalmente le relazioni: il risultato della proiezione contiene in questo caso tante tuple quante l'operando, definite però solo su parte degli attributi.

La Figura 3.10 mostra un'altra proiezione, in cui si nota una situazione diversa: il risultato contiene un numero di tuple inferiore rispetto a quelle dell'operando, perché alcune tuple, avendo uguali valori su tutti gli attributi della proiezione, danno lo stesso contributo alla proiezione stessa. Essendo le relazioni definite come insiemi, non

IMPIEGATI

Cognome	Nome	Reparto	Capo
Rossi	Mario	Vendite	Gatti
Neri	Luca	Vendite	Gatti
Verdi	Mario	Personale	Lupi
Rossi	Marco	Personale	Lupi

$\pi_{\text{Cognome,Nome}}$(IMPIEGATI)

Cognome	Nome
Rossi	Mario
Neri	Luca
Verdi	Mario
Rossi	Marco

Figura 3.9 Una proiezione.

IMPIEGATI

Cognome	Nome	Reparto	Capo
Rossi	Mario	Vendite	Gatti
Neri	Luca	Vendite	Gatti
Verdi	Mario	Personale	Lupi
Rossi	Marco	Personale	Lupi

$\pi_{\text{Reparto,Capo}}$(IMPIEGATI)

Reparto	Capo
Vendite	Gatti
Personale	Lupi

Figura 3.10 Una proiezione con meno tuple dell'operando.

possono in esse comparire più tuple uguali fra loro: i contributi uguali "collassano" in una sola tupla.

In generale, possiamo dire che il risultato di una proiezione contiene al più tante tuple quante l'operando, ma può contenerne di meno, come mostrato nella Figura 3.10. Notiamo anche che esiste un legame fra i vincoli di chiave e le proiezioni, relativamente a questo problema: $\pi_Y(r)$ contiene lo stesso numero di tuple di r se e solo se Y è superchiave per r. Infatti:

- se Y è superchiave, allora r non contiene tuple uguali su Y, quindi ogni tupla dà un contributo diverso alla proiezione;
- se la proiezione ha tante tuple quante l'operando, allora ciascuna tupla di r contribuisce alla proiezione con valori diversi, quindi r non contiene coppie di tuple uguali su Y: ma questa è proprio la definizione di superchiave.

Per la relazione IMPIEGATI nelle Figure 3.9 e 3.10, gli attributi Cognome e Nome formano una chiave (e perciò una superchiave), mentre Reparto e Capo non formano una superchiave: questo giustifica il comportamento riguardo al numero delle tuple. Come inciso, notiamo che una proiezione può produrre un numero di tuple pari a quelle dell'operando anche se gli attributi coinvolti non sono definiti come superchiave (nello schema). Per esempio, se riconsideriamo le relazioni discusse nel Capitolo 2 sullo schema:

STUDENTI(Matricola,Cognome,Nome,Nascita,Corso)

possiamo dire che, per tutte le relazioni, la proiezione su Matricola e quella su Cognome, Nome e Nascita hanno lo stesso numero di tuple dell'operando. Al contrario, una proiezione su Cognome e Corso può avere meno tuple; però, nel caso

r_1

Impiegato	Reparto
Rossi	vendite
Neri	produzione
Bianchi	produzione

r_2

Reparto	Capo
produzione	Mori
vendite	Bruni

$r_1 \bowtie r_2$

Impiegato	Reparto	Capo
Rossi	vendite	Bruni
Neri	produzione	Mori
Bianchi	produzione	Mori

Figura 3.11 Un join naturale.

particolare (come per esempio nella Figura 2.16) in cui non vi siano studenti diversi con lo stesso cognome iscritti allo stesso corso di laurea, allora anche la proiezione su Cognome e Corso ha lo stesso numero di tuple dell'operando.

3.1.5 Join

Passiamo ora a esaminare l'operatore di join[1] che è il più caratteristico dell'algebra relazionale, in quanto è l'operatore che permette di correlare dati contenuti in relazioni diverse, confrontando i valori contenuti in esse e utilizzando quindi la caratteristica fondamentale del modello, quella di essere basato su valori. Esistono, a parte alcune varianti, due versioni dell'operatore, comunque riconducibili l'una all'altra: la prima (il join naturale) utile per riflessioni di tipo astratto e la seconda (il theta-join) più rilevante dal punto di vista pratico.

Join naturale Il *join naturale* è un operatore (lo definiamo inizialmente in versione binaria, cioè con due operandi, per poi generalizzare) che correla dati in relazioni diverse, sulla base di valori uguali in attributi con lo stesso nome. Vediamo un esempio nella Figura 3.11, in cui l'operatore è denotato, come sarà sempre nel seguito, con il simbolo $\bowtie$. Il risultato del join è costituito da una relazione sull'unione degli insiemi di attributi degli operandi e le sue tuple sono ottenute combinando le tuple degli operandi con valori uguali sugli attributi comuni (in questo caso l'attributo Reparto): per esempio, la prima tupla del join deriva dalla combinazione della prima tupla della relazione r_1 e della seconda tupla della relazione r_2.

In generale, il *join naturale* $r_1 \bowtie r_2$ di $r_1(X_1)$ e $r_2(X_2)$ è una relazione definita su $X_1 X_2$ (cioè sull'unione degli insiemi X_1 e X_2), come segue:

$$r_1 \bowtie r_2 = \{ t \text{ su } X_1 X_2 \quad | \quad \text{esistono } t_1 \in r_1 \text{ e } t_2 \in r_2 \text{ con } t[X_1] = t_1 \text{ e } t[X_2] = t_2 \}$$

Più sinteticamente, ma in modo equivalente, possiamo scrivere:

$$r_1 \bowtie r_2 = \{ t \text{ su } X_1 X_2 \quad | \quad t[X_1] \in r_1 \text{ e } t[X_2] \in r_2 \}$$

[1] Utilizziamo per questo operatore il termine in lingua inglese, perché in effetti la sua traduzione ("giunzione" o "congiunzione") non viene mai usata in questo contesto.

La definizione conferma che le tuple del risultato sono ottenute combinando tuple degli operandi con valori uguali sugli attributi comuni. Infatti, se indichiamo con $X_{1,2}$ gli attributi comuni (cioè $X_{1,2} = X_1 \cap X_2$), le due condizioni $t[X_1] = t_1$ e $t[X_2] = t_2$ implicano (poiché $X_{1,2} \subseteq X_1$ e $X_{1,2} \subseteq X_2$) che $t[X_{1,2}] = t_1[X_{1,2}]$ e $t[X_{1,2}] = t_2[X_{1,2}]$, quindi $t_1[X_{1,2}] = t_2[X_{1,2}]$. Il grado della relazione ottenuta come risultato di un join è minore o uguale della somma dei gradi dei due operandi, perché gli attributi omonimi degli operandi compaiono una sola volta nel risultato.

È opportuno notare che è molto frequente eseguire join sulla base di valori della chiave di una delle relazioni coinvolte, esplicitando i riferimenti fra tuple che, come abbiamo più volte ripetuto, sono realizzati per mezzo di valori, soprattutto valori di chiavi. In molti di questi casi, è anche definito, fra gli attributi coinvolti, un vincolo di integrità referenziale. Riconsideriamo per esempio le relazioni INFRAZIONI e AUTO nella base di dati nella Figura 2.19, ripetute per comodità nella Figura 3.12 insieme al join di esse. Notiamo che ciascuna delle tuple di INFRAZIONI è stata combinata con una e una sola delle tuple di AUTO: (i) una sola perché Prov e Numero formano una chiave di AUTO (ii) almeno una perché è definito il vincolo di integrità referenziale fra Prov e Numero in INFRAZIONI e (la chiave primaria di) AUTO. Il join, quindi, ha esattamente tante tuple quante la relazione INFRAZIONI.

La Figura 3.13 mostra un altro esempio di join, sulle stesse relazioni riguardo alle quali abbiamo già visto (Figura 3.4) una unione preceduta da ridenominazione: qui combiniamo i dati nelle due relazioni sulla base del valore del figlio, ottenendo la

INFRAZIONI

Codice	Data	Ag	Art	Prov	Numero
143256	25/10/2012	567	44	RM	4E5432
987554	26/10/2012	456	34	RM	4E5432
987557	26/10/2012	456	34	RM	2F7643
630876	15/10/2012	456	53	MI	2F7643
539856	12/10/2012	567	44	MI	2F7643

AUTO

Prov	Numero	Proprietario	Indirizzo
RM	2F7643	Verdi Piero	Via Tigli
RM	1A2396	Verdi Piero	Via Tigli
RM	4E5432	Bini Luca	Via Aceri
MI	2F7643	Luci Gino	Via Aceri

INFRAZIONI ⋈ AUTO

Codice	Data	Ag	Art	Prov	Numero	Proprietario	Indirizzo
143256	25/10/2012	567	44	RM	4E5432	Bini Luca	Via Aceri
987554	26/10/2012	456	34	RM	4E5432	Bini Luca	Via Aceri
987557	26/10/2012	456	34	RM	2F7643	Verdi Piero	Via Tigli
630876	15/10/2012	456	53	MI	2F7643	Luci Gino	Via Aceri
539856	12/10/2012	567	44	MI	2F7643	Luci Gino	Via Aceri

Figura 3.12 Le relazioni INFRAZIONI e AUTO e il loro join.

PATERNITÀ

Padre	Figlio
Adamo	Caino
Adamo	Abele
Abramo	Isacco
Abramo	Ismaele

MATERNITÀ

Madre	Figlio
Eva	Caino
Eva	Set
Sara	Isacco
Agar	Ismaele

PATERNITÀ⋈MATERNITÀ

Padre	Figlio	Madre
Adamo	Caino	Eva
Abramo	Isacco	Sara
Abramo	Ismaele	Agar

Figura 3.13 Figli con entrambi i genitori.

coppia di genitori, per ogni persona per cui entrambi siano indicati nella base di dati. I due esempi considerati insieme ci mostrano quindi come i vari operatori dell'algebra relazionale permettano di combinare e correlare in vario modo, a seconda delle necessità, i dati contenuti in una base di dati.

Join completi e incompleti Discutiamo ora diversi esempi di join, per svolgere alcune importanti considerazioni, con riferimento alla dimensione del risultato e al contributo a esso portato dalle tuple degli operandi. Nell'esempio nella Figura 3.11, possiamo dire che ciascuna tupla di ciascuno degli operandi contribuisce ad almeno una tupla del risultato (il join si dice in questo caso *completo*): per ogni tupla t_1 di r_1, esiste una tupla t in $r_1 \bowtie r_2$ tale che $t[X_1] = t_1$ (e analogamente per r_2). Questa proprietà non è sempre verificata, perché richiede una corrispondenza fra le tuple delle due relazioni. La Figura 3.14 mostra un join in cui alcune tuple degli operandi (in particolare la prima di r_1 e la seconda di r_2) non contribuiscono al risultato, perché l'altra relazione non contiene tuple con gli stessi valori sull'attributo comune. In inglese tali tuple vengono chiamate *dangling* (cioè "dondolanti"). Come caso limite, è

r_1

Impiegato	Reparto
Rossi	vendite
Neri	produzione
Bianchi	produzione

r_2

Reparto	Capo
produzione	Mori
acquisti	Bruni

$r_1 \bowtie r_2$

Impiegato	Reparto	Capo
Neri	produzione	Mori
Bianchi	produzione	Mori

Figura 3.14 Un join con tuple *dangling*.

r_1

Impiegato	Reparto
Rossi	vendite
Neri	produzione
Bianchi	produzione

r_2

Reparto	Capo
concorsi	Mori
acquisti	Bruni

$r_1 \bowtie r_2$

Impiegato	Reparto	Capo

Figura 3.15 Un join vuoto.

ovviamente possibile che nessuna delle tuple degli operandi sia combinabile, e allora il risultato del join è la relazione vuota (Figura 3.15).

All'estremo opposto, è possibile che ciascuna delle tuple di ciascuno degli operandi sia combinabile con tutte le tuple dell'altro, come mostrato nella Figura 3.16. In tal caso, il risultato contiene un numero di tuple pari al prodotto delle cardinalità degli operandi e cioè $|r_1| \times |r_2|$ (dove $|r|$ indica la cardinalità della relazione r).

Ricapitolando, possiamo dire che il join di r_1 e r_2 contiene un numero di tuple compreso fra 0 e $|r_1| \times |r_2|$. Inoltre:

- se il join di r_1 e r_2 è completo, allora contiene almeno un numero di tuple pari al massimo fra $|r_1|$ e $|r_2|$;
- se $X_1 \cap X_2$ contiene una chiave per r_2, allora il join di $r_1(X_1)$ e $r_2(X_2)$ contiene al più $|r_1|$ tuple;
- se $X_1 \cap X_2$ coincide con una chiave per r_2 e sussiste il vincolo di riferimento fra $X_1 \cap X_2$ in r_1 e la chiave di r_2, allora il join di $r_1(X_1)$ e $r_2(X_2)$ contiene esattamente $|r_1|$ tuple.

r_1

Impiegato	Progetto
Rossi	A
Neri	A
Bianchi	A

r_2

Progetto	Capo
A	Mori
A	Bruni

$r_1 \bowtie r_2$

Impiegato	Reparto	Capo
Rossi	A	Mori
Neri	A	Mori
Bianchi	A	Mori
Rossi	A	Bruni
Neri	A	Bruni
Bianchi	A	Bruni

Figura 3.16 Un join con $|r_1| \times |r_2|$ tuple.

Join esterni La caratteristica dell'operatore di join di "tralasciare" le tuple di una relazione senza controparte nell'altra è utile in molti casi ma potenzialmente pericolosa in altri, in quanto può portare a omettere informazioni rilevanti. Consideriamo per esempio il join in Figura 3.14, supponendo di essere interessati a tutti gli impiegati, con l'indicazione del capo se noto. Allo scopo, è stata proposta (e recepita nelle ultime versioni dell'SQL, e la approfondiremo nel Paragrafo 4.3.2) una variante dell'operatore di join chiamata *join esterno* (in inglese *outer join*), che prevede che tutte le tuple diano un contributo al risultato, eventualmente estese con valori nulli ove non vi siano controparti opportune. Esistono tre varianti per l'operatore: il join esterno *sinistro*, che estende solo le tuple del primo operando, quello *destro*, che estende solo le tuple del secondo operando, e quello *completo*, che le estende tutte. Mostriamo in Figura 3.17 esempi di join esterno sulle relazioni già mostrate nella Figura 3.14, con una sintassi autoesplicativa.

Join n-ario, intersezione e prodotto cartesiano Vediamo alcune proprietà dell'operatore di join naturale.[2] In primo luogo, osserviamo che esso è commutativo, poiché $r_1 \bowtie r_2$ è sempre uguale a $r_2 \bowtie r_1$, e associativo, in quanto $r_1 \bowtie (r_2 \bowtie r_3)$ è ugua-

[2] Ci riferiamo qui al join naturale come tale, non al join esterno, per il quale alcune delle proprietà non valgono.

r_1

Impiegato	Reparto
Rossi	vendite
Neri	produzione
Bianchi	produzione

r_2

Reparto	Capo
produzione	Mori
acquisti	Bruni

$r_1 \bowtie_{\text{LEFT}} r_2$

Impiegato	Reparto	Capo
Rossi	vendite	NULL
Neri	produzione	Mori
Bianchi	produzione	Mori

$r_1 \bowtie_{\text{RIGHT}} r_2$

Impiegato	Reparto	Capo
Neri	produzione	Mori
Bianchi	produzione	Mori
NULL	acquisti	Bruni

$r_1 \bowtie_{\text{FULL}} r_2$

Impiegato	Reparto	Capo
Rossi	vendite	NULL
Neri	produzione	Mori
Bianchi	produzione	Mori
NULL	acquisti	Bruni

Figura 3.17 Alcuni join esterni.

le a $(r_1 \bowtie r_2) \bowtie r_3$. Pertanto, potremo scrivere, ove necessario, sequenze di join senza parentesi:

$$r_1 \bowtie r_2 \bowtie \ldots \bowtie r_n \quad \text{oppure} \quad \bowtie_{i=1}^{n} r_i$$

Notiamo poi che non abbiamo fatto alcuna ipotesi specifica sugli insiemi di attributi X_1 e X_2 su cui sono definiti gli operandi. Di conseguenza, ha senso considerare anche i casi estremi, quello in cui essi sono uguali e quello in cui sono disgiunti. In entrambi i casi, si applica la definizione generale vista sopra, ma è opportuno fare alcune osservazioni. Se $X_1 = X_2$, allora il join coincide in effetti con l'intersezione:

$$r_1(X_1) \bowtie r_2(X_1) = r_1(X_1) \cap r_2(X_1)$$

in quanto, nella definizione, si richiede che il risultato sia definito sull'unione dei due insiemi di attributi, e che contenga le tuple t tali che $t[X_1] \in r_1$ e $t[X_2] \in r_2$; se $X_1 = X_2$, allora l'unione di X_1 e X_2 è ancora pari a X_1, quindi t è definita su X_1: la definizione richiede pertanto che $t \in r_1$ e $t \in r_2$ e coincide perciò con la definizione di intersezione.

Il caso in cui i due insiemi di attributi sono disgiunti merita ancor maggiore attenzione. Il risultato è sempre definito sull'unione $X_1 X_2$, e ciascuna tupla deriva sempre da due tuple, una per ciascuno degli operandi, ma, in effetti, poiché tali tuple non hanno attributi in comune, non viene richiesta a esse nessuna condizione per partecipare insieme al join: la condizione che informalmente abbiamo prima discusso, e cioè che le due tuple devono avere gli stessi valori sugli attributi comuni, degenera in una condizione sempre verificata. Quindi, il risultato del join contiene le tuple ottenute combinando, in tutti i modi possibili, le tuple degli operandi. In questo caso particolare, si dice spesso che il join diventa un *prodotto cartesiano*. Potremmo dire che il prodotto cartesiano è un operatore definito (con la stessa definizione data sopra per il join naturale) su relazioni che non hanno attributi in comune. L'uso del termine è in effetti improprio, in quanto non si tratta di un prodotto cartesiano fra insiemi: il prodotto cartesiano di due insiemi è un insieme di coppie (con il primo elemento dal primo insieme e il secondo dal secondo), mentre qui abbiamo tuple, ottenute concatenando tuple della prima relazione e tuple della seconda. La Figura 3.18 mostra un esempio di prodotto cartesiano, confermando come il risultato contenga un numero di tuple pari al prodotto delle cardinalità degli operandi.

Theta-join ed equi-join Osservando la Figura 3.18 possiamo anche notare che un prodotto cartesiano ha di solito ben poca utilità, in quanto concatena tuple non necessariamente correlate dal punto di vista semantico. In effetti, il prodotto cartesiano viene spesso seguito da una selezione, che centra l'attenzione su tuple correlate secondo le esigenze. Per esempio, sulle relazioni IMPIEGATI e PROGETTI ha senso definire un prodotto cartesiano seguito dalla selezione che mantiene solo le tuple con valori uguali sull'attributo **Progetto** di IMPIEGATI e su **Codice** di PROGETTI (Figura 3.19).

Per questa ragione, viene spesso definito un operatore derivato (cioè esprimibile per mezzo di altri operatori), il *theta-join*, come prodotto cartesiano seguito da una

IMPIEGATI

Impiegato	Progetto
Rossi	A
Neri	A
Neri	B

PROGETTI

Codice	Nome
A	Venere
B	Marte

IMPIEGATI ⋈ PROGETTI

Impiegato	Progetto	Codice	Nome
Rossi	A	A	Venere
Neri	A	A	Venere
Neri	B	A	Venere
Rossi	A	B	Marte
Neri	A	B	Marte
Neri	B	B	Marte

Figura 3.18 Un prodotto cartesiano.

selezione, nel modo seguente (dove F è una formula proposizionale utilizzabile in una selezione e le relazioni r_1 e r_2 non hanno attributi in comune):

$$r_1 \bowtie_F r_2 = \sigma_F(r_1 \bowtie r_2)$$

La relazione nella Figura 3.19 può quindi essere ottenuta per mezzo del theta-join:

$$\text{IMPIEGATI} \bowtie_{\text{Progetto=Codice}} \text{PROGETTI}$$

Un theta-join in cui la condizione di selezione F sia una congiunzione di atomi di uguaglianza, con un attributo della prima relazione e uno della seconda, viene chia-

IMPIEGATI

Impiegato	Progetto
Rossi	A
Neri	A
Neri	B

PROGETTI

Codice	Nome
A	Venere
B	Marte

$\sigma_{\text{Progetto=Codice}}$(IMPIEGATI ⋈ PROGETTI)

Impiegato	Progetto	Codice	Nome
Rossi	A	A	Venere
Neri	A	A	Venere
Neri	B	B	Marte

Figura 3.19 Un prodotto cartesiano seguito da una selezione.

mato *equi-join*. Quindi la relazione nella Figura 3.19 è ottenuta per mezzo di un equi-join.

Dal punto di vista pratico il theta-join e ancor più l'equi-join hanno una grande importanza, in quanto la maggior parte dei sistemi di basi di dati effettivamente esistenti non utilizzano i nomi di attributo per correlare relazioni, e pertanto non utilizzano il join naturale ma l'equi-join e il theta-join. Approfondiremo questa osservazione quando discuteremo la specifica delle interrogazioni nel linguaggio SQL nel Capitolo 4. Peraltro, sottolineiamo che il join naturale, reso disponibile in pratica solo nelle ultime versioni dell'SQL, permette di elaborare in modo semplice riflessioni che sono poi estendibili all'equi-join. Notiamo anche come il join naturale possa essere simulato per mezzo della ridenominazione, dell'equi-join e della proiezione. Senza scendere nel dettaglio, mostriamo un esempio. Date due relazioni $r_1(ABC)$ e $r_2(BCD)$, il join naturale di r_1 e r_2 può essere espresso per mezzo degli altri operatori, in tre passi:

- ridenominando gli attributi in modo di ottenere relazioni su schemi disgiunti: $\rho_{B'C' \leftarrow BC}(r_2)$;
- effettuando l'equi-join, con condizioni di uguaglianza sugli attributi corrispondenti: $r_1 \bowtie_{B=B' \wedge C=C'} \rho_{B'C' \leftarrow BC}(r_2)$;
- concludendo con una proiezione che elimina gli attributi "doppioni," che presentano valori identici a quelli di altri attributi:

$$\pi_{ABCD}(r_1 \bowtie_{B=B' \wedge C=C'} \rho_{B'C' \leftarrow BC}(r_2))$$

3.1.6 Interrogazioni in algebra relazionale

In generale, un'interrogazione può essere definita come una funzione che, applicata a istanze di basi di dati, produce relazioni. Più precisamente, dato uno schema **R** di base di dati, un'interrogazione è una funzione che, per ogni istanza **r** di **R**, produce una relazione su un dato insieme di attributi X. Le espressioni dei vari linguaggi di interrogazione (per esempio dell'algebra relazionale) "rappresentano" o "realizzano" interrogazioni: ogni espressione definisce una funzione. Indichiamo con $E(\mathbf{r})$ il *risultato* dell'applicazione dell'espressione E alla base di dati **r**.

In algebra relazionale, le interrogazioni su uno schema di base di dati **R** vengono formulate con espressioni i cui atomi sono (nomi di) relazioni in **R** (le "variabili"). Concludiamo la presentazione dell'algebra relazionale mostrando la formulazione di alcune interrogazioni di crescente complessità, che fanno riferimento allo schema contenente le due relazioni:

IMPIEGATI(Matr,Nome,Età,Stipendio)
SUPERVISIONE(Capo,Impiegato)

Una base di dati su tale schema è mostrata nella Figura 3.20.

La prima interrogazione che consideriamo è molto semplice, in quanto coinvolge una sola relazione: *trovare matricola, nome ed età degli impiegati che guadagnano più di 40 mila euro*. In questo caso, con una selezione possiamo porre l'attenzione sulle sole tuple che soddisfano la condizione (stipendio maggiore di 40 mila euro) e con una proiezione eliminiamo gli attributi non richiesti:

IMPIEGATI

Matr	Nome	Età	Stipendio
101	Mario Rossi	34	40
103	Mario Bianchi	23	35
104	Luigi Neri	38	61
105	Nico Bini	44	38
210	Marco Celli	49	60
231	Siro Bisi	50	60
252	Nico Bini	44	70
301	Sergio Rossi	34	70
375	Mario Rossi	50	65

SUPERVISIONE

Capo	Impiegato
210	101
210	103
210	104
231	105
301	210
301	231
375	252

Figura 3.20 Una base di dati per gli esempi di espressioni.

$$\pi_{\text{Matr,Nome,Età}}(\sigma_{\text{Stipendio}>40}(\text{IMPIEGATI})) \tag{3.1}$$

Il risultato di questa espressione, applicata alla base di dati nella Figura 3.20, è mostrato nella Figura 3.21.

La seconda interrogazione coinvolge entrambe le relazioni, in modo molto naturale, *trovare le matricole dei capi degli impiegati che guadagnano più di 40 mila euro*:

$$\pi_{\text{Capo}}(\text{SUPERVISIONE} \bowtie_{\text{Impiegato=Matr}} \sigma_{\text{Stipendio}>40}(\text{IMPIEGATI})) \tag{3.2}$$

Matr	Nome	Età
104	Luigi Neri	38
210	Marco Celli	49
231	Siro Bisi	50
252	Nico Bini	44
301	Sergio Rossi	34
375	Mario Rossi	50

Figura 3.21 Il risultato dell'applicazione dell'Espressione 3.1 alla base di dati nella Figura 3.20.

Capo
210
301
375

Figura 3.22 Il risultato dell'applicazione dell'Espressione 3.2 alla base di dati nella Figura 3.20.

Nella Figura 3.22 è mostrato il risultato, sempre con riferimento alla base di dati nella Figura 3.20.

Passiamo a esempi un po' più complessi, in cui il coinvolgimento delle due relazioni è più articolato. Cominciamo aggiungendo solo un piccolo elemento all'interrogazione precedente: *trovare nome e stipendio dei capi degli impiegati che guadagnano più di 40 mila euro*. In questo caso, possiamo ovviamente far uso dell'espressione precedente, ma dobbiamo poi produrre, per ciascuna tupla del risultato, le informazioni richieste sul capo, che vanno estratte dalla relazione IMPIEGATI. È evidente, quindi, che ogni tupla del risultato è costruita a partire da tre tuple, la prima di IMPIEGATI, relativa a un impiegato che guadagna più di 40 mila euro, la seconda di SUPERVISIONE, che indica la matricola del capo dell'impiegato in questione, e la terza di nuovo di IMPIEGATI, con le informazioni relative al capo. Intuitivamente, la soluzione prevede il join della relazione IMPIEGATI con il risultato dell'espressione precedente, ma con una avvertenza: in generale, il capo e l'impiegato differiscono, quindi le due tuple di IMPIEGATI che contribuiscono a una tupla del join sono diverse. Il join deve quindi essere preceduto da una ridenominazione che "cambi" tutti i nomi degli attributi. Una possibile espressione allo scopo è la seguente (in cui alcuni nomi di attributo sono stati abbreviati per ragioni di spazio):

$$\pi_{\text{NomeC,StipC}}(\rho_{\text{MatrC,NomeC,StipC,EtàC}\leftarrow\text{Matr,Nome,Stip,Età}}(\text{IMPIEGATI}) \bowtie_{\text{MatrC=Capo}} \text{SUPERVISIONE} \bowtie_{\text{Imp=Matr}} \sigma_{\text{Stip>40}}(\text{IMPIEGATI})) \qquad (3.3)$$

Nella Figura 3.23 è mostrato il risultato, sempre con riferimento alla base di dati nella Figura 3.20.

NomeC	StipC
Marco Celli	60
Sergio Rossi	70
Mario Rossi	65

Figura 3.23 Il risultato dell'applicazione dell'Espressione 3.3 alla base di dati nella Figura 3.20.

Matr	Nome	Stip	MatrC	NomeC	StipC
104	Luigi Neri	61	210	Marco Celli	60
252	Nico Bini	70	375	Mario Rossi	65

Figura 3.24 Il risultato dell'applicazione dell'Espressione 3.4 alla base di dati nella Figura 3.20.

Il prossimo esempio è una variante del precedente, in quanto richiede il confronto di due valori dello stesso attributo, di tuple diverse: *trovare gli impiegati che guadagnano più del rispettivo capo, mostrando matricola, nome e stipendio di ciascuno di essi e del capo*. L'espressione è simile a quella precedente, e si nota ancora di più la necessità delle ridenominazioni (il risultato è nella Figura 3.24):

$$\pi_{\text{Matr,Nome,Stip,MatrC,NomeC,StipC}}$$
$$(\sigma_{\text{Stip>StipC}}(\rho_{\text{MatrC,NomeC,StipC,EtàC}\leftarrow\text{Matr,Nome,Stip,Età}}(\text{IMPIEGATI})$$
$$\bowtie_{\text{MatrC=Capo}}\text{SUPERVISIONE}\bowtie_{\text{Imp=Matr}}\text{IMPIEGATI})) \qquad (3.4)$$

L'ultimo esempio richiede ancora più attenzione: *trovare matricola e nome dei capi i cui impiegati guadagnano tutti più di 40 mila euro*. L'interrogazione include una sorta di quantificazione universale, ma l'algebra relazionale non contiene alcun costrutto direttamente utile allo scopo. Però, possiamo procedere con una doppia negazione, cercando i capi per i quali non vi sia alcun impiegato con stipendio non superiore a 40 mila euro. Questa interrogazione, pur contorta, può essere realizzata in algebra relazionale per mezzo dell'operatore di differenza: prendiamo tutti i capi meno quelli che hanno un impiegato che guadagna non più di 40 mila euro. L'espressione è la seguente:

$$\pi_{\text{Matr,Nome}}(\text{IMPIEGATI}\bowtie_{\text{Matr=Capo}}$$
$$(\pi_{\text{Capo}}(\text{SUPERVISIONE}) -$$
$$\pi_{\text{Capo}}(\text{SUPERVISIONE}\bowtie_{\text{Imp=Matr}}\sigma_{\text{Stip}\leq 40}(\text{IMPIEGATI}))) \qquad (3.5)$$

Il risultato di questa espressione sulla base di dati nella Figura 3.20 è mostrato nella Figura 3.25.

Matr	Nome
301	Sergio Rossi
375	Mario Rossi

Figura 3.25 Il risultato dell'Espressione 3.5 sulla base di dati nella Figura 3.20.

3.1.7 Equivalenza di espressioni algebriche

L'algebra relazionale, come molti altri strumenti formali in contesti diversi, permette di formulare espressioni fra loro *equivalenti*, cioè che producono lo stesso risultato. Per esempio, con riferimento ai numeri reali e agli operatori di addizione e moltiplicazione, vale l'equivalenza:

$$x \times (y + z) \equiv x \times y + x \times z$$

nel senso che, per ogni valore sostituito alle tre variabili, i due membri risultano uguali. Nell'algebra relazionale possiamo dare una definizione analoga, facendo attenzione al fatto che l'equivalenza può dipendere dallo schema, oppure essere assoluta:

- $E_1 \equiv_{\mathbf{R}} E_2$ se $E_1(\mathbf{r}) = E_2(\mathbf{r})$, per ogni istanza $\mathbf{r}$ di $\mathbf{R}$;
- $E_1 \equiv E_2$ se $E_1 \equiv_{\mathbf{R}} E_2$ per ogni schema $\mathbf{R}$.

La distinzione fra i due casi è dovuta al fatto che gli schemi degli operandi non vengono esplicitati nelle espressioni (in particolare nelle operazioni di join naturale), e il comportamento può variare a seconda degli attributi nei vari schemi di relazione. Un esempio di equivalenza assoluta è:

$$\pi_{AB}(\sigma_{A>0}(R)) \equiv \sigma_{A>0}(\pi_{AB}(R))$$

mentre la seguente equivalenza:

$$\pi_{AB}(R_1) \bowtie \pi_{AC}(R_2) \equiv_{\mathbf{R}} \pi_{ABC}(R_1 \bowtie R_2)$$

sussiste se e solo se nello schema $\mathbf{R}$ l'intersezione fra gli insiemi di attributi di R_1 e R_2 è pari ad A. Infatti, se ci fossero anche altri attributi, il join opererebbe solo su A nella prima espressione e su A e tali altri attributi nella seconda, con risultati in generale diversi.

L'equivalenza di espressioni dell'algebra risulta particolarmente importante dal punto di vista applicativo, nella fase di esecuzione delle interrogazioni (che approfondiremo nel Capitolo 11). Infatti, le interrogazioni, specificate in linguaggio SQL (Capitolo 4), vengono tradotte in algebra relazionale e, appunto con riferimento all'algebra, viene valutato il costo, sostanzialmente in termini di dimensioni dei risultati intermedi. In presenza di varie alternative equivalenti, viene scelta quella con costo minore. In questo contesto, vengono spesso utilizzate *trasformazioni di equivalenza*, cioè operazioni che sostituiscono un'espressione con un'altra a essa equivalente. In particolare, risultano interessanti le trasformazioni che riducono le dimensioni dei risultati intermedi e quelle che preparano un'espressione all'applicazione di una trasformazione che riduce le dimensioni dei risultati intermedi. Vediamo un primo insieme di trasformazioni interessanti.

1. Atomizzazione delle selezioni: una selezione congiuntiva può essere sostituita da una cascata di selezioni atomiche:

$$\sigma_{F_1 \wedge F_2}(E) \equiv \sigma_{F_1}(\sigma_{F_2}(E))$$

dove E è una qualunque espressione. Questa trasformazione permette di applicare successive trasformazioni che operano su selezioni con condizioni atomiche.

2. Idempotenza delle proiezioni: una proiezione può essere trasformata in una cascata di proiezioni che "eliminano" i vari attributi in fasi successive:

$$\pi_X(E) \equiv \pi_X(\pi_{XY}(E))$$

se E è definita su un insieme di attributi che contiene Y (oltre a X). Anche questa è una trasformazione preliminare ad altre.

3. Anticipazione della selezione rispetto al join (descritta spesso in inglese con *pushing selections down*):

$$\sigma_F(E_1 \bowtie E_2) \equiv E_1 \bowtie \sigma_F(E_2)$$

se la condizione F fa riferimento solo ad attributi nella sottoespressione E_2.

4. Anticipazione della proiezione rispetto al join (*pushing projections down*): siano E_1 ed E_2 definite rispettivamente su X_1 e X_2; se $Y_2 \subseteq X_2$ e $Y_2 \supseteq (X_1 \cap X_2)$ (cioè gli attributi in $X_2 - Y_2$ non sono coinvolti nel join) allora vale l'equivalenza:

$$\pi_{X_1 Y_2}(E_1 \bowtie E_2) \equiv E_1 \bowtie \pi_{Y_2}(E_2)$$

Combinando questa regola con quella della idempotenza delle proiezioni, possiamo ottenere la seguente equivalenza:

$$\pi_Y(E_1 \bowtie_F E_2) \equiv \pi_Y(\pi_{Y_1}(E_1) \bowtie_F \pi_{Y_2}(E_2))$$

dove, indicando con X_1 e X_2 gli attributi di E_1 ed E_2 rispettivamente e con J_1 e J_2 i rispettivi sottoinsiemi coinvolti nella condizione F di join:

$$Y_1 = (X_1 \cap Y) \cup J_1$$
$$Y_2 = (X_2 \cap Y) \cup J_2$$

In sostanza, possiamo eliminare subito da ciascuna relazione gli attributi che non compaiono nel risultato finale e non sono coinvolti nel join.

5. Inglobamento di una selezione in un prodotto cartesiano a formare un join:

$$\sigma_F(E_1 \bowtie E_2) \equiv E_1 \bowtie_F E_2$$

con $X_1 \cap X_2 = \emptyset$, indicando ancora con X_1 e X_2 rispettivamente gli attributi di E_1 e quelli di E_2.

Vediamo un esempio che chiarisce l'uso delle trasformazioni preparatorie e l'importante regola di anticipazione delle selezioni. Supponiamo di voler trovare, con riferimento alla base di dati di Figura 3.20, *i numeri di matricola dei capi di impiegati con meno di trenta anni*. Una prima espressione per specificare tale interrogazione potrebbe prevedere il prodotto cartesiano delle due relazioni seguito da una selezione congiuntiva e poi da una proiezione:

$$\pi_{\text{Capo}}(\sigma_{\text{Matr}=\text{Imp} \wedge \text{Età}<30}(\text{IMPIEGATI} \bowtie \text{SUPERVISIONE}))$$

La qualità di questa espressione è in effetti molto bassa, perché, per calcolare pochi valori (nel caso specifico, uno), effettua un prodotto cartesiano (che ha cardinalità pari al prodotto delle cardinalità degli operandi). Utilizzando le regole precedenti, possiamo pensare di migliorarla significativamente. Con la regola 1, spezziamo la selezione:

$$\pi_{\text{Capo}}(\sigma_{\text{Matr=Imp}}(\sigma_{\text{Età<30}}(\text{IMPIEGATI}\bowtie\text{SUPERVISIONE})))$$

Possiamo poi fondere la prima selezione con il prodotto cartesiano, e formare un join (regola 5) e anticipare la seconda selezione rispetto al join (regola 3), ottenendo:

$$\pi_{\text{Capo}}(\sigma_{\text{Età<30}}(\text{IMPIEGATI})\bowtie_{\text{Matr=Imp}}\text{SUPERVISIONE})$$

Infine, possiamo eliminare dal primo argomento del join (con una proiezione) gli attributi non necessari, utilizzando la regola 4:

$$\pi_{\text{Capo}}(\pi_{\text{Matr}}(\sigma_{\text{Età<30}}(\text{IMPIEGATI}))\bowtie_{\text{Matr=Imp}}\text{SUPERVISIONE})$$

Vediamo ora altre trasformazioni interessanti, in primo luogo ulteriori anticipazioni di selezioni e proiezioni.

6. Distributività della selezione rispetto all'unione:

$$\sigma_F(E_1 \cup E_2) \equiv \sigma_F(E_1) \cup \sigma_F(E_2)$$

7. Distributività della selezione rispetto alla differenza:

$$\sigma_F(E_1 - E_2) \equiv \sigma_F(E_1) - \sigma_F(E_2)$$

8. Distributività della proiezione rispetto all'unione:

$$\pi_X(E_1 \cup E_2) \equiv \pi_X(E_1) \cup \pi_X(E_2)$$

Vale la pena notare che la proiezione non è distributiva rispetto alla differenza, come si può verificare applicando le espressioni

$$\pi_A(R_1 - R_2) \quad \text{e} \quad \pi_A(R_1) - \pi_A(R_2)$$

a due relazioni su AB che contengano tuple uguali su A e diverse su B.

Un altro gruppo di trasformazioni interessanti contiene quelle che si basano sull'interazione fra gli operatori insiemistici e le selezioni complesse:

9. $\sigma_{F_1 \vee F_2}(R) \equiv \sigma_{F_1}(R) \cup \sigma_{F_2}(R)$
10. $\sigma_{F_1 \wedge F_2}(R) \equiv \sigma_{F_1}(R) \cap \sigma_{F_2}(R) \equiv \sigma_{F_1}(R) \bowtie \sigma_{F_2}(R)$
11. $\sigma_{F_1 \wedge \neg(F_2)}(R) \equiv \sigma_{F_1}(R) - \sigma_{F_2}(R)$

Abbiamo poi la proprietà commutativa e associativa di tutti gli operatori binari, esclusa la differenza, e la proprietà distributiva del join rispetto all'unione:

12. $E \bowtie (E_1 \cup E_2) \equiv (E \bowtie E_1) \cup (E \bowtie E_2)$

Infine, vale la pena segnalare che la presenza di risultati intermedi vuoti (relazioni con zero tuple) permette di semplificare le espressioni, in modo abbastanza naturale. Lasciando per esercizio i dettagli, notiamo che un join (o anche un prodotto cartesiano) in cui uno degli operandi sia la relazione vuota produce un risultato vuoto, perché non è possibile concatenare con alcuna tupla le tuple dell'altra relazione.

3.1.8 Algebra con valori nulli

Nella discussione dei paragrafi precedenti, abbiamo sempre supposto, per gradualità di presentazione, che le espressioni dell'algebra venissero applicate a relazioni prive di valori nulli. Avendo viceversa sottolineato, nel Paragrafo 2.1.5, l'importanza dei valori nulli nelle applicazioni reali, dobbiamo almeno accennare all'impatto che essi hanno sui linguaggi trattati in questo capitolo. La discussione sarà breve e volta prevalentemente a presentare i problemi; sarà ripresa nel Capitolo 4 dedicato al linguaggio SQL.

Consideriamo la relazione nella Figura 3.26 e la seguente selezione:

$$\sigma_{\text{Età}>30}(\text{PERSONE})$$

Ora, è indiscutibile che la prima tupla della relazione debba far parte del risultato e che la seconda invece no. Ma che cosa possiamo dire della terza? Intuitivamente, il valore di Età è un nullo di tipo sconosciuto, in quanto per ogni persona il valore esiste; tutt'al più, come in questo caso, può non essere noto. A proposito di queste interrogazioni, è stato proposto di utilizzare una logica a tre valori, in cui un predicato può essere vero, falso oppure assumere un terzo, nuovo valore di verità che chiamiamo *unknown* (sconosciuto) e rappresentiamo con il simbolo U. Un predicato assume questo valore quando almeno uno dei termini del confronto assume il valore nullo. Quindi, con riferimento al caso in discussione, la prima tupla certamente appartiene al risultato (appartenenza *vera*), la seconda certamente non appartiene (appartenenza *falsa*) e la terza forse appartiene e forse no (appartenenza *sconosciuta*). Le tabelle di verità dei connettivi logici *not*, *and* e *or*, per tenere conto del nuovo valore logico, si estendono nel modo seguente:

not	
F	V
U	U
V	F

and	V	U	F
V	V	U	F
U	U	U	F
F	F	F	F

or	V	U	F
V	V	V	V
U	V	U	U
F	V	U	F

Una selezione su relazioni con valori nulli produce come risultato le tuple per cui il predicato risulta vero. Il valore *unknown* rappresenta un valore di verità intermedio tra vero e falso, e il significato dei tre connettivi in questo contesto diventa il seguente:

PERSONE

Nome	Età	Reddito
Aldo	35	15
Andrea	27	21
Maria	NULL	42

Figura 3.26 Una relazione con valori nulli.

il *not* è vero solo se il valore di partenza è falso, l'*and* è vero solo se tutti i termini sono veri, e l'*or* è vero se almeno uno dei termini è vero.

Vale la pena sottolineare che la logica a tre valori risulta effettivamente significativa solo nel caso di espressioni complesse, in cui peraltro il risultato presenta proprietà poco soddisfacenti. Per esempio, consideriamo l'espressione dell'algebra:

$$\sigma_{\text{Età}>30}(\text{PERSONE}) \cup \sigma_{\text{Età}\leq 30}(\text{PERSONE})$$

Logica vorrebbe che questa espressione restituisse esattamente la relazione PERSONE, in quanto il valore dell'età o è maggiore di 30 (prima sottoespressione) oppure è non maggiore di 30 (seconda sottoespressione). D'altra parte, se le due sottoespressioni sono valutate separatamente, la terza tupla dell'esempio (così come ogni altra tupla con valore nullo per l'età) ha un'appartenenza sconosciuta a ciascuna sottoespressione e perciò all'unione. Solo attraverso una valutazione globale dell'espressione (cosa impraticabile per espressioni complesse) si arriva alla conclusione che tale tupla deve certamente apparire nel risultato. Stesso discorso potremmo fare per l'espressione:

$$\sigma_{\text{Età}>30 \vee \text{Età}\leq 30}(\text{PERSONE})$$

in cui la disgiunzione viene valutata secondo la logica a tre valori.

Il metodo migliore per superare in pratica gli inconvenienti appena discussi consiste nel trattare i valori nulli da un punto di vista meramente sintattico (rinunciando quindi a ragionare sui valori reali che essi potrebbero rappresentare) e può essere utilizzato sostanzialmente nello stesso modo sia con una logica a due valori sia con una a tre valori. Allo scopo, si introducono due nuove forme di condizioni atomiche di selezione, che valutano se un valore è specificato oppure nullo:

- A IS NULL assume valore vero su una tupla t se il valore di t su A è nullo e falso se esso è specificato;
- A IS NOT NULL assume valore vero su una tupla t se il valore di t su A è specificato e falso se il valore è nullo.

In questo contesto, l'espressione:

$$\sigma_{\text{Età}>30}(\text{PERSONE})$$

restituisce le persone la cui età è nota e maggiore di 30, mentre per ottenere quelle che hanno o potrebbero avere più di trent'anni (cioè quelle per cui l'età è nota e maggiore di 30 oppure non nota), possiamo utilizzare l'espressione:

$$\sigma_{\text{Età}>30 \vee \text{Età IS NULL}}(\text{PERSONE})$$

Analogamente, le espressioni:

$$\sigma_{\text{Età}>30}(\text{PERSONE}) \cup \sigma_{\text{Età}\leq 30}(\text{PERSONE})$$
$$\sigma_{\text{Età}>30 \vee \text{Età}\leq 30}(\text{PERSONE})$$

non restituiscono l'intera relazione, ma solo le tuple che hanno un valore non nullo per Età. Se volessimo l'intera relazione come risultato, allora dovremmo includere una condizione IS NULL:

$$\sigma_{\text{Età}>30 \vee \text{Età}\leq 30 \vee \text{Età IS NULL}}(\text{PERSONE})$$

Questo approccio, come vedremo nel Capitolo 4, è utilizzato (e consigliabile) nella versione attuale di SQL, che prevede la gestione di una logica a tre valori, ed era utilizzabile in precedenti versioni, che adottavano una logica a due valori.

3.1.9 Viste

Abbiamo osservato nel Capitolo 1 che può risultare utile mettere a disposizione degli utenti rappresentazioni diverse per gli stessi dati. Nel modello relazionale, la tecnica prevista a questo scopo è quella delle *relazioni derivate*, relazioni il cui contenuto è funzione del contenuto di altre relazioni. In una base di dati relazionale possono quindi esistere relazioni *di base*, il cui contenuto è autonomo, e relazioni derivate, il cui contenuto è funzione di quello di altre relazioni. È possibile che una relazione derivata sia funzione di altre relazioni derivate, a condizione che esista un ordinamento fra le relazioni derivate tale che ogni relazione sia definita solo in termini di relazioni di base e di relazioni derivate che la precedono nell'ordinamento.[3]

In linea di principio, possono esistere due tipi di relazioni derivate:

- *viste materializzate*: relazioni derivate effettivamente memorizzate nella base di dati;
- *relazioni virtuali* (chiamate anche *viste*, senza ulteriori specificazioni): relazioni definite per mezzo di funzioni (espressioni del linguaggio di interrogazione), non memorizzate nella base di dati, ma utilizzabili nelle interrogazioni come se lo fossero.

Le viste materializzate hanno il vantaggio di essere immediatamente disponibili per le interrogazioni, ma è spesso oneroso mantenere il loro contenuto allineato con quello delle relazioni da cui derivano. Le relazioni virtuali devono essere ricalcolate per ogni interrogazione ma non presentano problemi di allineamento. Le viste materializzate risultano quindi convenienti quando gli aggiornamenti sono rari rispetto alle interrogazioni e il calcolo della vista è complesso. È comunque difficile fornire tecniche generalizzate per mantenere l'allineamento. Per questo motivo, i sistemi attuali forniscono quasi solo meccanismi per la gestione di relazioni virtuali, che nel seguito, non essendoci rischio di ambiguità, chiameremo semplicemente *viste*.

Le viste vengono definite nei sistemi relazionali per mezzo di espressioni del linguaggio di interrogazione. Eventuali interrogazioni che si riferiscano alle viste vengono risolte sostituendo alla vista la sua definizione, componendo cioè le due interrogazioni. Di solito, i sistemi relazionali stabiliscono la strategia di esecuzione

[3] Questa condizione viene rilasciata nelle recenti proposte di basi di dati deduttive, che permettono di definire *viste ricorsive*. Accenneremo all'argomento nel Paragrafo 3.3.

delle interrogazioni dopo aver sostituito alla vista la sua definizione. Per esempio, supponiamo di avere una base di dati sulle relazioni:

$$R_1(ABC),\ R_2(DEF),\ R_3(GH)$$

con una vista definita per mezzo di un prodotto cartesiano seguito da una selezione:

$$R = \sigma_{A>D}(R_1 \bowtie R_2)$$

Su questo schema, l'interrogazione:

$$\sigma_{B=G}(R \bowtie R_3)$$

viene eseguita sostituendo a R la sua definizione:

$$\sigma_{B=G}(\sigma_{A>D}(R_1 \bowtie R_2) \bowtie R_3)$$

L'uso delle viste può risultare vantaggioso per diversi ordini di motivi:

- Un utente interessato solo a una porzione di una base di dati può evitare di considerare le componenti non rilevanti. Per esempio, in una base di dati con due relazioni sugli schemi

  AFFERENZA(Impiegato,Dipartimento)
  DIREZIONE(Dipartimento,Direttore)

  un utente interessato solo agli impiegati e ai relativi direttori potrebbe trarre vantaggio da una vista definita come:

  $$\pi_{\text{Impiegato,Direttore}}(\text{AFFERENZA} \bowtie \text{DIREZIONE})$$

- Espressioni molto complesse possono essere definite tramite viste, con vantaggi rilevanti soprattutto nel caso di presenza di sottoespressioni ripetute.
- Attraverso la definizione di autorizzazioni di accesso rispetto alle viste, è possibile introdurre meccanismi di protezione della privatezza.
- In occasione di ristrutturazioni di una base di dati, può risultare conveniente definire viste che corrispondano a relazioni sostituite da altre e perciò non più presenti dopo la ristrutturazione stessa, ma ricavabili dalle nuove relazioni. In questo modo, le applicazioni scritte con riferimento alla versione precedente dello schema possono essere utilizzate sul nuovo senza bisogno di modifiche. Per esempio, se uno schema $R(ABC)$ viene sostituito dagli schemi $R_1(AB)$, $R_2(BC)$, è possibile definire una vista $R = R_1 \bowtie R_2$ e mantenere inalterate le applicazioni che fanno riferimento a R. I risultati che vedremo nel capitolo sulla normalizzazione (Capitolo 9) confermano che, se B è una chiave per R_2, allora la presenza della vista è trasparente.

Mentre per quanto riguarda le interrogazioni le viste possono essere trattate come le relazioni di base, lo stesso non si può dire per le operazioni di aggiornamento. Infatti, in molti casi non è possibile stabilire facilmente una semantica degli aggiornamenti sulle viste: dato un aggiornamento su una vista, in generale non esiste uno e un solo aggiornamento delle relazioni di base (che sono le uniche effettivamente memorizzate) che porti a un'istanza della base di dati cui corrisponda un'istanza della vista che sia il risultato effettivo dell'aggiornamento specificato sulla vista. Per esempio, consideriamo ancora la vista sopra discussa:

$$\pi_{\text{Impiegato,Direttore}}(\text{AFFERENZAMDIREZIONE})$$

L'inserimento di una tupla nella vista non corrisponde univocamente a un insieme di aggiornamenti sulle relazioni di base, in quanto non risulta disponibile alcun valore per l'attributo Dipartimento, che stabilisce la corrispondenza fra le due relazioni. Per questo motivo, molti sistemi pongono forti limitazioni riguardo alla possibilità di specificare aggiornamenti sulle viste.

Riprenderemo la discussione sulle viste e presenteremo ulteriori esempi nel Capitolo 5, in cui mostreremo come le viste vengono definite e utilizzate in SQL.

3.2 Calcolo relazionale

Con il termine *calcolo relazionale* si fa riferimento a una famiglia di linguaggi di interrogazione, basati sul calcolo dei predicati del primo ordine, che hanno la caratteristica di essere *dichiarativi*, cioè di specificare le proprietà del risultato delle interrogazioni, anziché la procedura seguita per generarlo. In contrasto, come abbiamo già visto, l'algebra relazionale è un linguaggio *procedurale*, in quanto le sue espressioni specificano passo passo (attraverso le singole applicazioni degli operatori) la costruzione del risultato.

Come abbiamo già accennato, esistono diverse versioni del calcolo relazionale e non è certamente possibile (né sarebbe sensato) in questa sede presentarle tutte. Illustreremo per prima la versione forse più vicina al calcolo dei predicati (il *calcolo relazionale su domini*), che presenta in modo naturale le caratteristiche originali di questi linguaggi, per poi discuterne le limitazioni e le modifiche che possono portare a linguaggi di interesse pratico. Presenteremo quindi il *calcolo su tuple con dichiarazioni di range*, che costituisce la base per molti dei costrutti disponibili per le interrogazioni nel linguaggio SQL, che vedremo nel Capitolo 4.

Il presente paragrafo non ha alcun requisito di conoscenza pregressa del calcolo dei predicati del primo ordine. Concludiamo però l'introduzione al paragrafo con alcuni commenti che (potendo essere ignorati senza pregiudicare la comprensione dei concetti successivi) permettono a chi viceversa abbia già familiarità con tale formalismo di notare subito le differenze principali.

Rispetto alla usuale definizione del calcolo dei predicati del primo ordine, nel calcolo relazionale vi sono alcune semplificazioni e modifiche. In primo luogo, mentre nel calcolo dei predicati abbiamo in generale simboli di predicato (interpretati come relazioni su un universo fissato) e simboli di funzione (interpretati come funzioni), nel calcolo relazionale i simboli di predicato corrispondono alle relazioni nelle

basi di dati (oltre ad altri predicati standard come uguaglianza e disuguaglianza) e non compaiono simboli di funzione perché non necessari (grazie alla struttura piatta delle relazioni).

Poi, nel calcolo dei predicati interessano di solito sia formule aperte (cioè con variabili libere), sia formule chiuse (con tutte variabili legate e nessuna libera). Le seconde hanno un valore di verità che, rispetto a una interpretazione, è fissato, mentre le prime hanno un valore che dipende dai valori associati alle variabili libere. Nel calcolo relazionale interessano prevalentemente formule aperte: un'interrogazione è definita per mezzo di una formula del calcolo e il risultato è costituito dalle tuple di valori che, sostituiti alle variabili libere, rendono vera la formula stessa.

Infine, per coerenza con gli argomenti già discussi riguardo al modello relazionale, utilizziamo nel calcolo relazionale una notazione non posizionale.

Segnaliamo ancora che, come detto nell'introduzione al capitolo, questo paragrafo e il successivo possono essere tralasciati senza pregiudicare la comprensione dei capitoli successivi.

3.2.1 Calcolo relazionale su domini

Le espressioni del calcolo relazionale su domini hanno la forma:

$$\{A_1 : x_1, \ldots, A_k : x_k \mid f\}$$

dove:

- $A_1, \ldots, A_k$ sono attributi distinti (che possono anche non comparire nello schema della base di dati rispetto a cui viene formulata l'interrogazione);
- $x_1, \ldots, x_k$ sono *variabili* (che per comodità supponiamo distinte, anche se non sarebbe strettamente necessario);
- f è una formula, secondo le seguenti regole:
  - Vi sono formule *atomiche*, di due tipi:
    - $R(A_1 : x_1, \ldots, A_p : x_p)$, dove $R(A_1 \ldots A_p)$ è uno schema di relazione e $x_1, \ldots, x_p$ sono variabili.
    - $x\theta y$ o $x\theta c$, con x e y variabili, c costante e θ *operatore di confronto* ($=, \neq, \leq, \geq, >, <$).
  - Se f_1 e f_2 sono formule, allora $f_1 \vee f_2$, $f_1 \wedge f_2$ e $\neg f_1$ sono formule; ove necessario, per disambiguare le precedenze, si possono usare le parentesi.
  - Se f è una formula e x una variabile (che di solito compare in f, anche se non è strettamente necessario), allora $\exists x(f)$ e $\forall x(f)$ sono formule ($\exists$ e $\forall$ sono rispettivamente il *quantificatore esistenziale* e il *quantificatore universale*).

La lista di coppie $A_1 : x_1, \ldots, A_k : x_k$ viene chiamata *target list* (cioè lista degli obiettivi) in quanto definisce la struttura del risultato, che è costituito dalla relazione su $A_1, \ldots, A_k$ che contiene le tuple i cui valori sostituiti a $x_1, \ldots, x_k$ rendono vera la formula rispetto a un'istanza di base di dati a cui l'espressione viene applicata. La definizione precisa del concetto di *valore di verità* di una formula va oltre gli obiettivi di questo testo, ma può essere illustrata informalmente. Seguiamo allo scopo

la struttura sintattica delle formule (con "valore" intendiamo "elemento del dominio", assumendo per semplicità che tutti gli attributi abbiano lo stesso dominio):

- una formula atomica $R(A_1 : x_1, \ldots, A_p : x_p)$ è vera sui valori di $x_1, \ldots, x_p$ che formano una tupla della relazione r sullo schema R, nell'istanza di base di dati a cui l'espressione viene applicata;
- una formula atomica $x\theta y$ (per esempio $x > y$) è vera sui valori a_1 e a_2 se il confronto $a_1\theta a_2$ è soddisfatto (nell'esempio, se $a_1 > a_2$); analogamente per $x\theta c$;
- per congiunzione, disgiunzione e negazione valgono le usuali definizioni;
- per le formule con i quantificatori:
  - $\exists x(f)$ è vera se esiste almeno un valore a che, sostituito alla variabile x, rende vera f;
  - $\forall x(f)$ è vera se, per ogni possibile valore a per la variabile x, la formula f risulta vera.

Mostriamo le espressioni del calcolo che realizzano le stesse interrogazioni che abbiamo già formulato in algebra relazionale nel Paragrafo 3.1.6, con riferimento allo schema di basi di dati sulle relazioni:

IMPIEGATI(Matr,Nome,Età,Stipendio)
SUPERVISIONE(Capo,Impiegato)

Cominciamo in effetti con una interrogazione ancora più semplice di quelle già viste: *trovare matricola, nome, età e stipendio degli impiegati che guadagnano più di 40 mila euro*, che formuleremmo in algebra con una selezione:

$$\sigma_{\text{Stipendio}>40}(\text{IMPIEGATI})$$

Nel calcolo relazionale su domini abbiamo una formulazione altrettanto semplice, con l'espressione:

$$\{\text{Matr} : m,\ \text{Nome} : n,\ \text{Età} : e,\ \text{Stipendio} : s \mid \text{IMPIEGATI}(\text{Matr} : m,\ \text{Nome} : n,\ \text{Età} : e,\ \text{Stipendio} : s) \wedge s > 40\} \quad (3.6)$$

Notiamo la presenza di due condizioni nella formula (connesse dall'operatore logico di *and*, indicato con $\wedge$):

- la prima, IMPIEGATI(Matr : m, Nome : n, Età : e, Stipendio : s), richiede che i valori rispettivamente sostituiti alle variabili m, n, e, s costituiscano una tupla della relazione IMPIEGATI;
- la seconda richiede che il valore della variabile s sia maggiore di 40.

Stante il significato dell'operatore di *and*, il risultato è costituito dai valori sulle quattro variabili che provengono dalle tuple di IMPIEGATI per le quali il valore dello stipendio sia maggiore di 40.

L'interrogazione appena più complessa che richiede solo alcuni degli attributi: *trovare matricola, nome ed età degli impiegati che guadagnano più di 40 mila euro*, e che quindi in algebra abbiamo formulato con una proiezione (Espressione 3.1):

$$\pi_{\text{Matr,Nome,Età}}(\sigma_{\text{Stipendio}>40}(\text{IMPIEGATI}))$$

può essere formulata in vari modi. Il più diretto, anche se non il più semplice, è basato sull'osservazione che ciò che ci interessa sono i valori di matricola, nome ed età che partecipano a tuple per le quali lo stipendio è maggiore di 40, cioè per i quali esiste un valore dello stipendio, maggiore di 40, che permetta di completare una tupla della relazione IMPIEGATI. Possiamo quindi usare un quantificatore esistenziale:

$$\{\text{Matr} : m,\ \text{Nome} : n,\ \text{Età} : e \mid \exists s(\text{IMPIEGATI}(\text{Matr} : m,\ \text{Nome} : n,\ \text{Età} : e,\ \text{Stipendio} : s) \wedge s > 40)\} \quad (3.7)$$

In effetti, l'uso del quantificatore non è necessario, poiché scrivendo semplicemente

$$\{\text{Matr} : m,\ \text{Nome} : n,\ \text{Età} : e \mid \text{IMPIEGATI}(\text{Matr} : m,\ \text{Nome} : n,\ \text{Età} : e,\ \text{Stipendio} : s) \wedge s > 40\} \quad (3.8)$$

otteniamo lo stesso risultato: le tuple con valori m, n, e che soddisfano la formula, cioè per le quali esiste un valore di s che permette di completare la tupla di IMPIEGATI e soddisfa la condizione $s > 40$.

La stessa struttura si estende a interrogazioni più complesse, che in algebra relazionale abbiamo formulato per mezzo dell'operatore di join: avremo bisogno di più condizioni atomiche, una per ciascuna relazione coinvolta, e possiamo utilizzare variabili ripetute per indicare le condizioni di join. Per esempio, l'interrogazione che vuole *trovare le matricole dei capi degli impiegati che guadagnano più di 40 mila euro*, formulata in algebra con l'Espressione 3.2:

$$\pi_{\text{Capo}}(\text{SUPERVISIONE} \bowtie_{\text{Impiegato=Matr}} \sigma_{\text{Stipendio}>40}(\text{IMPIEGATI}))$$

può essere formulata nel calcolo con:

$$\{\text{Capo} : c \mid \text{IMPIEGATI}(\text{Matr} : m,\ \text{Nome} : n,\ \text{Età} : e,\ \text{Stipendio} : s) \wedge \text{SUPERVISIONE}(\text{Impiegato} : m,\ \text{Capo} : c) \wedge s > 40\} \quad (3.9)$$

dove la variabile m, comune alle due condizioni atomiche, realizza la stessa correlazione fra tuple specificata nel join. Anche qui potremmo utilizzare quantificatori esistenziali per tutte le variabili che non compaiono nella target list, ma, come nel caso precedente, ciò non è necessario e appesantirebbe la formulazione.

Se in un'espressione è richiesto il coinvolgimento di tuple diverse di una stessa relazione (in algebra, il join di una relazione con se stessa), è sufficiente includere nella formula più condizioni sullo stesso predicato, con variabili diverse. L'interrogazione che vuole *trovare nome e stipendio dei capi degli impiegati che guadagnano più di 40 mila euro*, realizzata in algebra con l'Espressione 3.3:

$$\pi_{\text{NomeC,StipC}}(\rho_{\text{MatrC,NomeC,StipC,EtàC} \leftarrow \text{Matr,Nome,Stip,Età}}(\text{IMPIEGATI}) \bowtie_{\text{MatrC=Capo}} \text{SUPERVISIONE} \bowtie_{\text{Impiegato=Matr}} \sigma_{\text{stipendio}>40}(\text{IMPIEGATI}))$$

viene formulata nel calcolo richiedendo, per ciascuna tupla del risultato, l'esistenza di tre tuple, una relativa a un impiegato che guadagna più di 40 mila euro, una seconda che indica chi è il suo capo e l'ultima (di nuovo nella relazione IMPIEGATI) che fornisce le informazioni di dettaglio sul capo:

$$\begin{gathered}\{\mathsf{NomeC} : nc,\ \mathsf{StipC} : sc \mid \\ \textsc{Impiegati}(\mathsf{Matr} : m,\ \mathsf{Nome} : n,\ \mathsf{Età} : e,\ \mathsf{Stipendio} : s) \wedge s > 40 \\ \textsc{Supervisione}(\mathsf{Impiegato} : m,\ \mathsf{Capo} : c) \wedge \\ \textsc{Impiegati}(\mathsf{Matr} : c,\ \mathsf{Nome} : nc,\ \mathsf{Età} : ec,\ \mathsf{Stipendio} : sc)\}\end{gathered} \tag{3.10}$$

La successiva interrogazione, *trovare gli impiegati che guadagnano più del rispettivo capo, mostrando matricola, nome e stipendio di ciascuno di essi e del capo*, differisce dalla precedente solo per la necessità di confrontare valori dello stesso attributo provenienti da tuple diverse (Espressione 3.4 in algebra), il che non causa difficoltà particolari:

$$\begin{gathered}\{\mathsf{Matr} : m,\ \mathsf{Nome} : n,\ \mathsf{Stip} : s,\ \mathsf{MatrC} : c,\ \mathsf{NomeC} : nc,\ \mathsf{StipC} : sc \mid \\ \textsc{Impiegati}(\mathsf{Matr} : m,\ \mathsf{Nome} : n,\ \mathsf{Età} : e,\ \mathsf{Stip} : s) \wedge \\ \textsc{Supervisione}(\mathsf{Impiegato} : m,\ \mathsf{Capo} : c) \wedge \\ \textsc{Impiegati}(\mathsf{Matr} : c,\ \mathsf{Nome} : nc,\ \mathsf{Età} : ec,\ \mathsf{Stip} : sc) \wedge s > sc\}\end{gathered} \tag{3.11}$$

L'ultimo esempio richiede una soluzione più complessa. Dobbiamo *trovare matricola e nome dei capi i cui impiegati guadagnano tutti più di 40 mila euro*. In algebra abbiamo utilizzato una differenza (Espressione 3.5):

$$\begin{gathered}\pi_{\mathsf{Matr,Nome}}(\textsc{Impiegati} \bowtie_{\mathsf{Matr=Capo}} \\ (\pi_{\mathsf{Capo}}(\textsc{Supervisione}) - \\ \pi_{\mathsf{Capo}}(\textsc{Supervisione} \bowtie_{\mathsf{Imp=Matr}} \sigma_{\mathsf{Stip} \leq 40}(\textsc{Impiegati})))\end{gathered}$$

Nel calcolo dobbiamo utilizzare un quantificatore. Seguendo la stessa strada seguita nell'algebra (che genera l'insieme richiesto considerando tutti i capi esclusi quelli che hanno almeno un impiegato che guadagna meno di 40 mila euro), possiamo utilizzare un quantificatore esistenziale negato (in effetti ne usiamo diversi, uno per ciascuna variabile coinvolta), trovando i capi per i quali non esiste un impiegato che guadagna non più di 40 mila euro:

$$\begin{gathered}\{\mathsf{Matr} : c,\ \mathsf{Nome} : n \mid \\ \textsc{Impiegati}(\mathsf{Matr} : c,\ \mathsf{Nome} : n,\ \mathsf{Età} : e,\ \mathsf{Stip} : s) \wedge \\ \textsc{Supervisione}(\mathsf{Impiegato} : m,\ \mathsf{Capo} : c) \wedge \\ \neg\exists m'(\exists n'(\exists e'(\exists s'(\textsc{Impiegati}(\mathsf{Matr} : m',\ \mathsf{Nome} : n',\ \mathsf{Età} : e',\ \mathsf{Stip} : s') \wedge \\ \textsc{Supervisione}(\mathsf{Impiegato} : m',\ \mathsf{Capo} : c) \wedge s' \leq 40))))\}\end{gathered} \tag{3.12}$$

In alternativa, possiamo utilizzare quantificatori universali:

$$\begin{gathered}\{\mathsf{Matr} : c,\ \mathsf{Nome} : n \mid \\ \textsc{Impiegati}(\mathsf{Matr} : c,\ \mathsf{Nome} : n,\ \mathsf{Età} : e,\ \mathsf{Stip} : s) \wedge \\ \textsc{Supervisione}(\mathsf{Impiegato} : m,\ \mathsf{Capo} : c) \wedge \\ \forall m'(\forall n'(\forall e'(\forall s'(\neg(\textsc{Impiegati}(\mathsf{Matr} : m',\ \mathsf{Nome} : n',\ \mathsf{Età} : e',\ \mathsf{Stip} : s') \wedge \\ \textsc{Supervisione}(\mathsf{Impiegato} : m',\ \mathsf{Capo} : c)) \vee s' > 40))))\}\end{gathered} \tag{3.13}$$

Questa espressione seleziona un capo c se, per ogni quadrupla di valori m', n', e', s' relativi a impiegati di c, si ha che s' è maggiore di 40. La struttura $\neg f \vee g$ corrisponde alla condizione "se f allora g" (nel nostro caso, "se m' è un impiegato avente c come capo, allora lo stipendio di m' è maggiore di 40"), in quanto è vera in tutti i casi escluso quello in cui f è vera e g è falsa.

Vale la pena di notare che le leggi di de Morgan che valgono per gli operatori dell'algebra di Boole, in modo che:

$$\neg(f \wedge g) = \neg(f) \vee \neg(g)$$

$$\neg(f \vee g) = \neg(f) \wedge \neg(g)$$

valgono, *mutatis mutandis*, anche per i quantificatori:

$$\exists x(f) = \neg(\forall x(\neg(f)))$$

$$\forall x(f) = \neg(\exists x(\neg(f)))$$

In effetti, le due formulazioni che abbiamo mostrato per l'ultima interrogazione possono essere ottenute l'una dall'altra per mezzo di queste equivalenze. Più in generale, possiamo trarre anche la conseguenza che è possibile usare una forma ridotta del calcolo (ma senza perdita di potere espressivo), in cui compaiono la negazione, un solo connettivo (per esempio la congiunzione) e un solo quantificatore (per esempio l'esistenziale, che è di più naturale comprensione).

3.2.2 Pregi e difetti del calcolo su domini

Il calcolo relazionale presenta, come dimostrato dagli esempi, aspetti interessanti, soprattutto per la dichiaratività, ma anche alcuni difetti e limitazioni, che è opportuno discutere, per arrivare a versioni più interessanti e, soprattutto, significative dal punto di vista pratico.

In primo luogo, notiamo che il calcolo ammette espressioni che hanno veramente poco senso, almeno dal punto di vista pratico. Per esempio, l'espressione:

$$\{A_1 : x_1,\ A_2 : x_2 \mid R(A_1 : x_1) \wedge x_2 = x_2\}$$

produce come risultato una relazione su A_1 e A_2 costituita da tuple il cui valore su A_1 compare nella relazione R e il valore su A_2 è un qualunque valore del dominio (in

quanto non viene posta su di esso alcuna condizione, dato che la condizione $x_2 = x_2$ è sempre vera). In particolare, se cambia il dominio, per esempio gli interi compresi fra 0 e 99 o gli interi compresi fra 0 e 999, cambia anche la risposta all'interrogazione. Se il dominio è infinito, anche la risposta è infinita, il che è indesiderabile. Un discorso analogo può essere fatto per l'espressione:

$$\{A_1 : x_1 \mid \neg(R(A_1 : x_1))\}$$

il cui risultato contiene i valori del dominio che non compaiono in R. Al tempo stesso, possiamo notare che, per tutte le espressioni viste in precedenza (che tra l'altro sono significative da un punto di vista pratico), il valore, su un'istanza della base di dati, è lo stesso qualunque sia il dominio (purché contenga almeno i valori presenti nell'istanza e quelli nell'espressione).

Pertanto, può essere utile introdurre il seguente concetto: un'espressione di un linguaggio di interrogazione è *indipendente dal dominio* se il suo risultato, su ciascuna istanza di base di dati, non varia al variare del dominio rispetto al quale l'espressione è valutata. Un linguaggio è indipendente dal dominio se tutte le sue espressioni sono indipendenti dal dominio. Il requisito dell'indipendenza dal dominio è chiaramente fondamentale per i linguaggi reali, perché nella maggior parte dei casi le espressioni dipendenti dal dominio non hanno utilità pratica e possono produrre risultati di grandi dimensioni.

Sulla base delle espressioni viste prima, possiamo dire che il calcolo relazionale non è indipendente dal dominio. Al tempo stesso, si può vedere facilmente che l'algebra relazionale è indipendente dal dominio, perché costruisce i risultati a partire dalle relazioni nella base di dati, senza mai far riferimento ai domini degli attributi (i valori nei risultati compaiono tutti nell'istanza cui l'espressione viene applicata).

Se a questo punto diciamo che due linguaggi di interrogazione sono *equivalenti* quando per ogni espressione dell'uno esiste un'espressione dell'altro a essa equivalente e viceversa, possiamo affermare che algebra e calcolo non sono equivalenti, perché il calcolo, al contrario dell'algebra, ammette espressioni dipendenti dal dominio. Peraltro, se limitiamo la nostra attenzione al sottoinsieme del calcolo relazionale costituito dalle sole espressioni indipendenti dal dominio, otteniamo un linguaggio equivalente all'algebra relazionale. Infatti:

- per ogni espressione del calcolo relazionale che sia indipendente dal dominio esiste un'espressione dell'algebra relazionale equivalente a essa;
- per ogni espressione dell'algebra relazionale esiste un'espressione del calcolo relazionale equivalente a essa (e di conseguenza indipendente dal dominio).

La dimostrazione di equivalenza va oltre gli obiettivi di questo testo, ma possiamo accennare che è sostanzialmente costruttiva, basata, in ciascuno dei due versi, su una induzione sulla struttura dell'espressione. In particolare, esiste una corrispondenza fra selezioni e condizioni semplici, fra proiezioni e quantificazioni esistenziali, fra join e congiunzioni, fra unioni e disgiunzioni. I quantificatori universali possono essere ignorati in quanto ricondotti, attraverso le leggi di de Morgan, a quantificatori esistenziali.

Oltre al problema della possibile dipendenza dal dominio, il calcolo relazionale presenta un altro svantaggio, quello di richiedere numerose variabili, spesso una per ciascun attributo di ciascuna relazione coinvolta. Quando poi sono necessarie quantificazioni, come abbiamo visto negli esempi, anche i quantificatori si moltiplicano. In effetti, le uniche realizzazioni pratiche di linguaggi almeno in parte basati sul calcolo su domini, che vanno sotto il nome di *Query-by-Example* (QBE), utilizzano un'interfaccia grafica che libera l'utente dalla necessità di specificare dettagli tediosi. Nell'appendice dedicata al sistema Access (disponibile sul sito web del libro) accenniamo a una versione del QBE.

Per superare i limiti del calcolo su domini, è stata proposta un'altra versione del calcolo relazionale, in cui le variabili, anziché denotare singoli valori, denotano tuple. Molto spesso il numero di variabili si riduce notevolmente, perché si ha una variabile per ciascuna relazione coinvolta. Peraltro, diventa poi necessario associare una struttura (insieme di attributi su cui è definita) a ciascuna variabile e realizzare opportunamente le operazioni di confronto (in modo che facciano riferimento a singoli valori, cioè a singole componenti delle tuple denotate dalle variabili). Potremmo a questo punto definire un *calcolo relazionale su tuple* perfettamente corrispondente al calcolo su domini, ed equivalente a esso, quindi anche con la limitazione della dipendenza dal dominio. Preferiamo però omettere la presentazione di questo linguaggio, per passare direttamente a un linguaggio che, recependo le caratteristiche del calcolo su tuple, superi al tempo stesso il difetto della dipendenza dal dominio, attraverso la diretta associazione delle variabili alle relazioni della base di dati. A esso è dedicato il paragrafo seguente.

3.2.3 Calcolo su tuple con dichiarazioni di range

Le espressioni del *calcolo su tuple con dichiarazioni di range* hanno la forma:

$$\{\mathcal{T} \mid \mathcal{L} \mid f\}$$

dove:

- $\mathcal{T}$ è la *target list* (lista degli obiettivi dell'interrogazione), con elementi del tipo $Y : x.Z$ (o semplicemente $x.Z$, abbreviazione per $Z : x.Z$), con x variabile e Y e Z sequenze di attributi (di pari lunghezza); gli attributi in Z devono comparire nello schema della relazione che costituisce il *range* (cioè il campo di variabilità) di x; si può anche scrivere $x.*$, come abbreviazione di $X : x.X$, dove il range della variabile x è una relazione sull'insieme di attributi X;
- $\mathcal{L}$ è la *range list*, che elenca (ciascuna una e una sola volta) le variabili libere della formula f con i relativi range: infatti, $\mathcal{L}$ è una lista di elementi del tipo $x(R)$, con x variabile e R nome di relazione;
- f è una formula con:
  - atomi del tipo $x.A\theta c$ o $x_1.A_1\theta x_2.A_2$, che confrontano, rispettivamente, il valore di x sull'attributo A con la costante c e il valore di x_1 su A_1 con quello di x_2 su A_2;
  - connettivi come nel calcolo su domini;

- quantificatori che associano i range alle relative variabili:

$$\exists x(R)(f) \quad \forall x(R)(f)$$

Intuitivamente, $\exists x(R)(f)$ significa "esiste nella relazione R una tupla x che soddisfa la formula f".

Va sottolineato il ruolo giocato dalle dichiarazioni di range (nella range list e nelle quantificazioni), che, introducendo le variabili, specificano che esse possono assumere come valore solo tuple nella relazione rispettivamente associata e garantiscono così l'indipendenza dal dominio. Di conseguenza, questo linguaggio non ha bisogno di condizioni atomiche come quelle viste nel calcolo su domini, che specificano l'appartenenza di una tupla a una relazione.

Mostriamo come possono essere espresse in questo linguaggio le varie interrogazioni che abbiamo già formulato in algebra e in calcolo su domini.

La prima interrogazione, che richiede *matricola, nome, età e stipendio degli impiegati che guadagnano più di 40 mila euro*, diventa molto compatta e chiara (cfr. con l'Espressione 3.6):

$$\{i.* \mid i(\text{IMPIEGATI}) \mid i.\text{Stipendio} > 40\} \tag{3.14}$$

Per produrre solo alcuni degli attributi, *matricola, nome ed età degli impiegati che guadagnano più di 40 mila euro* (Espressione 3.1 in algebra e 3.8 in calcolo su domini), è sufficiente modificare la target list:

$$\{i.(\text{Matr}, \text{Nome}, \text{Età}) \mid i(\text{IMPIEGATI}) \mid i.\text{Stipendio} > 40\} \tag{3.15}$$

Per interrogazioni che coinvolgono più relazioni, sono necessarie più variabili, con specifica delle condizioni di correlazione sugli attributi. L'interrogazione che vuole *trovare le matricole dei capi degli impiegati che guadagnano più di 40 mila euro* (3.2 in algebra e 3.9 in calcolo su domini), può essere formulata con:

$$\{s.\text{Capo} \mid i(\text{IMPIEGATI}), s(\text{SUPERVISIONE}) \mid i.\text{Matr} = s.\text{Impiegato} \wedge i.\text{Stipendio} > 40\} \tag{3.16}$$

Notiamo come la formula preveda la congiunzione di due condizioni atomiche, una che corrisponde alla condizione di join ($i.\text{Matr} = s.\text{Impiegato}$) e l'altra alla solita condizione di selezione ($i.\text{Stipendio} > 40$).

Nel caso delle espressioni corrispondenti al join di una relazione con se stessa, abbiamo più variabili aventi la stessa relazione come range. L'interrogazione: *trovare nome e stipendio dei capi degli impiegati che guadagnano più di 40 mila euro* (Espressioni 3.3 e 3.10), può essere realizzata con l'espressione:

$$\{\text{NomeC}, \text{StipC} : i'.(\text{Nome}, \text{Stip}) \mid i'(\text{IMPIEGATI}), s(\text{SUPERVISIONE}), i(\text{IMPIEGATI}) \mid i'.\text{Matr} = s.\text{Capo} \wedge s.\text{Impiegato} = i.\text{Matr} \wedge i.\text{Stipendio} > 40\} \tag{3.17}$$

In modo analogo troviamo anche *gli impiegati che guadagnano più del rispettivo capo, mostrando matricola, nome e stipendio di ciascuno di essi e del capo*, (3.4 in algebra e 3.11 in calcolo su domini):

$$\{i.(\mathsf{Nome}, \mathsf{Matr}, \mathsf{Stip}),\ \mathsf{NomeC}, \mathsf{MatrC}, \mathsf{StipC} : i'.(\mathsf{Nome}, \mathsf{Matr}, \mathsf{Stip}) \mid$$
$$i(\textsc{Impiegati}),\ s(\textsc{Supervisione}),\ i'(\textsc{Impiegati}) \mid$$
$$i.\mathsf{Matr} = s.\mathsf{Impiegato} \wedge s.\mathsf{Capo} = i'.\mathsf{Matr} \wedge i.\mathsf{Stipendio} > i'.\mathsf{Stipendio}\} \quad (3.18)$$

Le interrogazioni con i quantificatori mostrano appieno la maggiore sinteticità e praticità del calcolo su tuple con dichiarazioni di range. L'interrogazione che richiede di *trovare matricola e nome dei capi i cui impiegati guadagnano tutti più di 40 mila euro* (Espressione 3.5 in algebra ed Espressione 3.12 o 3.13 in calcolo su domini) si esprime con un numero assai inferiore di quantificatori e variabili. Abbiamo di nuovo varie alternative, sulla base dell'uso dei due quantificatori e della negazione. Con quantificatori universali:

$$\{i.(\mathsf{Matr}, \mathsf{Nome}) \mid i(\textsc{Impiegati}),\ s(\textsc{Supervisione}) \mid$$
$$i.\mathsf{Matr} = s.\mathsf{Capo} \wedge \forall i'(\textsc{Impiegati})(\forall s'(\textsc{Supervisione})$$
$$(\neg(s.\mathsf{Capo} = s'.\mathsf{Capo} \wedge s'.\mathsf{Impiegato} = i'.\mathsf{Matr}) \vee i'.\mathsf{Stipendio} > 40))\} \quad (3.19)$$

Con quantificatori esistenziali negati:

$$\{i.(\mathsf{Matr}, \mathsf{Nome}) \mid i(\textsc{Impiegati}),\ s(\textsc{Supervisione}) \mid$$
$$i.\mathsf{Matr} = s.\mathsf{Capo} \wedge \neg(\exists i'(\textsc{Impiegati})(\exists s'(\textsc{Supervisione})$$
$$(s.\mathsf{Capo} = s'.\mathsf{Capo} \wedge s'.\mathsf{Impiegato} = i'.\mathsf{Matr} \wedge i'.\mathsf{Stipendio} \leq 40)))\} \quad (3.20)$$

Purtroppo, il calcolo su tuple con dichiarazioni di range non permette di esprimere tutte le interrogazioni che possono essere formulate in algebra relazionale (o, equivalentemente, nel calcolo su domini). In particolare, le interrogazioni i cui risultati possono provenire indifferentemente da due o più relazioni (che in algebra realizziamo con l'operatore di unione) non possono essere espresse in questa versione del calcolo: infatti, i risultati sono costruiti a partire da tutte le variabili libere, i cui range sono definiti nella target list, e ogni variabile ha come range una sola relazione. Consideriamo per esempio la semplice unione di due relazioni sugli stessi attributi: date $R_1(AB)$ e $R_2(AB)$, vogliamo formulare l'interrogazione che in algebra esprimeremmo con l'unione di R_1 e R_2. Se l'espressione avesse due variabili libere, allora ogni tupla del risultato dovrebbe corrispondere a una tupla di ciascuna delle relazioni, il che non è necessario, perché l'unione richiede alle tuple nel risultato di comparire in almeno uno degli operandi, non necessariamente in entrambi. Se viceversa l'espressione avesse una sola variabile libera, questa dovrebbe far riferimento a una sola delle relazioni, senza acquisire tuple dall'altra per il risultato.

Per questo motivo, SQL, il linguaggio pratico effettivamente utilizzato per l'interrogazione di basi di dati, che vedremo in dettaglio nel Capitolo 4, e che è basato sul calcolo su tuple con dichiarazioni di range, prevede un costrutto esplicito di unione, per esprimere interrogazioni che altrimenti risulterebbero non esprimibili.

Notiamo che se permettessimo di associare a una variabile un range costituito da più relazioni, risolveremmo il problema della semplice unione di due relazioni, ma non riusciremmo comunque a formulare unioni complesse i cui operandi siano sottoespressioni non direttamente corrispondenti a schemi di relazioni. Per esempio, date due relazioni $R_1(ABC)$ e $R_2(BCD)$, l'unione delle loro proiezioni su BC:

$$\pi_{BC}(R_1) \cup \pi_{BC}(R_2)$$

non potrebbe essere espressa, perché le due relazioni hanno schemi diversi, quindi non può una sola variabile essere associata a entrambe.

Sottolineiamo che, mentre l'operatore di unione non è esprimibile in questa versione del calcolo relazionale, gli operatori di intersezione e differenza risultano esprimibili.

- L'intersezione richiede che le tuple del risultato appartengano a entrambi gli operandi, quindi si può costruire il risultato a partire da una relazione, richiedendo l'esistenza di una tupla uguale nell'altra relazione; per esempio, l'intersezione:

  $$\pi_{BC}(R_1) \cap \pi_{BC}(R_2)$$

  può essere espressa con:

  $$\{x_1.BC \mid x_1(R_1) \mid \exists x_2(R_2)(x_1.B = x_2.B \wedge x_1.C = x_2.C)\}$$

- In modo simile, la differenza, che produce le tuple di un operando non contenute nell'altro, può essere specificata richiedendo appunto le tuple del minuendo che non compaiono nel sottraendo; per esempio:

  $$\pi_{BC}(R_1) - \pi_{BC}(R_2)$$

  può essere espressa con:

  $$\{x_1.BC \mid x_1(R_1) \mid \neg\exists x_2(R_2)(x_1.B = x_2.B \wedge x_1.C = x_2.C)\}$$

3.3 Datalog

Concludiamo il capitolo discutendo brevemente un altro linguaggio di interrogazione per basi di dati che ha riscosso un notevole interesse nella comunità scientifica a partire dalla metà degli anni Ottanta, pur non raggiungendo la diffusione a livello di tecnologia disponibile sul mercato. L'idea fondamentale su cui si basa il linguaggio *Datalog* è quella di adattare alle basi di dati il linguaggio di programmazione logica *Prolog*. Non abbiamo ovviamente qui la possibilità di illustrare in dettaglio il Datalog, né tantomeno il Prolog, ma possiamo indicare gli aspetti più interessanti, soprattutto in termini di confronto con gli altri linguaggi visti in questo capitolo.

Sintatticamente, nella versione base, il Datalog è una versione semplificata del Prolog,[4] linguaggio basato sul calcolo dei predicati del primo ordine, ma con un approccio diverso rispetto al calcolo relazionale discusso in precedenza. Abbiamo in Datalog due tipi di predicati:

[4] Per chi conosce il Prolog, possiamo dire che in Datalog non sono previsti simboli di funzione.

- i predicati *estensionali*, che corrispondono alle relazioni nella base di dati;
- i predicati *intensionali*, che sono specificati (ma non materializzati) per mezzo di regole logiche (le *regole Datalog* che vedremo fra poco). Concettualmente, questi predicati definiscono viste (relazioni virtuali) sulla base di dati.

Le *regole Datalog* hanno la forma:

$$testa \leftarrow corpo$$

in cui:

- la *testa* è un predicato atomico simile a quelli utilizzati nel calcolo relazionale su domini: $R(A_1 : a_1, \ldots, A_p : a_p)$, dove però ciascuno degli a_i può essere una costante o una variabile;
- il *corpo* è una lista di condizioni atomiche dello stesso tipo e/o di condizioni di confronto fra variabili o fra variabili e costanti.

Sono imposte le seguenti condizioni:

- i predicati estensionali possono comparire solo nel corpo delle regole;
- se una variabile compare nella testa di una regola, allora deve comparire anche nel corpo della stessa regola;
- se una variabile compare in un atomo di confronto, allora deve comparire anche in un atomo nel corpo della stessa regola.

La prima condizione garantisce che non vi sia il tentativo di ridefinire le relazioni memorizzate nella base di dati, mentre le altre due hanno lo scopo di garantire una proprietà che è analoga (in questo contesto) all'indipendenza dal dominio discussa a proposito del calcolo relazionale.

Una caratteristica fondamentale del Datalog, che lo distingue dagli altri linguaggi finora visti, è la *ricorsività*: è possibile che un predicato intensionale sia definito in termini di se stesso (direttamente o indirettamente). Torneremo su questo aspetto fra poco.

Le interrogazioni Datalog sono specificate semplicemente per mezzo di atomi $R(A_1 : a_1, \ldots, A_p : a_p)$ (preceduti talvolta da un punto interrogativo "?", per sottolineare appunto che si tratta di interrogazioni), che producono come risultato le tuple della relazione R che possono essere ottenute sostituendo correttamente le variabili. Per esempio, l'interrogazione:

?IMPIEGATI(Matr : m, Nome : n, Età : 30, Stipendio : s)

restituisce gli impiegati che hanno trenta anni. Per costruire interrogazioni più complesse è necessario ricorrere a regole. Per esempio, per *trovare le matricole dei capi degli impiegati che guadagnano più di 40 mila euro*, formulata in algebra con l'Espressione 3.2 e in calcolo su domini con la 3.9, definiamo un predicato intensionale CAPIDEIRICCHI, con la regola:

CAPIDEIRICCHI(Capo : c) $\leftarrow$
IMPIEGATI(Matr : m, Nome : n, Età : e, Stipendio : s),
SUPERVISIONE(Impiegato : m, Capo : c), $s > 40$ (3.21)

Per valutare un'interrogazione come questa (come qualunque interrogazione che coinvolga predicati intensionali), è necessario definire la semantica delle regole. L'idea di base è che il corpo di una regola va considerato come la congiunzione degli atomi che in esso compaiono, quindi la regola può essere valutata come un'espressione del calcolo su domini (in cui appunto il corpo, sostituendo le virgole con *and*, diventa la formula, e la testa, a parte il nome del predicato intensionale, la target list). La regola 3.21 definisce la relazione intensionale CAPIDEIRICCHI come costituita dalle stesse tuple che compaiono nel risultato dell'Espressione 3.9 del calcolo, che ha appunto la struttura sopra citata:

$$\{\text{Capo} : c \mid \text{IMPIEGATI}(\text{Matr} : m,\ \text{Nome} : n,\ \text{Età} : e,\ \text{Stipendio} : s) \wedge \text{SUPERVISIONE}(\text{Impiegato} : m,\ \text{Capo} : c) \wedge s > 40\}$$

In modo analogo possiamo scrivere regole (con predicati intensionali ausiliari) per molte delle interrogazioni che abbiamo visto nei paragrafi precedenti. In assenza di definizioni ricorsive, la semantica del Datalog è quindi molto semplice, nel senso che i vari predicati intensionali possono essere calcolati per mezzo di espressioni simili a quelle del calcolo. In effetti, con la definizione finora illustrata per il Datalog, non è possibile formulare tutte le interrogazioni esprimibili nel calcolo (e nell'algebra), perché non è disponibile un costrutto che corrisponda al quantificatore universale (o alla negazione nel senso pieno del termine). In effetti, si può dimostrare che:

- il Datalog non ricorsivo è equivalente al calcolo su domini senza negazione né quantificazione universale.[5]

Per far acquisire al Datalog lo stesso potere espressivo del calcolo è necessario aggiungere alla struttura base la possibilità di includere nel corpo, non solo condizioni atomiche, ma anche negazioni di condizioni atomiche (che indicheremo con il simbolo NOT).

Solo in questo modo è possibile esprimere l'interrogazione che richiede di *trovare matricola e nome dei capi i cui impiegati guadagnano tutti più di 40 mila euro*. Espressione 3.12:

$$\begin{gathered}\{\text{Matr} : c,\ \text{Nome} : n \mid \\ \text{IMPIEGATI}(\text{Matr} : c,\ \text{Nome} : n,\ \text{Età} : e,\ \text{Stip} : s) \wedge \\ \text{SUPERVISIONE}(\text{Impiegato} : m,\ \text{Capo} : c) \wedge \\ \neg\exists m'(\exists n'(\exists e'(\exists s'(\text{IMPIEGATI}(\text{Matr} : m',\ \text{Nome} : n',\ \text{Età} : e',\ \text{Stip} : s') \wedge \\ \text{SUPERVISIONE}(\text{Impiegato} : m',\ \text{Capo} : c) \wedge s' \leq 40))))\}\end{gathered}$$

[5] Per semplicità, in questo paragrafo, usiamo il termine "calcolo relazionale su domini" per riferirci, secondo la discussione già fatta nel Paragrafo 3.2.2, al "sottoinsieme del calcolo costituito dalle sole espressioni indipendenti dal dominio".

Procediamo definendo un predicato per i capi che non soddisfano la condizione:

CAPIDINONRICCHI(Capo : c) $\leftarrow$
SUPERVISIONE(Impiegato : m, Capo : c),
IMPIEGATI(Matr : m, Nome : n, Età : e, Stip : s), $s \leq 40$

quindi utilizziamo questo predicato in forma negata:

CAPISOLODIRICCHI(Matr : c, Nome : n) $\leftarrow$
IMPIEGATI(Matr : c, Nome : n, Età : e, Stip : s),
SUPERVISIONE(Impiegato : m, Capo : c),
NOT CAPIDINONRICCHI(Capo : c)

Si può dimostrare che:

- il Datalog non ricorsivo con negazione è equivalente al calcolo su domini.

Maggiore espressività viene ottenuta utilizzando infine regole ricorsive. Per esempio, sempre sulla base di dati con le relazioni IMPIEGATI e SUPERVISIONE, è possibile definire il predicato intensionale SUPERIORE, che descrive per ogni impiegato, il capo, il capo del capo, e così via, senza limiti. Allo scopo, abbiamo bisogno di due regole:

SUPERIORE(Impiegato : i, SuperCapo : c) $\leftarrow$
SUPERVISIONE(Impiegato : i, Capo : c)

SUPERIORE(Impiegato : i, SuperCapo : c) $\leftarrow$
SUPERVISIONE(Impiegato : i, Capo : c'),
SUPERIORE(Impiegato : c', SuperCapo : c)

La seconda regola è in effetti ricorsiva, in quanto definisce la relazione SUPERIORE in termini di se stessa. Per valutare questa regola, non possiamo procedere come visto finora, perché una singola valutazione del corpo non sarebbe sufficiente per calcolare completamente il predicato ricorsivo. Esistono varie teniche per definire formalmente la semantica in questo caso, ma la loro discussione sarebbe certamente oltre gli scopi di questo testo. Accenniamo alla modalità più semplice, che si basa sulla tecnica di *punto fisso* (dall'inglese *fixpoint*): le regole relative al predicato intensionale ricorsivo vengono valutate più volte, interrompendo il processo quando l'ultima iterazione non genera nuovi risultati. Nel nostro caso, la prima iterazione genererebbe una relazione SUPERIORE uguale alla relazione estensionale SUPERVISIONE, contenente cioè i capi degli impiegati. Al secondo passo verrebbero aggiunti i capi dei capi, al terzo i capi dei capi dei capi, e così via. È evidente che interrogazioni di questo genere non possono essere formulate in algebra relazionale (e analogamente in calcolo) perché non avremmo modo di specificare quante volte deve essere eseguito il join della relazione SUPERVISIONE con se stessa, mentre l'algebra ci richiede espressioni predefinite.

Per concludere, citiamo semplicemente il fatto che regole ricorsive con la negazione sono difficili da valutare, perché il punto fisso può non essere raggiungibile: perciò vengono imposte limitazioni alla presenza di negazione nelle regole ricorsive. In ogni caso, è possibile individuare un sottoinsieme ben utilizzabile del Datalog ricorsivo con la negazione che è strettamente più espressivo del calcolo e dell'algebra relazionale, in quanto:

- per ogni espressione dell'algebra esiste un'espressione del Datalog con negazione equivalente a essa;
- esistono espressioni del Datalog ricorsivo per le quali non esistono espressioni equivalenti dell'algebra e del calcolo.

Note bibliografiche

I concetti illustrati in questo capitolo possono essere approfonditi sugli stessi testi già segnalati con riferimento al capitolo precedente: quelli di Elmasri e Navathe [41] e Silberchatz, Korth e Sudarshan [71] per trattazioni generali e quelli di Atzeni, Batini e De Antonellis [5], Ullman [79], Maier [56], Atzeni e De Antonellis [7], Abiteboul, Hull e Vianu [1] per approfondimenti più formali e teorici. Per il Datalog, si può consultare il testo di Ceri, Gottlob e Tanca [22].

Esercizi

Soluzioni sul sito http://www.atenesonline.bruzzo.it

3.1 Considerare una relazione $R(\underline{A}, \underline{B}, \underline{C}, D, E)$. Indicare quali delle seguenti proiezioni hanno certamente lo stesso numero di ennuple di R:

1. $\pi_{ABCD}(R)$
2. $\pi_{AC}(R)$
3. $\pi_{BC}(R)$
4. $\pi_{C}(R)$
5. $\pi_{CD}(R)$.

3.2 Considerare le relazioni $R_1(\underline{A}, B, C)$ e $R_2(\underline{D}, E, F)$ aventi rispettivamente cardinalità N_1 e N_2. Assumere che sia definito un vincolo di integrità referenziale fra l'attributo C di R_1 e la chiave D di R_2. Indicare la cardinalità di ciascuno dei seguenti join (specificare l'intervallo nel quale essa può variare):

1. $R_1 \bowtie_{A=D} R_2$
2. $R_1 \bowtie_{C=D} R_2$
3. $R_1 \bowtie_{A=F} R_2$
4. $R_1 \bowtie_{B=E} R_2$

3.3 Considerare le seguenti relazioni (tutte senza valori nulli):

- $R_1(\underline{A}, B, C)$, con vincolo di integrità referenziale fra C e R_2 e con cardinalità $N_1 = 100$
- $R_2(\underline{D}, E, F)$, con vincolo di integrità referenziale fra F e R_3 e con cardinalità $N_2 = 200$
- $R_3(\underline{G}, H, I)$, con cardinalità $N_3 = 50$

Indicare la cardinalità del risultato di ciascuna delle seguenti espressioni (specificando l'intervallo nel quale essa può variare):

1. $\pi_{AB}(R_1)$
2. $\pi_{E}(R_2)$
3. $\pi_{BC}(R_1)$
4. $\pi_{G}(R_3)$
5. $R_1 \bowtie_{A=D} R_2$
6. $R_1 \bowtie_{C=D} R_2$
7. $R_3 \bowtie_{I=A} R_1$
8. $(R_3 \bowtie_{I=A} R_1) \bowtie_{C=D} R_2$
9. $(R_3 \bowtie_{I=A} R_1) \bowtie_{C=E} R_2$

3.4 Date le relazioni $R_1(A, B, C)$, $R_2(E, F, G, H)$, $R_3(J, K)$, $R_4(L, M)$ aventi rispettivamente cardinalità N_1, N_2, N_3 e N_4 quali vincoli di chiave e di integrità referenziale vanno definiti (se possibile) affinché nei casi seguenti valgano le condizioni indicate?

1. $|R_1 \bowtie_{B=G} R_2| = N_1$
2. $|R_2 \bowtie_{G=B} R_1| = N_1$
3. $|\pi_{J}(R_3)| = N_3$
4. $|\pi_{J}(R_3)| < N_3$
5. $|\pi_{L}(R_4) \bowtie_{L=J} R_3| = N_4$
6. $|R_4 \bowtie_{M=K} R_3| = N_3$
7. $|R_1 \bowtie_{BC=GK} R_2| = N_2$
8. $|R_1 \bowtie_{BC=GH} R_2| = N_1$
9. $0 \leq |R_1 \bowtie_{A=F} R_2| \leq N_1 \cdot N_2$
10. $|R_1 \bowtie_{A=F} R_2| = N_1 \cdot N_2$

3.5 Con riferimento ai punti 1 e 2 dell'esercizio precedente, considerando i vincoli di integrità imposti in ogni punto spiegare le differenze che si avrebbero nei risultati delle operazioni nel caso di join destro e join sinistro e come cambia di conseguenza la cardinalità del risultato.

3.6 Considerare lo schema di base di dati contenente le relazioni:

FILM(CodiceFilm,Titolo,Regista,Anno,CostoNoleggio)
ARTISTI(CodiceAttore,Cognome,Nome,Sesso,DataNascita,Nazionalità)
INTERPRETAZIONI(CodiceFilm,CodiceAttore,Personaggio)

1. Mostrare una base di dati su questo schema per la quale i join fra le varie relazioni siano tutti completi.
2. Supponendo che esistano due vincoli di integrità referenziale fra la relazione INTERPRETAZIONI e le altre due, discutere i possibili casi di join non completo.
3. Mostrare un prodotto cartesiano che coinvolga relazioni in questa base di dati.
4. Mostrare una base di dati per la quale uno (o più) dei join sia vuoto.

3.7 Con riferimento allo schema nell'Esercizio 3.6, formulare in algebra relazionale, in calcolo su domini, in calcolo su tuple e in Datalog le interrogazioni che trovano:

1. i titoli dei film nei quali Henry Fonda sia stato interprete;
2. i titoli dei film per i quali il regista sia stato anche interprete;
3. i titoli dei film in cui gli attori noti siano tutti dello stesso sesso.

3.8 Si consideri lo schema di base di dati che contiene le seguenti relazioni:

DEPUTATI(Codice,Cognome,Nome,Commissione,Provincia,Collegio)
COLLEGI (Provincia,Numero,Nome)
PROVINCE (Sigla,Nome,Regione)
REGIONI (Codice,Nome)
COMMISSIONI (Numero,Nome,Presidente)

Formulare in algebra relazionale, in calcolo su domini e in calcolo su tuple le seguenti interrogazioni:

1. trovare nome e cognome dei presidenti di commissioni cui partecipa almeno un deputato eletto in una provincia della Sicilia;
2. trovare nome e cognome dei deputati della commissione Bilancio;
3. trovare nome, cognome e provincia di elezione dei deputati della commissione Bilancio;
4. trovare nome, cognome, provincia e regione di elezione dei deputati della commissione Bilancio;
5. trovare le regioni in cui vi sia un solo collegio, indicando il nome e cognome del deputato ivi eletto;
6. trovare i collegi di una stessa regione in cui siano stati eletti deputati con lo stesso nome proprio.

3.9 Mostrare come le interrogazioni nell'Esercizio 3.8 possano trarre vantaggio, nella specifica, dalla definizione di viste.

3.10 Si consideri lo schema di base di dati sulle relazioni:

MATERIE(Codice,Facoltà,Denominazione,Professore)
STUDENTI(Matricola,Cognome,Nome,Facoltà)
PROFESSORI(Matricola,Cognome,Nome)
ESAMI(Studente,Materia,Voto,Data)
PIANIDISTUDIO(Studente,Materia,Anno)

Formulare, in algebra relazionale, in calcolo su domini, in calcolo su tuple e in Datalog le interrogazioni che producono

1. gli studenti che hanno riportato in almeno un esame una votazione pari a 30, mostrando, per ciascuno di essi, nome e cognome e data della prima di tali occasioni;
2. per ogni insegnamento della facoltà di ingegneria, gli studenti che hanno superato l'esame nell'ultima seduta svolta;
3. gli studenti che hanno superato tutti gli esami previsti dal rispettivo piano di studio;
4. per ogni insegnamento della facoltà di lettere, lo studente (o gli studenti) che hanno superato l'esame con il voto più alto;
5. gli studenti che hanno in piano di studio solo insegnamenti della propria facoltà;
6. nome e cognome degli studenti che hanno sostenuto almeno un esame con un professore che ha il loro stesso nome proprio.

3.11 Con riferimento al seguente schema di base di dati:

CITTÀ(Nome, Regione, Abitanti)
ATTRAVERSAMENTI(Città, Fiume)
FIUMI(Fiume, Lunghezza)

formulare, in algebra relazionale, in calcolo su domini, in calcolo su tuple e in Datalog le seguenti interrogazioni:

1. visualizzare nome, regione e abitanti per le città che (i) hanno più di 50 000 abitanti e (ii) sono attraversate dal Po o dall'Adige;
2. trovare le città che sono attraversate da (almeno) due fiumi, visualizzando il nome della città e quello del più lungo di tali fiumi.

3.12 Con riferimento al seguente schema di base di dati:

AFFLUENZA(Affluente,Fiume)
FIUMI(Fiume, Lunghezza)

formulare l'interrogazione in Datalog che trova tutti gli affluenti, diretti e indiretti dell'Adige.

3.13 Si consideri lo schema relazionale composto dalle seguenti relazioni:

PROFESSORI(Codice,Cognome,Nome)
CORSI(Codice,Denominazione,Professore)
STUDENTI(Matricola,Cognome,Nome)
ESAMI(Studente,Corso,Data,Voto)

Formulare, con riferimento a tale schema, le espressioni dell'algebra, del calcolo relazionale su tuple e del Datalog, che producano:

1. gli esami superati dallo studente Pico Della Mirandola (supposto unico), con indicazione, per ciascuno, della denominazione del corso, del voto e del cognome del professore;
2. i professori che tengono due corsi (e non più di due), con indicazione di cognome e nome del professore e denominazione dei due corsi.

3.14 Considerare uno schema relazionale contenente le relazioni:

$$R_1(ABC), R_2(DG), R_3(EF)$$

Formulare in calcolo relazionale su tuple e su domini l'interrogazione realizzata in algebra relazionale dalla seguente espressione:

$$(R_3 \bowtie_{G=E} R_2) \cup \rho_{DG \leftarrow AC}(\pi_{ACEF}(R_1 \bowtie_{B=F} R_3))$$

3.15 Con riferimento allo stesso schema dell'Esercizio 3.14, formulare in algebra relazionale le interrogazioni realizzate in calcolo su domini dalle seguenti espressioni:

$$\{H : g, B : b \mid R_1(A : a, B : b, C : c) \wedge R_2(D : c, G : g)\}$$

$$\{A : a, B : b \mid R_2(D : a, G : b) \wedge R_3(E : a, F : b)\}$$

$$\{A : a, B : b \mid R_1(A : a, B : b, C : c) \wedge \exists a'(R_1(A : a', B : b, C : c) \wedge a \neq a')\}$$

$$\{A : a, B : b \mid R_1(A : a, B : b, C : c) \wedge \forall a'(\neg R_1(A : a', B : b, C : c)) \vee a = a'\}$$

$$\{A : a, B : b \mid R_1(A : a, B : b, C : c) \wedge \neg\exists a'(R_1(A : a', B : b, C : c)) \wedge a \neq a'\}$$

3.16 Facendo riferimento allo schema:

$$R_1(AB), R_2(CDE), R_3(FGH)$$

trasformare la seguente espressione dell'algebra:

$$\pi_{ADH}(\sigma_{(B=C)\wedge(E=F)\wedge(A>20)\wedge(G=10)}((R_1 \bowtie R_3) \bowtie R_2)$$

con l'obiettivo di ridurre le dimensioni dei risultati intermedi.

3.17 Considerare la seguente base di dati relazionale:

- FARMACI(Codice, NomeFarmaco, PrincipioAttivo, Produttore, Prezzo)
- PRODUTTORI(CodProduttore, Nome, Nazione)
- SOSTANZE(ID, NomeSostanza, Categoria)

Con vincoli di integrità referenziale tra Produttore e la relazione PRODUTTORI, tra PrincipioAttivo e la relazione SOSTANZE. Formulare in algebra relazionale le seguenti interrogazioni:

- l'interrogazione che fornisce, per i farmaci il cui principio attivo è nella categoria "sulfamidico," il nome del farmaco e quello del suo produttore;
- l'interrogazione che fornisce, per i farmaci con produttore italiano, il nome del farmaco e quello della sostanza del suo principio attivo.

4 SQL: concetti base

SQL è il linguaggio di riferimento per le basi di dati relazionali. Il nome SQL[1] rappresentava originariamente l'acronimo di *Structured Query Language*, ma lo standard specifica ora che SQL deve essere considerato come un nome proprio. SQL era originariamente il linguaggio di interrogazione del DBMS relazionale *System R*, sviluppato presso il laboratorio di ricerca IBM di San José in California nella seconda metà degli anni Settanta. Il linguaggio è stato poi adottato da molti altri sistemi ed è stato oggetto di un'intensa attività di standardizzazione.

SQL è ben più di un linguaggio per scrivere interrogazioni. Contiene infatti al suo interno sia le funzionalità di un *Data Definition Language*, DDL (con un insieme di comandi per la definizione dello schema di una base di dati relazionale), sia quelle di un *Data Manipulation Language*, DML (con un insieme di comandi per la modifica e l'interrogazione dell'istanza di una base di dati). In questo capitolo mostreremo le caratteristiche di base di SQL, illustrando dapprima l'uso di SQL per la definizione dello schema di una base di dati (Paragrafo 4.2), per poi descrivere la specifica di interrogazioni (Paragrafo 4.3) e modifiche (Paragrafo 4.4). Nei capitoli successivi continueremo la presentazione di SQL, mostrando alcune caratteristiche evolute del linguaggio (Capitolo 5). La trattazione verrà ripresa nel Capitolo 10, dove si descriveranno l'integrazione tra SQL e i tradizionali linguaggi di programmazione.

4.1 Il linguaggio SQL e gli standard

La diffusione di SQL è dovuta in buona parte alla intensa opera di standardizzazione dedicata a questo linguaggio, svolta principalmente nell'ambito degli organismi ANSI (*American National Standards Institute*, l'organismo nazionale statunitense degli standard) e ISO (l'organismo internazionale che coordina i vari organismi nazionali). Gran parte dei produttori del settore hanno avuto modo di partecipare al processo decisionale. Il processo di standardizzazione ha avuto inizio nella prima metà degli anni Ottanta e continua tuttora. Sono state così prodotte nel tempo diverse versioni, sempre più complete e sofisticate, dello standard del linguaggio. La tabella in Figura 4.1 sintetizza questa evoluzione.

La prima definizione di uno standard per il linguaggio SQL è stata emanata nel 1986 dall'ANSI. Questo primo standard possedeva già gran parte delle primitive di formulazione di interrogazioni, mentre offriva un supporto limitato per la definizione e manipolazione degli schemi e delle istanze. SQL-86 è stato esteso in

[1] Vi sono due diverse pronunce anglosassoni; la prima enuncia le singole lettere es-que-el (corrisponde alla pronuncia italiana esse-qu-elle), la seconda legge la sigla come se fosse la parola "sequel". *Sequel* è il primo nome dato al linguaggio.

Nome informale	Nome ufficiale	Caratteristiche
SQL base	SQL-86	Costrutti base
	SQL-89	Integrità referenziale
SQL-2	SQL-92	Modello relazionale Vari costrutti nuovi 3 livelli: entry, intermediate, full
SQL-3	SQL:1999	Modello relazionale a oggetti Organizzato in diverse parti Trigger, funzioni esterne, ...
	SQL:2003	Estensioni del modello a oggetti Eliminazione di costrutti non usati Nuove parti: SQL/JRT, SQL/XML, ...
	SQL:2006	Estensione della parte XML
	SQL:2008	Lievi aggiunte (per esempio, trigger `instead of`)
	SQL:2011	Ricco supporto per dati temporali

Figura 4.1 Evoluzione dello standard SQL.

modo limitato nel 1989, producendo lo standard il cui nome formale è SQL-89; l'aggiunta più significativa di questa versione è stata la definizione dell'integrità referenziale.

Una seconda versione, in gran parte compatibile con la versione precedente ma arricchita da un gran numero di nuove funzionalità, è stata pubblicata nel 1992. A questa versione si fa riferimento con il nome ufficiale SQL-92 o con il nome informale SQL-2; noi indicheremo questa versione come SQL-2.

Sono state preparate successivamente altre versioni dello standard, cui si fa riferimento con il nome informale SQL-3, aventi nome ufficiale SQL:1999, SQL:2003, SQL:2006, SQL:2008 e SQL:2011. SQL-3 rappresenta un'estensione molto significativa rispetto a SQL-2, come è testimoniato dal fatto che lo standard è ora organizzato in diverse parti opportunamente numerate e denominate, ciascuna dedicata a un particolare aspetto del linguaggio. Per esempio, la parte 13 di SQL-3, SQL/JRT, descrive l'integrazione con il linguaggio Java; la parte 14, SQL/XML, illustra la gestione di dati XML; ciascuna parte è prodotta da uno specifico comitato tecnico e può essere rinnovata in tempi diversi rispetto alle altre, dando luogo a un contesto molto più dinamico. SQL-3 è pienamente compatibile con SQL-2; questo è un importante requisito che garantisce che applicazioni scritte facendo riferimento allo standard precedente possano operare senza modifiche su sistemi che rispettano il nuovo standard.

Ad alcuni anni di distanza dalla pubblicazione di SQL-3, esso è ancora lontano dall'essere comunemente adottato. Per questa ragione, nel testo faremo sempre rife-

rimento a SQL-2, cercando di mettere in evidenza le caratteristiche sintattiche che non erano presenti nelle versioni precedenti. SQL-3 include nuovi servizi che sono il risultato dell'evoluzione più recente della tecnologia delle basi di dati, tra cui i trigger, le viste ricorsive e il supporto per il paradigma a oggetti. Rispetto a SQL:1999, SQL:2003 introduce alcune estensioni relative al modello a oggetti, elimina alcuni costrutti degli standard precedenti che nessun sistema aveva implementato e che sono stati considerati obsoleti, e introduce nuove parti. SQL:2006 ha ulteriormente esteso il supporto all'integrazione con XML, definendo per esempio un legame con il linguaggio XQuery. SQL:2008 ha introdotto una serie di lievi modifiche, come per esempio il supporto per i trigger con modo `instead of`. SQL:2011 ha introdotto un ricco supporto per la gestione di query temporali. Mostreremo in questo capitolo e in quelli successivi alcune delle novità introdotte da SQL-3; non parleremo però delle estensioni a oggetti (vengono descritte nel secondo volume [6]).

Pur senza le estensioni introdotte in SQL-3, SQL-2 è un linguaggio ricco e complesso, tanto che, a molti anni dalla comparsa del documento di definizione, ancora nessun sistema commerciale mette a disposizione tutte le funzionalità previste dal linguaggio. Per quantificare in modo preciso l'aderenza allo standard, sono stati definiti tre livelli di supporto dei costrutti del linguaggio, denominati rispettivamente *Entry SQL*, *Intermediate SQL* e *Full SQL*. I sistemi possono così essere caratterizzati in base al livello cui aderiscono. Il livello Entry SQL è abbastanza simile a SQL-89, da cui differisce solo per poche e lievi imprecisioni della definizione di SQL-89 che sono state corrette nel passaggio a SQL-2. Il livello Intermediate SQL contiene le caratteristiche ritenute più importanti per rispondere alle esigenze del mercato. Il livello Full SQL rappresenta lo standard nella sua interezza, comprese molte funzioni avanzate che non hanno ancora trovato riscontro nelle implementazioni di SQL.

Analizzando con cura i sistemi relazionali, si osserva che ciascuno di essi presenta in effetti piccole differenze nell'implementazione del linguaggio SQL; le differenze emergono soprattutto quando si confrontano fra di loro le funzionalità innovative. Invece, per quanto riguarda gli aspetti più consolidati del linguaggio, l'adesione allo standard è maggiore e questo permette agli utenti di dialogare in SQL standard con sistemi completamente diversi, come possono essere l'implementazione di un DBMS per uno smartphone destinato a esigenze individuali e una base di dati su mainframe su cui si appoggia il sistema informativo di una grossa azienda.

Noi supporremo che si usi direttamente SQL per definire, aggiornare e interrogare la base di dati. In effetti, sempre più frequentemente i sistemi sono dotati di interfacce molto più facili da usare e la specifica degli schemi, o le modifiche e interrogazioni sulle istanze avvengono spesso mediante l'uso di programmi di tipo grafico, che offrono una modalità di interazione tramite menu e interfacce; questi programmi generano le istruzioni SQL corrispondenti. Ciò comunque non sminuisce l'importanza di conoscere la "lingua franca" dei sistemi di basi di dati, in quanto la sua conoscenza è quasi sempre indispensabile per realizzare applicazioni sofisticate che accedono a basi di dati, indipendentemente dall'interfaccia offerta dal sistema. Dato il ruolo sempre crescente dei DBMS negli attuali sistemi informatici, è sempre più importante per gli sviluppatori avere una conoscenza approfondita di SQL.

4.2 Definizione dei dati in SQL

In questo paragrafo illustriamo l'uso di SQL per la definizione degli schemi delle basi di dati. Conviene prima di tutto descrivere la notazione che verrà usata per la sintassi dei comandi del linguaggio. In generale rappresenteremo i termini del linguaggio usando un font `macchina da scrivere`, mentre i termini variabili verranno scritti in *corsivo*. Usiamo inoltre le parentesi angolari, quadre e graffe e la barra verticale con il significato consueto nella rappresentazione di sintassi.

- Le parentesi angolari (⟨,⟩) permettono di isolare un termine della sintassi.
- Le parentesi quadre ([,]) indicano che il termine all'interno è opzionale, ossia può non comparire o comparire una sola volta.
- Le parentesi graffe ({, }) indicano invece che il termine racchiuso può non comparire o essere ripetuto un numero arbitrario di volte.
- Le barre verticali (|) indicano che deve essere scelto uno tra i termini separati dalle barre; un elenco di termini in alternativa può essere racchiuso tra parentesi angolari.

Le parentesi tonde dovranno essere sempre intese come termini del linguaggio SQL e non come simboli per la definizione della grammatica.

4.2.1 I domini elementari

SQL mette a disposizione alcune famiglie di domini elementari, a partire dai quali si possono definire i domini da associare agli attributi dello schema.

Caratteri Il dominio `character` permette di rappresentare singoli caratteri oppure stringhe. La lunghezza delle stringhe di caratteri può essere fissa o variabile; per le stringhe di lunghezza variabile si indica la lunghezza massima. Per ogni schema si può definire una famiglia di caratteri di default (per esempio, alfabeto latino, cirillico, greco ecc.). La sintassi è:

`character` [`varying`] [(*Lunghezza*)]
[`character set` *NomeFamigliaCaratteri*]

Per definire con questa sintassi un dominio "stringa di 20 caratteri" si potrà scrivere `character(20)`, mentre un dominio "stringa di caratteri dell'alfabeto greco a lunghezza variabile, di lunghezza massima 1000", sarà descritto da `character varying (1000) character set Greek`. Se la lunghezza non è specificata, il dominio rappresenta un singolo carattere. Per le stringhe a lunghezza variabile è obbligatorio specificare la lunghezza massima. SQL ammette anche le forme compatte, e molto utilizzate, `char` e `varchar`, rispettivamente per `character` e `varying character`.

Tipi numerici esatti Questa famiglia contiene i domini che permettono di rappresentare valori esatti, interi o con una parte decimale di lunghezza prefissata (come i tipici valori monetari in euro o in dollari). SQL mette a disposizione quattro diversi tipi numerici esatti:

```
numeric [ ( Precisione [ , Scala ] ) ]
decimal [ ( Precisione [ , Scala ] ) ]
integer
smallint
```

I domini `numeric` e `decimal` rappresentano numeri in base *decimale*. Il parametro *Precisione* specifica il numero di cifre significative; con un dominio `decimal(4)` si possono rappresentare valori tra -9.999 e +9.999. Mediante il parametro *Scala* si specifica la scala di rappresentazione, ovvero si indica quante cifre devono comparire dopo la virgola. Se per esempio si vogliono rappresentare valori precisi sino al centesimo, si assegnerà a *Scala* il valore 2. Per specificare la scala bisogna anche specificare la precisione, e la precisione comprende la rappresentazione della parte frazionaria; quindi, con un dominio `numeric(6,4)` si potranno rappresentare i valori compresi tra -99.9999 e +99.9999. La differenza tra i domini `numeric` e `decimal` consiste nel fatto che la precisione per il dominio `numeric` rappresenta un valore esatto, mentre per il dominio `decimal` costituisce un requisito minimo. Qualora la precisione non sia specificata, il sistema usa un valore caratteristico della implementazione. Se la scala non è specificata, si assume che valga zero.

Nei casi in cui non interessa avere una rappresentazione della parte frazionaria e non è importante controllare in modo preciso la dimensione della rappresentazione decimale, allora diventa possibile usare i domini predefiniti `integer` e `smallint`. Per questi domini non esiste un vincolo sulla rappresentazione ed essi sono generalmente basati sulla rappresentazione interna binaria del calcolatore. La precisione di questi tipi non viene specificata nello standard, ma viene lasciata all'implementazione.

Tipi numerici approssimati Per la rappresentazione di valori reali approssimati (utili per esempio per la rappresentazione di grandezze fisiche), SQL fornisce i seguenti tipi:

```
float [ ( Precisione ) ]
real
double precision
```

Tutti questi domini permettono di descrivere numeri approssimati mediante una rappresentazione in virgola mobile, in cui a ciascun numero corrisponde una coppia di valori: la mantissa e l'esponente. La mantissa è un valore frazionario, mentre l'esponente è un numero intero. Il valore approssimato del numero reale si ottiene moltiplicando la mantissa per la potenza di 10 con grado pari all'esponente. Per esempio, `0.17E16` rappresenta il valore $1{,}7 \times 10^{15}$, e `-0.4E-6` rappresenta -4×10^{-7}.

Al dominio `float` può essere associata una precisione, che rappresenta il numero di cifre dedicate alla rappresentazione della mantissa, mentre la precisione nell'esponente dipende dall'implementazione. La precisione del dominio `real` è invece fissa. Il dominio `double precision` dedica a un numero approssimato una rappresentazione di dimensione doppia rispetto a quella del dominio `real`.

Istanti temporali Questi domini, come quelli della famiglia successiva, sono stati introdotti in SQL-2 per descrivere informazioni temporali, importanti in numerosi contesti applicativi. Questa famiglia di domini permette di rappresentare *istanti* di tempo e offre tre diverse forme:

```
date
time [ ( Precisione ) ] [ with time zone ]
timestamp [ ( Precisione ) ] [ with time zone ]
```

Ciascuno di questi domini è strutturato e decomponibile in un insieme di campi. Il dominio `date` ammette i campi `year`, `month` e `day`, il dominio `time` ammette i campi `hour`, `minute` e `second`, infine `timestamp` ammette tutti i campi, da `year` a `second`. Sia per `time` sia per `timestamp` è possibile specificare una precisione, che rappresenta il numero di cifre decimali che si devono utilizzare nella rappresentazione delle frazioni di secondo.

Se l'opzione `with time zone` è specificata, allora risulta possibile accedere a due campi `timezone_hour` e `timezone_minute` che rappresentano la differenza di fuso orario tra l'ora locale e l'ora universale (Coordinated Universal Time o UTC, che ha sostituito la precedente "ora di Greenwich"); così `21:03:04+1:00` e `20:03:04+0:00` corrispondono allo stesso istante temporale, il primo rappresentato nell'ora solare italiana (differenza dovuta al fuso orario di `+1:00`), il secondo nell'ora universale.

Intervalli temporali Questa famiglia di domini permette di rappresentare *intervalli* di tempo, come per esempio la durata di un evento. La sintassi è:

`interval` *PrimaUnitàDiTempo* [`to` *UltimaUnitàDiTempo*]

PrimaUnitàDiTempo e *UltimaUnitàDiTempo* definiscono le unità di misura che devono essere usate, dalla più precisa alla meno precisa. È così possibile definire domini come `interval year to month` per indicare che la durata dell'intervallo di tempo deve essere misurata in numero di anni e di mesi. Si noti che l'insieme delle unità di misura è diviso in due insiemi distinti: `year` e `month` da una parte, e le unità da `day` a `second` dall'altra, in quanto non si possono paragonare esattamente giorni e mesi (a un mese possono corrispondere da 28 a 31 giorni) e sarebbe altrimenti molto difficile effettuare operazioni aritmetiche sugli intervalli. La prima unità che compare nella definizione, qualunque essa sia, può essere caratterizzata dalla precisione, che rappresenta il numero di cifre, in base 10, usate nella rappresentazione. Quando l'unità più piccola è `second`, si può specificare una precisione che rappresenta il numero di cifre decimali dopo la virgola. Così `interval year(5) to month` permette di rappresentare intervalli fino a 99.999 anni e 11 mesi, mentre `interval day(4) to second(6)` permette di rappresentare intervalli sino a 9.999 giorni, 23 ore, 59 minuti e 59,999999 secondi, con una precisione al milionesimo di secondo.

Domini introdotti in SQL-3 Le estensioni più significative di SQL-3 per quanto riguarda la definizione di domini consistononell'offerta di un insieme di costrutto-

ri, come `ref`, `array` e `row`, che sono parte di estensioni del modello relazionale che recepiscono elementi del paradigma a oggetti. Non parliamo di questi aspetti, che richiedono una trattazione specifica, presente nel secondo volume [6], delle caratteristiche di un modello dei dati a oggetti. Presentiamo qui solo i nuovi domini elementari.

Boolean: Il dominio `boolean` permette di rappresentare singoli valori booleani (`true` e `false`). In SQL-2 era stato previsto a questo scopo un dominio `bit`, il quale però non è stato implementato dai sistemi ed è stato quindi rimosso da SQL:2003.

Bigint: Il dominio `bigint` si aggiunge ai domini numerici esatti `smallint` e `integer`. Lo standard non specifica i limiti di rappresentazione del dominio, ma impone solamente che essi non siano inferiori a quelli del dominio `integer`.

BLOB e CLOB: I due domini `blob` e `clob` permettono di rappresentare oggetti di grandi dimensioni, costituiti da una sequenza arbitraria di valori binari (`blob`, *binary large object*) o di caratteri (`clob`, *character large object*). Per entrambi i domini il sistema garantisce solo di memorizzare il valore, ma non permette che il valore venga utilizzato come criterio di selezione per le interrogazioni. Spesso i valori degli attributi di questa famiglia vengono memorizzati separatamente dagli altri attributi, in un'area di sistema apposita. SQL-3 ha introdotto questi domini in quanto le basi di dati costituiscono il cuore dei servizi di archiviazione del sistema informatico, che sempre più ha l'esigenza di gestire informazioni di tipo semi-strutturato e multimediale (come immagini, documenti, video). Questi contenuti sono caratterizzati da grandi dimensioni e la loro fruizione richiede l'uso di applicazioni specifiche.

4.2.2 Definizione di schema

SQL consente la definizione di uno schema di base di dati come collezione di oggetti (tabelle, domini, viste ecc.). Uno schema viene definito dalla seguente sintassi:

`create schema` [*NomeSchema*] [[`authorization`] *Autorizzazione*]
{ *DefElementoSchema* }

Autorizzazione rappresenta il nome dell'utente proprietario dello schema; se il termine viene omesso, si assume che il proprietario sia l'utente che ha lanciato il comando. Il nome dello schema può essere omesso, e in tal caso si assume come nome dello schema il nome del proprietario. Dopo il comando di `create schema`, compaiono le definizioni dei suoi componenti. Non è necessario che la definizione di tutti i componenti avvenga contemporaneamente alla creazione dello schema, ma può anzi avvenire in più fasi successive. Vediamo ora la definizione di tabelle e domini; altri elementi dello schema verranno descritti nel Paragrafo 5.1.

4.2.3 Definizione delle tabelle

Una tabella[2] SQL è costituita da una collezione ordinata di attributi e da un insieme (eventualmente vuoto) di vincoli. Lo schema della tabella DIPARTIMENTO viene per esempio definito tramite la seguente istruzione SQL:

```
create table Dipartimento
(
    Nome        varchar(20) primary key,
    Indirizzo   varchar(50),
    Città       varchar(20)
)
```

La tabella possiede tre attributi di tipo stringa di caratteri e l'attributo Nome costituisce la chiave primaria della tabella. Osserviamo che, come avviene normalmente nei linguaggi di programmazione, una qualsiasi sequenza di spazi e di caratteri di fine linea è equivalente a un singolo spazio; ciò deve essere sfruttato per aumentare la leggibilità dei comandi SQL, usando strutture allineate come nel comando visto sopra.

La sintassi per la definizione delle tabelle è:

```
create table NomeTabella
( NomeAttributo Dominio [ValoreDiDefault] [ Vincoli ]
  { , NomeAttributo Dominio [ ValoreDiDefault ] [ Vincoli ] }
  AltriVincoli
)
```

Ogni tabella viene quindi definita associandole un nome ed elencando gli attributi che ne compongono lo schema. Per ogni attributo si definiscono un nome, un dominio ed eventualmente un insieme di vincoli che devono essere rispettati dai valori dell'attributo. Dopo aver definito gli attributi, si possono definire i vincoli che coinvolgono più attributi della tabella. Una tabella è inizialmente vuota e il creatore possiede tutti i privilegi sulla tabella, cioè i diritti di accedere al suo contenuto e di modificarlo.

4.2.4 Definizione dei domini

Nella definizione delle tabelle si può far riferimento ai domini predefiniti del linguaggio, descritti nel Paragrafo 4.2.1, o a domini definiti dall'utente a partire dai domini predefiniti. Partendo dai domini predefiniti è possibile costruire nuovi domini, tramite la primitiva `create domain`:

[2] Una convenzione che adotteremo in questo capitolo è quella di usare al posto di *relazione* il termine *tabella* (in inglese *table*) e al posto di *tupla* il termine *riga* (in inglese *row*), rispettando la scelta di termini di SQL; in SQL si usa anche far riferimento agli *attributi* parlando di *colonne*, ma in questo caso preferiamo rimanere aderenti alla classica terminologia relazionale.

```
create domain NomeDominio as TipoDiDato
                [ ValoreDiDefault ]
                [ Vincolo ]
```

Un dominio è così caratterizzato dal proprio nome, da un dominio elementare (che può essere predefinito o definito dall'utente in precedenza), da un eventuale valore di default, e infine da un insieme di vincoli (eventualmente vuoto) che rappresenta un insieme di condizioni che devono essere rispettate dai valori del dominio.

La dichiarazione di nuovi domini permette di associare un insieme di vincoli a un nome di dominio, il che è importante quando per esempio si deve ripetere la stessa definizione di attributo nell'ambito di diverse tabelle. Definendo un dominio apposito si rende la definizione più facilmente modificabile; se si vuole modificare la definizione di un insieme di attributi con lo stesso dominio (in modo particolare il valore di default e i vincoli), risulta sufficiente modificare la definizione del dominio e la modifica si applicherà a tutte le tabelle in cui il dominio viene usato.

Esiste una stretta relazione tra la definizione dei domini degli attributi e la definizione dei tipi delle variabili in un linguaggio di programmazione di alto livello (C, Java ecc.). In entrambi i casi si definisce un insieme di valori ammissibili per un oggetto. D'altra parte, vi sono anche importanti differenze. Infatti, i vincoli sui domini definibili in SQL non hanno un corrispondente nei normali linguaggi di programmazione; allo stesso tempo, facendo riferimento a SQL-2, l'insieme dei costruttori di tipo è molto più limitato. Al contrario dei meccanismi di definizione dei tipi dei linguaggi di programmazione, SQL-2 non mette a disposizione costruttori di tipo come il record o l'array (a parte la possibilità di definire stringhe di caratteri). Questa caratteristica deriva dal modello relazionale dei dati, il quale richiede che tutti gli attributi siano caratterizzati da un dominio elementare.

4.2.5 Specifica di valori di default

Nella sintassi per la definizione dei domini e delle tabelle, si può osservare la presenza di un termine *ValoreDiDefault* in corrispondenza di ogni dominio e attributo. Questo termine permette di specificare il valore di *default*, ovvero il valore che deve assumere l'attributo quando viene inserita una riga nella tabella senza che sia specificato un valore per l'attributo stesso. Quando il valore di default non è specificato, si assume come default il valore *nullo*.

La sintassi per la specifica dei valori di default è:

```
default ( GenericoValore | user | null )
```

GenericoValore rappresenta un valore compatibile con il dominio. L'opzione `user` impone come valore di default l'identificativo dell'utente che esegue il comando di aggiornamento della tabella. Quando un attributo o un dominio è definito a partire da un dominio per il quale è già stato specificato un valore di default, l'eventuale nuovo valore di default ha la priorità e diventa il valore effettivo. L'opzione `null` corrisponde al valore di default di base.

Per esempio, un attributo NumeroFigli che ammetta come valore un numero intero e che abbia il valore di default zero è definito da:

```
NumeroFigli smallint default 0
```

In base a questa definizione, quando in un inserimento il valore dell'attributo non viene specificato, a esso viene assegnato il valore zero.

4.2.6 Vincoli intrarelazionali

Sia nella definizione dei domini sia nella definizione delle tabelle è possibile definire dei vincoli, ovvero delle proprietà che devono essere verificate da ogni istanza della base di dati. I vincoli sono stati introdotti nel Capitolo 2, distinguendo tra vincoli intrarelazionali (che coinvolgono una sola relazione) e vincoli interrelazionali (in cui il predicato considera diverse relazioni). Il costrutto più potente per specificare vincoli generici, sia interrelazionali che intrarelazionali, è il costrutto di `check`, che richiede però di formulare delle interrogazioni sulla base di dati e viene perciò rimandato al Paragrafo 5.1, dopo che avremo illustrato le interrogazioni SQL. Illustriamo invece in questo paragrafo i vincoli intrarelazionali predefiniti.

I più semplici vincoli di tipo intrarelazionale sono i vincoli *not null*, *unique* e *primary key*.

Not null Come detto nel Capitolo 2, il valore *nullo* è un particolare valore che indica assenza di informazioni. Un valore nullo può rappresentare in generale diverse situazioni, come abbiamo visto nel Paragrafo 2.2.

SQL non permette però di distinguere tra le diverse interpretazioni del valore nullo. Le applicazioni che hanno bisogno di distinguere tra questi diversi casi devono ricorrere a soluzioni *ad hoc*, come l'introduzione di altri attributi o l'uso di una particolare codifica.

Il vincolo *not null* indica che il valore *nullo* non è ammesso come valore dell'attributo; in tal caso, il valore dell'attributo deve sempre essere specificato, tipicamente in fase di inserimento. Se all'attributo è però associato un valore di default diverso dal valore nullo, allora diventa possibile effettuare l'inserimento anche senza fornire un valore per l'attributo, in quanto all'attributo viene automaticamente assegnato il valore di default.

Il vincolo viene specificato facendo seguire alla definizione dell'attributo la dichiarazione `not null`:

```
Cognome varchar(20) not null
```

Unique Un vincolo *unique* si applica a un attributo o a un insieme di attributi di una tabella e impone che i valori dell'attributo (o le ennuple di valori sull'insieme di attributi) siano una (super)chiave, cioè righe differenti della tabella non possano avere gli stessi valori; viene fatta un'eccezione per il valore nullo, il quale può comparire su diverse righe senza violare il vincolo, in quanto si assume che i valori nulli siano tutti diversi tra loro.

La definizione di questo vincolo può avvenire in due modi; la prima alternativa può essere usata unicamente quando bisogna definire il vincolo su un solo attributo;

in questo caso si fa seguire la specifica dell'attributo dalla parola chiave `unique` (analogamente a quanto avviene per la specifica del vincolo *not null*):

```
Matricola character(6) unique
```

La seconda alternativa è necessaria quando il vincolo opera su un insieme di attributi. Dopo aver definito gli attributi della tabella, si usa la sintassi:

`unique` (*Attributo* { , *Attributo* })

Un esempio d'uso della sintassi è il seguente:

```
Nome     varchar(20)  not null,
Cognome  varchar(20)  not null,
unique (Cognome,Nome)
```

Si noti che la precedente definizione è ben diversa da una definizione come:

```
Nome     varchar(20)  not null unique,
Cognome  varchar(20)  not null unique
```

Nel primo caso si impone che non ci siano due righe che abbiano uguali sia il nome sia il cognome, nel secondo (più restrittivo) si ha una violazione se nelle righe compaiono più di una volta o lo stesso nome o lo stesso cognome.

Primary key Come è stato detto nel Paragrafo 2.2.2, è di norma necessario specificare per ogni relazione la *chiave primaria*, il più importante tra gli identificatori della relazione. SQL permette così di specificare il vincolo *primary key* una sola volta per ogni tabella (mentre è possibile utilizzare un numero arbitrario di volte i vincoli *unique* e *not null*). Come il vincolo *unique*, il vincolo *primary key* può essere definito direttamente su di un singolo attributo, oppure essere definito elencando più attributi che costituiscono l'identificatore. Gli attributi che fanno parte della chiave primaria non possono assumere il valore nullo; pertanto la definizione di *primary key* implica per tutti gli attributi della chiave primaria una definizione di *not null*, che può essere omessa.

Per esempio, la definizione seguente impone che la coppia di attributi Nome e Cognome costituiscano la chiave primaria:

```
Nome       varchar(20),
Cognome    varchar(20),
primary key (Cognome,Nome)
```

4.2.7 Vincoli interrelazionali

I vincoli interrelazionali più diffusi e significativi sono i *vincoli di integrità referenziale*, come abbiamo visto nel Paragrafo 2.2.4. In SQL per la loro definizione si usa l'apposito vincolo di *foreign key*, ovvero di *chiave esterna*.

Questo vincolo crea un legame tra i valori di un attributo della tabella su cui è definito (che chiameremo *interna*) e i valori di un attributo di un'altra tabella (che chiameremo *esterna*). Il vincolo impone che per ogni riga della tabella interna il valore dell'attributo specificato, se diverso dal valore nullo, sia presente nelle righe della tabella esterna tra i valori del corrispondente attributo. L'unico requisito che la sintassi impone è che l'attributo cui si fa riferimento nella tabella esterna sia soggetto a un vincolo *unique*, cioè sia un identificatore della tabella; tipicamente l'attributo della tabella esterna cui si fa riferimento rappresenta in effetti la chiave primaria della tabella. Più attributi possono essere coinvolti nel vincolo, quando la chiave della tabella esterna è costituita da un insieme di attributi; in tal caso l'unica differenza è che bisognerà confrontare ennuple di valori invece che singoli valori.

Il vincolo può essere definito in due modi, come i vincoli *unique* e *primary key*. Se c'è un solo attributo coinvolto, si può usare il costrutto sintattico `references`, con il quale si specificano la tabella esterna e l'attributo della tabella esterna al quale l'attributo in questione deve essere legato; una definizione alternativa, necessaria quando il legame è rappresentato da un insieme di attributi, fa uso invece del costrutto `foreign key`, posto al termine della definizione degli attributi. Il costrutto elenca gli attributi della tabella coinvolti nel legame, cui segue la definizione dei corrispondenti attributi della tabella esterna mediante il costrutto `references`. Forniamo un esempio del primo uso:

```
create table Impiegato
(
    Matricola    character(6) primary key,
    Nome         varchar(20) not null,
    Cognome      varchar(20) not null,
    Dipart       varchar(15)
                 references Dipartimento(NomeDip),
    Ufficio      numeric(3),
    Stipendio    numeric(9) default 0,
    unique (Cognome,Nome)
)
```

Il vincolo impone che l'attributo Dipart della tabella IMPIEGATO possa solamente assumere uno dei valori che le righe della tabella DIPARTIMENTO possiedono per l'attributo NomeDip.

Se si volesse inoltre imporre che gli attributi Nome e Cognome debbano comparire in una tabella anagrafica, si potrebbe aggiungere il vincolo:

```
foreign key (Nome,Cognome)
            references Anagrafica(Nome,Cognome)
```

La corrispondenza tra gli attributi locali e quelli esterni avviene in base all'ordine: al primo attributo argomento di `foreign key` corrisponde il primo attributo argomento di `references`, e via via gli altri attributi. In questo caso a Nome e Cognome di IMPIEGATO corrispondono rispettivamente Nome e Cognome di ANAGRAFICA.

Per tutti gli altri vincoli visti fino a ora, quando il sistema rileva una violazione, il comando di aggiornamento viene rifiutato, segnalando l'errore all'utente. Per i vincoli di integrità referenziale, invece, SQL permette di scegliere altre reazioni da adottare quando viene rilevata una violazione.

Prendiamo come esempio di riferimento la definizione del vincolo *foreign key* sull'attributo Dipart di IMPIEGATO. Il vincolo può essere violato operando sia sulle righe della tabella interna, nell'esempio IMPIEGATO, sia sulle righe della tabella esterna, nell'esempio DIPARTIMENTO. Si possono introdurre violazioni modificando il contenuto della tabella interna solo in due modi: inserendo una nuova riga o modificando il valore dell'attributo referente; per entrambe queste violazioni non viene offerto un particolare supporto e l'operazione viene semplicemente rifiutata.

Vengono invece offerte diverse alternative per rispondere alle violazioni generate da modifiche sulla tabella esterna. Il motivo di questa asimmetria è dovuto al particolare significato della tabella esterna, che sul piano applicativo rappresenta la tabella principale (o *master*) alle cui variazioni la tabella interna (o *slave*) deve adeguarsi. Infatti, tutte le reazioni alle violazioni opereranno sulla tabella interna.

Le operazioni sulla tabella esterna che possono introdurre delle violazioni sono le modifiche del valore dell'attributo riferito e la cancellazione di righe (nell'esempio, cancellazioni di righe da DIPARTIMENTO e modifiche dell'attributo Nome). La politica di reazione può essere diversa a seconda del comando di aggiornamento che introduce le violazioni.

In particolare, per le operazioni di modifica, è possibile reagire in uno dei seguenti modi:

- `cascade`: il nuovo valore dell'attributo della tabella esterna viene riportato su tutte le corrispondenti righe della tabella interna;
- `set null`: all'attributo referente viene assegnato il valore nullo al posto del valore modificato nella tabella esterna;
- `set default`: all'attributo referente viene assegnato il valore di default al posto del valore modificato nella tabella esterna;
- `no action`: l'azione di modifica non viene consentita e il sistema non ha quindi bisogno di riparare la violazione.

Per le violazioni prodotte dalla cancellazione di un elemento della tabella esterna si ha a disposizione lo stesso insieme di reazioni:

- `cascade`: tutte le righe della tabella interna corrispondenti alla riga cancellata vengono cancellate;
- `set null`: all'attributo referente viene assegnato il valore nullo al posto del valore cancellato nella tabella esterna;
- `set default`: all'attributo referente viene assegnato il valore di default al posto del valore cancellato nella tabella esterna;

- `no action`: la cancellazione non viene consentita.

È possibile associare politiche diverse ai diversi eventi (per esempio utilizzare una politica di `cascade` per le modifiche e una politica di `set null` per le cancellazioni).

Utilizzando la politica `cascade` si assume che le righe della tabella interna siano strettamente legate alle corrispondenti righe della tabella esterna, per cui se si apporta una modifica alla tabella esterna si devono modificare in modo conseguente tutte le righe della tabella interna. Le altre politiche invece assumono una dipendenza meno stretta tra le righe della prima tabella e quelle della seconda. Si noti che le reazioni alle violazioni possono generare una reazione a catena, qualora la tabella interna compaia a sua volta come tabella esterna in un altro vincolo di integrità.

La politica di reazione viene specificata immediatamente dopo il vincolo di integrità, secondo la seguente sintassi:

```
on {delete | update}
   {cascade | set null | set default | no action}
```

Con la seguente definizione si stabilisce una politica di `set null` sulle cancellazioni e di `cascade` per gli update:

```
create table Impiegato
(
    Matricola      character(6),
    Nome           varchar(20) not null,
    Cognome        varchar(20) not null,
    Dipart         varchar(15)
                   references Dipartimento(NomeDip)
                   on delete set null
                   on update cascade,
    Ufficio        numeric(3),
    Stipendio      numeric(9) default 0,
    primary key(Matricola),
    unique (Cognome,Nome)
)
```

4.2.8 Modifica degli schemi

SQL fornisce primitive per la manipolazione degli schemi delle basi di dati, che permettono di modificare le definizioni di tabelle precedentemente introdotte. I comandi che vengono utilizzati a questo fine sono `alter` e `drop`.

Alter Il comando `alter` permette di modificare domini e schemi di tabelle. Il comando può assumere varie forme:

```
alter domain NomeDominio ( set default ValoreDefault |
                drop default |
                add constraint DefVincolo |
                drop constraint NomeVincolo )
alter table NomeTabella (
        alter column NomeAttributo ( set default NuovoDefault |
                                     drop default ) |
        add constraint DefVincolo |
        drop constraint NomeVincolo |
        add column DefAttributo |
        drop column NomeAttributo )
```

Tramite `alter domain` e `alter table` è possibile aggiungere e rimuovere vincoli e modificare i valori di default associati ai domini e agli attributi; è inoltre possibile aggiungere ed eliminare attributi e vincoli sullo schema di una tabella. Si noti che quando si definisce un nuovo vincolo, questo deve essere soddisfatto dai dati già presenti; se l'istanza contiene delle violazioni per il nuovo vincolo, l'inserimento viene rifiutato.

Per esempio, il comando qui descritto estende lo schema della tabella DIPARTIMENTO con un attributo NroUff che permette di rappresentare il numero di uffici di cui il dipartimento è dotato:

```
alter table Dipartimento add column NroUff numeric(4)
```

Drop Mentre il comando `alter` effettua delle modifiche sui domini o sullo schema delle tabelle, il comando `drop` permette di rimuovere dei componenti, siano essi schemi, domini, tabelle, viste o asserzioni (le asserzioni sono dei vincoli che non sono associati ad alcuna tabella in particolare; verranno descritte nel Paragrafo 5.1). Il comando rispetta la sintassi:

```
drop ( schema | domain | table | view | assertion ) NomeElemento
     [ restrict | cascade ]
```

L'opzione `restrict` specifica che il comando non deve essere eseguito in presenza di oggetti *non vuoti*: uno schema non è rimosso se contiene tabelle o altri oggetti; un dominio non è rimosso se appare in qualche definizione di tabella; una tabella non è rimossa se possiede delle righe o se è presente in qualche definizione di tabella o vista; infine, una vista non è rimossa se è utilizzata nella definizione di altre tabelle o viste. L'opzione `restrict` è l'opzione di default.

Con l'opzione `cascade` invece, tutti gli oggetti specificati devono essere rimossi. Quando si rimuove uno schema non vuoto, anche tutti gli oggetti che fanno parte dello schema vengono eliminati. Rimuovendo un dominio che compare nella definizione di qualche attributo, l'opzione `cascade` fa sì che il nome di dominio venga rimosso, ma gli attributi che sono stati definiti utilizzando quel dominio rimangono associati al medesimo dominio elementare. Se, per esempio, viene eliminato il

dominio StringaLunga, definito come `varchar(100)`, tramite il comando `drop domain StringaLunga cascade`, allora tutti gli attributi definiti su quel dominio assumeranno direttamente il dominio `varchar(100)`. Quando si rimuove una tabella con l'opzione `cascade`, tutte le righe vengono perse; se la tabella compariva in qualche definizione di tabella o vista, anche queste vengono rimosse. Eliminando una vista che compare nella definizione di altre tabelle o viste, anche queste tabelle e viste vengono rimosse.

In generale l'opzione `cascade` attiva una reazione a catena, per cui tutti gli elementi che dipendono da un elemento rimosso vengono rimossi, e questo fino a che non si giunge in una situazione in cui non esistono dipendenze non risolte, ossia non vi sono elementi nella cui definizione compaiono elementi che sono stati rimossi. Bisogna perciò usare estrema cautela nell'uso di questa opzione, in quanto può capitare che, a causa di qualche dipendenza sfuggita all'analisi, il comando abbia un effetto molto diverso da quello voluto.[3]

4.2.9 Cataloghi relazionali

Anche se solo in parte previsto dallo standard, tutti i DBMS relazionali gestiscono il proprio *dizionario dei dati* (ovvero la descrizione delle tabelle presenti nella base di dati) mediante una struttura relazionale, cioè tramite tabelle. La base di dati contiene quindi due tipi di tabelle: quelle che contengono i dati e quelle che contengono i cosiddetti *metadati* (dati che descrivono i dati). Questo secondo insieme di tabelle costituisce il *catalogo* della base di dati.

Tale caratteristica delle implementazioni dei sistemi relazionali viene detta *riflessività*. Quasi sempre una base di dati gestisce il catalogo mediante strutture analoghe a quelle che vengono utilizzate per conservare l'istanza, per cui per esempio una base di dati a oggetti avrà un dizionario dei dati definito tramite un modello a oggetti. In questo modo la base di dati può utilizzare per la gestione interna dei metadati le stesse funzioni che vengono usate per la gestione dell'istanza.

I comandi di definizione e modifica dello schema della base di dati potrebbero così essere sostituiti da comandi di manipolazione operanti direttamente sulle tabelle del dizionario dei dati, rendendo superflua l'introduzione di appositi comandi per la definizione dello schema. Questa alternativa va però scartata per diversi motivi. In primo luogo, è utile rendere chiari e immediatamente riconoscibili i comandi di manipolazione degli schemi, distinguendoli anche dal punto di vista sintattico dai comandi che modificano l'istanza della base dati. Inoltre, dato che il dizionario è differente in tutti i prodotti, una manipolazione diretta sarebbe una soluzione applicabile solamente su un particolare sistema e quindi non portabile. Infine, la realizzazione di un comando DDL può richiedere di manipolare diverse componenti del dizionario e una modifica diretta corre il rischio di non realizzare completamente tutti i passi necessari e quindi di produrre un catalogo inconsistente.

[3] Si suggerisce quindi di sfruttare a proprio vantaggio il supporto transazionale offerto dai sistemi (Capitolo 12), analizzando con attenzione qual è il risultato dell'esecuzione di un comando di `drop cascade` prima di rendere definitivi i suoi effetti mediante un commit.

Lo standard SQL-2 prevede per il dizionario dei dati una descrizione in due livelli. Un primo livello è quello del DEFINITION_SCHEMA, costituito da un insieme di tabelle che contengono la descrizione di tutte le strutture della base di dati. Nello standard compare un insieme di tabelle di esempio che però non corrisponde a nessuna delle implementazioni di SQL, in quanto le tabelle forniscono una descrizione dei soli aspetti di un sistema di base di dati che vengono gestiti dallo standard SQL, tralasciando in particolare tutti i problemi di definizione delle strutture di memorizzazione che, pur se non presenti nello standard, costituiscono un componente fondamentale di uno schema. Le tabelle dello standard costituiscono quindi una traccia che potrebbe (ma non deve necessariamente) essere seguita dai sistemi.

Il secondo componente dello standard è l'INFORMATION_SCHEMA, un insieme di viste costruite sul DEFINITION_SCHEMA che invece fanno parte a pieno titolo dello standard e che costituiscono un'interfaccia verso il dizionario dei dati che deve essere garantita dai sistemi che vogliono essere conformi allo standard. L'INFORMATION_SCHEMA contiene viste come TABLES, VIEWS, COLUMNS, DOMAINS, DOMAIN_CONSTRAINTS e altre, per un totale di ventitré viste che descrivono la struttura della base di dati.

Non descriviamo né il nome né la struttura di tutte queste tabelle, ma forniamo un semplice esempio del contenuto di queste viste. In Figura 4.2 vediamo (in versione semplificata) il contenuto della vista COLUMNS del catalogo per le tabelle IMPIEGATO e DIPARTIMENTO. Table_Name rappresenta il nome della tabella; Column_Name è il nome dell'attributo; Ordinal_Position descrive la posizione dell'attributo nello schema; Column_Default specifica il valore di default per l'attributo; infine, Is_Nullable è un valore booleano che specifica se l'attributo può assumere il valore nullo.

In Figura 4.3 si vede una dimostrazione della riflessività del dizionario dati, con la descrizione in COLUMNS della tabella stessa.

COLUMNS

Table_Name	Column_Name	Ordinal_ Position	Column_ Default	Is_Nullable
Impiegato	Matricola	1	NULL	N
Impiegato	Cognome	2	NULL	N
Impiegato	Nome	3	NULL	N
Impiegato	Dipart	4	NULL	Y
Impiegato	Ufficio	5	NULL	Y
Impiegato	Stipendio	6	0	Y
Dipartimento	Nome	1	NULL	N
Dipartimento	Indirizzo	2	NULL	Y
Dipartimento	Città	3	NULL	Y

Figura 4.2 Una parte del contenuto della vista COLUMNS del dizionario dei dati.

COLUMNS

Table_Name	Column_Name	Ordinal_Position	Column_Default	Is_Nullable
Columns	Table_Name	1	NULL	N
Columns	Column_Name	2	NULL	N
Columns	Ordinal_Position	3	NULL	N
Columns	Column_Default	4	NULL	Y
Columns	Is_Nullable	5	Y	N

Figura 4.3 La descrizione riflessiva di COLUMNS.

4.3 Interrogazioni in SQL

La parte di SQL dedicata alla formulazione di interrogazioni fa parte del DML. D'altro canto, la separazione tra DML e DDL non è rigida e parte dei servizi di definizione di interrogazioni vengono riutilizzati nella specifica di alcuni aspetti avanzati dello schema (Paragrafo 5.1).

4.3.1 Dichiaratività di SQL

SQL esprime le interrogazioni in modo *dichiarativo*, ovvero si specifica l'obiettivo dell'interrogazione e non il modo in cui ottenerlo. In ciò SQL segue i principi del calcolo relazionale e si contrappone a linguaggi di interrogazione *procedurali*, come l'algebra relazionale, in cui l'interrogazione specifica i passi da compiere per estrarre le informazioni dalla base di dati. L'interrogazione SQL per essere eseguita viene passata all'ottimizzatore di interrogazioni (*query optimizer*), un componente del DBMS il quale analizza l'interrogazione e formula a partire da questa un'interrogazione equivalente nel linguaggio procedurale interno del sistema di gestione di basi di dati. Questo linguaggio procedurale è nascosto all'utente. Per questo, chiunque scriva interrogazioni in SQL può trascurare gli aspetti di traduzione e ottimizzazione. Il grande sforzo dedicato allo sviluppo di tecniche di ottimizzazione ha permesso di costruire strumenti che sono in grado di produrre traduzioni molto efficienti per la maggior parte dei DBMS relazionali.

Esistono in generale molti modi diversi per esprimere la stessa interrogazione in SQL: il programmatore dovrà effettuare una scelta basandosi non sull'efficienza, bensì su caratteristiche come la leggibilità e la modificabilità dell'interrogazione. SQL agevola così il lavoro del programmatore permettendogli di descrivere le interrogazioni in un modo astratto e di alto livello.

4.3.2 Interrogazioni semplici

Le operazioni di interrogazione in SQL vengono specificate per mezzo dell'istruzione `select`. Vediamo prima la struttura essenziale di una `select`.

select *ListaAttributi*
from *ListaTabelle*
[where *Condizione*]

Le tre parti di cui si compone un'istruzione select vengono spesso chiamate *clausola* select (detta anche *target list*), *clausola* from e *clausola* where. Una descrizione più precisa della stessa sintassi è la seguente:

select *AttrEspr* [[as] *Alias*] {, *AttrEspr* [[as] *Alias*] }
from *Tabella* [[as] *Alias*] { , *Tabella* [[as] *Alias*] }
[where *Condizione*]

L'interrogazione SQL seleziona, tra le righe che appartengono al prodotto cartesiano delle tabelle elencate nella clausola from, quelle che soddisfano le condizioni espresse nell'argomento della clausola where. Il risultato dell'esecuzione di un'interrogazione SQL è così una tabella con una riga per ogni riga prodotta dalla clausola from e filtrata dalla clausola where, le cui colonne si ottengono dalla valutazione delle espressioni *AttrEspr* che appaiono nella clausola select. Ogni colonna del risultato viene eventualmente ridenominata con l'*Alias*, se questo compare dopo l'espressione. Anche le tabelle nella clausola from possono essere ridenominate con un *Alias*; l'alias per le tabelle ha però un'importante funzione che approfondiremo nel seguito, quando parleremo dell'uso delle variabili nelle interrogazioni.

Per formulare le prime interrogazioni SQL, si consideri una base di dati avente le due tabelle IMPIEGATO(Nome,Cognome,Dipart,Ufficio,Stipendio,Città) e DIPARTIMENTO(Nome,Indirizzo,Città).

Interrogazione 1: estrarre lo stipendio degli impiegati di cognome "Rossi".

```
select Stipendio as Salario
from Impiegato
where Cognome = 'Rossi'
```

Se non vi sono impiegati di cognome "Rossi", l'interrogazione restituirà un insieme vuoto, altrimenti, un insieme con tante righe quanti sono tali impiegati. Applicando l'interrogazione alla tabella in Figura 4.4 si ottiene il risultato in Figura 4.5: vi sono due impiegati di cognome "Rossi", quindi il risultato contiene due righe.

Estendiamo ora l'analisi delle interrogazioni SQL, introducendo man mano costrutti più complicati.

Clausola select La clausola select specifica gli elementi dello schema della tabella risultato. Come argomento della clausola select può anche comparire il carattere speciale * (asterisco), che rappresenta la selezione di tutti gli attributi delle tabelle elencate nella clausola from.

Interrogazione 2: estrarre tutte le informazioni relative agli impiegati di cognome "Rossi". Il risultato compare in Figura 4.6.

IMPIEGATO

Nome	Cognome	Dipart	Ufficio	Stipendio	Città
Mario	Rossi	Amministrazione	10	45	Milano
Carlo	Bianchi	Produzione	20	36	Torino
Giovanni	Verdi	Amministrazione	20	40	Roma
Franco	Neri	Distribuzione	16	45	Napoli
Carlo	Rossi	Direzione	14	80	Milano
Lorenzo	Gialli	Direzione	7	73	Genova
Paola	Rosati	Amministrazione	75	40	Venezia
Marco	Franco	Produzione	20	46	Roma

Figura 4.4 Contenuto della tabella IMPIEGATO.

Salario
45
80

Figura 4.5 Risultato dell'Interrogazione 1.

Nome	Cognome	Dipart	Ufficio	Stipendio	Città
Mario	Rossi	Amministrazione	10	45	Milano
Carlo	Rossi	Direzione	14	80	Milano

Figura 4.6 Risultato dell'Interrogazione 2.

```
select *
from Impiegato
where Cognome = 'Rossi'
```

Nella clausola `select` possono comparire generiche espressioni sul valore degli attributi di ciascuna riga selezionata.

Interrogazione 3: estrarre lo stipendio mensile dell'impiegato che ha cognome "Bianchi". Il risultato è in Figura 4.7.

```
select Stipendio/12 as StipendioMensile
from Impiegato
where Cognome = 'Bianchi'
```

StipendioMensile
3,00

Figura 4.7 Risultato dell'Interrogazione 3.

Clausola from Quando si desidera formulare un'interrogazione che coinvolge righe appartenenti a più di una tabella, si pone come argomento della clausola `from` l'insieme di tabelle alle quali si vuole accedere. Sul prodotto cartesiano delle tabelle elencate verranno applicate le condizioni contenute nella clausola `where`. Quindi, un join può essere specificato indicando in modo esplicito le condizioni che esprimono il legame tra le diverse tabelle.

Interrogazione 4: estrarre i nomi degli impiegati e le città in cui lavorano.

```
select Impiegato.Nome, Impiegato.Cognome,
       Dipartimento.Città
from Impiegato, Dipartimento
where Impiegato.Dipart = Dipartimento.Nome
```

Supponendo che il contenuto di IMPIEGATO sia quello della Figura 4.4, mentre DIPARTIMENTO sia rappresentato dalla tabella in Figura 4.8, il risultato della valutazione dell'interrogazione sarà pari alla tabella rappresentata in Figura 4.9.

Rispetto alla interrogazione precedente si nota l'uso dell'operatore *punto* per identificare le tabelle da cui vengono estratti gli attributi. Per esempio, il termine `Impiegato.Nome` identifica l'attributo Nome della tabella IMPIEGATO. Viene fatto un uso analogo dell'operatore punto in molti linguaggi di programmazione, per identificare i campi di una variabile strutturata. È necessario specificare il nome della tabella quando le tabelle presenti nella clausola `from` posseggono più attributi con lo stesso nome. Qualora non vi sia possibilità di ambiguità, è possibile specificare l'attributo senza dichiarare la tabella di appartenenza.

Interrogazione 5: gli attributi per cui sorge un'ambiguità sono Nome e Città. L'interrogazione precedente può essere espressa facendo uso degli alias per le tabelle allo scopo di abbreviare i riferimenti a esse.

DIPARTIMENTO

Nome	Indirizzo	Città
Amministrazione	Via Tito Livio, 27	Milano
Produzione	P.le Lavater, 3	Torino
Distribuzione	Via Segre, 9	Roma
Direzione	Via Tito Livio, 27	Milano
Ricerca	Via Venosa, 6	Milano

Figura 4.8 Contenuto della tabella DIPARTIMENTO.

Impiegato.Nome	Impiegato.Cognome	Dipartimento.Città
Mario	Rossi	Milano
Carlo	Bianchi	Torino
Giovanni	Verdi	Milano
Franco	Neri	Roma
Carlo	Rossi	Milano
Lorenzo	Gialli	Milano
Paola	Rosati	Milano
Marco	Franco	Torino

Figura 4.9 Risultato dell'Interrogazione 4.

```
select I.Nome, Cognome, D.Città
from Impiegato as I, Dipartimento as D
where Dipart = D.Nome
```

Clausola where La clausola `where` ammette come argomento un'espressione booleana costruita combinando predicati semplici con gli operatori `and`, `or` e `not`. Ciascun predicato semplice usa gli operatori =, <>, <, >, <= e >= per confrontare da un lato un'espressione costruita a partire dai valori degli attributi per la riga, e dall'altro lato un valore costante o un'altra espressione. Nel caso più semplice l'espressione è rappresentata dal nome di un attributo. Quando i predicati sono separati dall'operatore `and`, saranno selezionate solo le righe per cui *tutti* i predicati sono veri; quando i predicati sono separati dall'operatore `or`, saranno selezionate solo le righe per cui *almeno uno* dei predicati risulta vero. L'operatore logico `not` è unario e inverte il valore di verità del predicato. La sintassi assegna la precedenza nella valutazione all'operatore `not`, ma non definisce una relazione di precedenza tra gli operatori `and` e `or`. Se è necessario esprimere un'interrogazione che richieda l'uso sia di `and` sia di `or`, conviene esplicitare l'ordine di valutazione mediante parentesi.

Interrogazione 6: estrarre il nome e il cognome degli impiegati che lavorano nell'ufficio 20 del dipartimento Amministrazione.

```
select Nome, Cognome
from Impiegato
where Ufficio = 20 and Dipart = 'Amministrazione'
```

Sulla base di dati di Figura 4.4, si ottiene il risultato in Figura 4.10.

Interrogazione 7: estrarre i nomi e i cognomi degli impiegati che lavorano nel dipartimento Amministrazione o nel dipartimento Produzione.

```
select Nome, Cognome
from Impiegato
where Dipart = 'Amministrazione' or
      Dipart = 'Produzione'
```

Nome	Cognome
Giovanni	Verdi

Figura 4.10 Risultato dell'Interrogazione 6.

Nome	Cognome
Mario	Rossi
Carlo	Bianchi
Giovanni	Verdi
Paola	Rosati
Marco	Franco

Figura 4.11 Risultato dell'Interrogazione 7.

Nome
Mario

Figura 4.12 Risultato dell'Interrogazione 8.

Applicando l'interrogazione alla tabella in Figura 4.4 si ottiene il risultato in Figura 4.11.

Interrogazione 8: estrarre i nomi propri degli impiegati di cognome "Rossi" che lavorano nei dipartimenti Amministrazione o Produzione. Il risultato è rappresentato in Figura 4.12.

```
select Nome
from Impiegato
where Cognome = 'Rossi' and
      (Dipart = 'Amministrazione' or
       Dipart = 'Produzione')
```

Oltre ai normali predicati di confronto relazionali, SQL mette a disposizione un operatore `like` per il confronto di stringhe, che permette di effettuare confronti con stringhe in cui compaiono i caratteri speciali _ (trattino sottolineato) e % (percentuale). Il primo carattere speciale può rappresentare nel confronto un carattere arbitrario, il secondo una stringa di un numero arbitrario (eventualmente anche nullo) di caratteri arbitrari. Un confronto `like 'ab%ba_'` sarà perciò soddisfatto da una qualsiasi stringa di caratteri che inizia con `ab` e che ha la coppia di caratteri `ba` prima dell'ultima posizione (per esempio `abcdedcbac`, oppure `abbaf`).

Nome	Cognome	Dipart	Ufficio	Stipendio	Città
Mario	Rossi	Amministrazione	10	45	Milano
Carlo	Rossi	Direzione	14	80	Milano
Paola	Rosati	Amministrazione	75	40	Venezia

Figura 4.13 Risultato dell'Interrogazione 9.

Interrogazione 9: estrarre gli impiegati che hanno un cognome che ha una "o" in seconda posizione e finisce per "i". Il risultato è rappresentato in Figura 4.13.

```
select *
from Impiegato
where Cognome like '_o%i'
```

Gestione dei valori nulli Come abbiamo visto nel Paragrafo 2.1.5, un valore nullo in un attributo può significare che un certo attributo non è applicabile, o che il valore è applicabile ma non è conosciuto, o anche che non si sa quale delle due situazioni vale.

Per selezionare i termini con valori nulli SQL fornisce il predicato `is null`, la cui sintassi è semplicemente:

Attributo `is [ not ] null`

Il predicato risulta vero solo se l'attributo ha valore *nullo*. Il predicato `is not null` è la sua negazione.

I valori nulli hanno un particolare impatto sulla valutazione dei normali predicati. Consideriamo un semplice predicato di confronto fra un attributo e un valore costante:

`Stipendio > 40`

Questo predicato sarà vero per le righe in cui l'attributo Stipendio è superiore a 40. Richiamando quanto detto nel Paragrafo 3.1.8, osserviamo che ci sono due diverse soluzioni per gestire il caso in cui l'attributo Stipendio abbia valore nullo. La prima soluzione, più immediata e adottata dallo standard SQL-89, usa la tradizionale logica a due valori e prevede semplicemente di considerare falso il predicato. La seconda soluzione è invece quella adottata in SQL a partire da SQL-2 e fa uso di una logica a tre valori, in cui un predicato semplice restituisce il valore *unknown* quando uno qualsiasi dei termini del predicato ha valore nullo. Si noti che il predicato `is null` costituisce un'eccezione, restituendo sempre il valore *vero* o il valore *falso*, e mai il valore *unknown*.

La differenza tra le soluzioni basate sulle logiche rispettivamente a due e tre valori emerge solo quando si valutano espressioni complicate. In alcuni casi il comportamento del sistema in presenza di valori nulli può diventare molto poco intuitivo,

particolarmente quando si costruiscono predicati complessi che usano l'operatore di negazione o interrogazioni nidificate (Paragrafo 4.3.6), richiedendo molta attenzione anche a programmatori esperti.

Interpretazione formale delle interrogazioni SQL È possibile costruire una corrispondenza tra le interrogazioni SQL ed equivalenti interrogazioni espresse in algebra relazionale.

Data un'interrogazione SQL nella sua forma più semplice:

```
select T1.Attributo11, ..., Th.Attributohm
from TABELLA1 T1, ..., TABELLAn Tn
where Condizione
```

si può costruire un'interrogazione equivalente in algebra relazionale utilizzando la seguente traduzione (in cui per semplicità omettiamo le ridenominazioni che ci permettono di considerare tutti i join come prodotti cartesiani):

$$\pi_{\mathsf{T}_1.\mathsf{Attributo}_{11},\ldots,\mathsf{T}_h.\mathsf{Attributo}_{hm}}(\sigma_{Condizione}(\mathrm{TABELLA}_1 \bowtie \ldots \bowtie \mathrm{TABELLA}_n))$$

Per interrogazioni SQL più complicate la formula di conversione sopra rappresentata non è più direttamente applicabile. Sarebbe comunque possibile mostrare una tecnica per tradurre ogni interrogazione SQL in una equivalente interrogazione in algebra relazionale, al più utilizzando gli operatori di assegnamento e di ridenominazione.

Ancora più stretto è il legame tra SQL e il calcolo relazionale su tuple con dichiarazioni di range (Paragrafo 3.2.3).

Se si assume che le variabili $\mathsf{T}_1, \mathsf{T}_2, \ldots, \mathsf{T}_h$ siano presenti nella clausola `select` e che $\mathsf{T}_{h+1}, \mathsf{T}_{h+2}, \ldots, \mathsf{T}_n$ non lo siano[4], la generica istruzione `select` ha una semantica che è uguale a quella della seguente espressione del calcolo relazionale su tuple con dichiarazioni di range:

$$\begin{aligned}\{t_1.\mathsf{Attributo}_{11}, \ldots, t_h.\mathsf{Attributo}_{hm} \\ \mid t_1(\mathrm{TABELLA}_1), \ldots, t_h(\mathrm{TABELLA}_h) \\ \mid \exists t_{h+1}(\mathrm{TABELLA}_{h+1})(\ldots(\exists t_n(\mathrm{TABELLA}_n) \\ (Condizione')) \ldots)\}\end{aligned}$$

dove *Condizione'* è la formula ottenuta da *Condizione* sostituendo nel modo naturale la notazione SQL a quella usata per il calcolo relazionale. Si può osservare come a ogni alias di tabella corrisponda una variabile del calcolo. Vedremo che questa interpretazione degli alias è in effetti analoga all'interpretazione che dà SQL.

Una condizione essenziale per l'esecuzione di queste traduzioni è però che l'interrogazione di partenza non usi funzionalità di SQL non presenti nell'algebra e nel calcolo relazionale, come la valutazione di operatori aggregati (che non abbiamo ancora trattato e che saranno l'oggetto del Paragrafo 4.3.3). I risultati delle interrogazioni SQL differiscono anche dalle espressioni dell'algebra e del calcolo relazionale nella gestione dei duplicati, come discutiamo qui sotto.

[4]Si noti che è sempre possibile riordinare le variabili in modo tale che questa condizione sia soddisfatta.

Duplicati Una significativa differenza tra SQL e algebra relazionale è data dalla gestione dei duplicati. Mentre in algebra una tabella viene vista come una relazione dal punto di vista matematico, e quindi come un insieme di elementi (tuple) diversi tra loro, in SQL si possono avere in una tabella più righe uguali (dette duplicati), ovvero righe con gli stessi valori per tutti gli attributi.

Per emulare il comportamento dell'algebra relazionale, sarebbe necessario effettuare l'eliminazione dei duplicati tutte le volte in cui si eseguono operazioni di proiezione. L'operazione di rimozione di duplicati è però molto costosa e spesso non necessaria, in quanto in molti casi il risultato non contiene duplicati. Per esempio, quando il risultato include una chiave per ogni tabella che compare nella clausola `from`, la tabella risultato non può contenere più esemplari della stessa riga. Per questo in SQL si è stabilito di permettere la presenza di duplicati all'interno delle tabelle, lasciando a chi scrive l'interrogazione il compito di specificare esplicitamente quando l'operazione di rimozione di duplicati è necessaria.

L'eliminazione dei duplicati è specificata con la parola chiave `distinct`, da porre immediatamente dopo la parola chiave `select`. La sintassi prevede che si possa anche specificare la parola chiave `all` al posto di `distinct`, indicando che si intendono mantenere tutti i duplicati. L'indicazione della parola `all` è opzionale in quanto, come abbiamo detto, il mantenimento dei duplicati costituisce l'opzione di default.

Data la relazione PERSONA(CodFiscale,Nome,Cognome,Città)(Figura 4.14), si vogliono determinare le città in cui abitano persone con cognome "Rossi"; mostriamo due esempi, il primo dei quali ammette la presenza di duplicati mentre il secondo fa uso dell'opzione `distinct` e quindi li rimuove.

Interrogazione 10: estrarre le città delle persone il cui cognome è "Rossi", presentando eventualmente più volte lo stesso valore di Città.

```
select Città
from Persona
where Cognome = 'Rossi'
```

Interrogazione 11: estrarre le città delle persone con cognome "Rossi", facendo comparire ogni città al più una volta.

PERSONA

CodFiscale	Nome	Cognome	Città
RSSMRA55B21T234J	Mario	Rossi	Verona
BNCCLR69T30H745Z	Carlo	Bianchi	Roma
RSSGNN41A31B344C	Giovanni	Rossi	Verona
RSSPRT75C12F205V	Pietro	Rossi	Milano

Figura 4.14 Tabella PERSONA.

Città
Verona
Verona
Milano

Città
Verona
Milano

Figura 4.15 Il risultato delle interrogazioni 10 e 11.

```
select distinct Città
from Persona
where Cognome = 'Rossi'
```

Eseguendo le due interrogazioni sopra riportate sulla tabella descritta in Figura 4.14, otteniamo i risultati che compaiono in Figura 4.15.

Join interni ed esterni Una sintassi alternativa per la specifica dei join (introdotta in SQL-2 e ancora in via di diffusione) permette di distinguere, tra le condizioni che compaiono nell'interrogazione, quelle che rappresentano condizioni di join e quelle che rappresentano condizioni di selezione sulle righe. In tal modo si possono anche specificare le forme esterne dell'operatore di join.
La sintassi proposta è la seguente:

```
select AttrEspr [ [as] Alias ] {, AttrEspr [ [as] Alias ] }
from Tabella [ [ as ] Alias ]
      { [ TipoJoin ] join Tabella [ [ as ] Alias ] on CondizioneDiJoin }
[ where AltraCondizione ]
```

Mediante questa sintassi la condizione di join non compare come argomento della clausola `where`, ma viene invece spostata nell'ambito della clausola `from`, associata alle tabelle che vengono coinvolte nel join.

Il parametro *TipoJoin* specifica qual è il tipo di join da usare, e a esso si possono sostituire i termini `inner` (interno, valore di default che può essere omesso), `right outer`, `left outer` o `full outer` (il qualificatore `outer` è opzionale). L'`inner join` rappresenta il tradizionale theta-join dell'algebra relazionale.

Interrogazione 12: l'Interrogazione 5 può essere riscritta in un modo diverso.

```
select I.Nome, Cognome, D.Città
from Impiegato I join Dipartimento D
     on Dipart = D.Nome
```

Con il join interno le righe che vengono coinvolte nel join sono in generale un sottoinsieme delle righe di ciascuna tabella. Può infatti capitare che alcune righe non vengano considerate in quanto non esiste una corrispondente riga nell'altra tabella per cui la condizione sia soddisfatta. Questo comportamento spesso non rispetta le

esigenze delle applicazioni, le quali, all'eliminazione delle righe operata dal join, possono preferire di mantenere le righe, introducendo dei valori nulli per rappresentare l'assenza di informazioni provenienti dall'altra tabella. Come abbiamo visto nel Paragrafo 3.1.5, il join esterno (`outer join`) esegue un join mantenendo però tutte le righe che fanno parte di una o entrambe le tabelle coinvolte.

Esistono appunto tre varianti dei join esterni: `left`, `right` e `full`. Il `left join` fornisce come risultato il join interno esteso con le righe della tabella che compare a sinistra per le quali non esiste una corrispondente riga nella tabella di destra; il `right join` si comporta in modo simmetrico (conserva le righe escluse della tabella di destra); infine, il `full join` restituisce il join interno esteso con le righe escluse di entrambe le tabelle.

Si considerino le tabelle GUIDATORE e AUTOMOBILE rappresentate in Figura 4.16.

Interrogazione 13: estrarre i guidatori con le automobili loro associate, mantenendo nel risultato anche i guidatori senza automobile.

```
select Nome, Cognome, G.NroPatente,
       Targa, Marca, Modello
from Guidatore G left join Automobile A on
            (G.NroPatente=A.NroPatente)
```

Il risultato compare in Figura 4.17. Si noti come l'ultima riga del risultato rappresenti un guidatore cui non risulta associata nessuna automobile.

Interrogazione 14: estrarre tutti i guidatori e tutte le auto, mostrando tutte le relazioni esistenti tra di essi.

```
select Nome, Cognome, G.NroPatente,
       Targa, Marca, Modello
from Guidatore G full join Automobile A on
            (G.NroPatente=A.NroPatente)
```

GUIDATORE

Nome	Cognome	NroPatente
Mario	Rossi	VR 2030020Y
Carlo	Bianchi	PZ 1012436B
Marco	Neri	AP 4544442R

AUTOMOBILE

Targa	Marca	Modello	NroPatente
AB 574 WW	Fiat	Punto	VR 2030020Y
AA 652 FF	Fiat	Brava	VR 2030020Y
BJ 747 XX	Lancia	Delta	PZ 1012436B
BB 421 JJ	Fiat	Uno	MI 2020030U

Figura 4.16 Tabelle GUIDATORE e AUTOMOBILE.

Nome	Cognome	G.NroPatente	Targa	Marca	Modello
Mario	Rossi	VR 2030020Y	AB 574 WW	Fiat	Punto
Mario	Rossi	VR 2030020Y	AA 652 FF	Fiat	Brava
Carlo	Bianchi	PZ 1012436B	BJ 747 XX	Lancia	Delta
Marco	Neri	AP 4544442R	NULL	NULL	NULL

Figura 4.17 Risultato dell'Interrogazione 13.

Nome	Cognome	G.NroPatente	Targa	Marca	Modello
Mario	Rossi	VR 2030020Y	AB 574 WW	Fiat	Punto
Mario	Rossi	VR 2030020Y	AA 652 FF	Fiat	Brava
Carlo	Bianchi	PZ 1012436B	BJ 747 XX	Lancia	Delta
Marco	Neri	AP 4544442R	NULL	NULL	NULL
NULL	NULL	NULL	BB 421 JJ	Fiat	Uno

Figura 4.18 Risultato dell'Interrogazione 14.

L'interrogazione produce come risultato la tabella in Figura 4.18. Si noti che l'ultimo elemento della tabella descrive un'automobile per la quale non esiste un corrispondente elemento in GUIDATORE.

In alcune implementazioni di SQL si rappresenta il join esterno aggiungendo all'identificativo degli attributi un particolare carattere o sequenza di caratteri (per esempio `*` o `(+)`). In questo modo diventa possibile formulare il join esterno senza ricorrere alla sintassi che abbiamo visto.

Interrogazione 15: l'Interrogazione 13 potrebbe essere formulata in un modo diverso.

```
select Nome, Cognome, G.NroPatente,
       Targa, Marca, Modello
from Guidatore G, Automobile A
where G.NroPatente * = A.NroPatente
```

Queste soluzioni sono però al di fuori dello standard SQL e non sono perciò portabili da un sistema all'altro.

Un'ulteriore estensione di SQL-2 permette di far precedere a ogni join la parola chiave `natural`. In questo modo si consente la specifica del join naturale dell'algebra relazionale, che prevede di utilizzare nel join di due tabelle una condizione implicita di uguaglianza su tutti gli attributi caratterizzati dallo stesso nome (Paragrafo 3.1.5). Per esempio, la query 14 potrebbe essere rappresentata come:

Interrogazione 16:

```
select Nome, Cognome, G.NroPatente,
       Targa, Marca, Modello
from Guidatore G natural full join Automobile
```

Nonostante il vantaggio di una rappresentazione più compatta, il join naturale non è normalmente consigliabile (e spesso non è offerto dai sistemi commerciali). Un motivo è che un'interrogazione che usa il join naturale può introdurre dei rischi nelle applicazioni, in quanto il suo comportamento può mutare profondamente al variare dello schema delle tabelle.

Uso di variabili Abbiamo già visto come nelle interrogazioni SQL sia possibile associare un nome alternativo, detto *alias*, alle tabelle che compaiono come argomento della clausola `from`. Il nome viene usato per far riferimento alla tabella nel contesto dell'interrogazione. Questa funzionalità può essere sfruttata per far riferimento a una tabella in modo compatto, ricorrendo a brevi alias ed evitando così di scrivere per esteso il nome della tabella tutte le volte che ne viene richiesto l'uso (Interrogazione 5). Vi sono però altre ragioni per usare gli alias.

Per prima cosa, utilizzando gli alias è possibile fare accesso più volte alla stessa tabella, come avviene nel calcolo relazionale quando si usano più variabili associate alla stessa tabella e in modo simile all'uso dell'operatore di ridenominazione ρ dell'algebra relazionale. Tutte le volte che si introduce un alias per una tabella si dichiara in effetti una variabile che rappresenta le righe della tabella di cui è alias. Quando una tabella compare una sola volta in un'interrogazione, non c'è differenza tra l'interpretare l'alias come uno pseudonimo o come una nuova variabile. Quando una tabella compare invece più volte, è necessario considerare l'alias come una nuova variabile.

Interrogazione 17: estrarre tutti gli impiegati che hanno lo stesso cognome (ma diverso nome) di impiegati del dipartimento Produzione.

```
select I1.Cognome, I1.Nome
from Impiegato I1, Impiegato I2
where I1.Cognome = I2.Cognome and
      I1.Nome <> I2.Nome and
      I2.Dipart = 'Produzione'
```

Questa interrogazione confronta ciascuna riga di IMPIEGATO con tutte le righe di IMPIEGATO associate al dipartimento Produzione. Si osservi che in questa interrogazione, ogni riga con "Produzione" come valore dell'attributo **Dipart** viene confrontata anche con se stessa, ma il confronto della riga con se stessa non sarà mai soddisfatto, in quanto il predicato di disuguaglianza sull'attributo **Nome** non potrà mai essere vero.

Per illustrare l'esecuzione di questa interrogazione, si può immaginare che al momento della definizione degli alias, vengano create due diverse tabelle associate alle variabili `I1` e `I2`, ciascuna con tutte le righe di IMPIEGATO; ciascuna variabile assumerà quindi ciascun valore di tupla in modo indipendente dall'altra variabile. La Figura 4.19 descrive l'operazione di copia della tabella IMPIEGATO in `I1` e `I2` e il successivo confronto di ogni riga di `I1` con ciascuna riga di `I2`.

La definizione di alias risulta anche molto importante per la realizzazione di sofisticate interrogazioni nidificate, come vedremo nel Paragrafo 4.3.6.

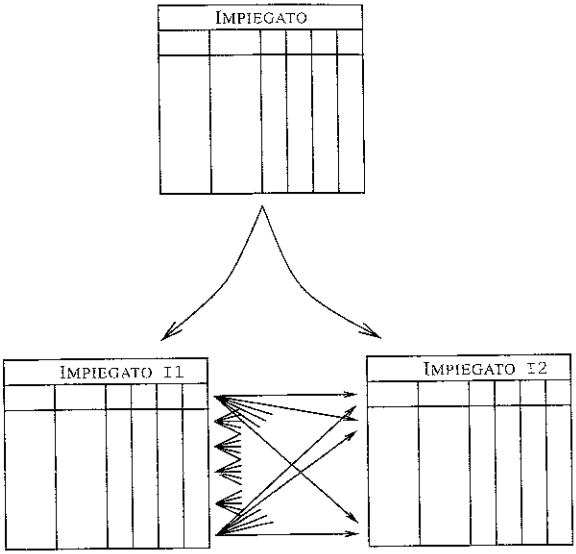

Figura 4.19 Descrizione dell'esecuzione dell'Interrogazione 17.

Per mostrare la corrispondenza tra l'operatore di ridenominazione dell'algebra e l'uso di variabili in SQL, possiamo formulare in SQL un'interrogazione già espressa nel Paragrafo 3.1.6 (Espressione 3.2).

Interrogazione 18: estrarre il nome e lo stipendio dei capi degli impiegati che guadagnano più di 40.

```
select I1.Nome as NomeC, I1.Stipendio as StipC
from Impiegati I1, Supervisione, Impiegati I2
where I1.Matricola = Supervisione.Capo and
      I2.Matricola = Supervisione.Impiegato and
      I2.Stipendio > 40
```

Ordinamento Mentre una relazione è costituita da un insieme non ordinato di tuple, nell'uso reale delle basi di dati sorge spesso il bisogno di costruire un ordine sulle righe delle tabelle. Pensiamo al caso in cui un utente vuole sapere quali sono gli stipendi più elevati che vengono elargiti dall'azienda. Una interrogazione che restituisce i dati degli impiegati ordinati in base al valore dell'attributo Stipendio permette di soddisfare questa esigenza.

Targa	Marca	Modello	NroPatente
BJ 747 XX	Lancia	Delta	PZ 1012436B
AA 652 FF	Fiat	Brava	VR 2030020Y
AB 574 WW	Fiat	Punto	VR 2030020Y
BB 421 JJ	Fiat	Uno	MI 2020030U

Figura 4.20 Risultato dell'interrogazione 19.

SQL permette di specificare un ordinamento delle righe del risultato di un'interrogazione tramite la clausola `order by`, con la quale si chiude l'interrogazione. La clausola rispetta la seguente sintassi:

`order by` *AttrDiOrdinamento* [`asc` | `desc`]
{ , *AttrDiOrdinamento* [`asc` | `desc`] }

In questo modo si specificano gli attributi che devono essere usati per l'ordinamento. Per prima cosa, le righe vengono ordinate in base al primo attributo nell'elenco. Per righe che hanno lo stesso valore del primo attributo, si considerano i valori degli attributi successivi, in sequenza. L'ordine su ciascun attributo può essere ascendente o discendente, a seconda che si sia usato il qualificatore `asc` o `desc`. Se il qualificatore è omesso, si assume un ordinamento ascendente.

Si consideri la base di dati in Figura 4.16.

Interrogazione 19: estrarre il contenuto della tabella AUTOMOBILE ordinato in base alla marca (in modo discendente) e al modello.

```
select *
from Automobile
order by Marca desc, Modello
```

Il risultato è rappresentato in Figura 4.20.

4.3.3 Operatori aggregati

Gli operatori aggregati costituiscono una delle più importanti estensioni di SQL rispetto all'algebra relazionale.

In algebra relazionale tutte le condizioni vengono valutate su una tupla alla volta: la condizione è sempre un predicato che viene valutato su ciascuna tupla indipendentemente da tutte le altre.

Spesso però nei contesti reali viene richiesto di valutare delle proprietà che dipendono da insiemi di tuple. Supponiamo che si voglia determinare il numero degli impiegati del dipartimento Produzione. Il numero di impiegati corrisponderà al numero di tuple della relazione IMPIEGATO che possiedono "Produzione" come valore dell'attributo Dipart. Questo numero non è però una proprietà posseduta da una tupla

in particolare e perciò l'interrogazione non è esprimibile in algebra relazionale. Per esprimerla in SQL usiamo l'operatore aggregato di conteggio `count`.

Interrogazione 20: estrarre il numero di impiegati del dipartimento Produzione.

```
select count(*)
from Impiegato
where Dipart = 'Produzione'
```

Gli operatori aggregati vengono gestiti come un'estensione delle normali interrogazioni. Prima viene normalmente eseguita l'interrogazione, considerando solo le parti `from` e `where`. L'operatore aggregato viene poi applicato alla tabella contenente il risultato dell'interrogazione. Nell'esempio appena visto, prima si costruisce la tabella che contiene tutte le righe di IMPIEGATO che hanno "Produzione" come valore dell'attributo Dipart, dopodiché su questa tabella si applica l'operatore aggregato che conta il numero di righe che compaiono nella tabella.

Lo standard SQL prevede cinque operatori aggregati: `count`, `sum`, `max`, `min` e `avg`.

L'operatore `count` usa la seguente sintassi:

`count` ({ * | [`distinct` | `all`] *ListaAttributi* })

La prima opzione (`*`) restituisce il numero di righe; l'opzione `distinct` restituisce il numero di diversi valori degli attributi in *ListaAttributi*; l'opzione `all` invece restituisce il numero di righe che possiedono valori diversi dal valore nullo per gli attributi in *ListaAttributi*. Se si specifica un attributo e si omette `distinct` o `all`, si assume `all` come default.

Interrogazione 21: estrarre il numero di diversi valori dell'attributo Stipendio fra tutte le righe di IMPIEGATO.

```
select count(distinct Stipendio)
from Impiegato
```

Interrogazione 22: estrarre il numero di righe che possiedono un valore non nullo per l'attributo Nome.

```
select count(all Nome)
from Impiegato
```

Gli altri quattro operatori aggregati invece ammettono come argomento un attributo o un'espressione, eventualmente preceduta dalle parole chiave `distinct` o `all`. Le funzioni aggregate `sum` e `avg` ammettono come argomento solo espressioni che rappresentano valori numerici o intervalli di tempo. Le funzioni `max` e `min` richiedono solamente che sull'espressione sia definito un ordinamento, per cui si possono applicare anche su stringhe di caratteri o su istanti di tempo.

{ `sum` | `max` | `min` | `avg` } ([`distinct` | `all`] *AttrEspr*)

Gli operatori si applicano sulle righe che soddisfano la condizione presente nella clausola `where` e hanno il seguente significato:

- `sum`: restituisce la somma dei valori posseduti dall'espressione;
- `max` e `min`: restituiscono rispettivamente il valore massimo e minimo;
- `avg`: restituisce la media dei valori (vale a dire, il risultato della divisione di `sum` per `count`).

Le parole chiave `distinct` e `all` hanno il significato che abbiamo già visto: `distinct` elimina i duplicati, mentre `all` trascura solo i valori nulli; l'uso di `distinct` o `all` con gli operatori `max` e `min` non ha effetto sul risultato.

Le varie implementazioni di SQL spesso offrono un repertorio di operatori aggregati più vasto, fornendo operatori di tipo statistico come varianza, mediana, scarto quadratico medio ecc.

Interrogazione 23: estrarre la somma degli stipendi del dipartimento Amministrazione.

```
select sum(Stipendio)
from Impiegato
where Dipart = 'Amministrazione'
```

Possiamo anche valutare diversi operatori aggregati nell'ambito della stessa interrogazione:

Interrogazione 24: estrarre gli stipendi minimo, massimo e medio fra quelli di tutti gli impiegati.

```
select min(Stipendio), max(Stipendio), avg(Stipendio)
from Impiegato
```

La valutazione degli operatori aggregati può avvenire su una generica interrogazione. Per esempio, l'interrogazione seguente applica l'operatore aggregato sul risultato di un join.

Interrogazione 25: estrarre il massimo stipendio tra quelli degli impiegati che lavorano in un dipartimento con sede a Milano.

```
select max(Stipendio)
from Impiegato, Dipartimento D
where Dipart = D.Nome and
      D.Città = 'Milano'
```

Considerando l'esempio precedente, vale la pena di osservare che la seguente interrogazione *non è corretta*:

Interrogazione 26:

```
select Cognome, Nome, max(Stipendio)
from Impiegato, Dipartimento D
where Dipart = D.Nome and
      D.Città = 'Milano'
```

Si potrebbe pensare che questa interrogazione riesca a selezionare il valore massimo dell'attributo Stipendio, e quindi automaticamente selezioni gli attributi Nome e Cognome dell'impiegato corrispondente. Questa interpretazione non può però essere usata. Infatti, gli operatori aggregati non rappresentano un meccanismo di selezione, ma solo delle funzioni che restituiscono un valore quando sono applicate a un insieme. La clausola `select` contiene quindi due attributi, che genereranno una coppia di valori per ogni tupla selezionata, e una funzione aggregata, che restituisce un valore per l'intero insieme di tuple. Il linguaggio non offre un meccanismo per gestire questa eterogeneità e perciò la sintassi SQL non ammette che nella stessa clausola `select` compaiano funzioni aggregate ed espressioni al livello di riga, a meno che non si faccia uso della clausola `group by` descritta nel prossimo paragrafo.

4.3.4 Interrogazioni con raggruppamento

Abbiamo caratterizzato gli operatori aggregati come gli operatori che vengono applicati a un insieme di righe. Gli esempi che abbiamo per ora visto operano su tutte le righe che vengono prodotte come risultato dell'interrogazione. Molto spesso sorge l'esigenza di applicare l'operatore aggregato separatamente a sottoinsiemi di righe. Per poter utilizzare in questo modo l'operatore aggregato, SQL mette a disposizione la clausola `group by`, che permette di specificare come dividere le tabelle in sottoinsiemi. La clausola ammette come argomento un insieme di attributi e l'interrogazione raggrupperà le righe che possiedono gli stessi valori per questo insieme di attributi.

Analizziamo come viene eseguita un'interrogazione SQL che fa uso della clausola `group by`, considerando la seguente interrogazione.

Interrogazione 27: estrarre la somma degli stipendi di tutti gli impiegati dello stesso dipartimento.

```
select Dipart, sum(Stipendio)
from Impiegato
group by Dipart
```

Supponiamo che la tabella inizialmente contenga le informazioni rappresentate in Figura 4.21. Per prima cosa l'interrogazione viene eseguita come se la clausola `group by` non esistesse, selezionando gli attributi che appaiono come argomento della clausola `group by` o che compaiono all'interno dell'espressione argomento dell'operatore aggregato. Nella query in esame, è come se venisse eseguita l'interrogazione:

IMPIEGATO

Nome	Cognome	Dipart	Ufficio	Stipendio
Mario	Rossi	Amministrazione	10	45
Carlo	Bianchi	Produzione	20	36
Giovanni	Verdi	Amministrazione	20	40
Franco	Neri	Distribuzione	16	45
Carlo	Rossi	Direzione	14	80
Lorenzo	Gialli	Direzione	7	73
Paola	Rosati	Amministrazione	75	40
Marco	Franco	Produzione	20	46

Figura 4.21 Contenuto della tabella IMPIEGATO.

Dipart	Stipendio
Amministrazione	45
Produzione	36
Amministrazione	40
Distribuzione	45
Direzione	80
Direzione	73
Amministrazione	40
Produzione	46

Figura 4.22 Proiezione sugli attributi Dipart e Stipendio della tabella IMPIEGATO.

```
select Dipart, Stipendio
from Impiegato
```

Il risultato ottenuto a questo punto è mostrato in Figura 4.22.
La tabella ottenuta viene poi analizzata, dividendo le righe in insiemi caratterizzati dallo stesso valore degli attributi che compaiono come argomento della clausola `group by`. Nell'esempio le righe vengono raggruppate in base al valore dell'attributo Dipart; in Figura 4.23 compare il risultato del raggruppamento.

Dopo che le righe sono state raggruppate in sottoinsiemi, l'operatore aggregato viene applicato separatamente su ogni sottoinsieme. Il risultato dell'interrogazione è costituito da una tabella con righe che contengono l'esito della valutazione dell'operatore aggregato affiancato al valore dell'attributo che è stato usato per l'aggregazione. In Figura 4.24 compare il risultato finale dell'interrogazione, ovvero l'ammontare totale degli stipendi elargiti agli impiegati di ogni dipartimento, divisi per dipartimento.

Dipart	Stipendio
Amministrazione	45
Amministrazione	40
Amministrazione	40
Produzione	36
Produzione	46
Distribuzione	45
Direzione	80
Direzione	73

Figura 4.23 Raggruppamento in base al valore dell'attributo Dipart.

Dipart	sum(Stipendio)
Amministrazione	125
Produzione	82
Distribuzione	45
Direzione	153

Figura 4.24 Risultato dell'interrogazione 27.

La sintassi SQL impone che, in un'interrogazione che fa uso della clausola `group by`, possa comparire come argomento della `select` solamente un sottoinsieme degli attributi usati nella clausola `group by`. Per questi attributi, infatti, ciascuna tupla del sottoinsieme sarà caratterizzata dallo stesso valore. L'esempio seguente mostra i problemi che possono essere introdotti da interrogazioni che presentano nella clausola `select` attributi che non appaiono nella clausola `group by`.

Interrogazione 28:

```
select Ufficio
from Impiegato
group by Dipart
```

Questa interrogazione risulta scorretta, in quanto a ogni valore dell'attributo Dipart corrisponderanno diversi valori dell'attributo Ufficio. Dopo l'esecuzione del raggruppamento, invece, ogni sottoinsieme di righe deve corrispondere a una sola riga nella tabella risultato dell'interrogazione.

D'altra parte, questa restrizione può a volte risultare eccessiva, come quando si desidera mostrare il valore di attributi che possiedono valori univoci per un dato valore degli attributi di raggruppamento (si dice che gli attributi *dipendono funzionalmente* dagli attributi utilizzati per il raggruppamento; vedi il Paragrafo 9.2 per una discussione delle dipendenze funzionali).

Interrogazione 29:

```
select Dipart, count(*), D.Città
from Impiegato I join Dipartimento D
     on (I.Dipart = D.Nome)
group by Dipart
```

Questa interrogazione dovrebbe restituire i dipartimenti, il numero di impiegati di ciascun dipartimento, e la città in cui il dipartimento ha sede. Visto che l'attributo Nome è chiave di DIPARTIMENTO, a ogni valore di Dipart corrisponde un preciso valore di Città. Il sistema potrebbe quindi fornire una risposta corretta, ma SQL vieta interrogazioni di questo tipo. Teoricamente si potrebbe arricchire il linguaggio in modo tale da analizzare quali sono gli attributi chiave nello schema delle tabelle, e derivare quali attributi possono comparire come argomento della `select`. In pratica si preferisce mantenere la sintassi semplice, richiedendo eventualmente che l'interrogazione utilizzi un insieme di attributi di raggruppamento ridondante. L'interrogazione può infatti essere resa conforme alle regole sintattiche riscrivendola in questo modo:

Interrogazione 30:

```
select Dipart, count(*), D.Città
from Impiegato I join Dipartimento D
     on (I.Dipart = D.Nome)
group by Dipart, D.Città
```

Predicati sui gruppi Abbiamo visto come tramite la clausola `group by` le righe possano venire raggruppate in sottoinsiemi. Una applicazione può aver bisogno di considerare solo i sottoinsiemi che soddisfano certe condizioni. Se le condizioni che i sottoinsiemi devono soddisfare sono verificabili al livello delle singole righe, allora basta porre gli opportuni predicati come argomento della clausola `where`. Se invece le condizioni sono delle condizioni di tipo aggregato, sarà necessario utilizzare un nuovo costrutto, la clausola `having`.

La clausola `having` descrive le condizioni che si devono applicare al termine dell'esecuzione di un'interrogazione che fa uso della clausola `group by`. Ogni sottoinsieme di righe costruito dalla `group by` fa parte del risultato dell'interrogazione solo se il predicato argomento della `having` risulta soddisfatto.

Interrogazione 31: estrarre i dipartimenti che spendono più di 100 in stipendi.

```
select Dipart, sum(Stipendio) as SommaStipendi
from Impiegato
group by Dipart
having sum(Stipendio) > 100
```

Applicando l'interrogazione alla tabella rappresentata in Figura 4.21, si procede seguendo gli stessi passi descritti per le interrogazioni con `group by`. Dopo aver raggruppato le righe in base al valore dell'attributo Dipart, viene valutato il predicato

Dipart	SommaStipendi
Amministrazione	125
Direzione	153

Figura 4.25 Risultato dell'Interrogazione 31.

argomento della clausola `having`, che seleziona i dipartimenti per cui la somma degli stipendi, per tutti gli elementi del sottoinsieme, è superiore a 100. Il risultato dell'interrogazione è rappresentato dalla tabella in Figura 4.25.

La sintassi permette anche la definizione di interrogazioni che presentano la clausola `having` senza una corrispondente clausola `group by`. In questo caso, l'intero insieme di righe è trattato come un unico raggruppamento, ma questo ha in generale un limitato campo di applicabilità, perché, se la condizione non è soddisfatta, il risultato sarà vuoto. Come la clausola `where`, anche la clausola `having` ammette come argomento un'espressione booleana su predicati semplici. I predicati semplici sono normalmente confronti tra il risultato della valutazione di un operatore aggregato e una generica espressione; sintatticamente è ammessa anche la presenza diretta degli attributi argomento della `group by`, ma è preferibile raccogliere tutte le condizioni su questi attributi nell'ambito della clausola `where`. Per sapere quali predicati di un'interrogazione che fa uso del raggruppamento vanno dati come argomento della clausola `where` e quali come argomento della clausola `having`, basta rispettare il seguente criterio: solo i predicati in cui compaiono operatori aggregati devono essere argomento della clausola `having`.

Interrogazione 32: estrarre i dipartimenti per cui la media degli stipendi degli impiegati che lavorano nell'ufficio 20 è superiore a 25.

```
select Dipart
from Impiegato
where Ufficio = 20
group by Dipart
having avg(Stipendio) > 25
```

La forma sintattica generale di un'interrogazione SQL diventa (riassumendo i vari arricchimenti che abbiamo apportato nei paragrafi precedenti):

SelectSQL ::= `select` *ListaAttributiOEspressioni*
`from` *ListaTabelle*
[`where` *CondizioniSemplici*]
[`group by` *ListaAttributiDiRaggruppamento*]
[`having` *CondizioniAggregate*]
[`order by` *ListaAttributiDiOrdinamento*]

4.3.5 Interrogazioni di tipo insiemistico

SQL mette a disposizione anche degli operatori insiemistici, simili a quelli disponibili nell'algebra relazionale. Gli operatori disponibili sono gli operatori di `union` (unione), `intersect` (intersezione) ed `except` (chiamato anche `minus`, differenza), di significato analogo ai corrispondenti operatori dell'algebra relazionale.

Si noti che ogni interrogazione che faccia uso degli operatori `intersect` ed `except` può essere espressa utilizzando altri costrutti del linguaggio (tipicamente tramite interrogazioni nidificate, argomento del Paragrafo 4.3.6). Al contrario, come è già stato discusso nel Paragrafo 3.2.3 parlando del calcolo relazionale con dichiarazioni di range, l'operatore di unione arricchisce il potere espressivo di SQL e permette di scrivere interrogazioni altrimenti non formulabili.

La sintassi per l'uso degli operatori insiemistici è la seguente:

SelectSQL { (`union` | `intersect` | `except`) [`all`] *SelectSQL* }

Gli operatori insiemistici, al contrario del resto del linguaggio, assumono come default di eseguire una eliminazione dei duplicati. Ci sono due ragioni che giustificano questa differenza: in primo luogo, l'eliminazione dei duplicati rispetta molto meglio il tipico significato di questi operatori; in secondo luogo, l'esecuzione di queste operazioni (in particolare differenza e intersezione) richiede di effettuare un'analisi delle righe che rende molto limitato il costo aggiuntivo dell'eliminazione dei duplicati. Qualora nell'interrogazione si voglia adottare una diversa interpretazione degli operatori e si vogliano utilizzare gli operatori insiemistici che preservano i duplicati, sarà sufficiente utilizzare l'operatore con la parola chiave `all`. Negli esempi successivi confrontiamo il comportamento ottenuto usando l'una e l'altra delle scelte. Un'altra osservazione è che SQL non richiede che gli schemi su cui vengono effettuate le operazioni insiemistiche siano identici (come è invece richiesto dall'algebra relazionale), ma solo che gli attributi siano in pari numero e che abbiano domini compatibili. La corrispondenza tra gli attributi non si basa sul nome ma sulla posizione degli attributi. Se gli attributi hanno nome diverso, il risultato normalmente usa i nomi del primo operando.

Interrogazione 33: estrarre i nomi e i cognomi degli impiegati.

```
select Nome
from Impiegato
   union
select Cognome
from Impiegato
```

L'interrogazione ottiene dapprima i valori dell'attributo **Nome** per le righe di IMPIEGATO, ricava quindi i valori dell'attributo **Cognome** per le stesse righe, e infine costruisce la tabella risultato unendo i due risultati parziali. Visto che le operazioni insiemistiche eliminano i duplicati, non vi saranno elementi ripetuti nella tabella risultato, nonostante la presenza di duplicati in entrambe le tabelle di partenza e nonostante la presenza di alcuni valori identici in entrambe le tabelle. Supponendo che

Nome
Mario
Carlo
Giovanni
Franco
Lorenzo
Paola
Marco
Rossi
Bianchi
Verdi
Neri
Gialli
Rosati

Figura 4.26 Risultato dell'Interrogazione 33.

i dati di partenza siano quelli contenuti nella tabella in Figura 4.21, il risultato della valutazione dell'interrogazione è rappresentato in Figura 4.26.

Interrogazione 34: estrarre i nomi e i cognomi di tutti gli impiegati, eccetto quelli appartenenti al dipartimento Amministrazione, mantenendo i duplicati.

```
select Nome
from Impiegato
where Dipart <> 'Amministrazione'
   union all
select Cognome
from Impiegato
where Dipart <> 'Amministrazione'
```

In questo caso tutti i duplicati vengono tenuti e il risultato della query, sempre partendo dalla tabella in Figura 4.21, è quello rappresentato in Figura 4.27.

Interrogazione 35: estrarre i cognomi di impiegati che sono anche nomi.

```
select Nome
from Impiegato
   intersect
select Cognome
from Impiegato
```

Si ottiene da questa interrogazione il semplice risultato in Figura 4.28.

Interrogazione 36: estrarre i nomi degli impiegati che non sono cognomi di qualche impiegato.

Nome
Carlo
Franco
Carlo
Lorenzo
Marco
Bianchi
Neri
Rossi
Gialli
Franco

Figura 4.27 Risultato dell'Interrogazione 34.

Nome
Franco

Figura 4.28 Risultato dell'Interrogazione 35.

Nome
Mario
Carlo
Giovanni
Lorenzo
Paola
Marco

Figura 4.29 Risultato dell'Interrogazione 36.

```
select Nome
from Impiegato
   except
select Cognome
from Impiegato
```

Il risultato di questa interrogazione è mostrato in Figura 4.29.

4.3.6 Interrogazioni nidificate

Fino a ora abbiamo visto interrogazioni in cui l'argomento della clausola `where` si basa su condizioni composte (tramite gli operatori logici `and`, `or` e `not`) da pre-

dicati semplici, in cui ciascun predicato rappresenta un semplice confronto tra due valori. SQL ammette anche l'uso di predicati con una struttura più complessa, in cui si confronta un valore (ottenuto come risultato di una espressione valutata sulla singola riga) con il risultato dell'esecuzione di un'interrogazione SQL. L'interrogazione che viene usata per il confronto viene definita direttamente nel predicato interno alla clausola `where`. Si parla in questo caso di *interrogazioni nidificate*.

Nel caso più tipico, l'espressione che compare come primo membro del confronto è il semplice nome di un attributo. Se in un predicato si confronta un attributo con il risultato di un'interrogazione, sorge il problema di disomogeneità dei termini del confronto. Infatti, da una parte abbiamo il risultato dell'esecuzione di un'interrogazione SQL (in generale un insieme di valori), mentre dall'altra abbiamo il valore dell'attributo per la particolare riga. La soluzione offerta da SQL consiste nell'estendere, con le parole chiave `all` o `any`, i normali operatori di confronto (=, <>, <, >, <= e >=). La parola chiave `any` specifica che la riga soddisfa la condizione se risulta vero il confronto (con l'operatore specificato) tra il valore dell'attributo per la riga e almeno uno degli elementi restituiti dall'interrogazione. La parola chiave `all` invece specifica che la riga soddisfa la condizione solo se tutti gli elementi restituiti dall'interrogazione nidificata rendono vero il confronto. La sintassi richiede la compatibilità di dominio tra l'attributo restituito dall'interrogazione nidificata e l'attributo con cui avviene il confronto.

Interrogazione 37: estrarre gli impiegati che lavorano in dipartimenti situati a Firenze.

```
select *
from Impiegato
where Dipart = any (select Nome
                    from Dipartimento
                    where Città = 'Firenze')
```

L'interrogazione seleziona le righe di IMPIEGATO per cui il valore dell'attributo Dipart è uguale ad almeno uno dei valori dell'attributo Nome delle righe di DIPARTIMENTO.

Questa interrogazione poteva anche essere espressa mediante un join tra le tabelle IMPIEGATO e DIPARTIMENTO, e in effetti gli ottimizzatori sono generalmente in grado di trattare allo stesso modo le due diverse formulazioni di questa interrogazione. La scelta tra l'una e l'altra rappresentazione può essere dettata dal grado di leggibilità della soluzione. In casi così semplici non vi sono differenze significative, ma per interrogazioni più complicate la scomposizione in interrogazioni distinte può migliorare la leggibilità.

Consideriamo un'interrogazione che permette di trovare gli impiegati che hanno lo stesso nome di un impiegato del dipartimento Produzione. L'interrogazione ammette due formulazioni. La prima è più compatta e fa uso di variabili.

Interrogazione 38:

```
select I1.Nome
from Impiegato I1, Impiegato I2
where I1.Nome = I2.Nome and
      I2.Dipart = 'Produzione'
```

La seconda interrogazione fa uso di un'interrogazione nidificata, risolvendo l'interrogazione senza bisogno di introdurre alias.

Interrogazione 39:

```
select Nome
from Impiegato
where Nome = any (select Nome
                  from Impiegato
                  where Dipart = 'Produzione')
```

Consideriamo ora una diversa interrogazione.

Interrogazione 40: estrarre i dipartimenti in cui non lavorano persone di cognome "Rossi".

```
select Nome
from Dipartimento
where Nome <> all (select Dipart
                   from Impiegato
                   where Cognome = 'Rossi')
```

L'interrogazione nidificata seleziona i valori di Dipart di tutte le righe in cui il cognome vale "Rossi". La condizione è quindi soddisfatta da quelle righe di DIPARTIMENTO per cui il valore dell'attributo Nome non fa parte dei nomi prodotti dall'interrogazione nidificata. Questa interrogazione non poteva essere espressa mediante un join. È interessante notare che tale interrogazione poteva essere implementata in algebra relazionale con l'espressione:

$$(\pi_{\text{Nome}}(\text{DIPARTIMENTO}) - \pi_{\text{Dipart}}(\sigma_{\text{Cognome='Rossi'}}(\text{IMPIEGATO})))$$

quindi poteva anche essere espressa tramite l'operatore insiemistico `except`:

Interrogazione 41:

```
select Nome
from Dipartimento
   except
select Dipart
from Impiegato
where Cognome = 'Rossi'
```

Per rappresentare il controllo di appartenenza e di esclusione rispetto a un insieme, SQL mette a disposizione due appositi operatori, `in` e `not in`, i quali risultano del tutto identici agli operatori che abbiamo visto nei due precedenti esempi, `= any` e `<> all`. Mostreremo esempi del loro uso nel prossimo paragrafo.

Si può osservare infine come in alcuni casi interrogazioni che fanno uso degli operatori `max` e `min` possono essere rappresentate senza gli operatori stessi, tramite un uso opportuno delle interrogazioni nidificate.

Interrogazione 42: estrarre il dipartimento dell'impiegato che guadagna lo stipendio massimo (usando l'operatore aggregato `max`).

```
select Dipart
from Impiegato
where Stipendio = any
          (select max(Stipendio)
           from Impiegato)
```

Interrogazione 43: estrarre il dipartimento dell'impiegato che guadagna lo stipendio massimo (usando solo un'interrogazione nidificata).

```
select Dipart
from Impiegato
where Stipendio >= all
          (select Stipendio
           from Impiegato)
```

Le due interrogazioni sono equivalenti, in quanto il valore massimo è esattamente il valore che è superiore o uguale a tutti i valori dello stesso attributo nelle altre righe della relazione. In questi casi è comunque consigliabile l'utilizzo dell'operatore aggregato, che fornisce un'interrogazione più leggibile (e può essere implementato da qualche sistema in modo più efficiente). È anche interessante notare che nella prima interrogazione è indifferente usare le parole chiave `any` o `all`, poiché l'interrogazione nidificata restituisce sempre un unico valore; in effetti, la sintassi in quel caso ammette anche l'omissione della parola chiave `any`.

Interrogazioni nidificate complesse Un'interpretazione molto semplice e intuitiva delle interrogazioni nidificate consiste nell'assumere che l'interrogazione nidificata venga eseguita prima di analizzare le righe dell'interrogazione esterna. Il risultato dell'interrogazione può essere salvato in una tabella temporanea e il controllo sulle righe dell'interrogazione esterna può essere fatto accedendo direttamente al risultato temporaneo. Questa interpretazione corrisponde tra l'altro a un meccanismo di esecuzione efficiente, in cui l'interrogazione nidificata viene eseguita una sola volta. Consideriamo ancora l'Interrogazione 40. Il sistema può eseguire dapprima l'interrogazione nidificata, che estrae il valore dell'attributo Dipart per tutti gli impiegati di cognome "Rossi". Dopo aver fatto ciò, per ciascun dipartimento si controlla che il nome non sia incluso nella tabella prodotta, utilizzando l'operatore `<> all`.

Talvolta però l'interrogazione nidificata fa riferimento al contesto dell'interrogazione che la racchiude; tipicamente ciò accade tramite una variabile definita nell'ambito della query più esterna e usata nell'ambito della query più interna (si parla di un *passaggio di binding* da un contesto all'altro). La presenza del meccanismo di passaggio di binding arricchisce il potere espressivo di SQL. In questo caso l'interpretazione semplice data precedentemente alle query nidificate non vale più; bisogna a questo punto riconsiderare l'interpretazione standard delle interrogazioni SQL, per cui prima si costruisce il prodotto cartesiano delle tabelle e successivamente si applicano a ciascuna riga del prodotto le condizioni che compaiono nella clausola `where`.

L'interrogazione nidificata è un componente della clausola `where` e dovrà anch'essa essere valutata separatamente per ogni riga prodotta nella valutazione della query esterna.

Così, la nuova interpretazione è la seguente: per ogni riga della query esterna, valutiamo per prima cosa la query nidificata, quindi calcoliamo il predicato a livello di riga sulla query esterna. Tale processo può essere ripetuto un numero arbitrario di volte, pari al numero arbitrario di nidificazioni che possono essere utilizzate nella query; con query così complicate si perdono però le caratteristiche di leggibilità delle interrogazioni SQL.

Per quanto riguarda la *visibilità* (o *scope*) delle variabili SQL, vale la restrizione che una variabile è usabile solo nell'ambito della query in cui è definita o nell'ambito di una query nidificata (a un qualsiasi livello) all'interno di essa. Se un'interrogazione possiede interrogazioni nidificate allo stesso livello (su predicati distinti), le variabili introdotte nella clausola `from` di una query non potranno essere usate nell'ambito dell'altra query. Una interrogazione come la seguente, per esempio, è scorretta:

Interrogazione 44: estrarre gli impiegati che afferiscono al dipartimento Produzione o a un dipartimento che risiede nella stessa città del dipartimento Produzione (query scorretta).

```
select *
from Impiegato
where Dipart in (select Nome
                 from Dipartimento D1
                 where Nome = 'Produzione') or
      Dipart in (select Nome
                 from Dipartimento D2
                 where D1.Città = D2.Città)
```

La query non rispetta la sintassi SQL perché utilizza la variabile `D1` dove non è visibile.

Introduciamo ora l'operatore logico `exists`. Questo operatore ammette come parametro un'interrogazione nidificata e restituisce il valore vero solo se l'interrogazione fornisce un risultato non vuoto (corrisponde al quantificatore esistenziale della logica). Questo operatore può essere usato in modo significativo solo quando si ha un passaggio di binding tra l'interrogazione esterna e quella nidificata.

Si consideri una relazione che descrive dati anagrafici, avente il seguente schema: PERSONA(CodFiscale, Nome, Cognome, Città).

Interrogazione 45: estrarre le persone che hanno degli omonimi (ovvero persone con lo stesso nome e cognome, ma diverso codice fiscale).

```
select *
from Persona P
where exists (select *
              from Persona P1
              where P1.Nome = P.Nome and
                    P1.Cognome = P.Cognome and
                    P1.CodFiscale <> P.CodFiscale)
```

L'interrogazione ricerca le righe della tabella PERSONA per le quali esiste un'ulteriore riga in PERSONA con lo stesso Nome e Cognome, ma diverso CodFiscale.

Si può osservare che in questo caso non risulta possibile eseguire l'interrogazione nidificata prima di valutare l'interrogazione più esterna, in quanto senza avere associato un valore alla variabile P l'interrogazione nidificata non risulta completamente definita. Si richiede invece che venga prima valutata l'interrogazione esterna; per ogni singola riga esaminata nell'ambito dell'interrogazione esterna si deve valutare l'interrogazione nidificata. Così, nell'esempio, prima di tutto verranno considerate una a una le righe associate alla variabile P; per ciascuna di queste righe sarà poi eseguita l'interrogazione nidificata che restituirà o meno l'insieme vuoto a seconda che vi siano o meno degli omonimi della persona. Questa interrogazione si sarebbe potuta formulare anche con un join tra due diverse istanze della tabella PERSONA.

Interrogazione 46: estrarre le persone che hanno degli omonimi (senza query nidificata).

```
select P.*
from Persona P, Persona P1
where P1.Nome = P.Nome and
      P1.Cognome = P.Cognome and
      P1.CodFiscale <> P.CodFiscale)
```

Presentiamo ora la richiesta opposta alla precedente.

Interrogazione 47: estrarre le persone che *non* hanno degli omonimi.

```
select *
from Persona P
where not exists (select *
                  from Persona P1
                  where P1.Nome = P.Nome and
                        P1.Cognome = P.Cognome and
                        P1.CodFiscale <> P.CodFiscale)
```

L'intepretazione è analoga a quella dell'Interrogazione 45, con l'unica differenza che il predicato è soddisfatto nel caso che il risultato dell'interrogazione nidificata sia vuoto. Questa interrogazione poteva anche essere implementata con una differenza che sottraesse ai nomi e cognomi delle persone i nomi e cognomi delle persone che possiedono un omonimo, determinati tramite un join.

Un altro modo per formulare la stessa interrogazione può far uso del *costruttore di tupla*, rappresentato da una coppia di parentesi tonde che racchiudono la lista di attributi.

Interrogazione 48: estrarre le persone che *non* hanno degli omonimi.

```
select *
from Persona P
where (Nome,Cognome) not in
            (select Nome, Cognome
             from Persona Q
             where Q.CodFiscale <> P.CodFiscale)
```

Si consideri una base di dati con una tabella CANTANTE(Nome,Canzone) e una tabella AUTORE(Nome,Canzone).

Interrogazione 49: estrarre i cantautori puri, ovvero i cantanti che hanno eseguito solo canzoni di cui erano anche autori.

```
select Nome
from Cantante
where Nome not in
      (select Nome
       from Cantante C
       where Nome not in
             (select Nome
              from Autore
              where Autore.Canzone=C.Canzone))
```

La prima interrogazione nidificata (`select Nome from Cantante C ...`) non ha alcun legame con l'interrogazione esterna, e può quindi essere eseguita in modo del tutto indipendente. L'interrogazione al livello successivo invece presenta un legame (`Autore.Canzone = C.Canzone`). L'esecuzione dell'interrogazione può così avvenire seguendo queste fasi.

1. L'interrogazione `select Nome from Cantante C ...` legge tutte le righe della tabella CANTANTE.
2. Per ognuna delle righe di `C` viene valutata l'interrogazione più interna, che restituisce i nomi degli autori della canzone il cui titolo compare nella riga di `C` che viene considerata. Se il nome del cantante non compare tra gli autori (e quindi il cantante non è un cantautore puro), allora il nome viene selezionato.
3. Dopo che l'interrogazione nidificata ha terminato di analizzare le righe di `C`, costruendo la tabella contenente i nomi dei cantanti che non sono cantautori puri, viene eseguita l'interrogazione più esterna, la quale restituirà tutti i nomi di cantanti che non compaiono nella tabella ottenuta come risultato dell'interrogazione nidificata.

La correttezza di questa esecuzione appare evidente se si considera che la query può essere espressa in modo equivalente tramite l'operatore insiemistico `except`:

Interrogazione 50:

```
select Nome
from Cantante
   except
select Nome
from Cantante C
where Nome not in (select Nome
                   from Autore
                   where Autore.Canzone=C.Canzone)
```

Si noti che non è affatto detto che i sistemi SQL commerciali eseguano al loro interno l'interrogazione scandendo sempre la tabella esterna e producendo un'interrogazione per ogni riga di questa relazione. I sistemi cercano anzi di eseguire il più possibile le interrogazioni in un modo *set-oriented* (ovvero orientato agli insiemi), con l'obiettivo di effettuare poche operazioni su tanti dati. Per far questo, il sistema può trasformare l'interrogazione e cercare di applicare diverse ottimizzazioni, come la memorizzazione dei risultati delle query nidificate, o la scelta di un opportuno ordine di valutazione dei predicati.

4.4 Modifica dei dati in SQL

La parte di Data Manipulation Language comprende i comandi per interrogare e modificare il contenuto della base di dati. I comandi che permettono di modificare la base di dati sono `insert`, `delete` e `update`. Analizziamo separatamente i singoli comandi, anche se, come vedremo, sono tutti caratterizzati da uno schema simile.

4.4.1 Inserimento

Il comando di inserimento di righe nella base di dati presenta due sintassi alternative:

`insert into` *NomeTabella* [*ListaAttributi*]
{ `values (` *ListaValori*) |
SelectSQL }

La prima forma permette di inserire *singole* righe all'interno delle tabelle. L'argomento della clausola `values` rappresenta esplicitamente i valori degli attributi della singola riga. Per esempio:

```
insert into Dipartimento(NomeDip,Città)
            values('Produzione','Torino')
```

La seconda forma invece permette di aggiungere degli insiemi di righe, estratti dal contenuto della base di dati.

Il seguente comando inserisce nella tabella PRODOTTIMILANESI il risultato della selezione dalla relazione PRODOTTO di tutte le righe aventi "Milano" come valore dell'attributo LuogoProd.

```
insert into ProdottiMilanesi
        (select Codice, Descrizione
         from Prodotto
         where LuogoProd = 'Milano')
```

Ciascuna forma del comando possiede uno specifico campo di applicazione. La prima forma è quella tipicamente usata all'interno dei programmi per riempire una tabella con i dati forniti direttamente dagli utenti. Ogni uso del comando di `insert` è generalmente associato al riempimento di una *maschera* (o *form*), ovvero un'interfaccia di facile uso in cui all'utente vengono presentati sul video il nome dei vari attributi e appositi spazi in cui immettere i relativi valori. La seconda forma permette invece di inserire dati in una tabella a partire da altre informazioni presenti nella base di dati.

Se in un inserimento non vengono specificati i valori di tutti gli attributi della tabella, agli attributi mancanti viene assegnato il valore di default, o in assenza di questo il valore nullo; come è stato già detto nel Paragrafo 4.2.6, se l'inserimento viola un vincolo di *not null* definito sull'attributo, l'inserimento viene rifiutato. Si noti infine che la corrispondenza tra gli attributi della tabella e i valori da inserire è data dall'ordine in cui compaiono i termini nella definizione della tabella. Perciò, al primo attributo che compare in *ListaValori* (per la prima forma del comando) o al primo elemento della clausola `select` (per la seconda forma) deve corrispondere il primo attributo che compare in *ListaAttributi* (o nella definizione della tabella se *ListaAttributi* è omesso), e così via per gli altri attributi.

4.4.2 Cancellazione

Il comando `delete` elimina righe dalle tabelle della base di dati, seguendo la semplice sintassi:

`delete from` *NomeTabella* [`where` *Condizione*]

Quando la condizione argomento della clausola `where` non viene specificata, il comando cancella tutte le righe dalla tabella, altrimenti vengono rimosse solo le righe che soddisfano la condizione. Si ricorda che qualora esista un vincolo di integrità referenziale con politica di `cascade` in cui la tabella viene referenziata, allora la cancellazione di righe dalla tabella può comportare la cancellazione di righe appartenenti ad altre tabelle (e si può generare una reazione a catena se queste cancellazioni a loro volta causano la cancellazione di righe di altre tabelle).

```
delete from Dipartimento
        where NomeDip = 'Produzione'
```

Il comando elimina la riga di DIPARTIMENTO avente nome "Produzione" (visto che NomeDip era stata dichiarata come `primary key` della tabella, vi può essere una sola riga avente quel valore).

La condizione rispetta la sintassi della `select`, per cui possono comparire al suo interno anche interrogazioni nidificate che fanno riferimento ad altre tabelle. Un semplice esempio è il comando che elimina i dipartimenti senza impiegati:

```
delete from Dipartimento
        where Nome not in (select Dipart
                           from Impiegato)
```

Si noti la differenza tra il comando `delete` appena visto e il comando `drop` descritto nel Paragrafo 4.2.8. Un comando come:

```
delete from Dipartimento
```

elimina tutte le righe dalla tabella DIPARTIMENTO, eventualmente eliminando anche tutte le righe dalle tabelle che sono legate da vincolo di integrità referenziale con la tabella, se per il vincolo è specificata la politica `cascade` sull'evento di cancellazione. Lo schema della base di dati rimane però immutato, e il comando modifica solamente l'istanza della base di dati. Il comando:

```
drop table Dipartimento cascade
```

ha lo stesso effetto del comando `delete`, ma in più anche lo schema della base di dati viene modificato, eliminando dallo schema non solo la tabella DIPARTIMENTO, ma anche tutte le viste e tabelle che nella loro definizione fanno riferimento a essa. Invece, il comando:

```
drop table Dipartimento restrict
```

fallisce se vi sono righe nella tabella DIPARTIMENTO.

4.4.3 Modifica

Il comando di `update` presenta una sintassi leggermente più complicata:

`update` *NomeTabella*
  `set` *Attributo* = ⟨ *Espressione* | *SelectSQL* | `null` | `default` ⟩
  {, *Attributo* = ⟨ *Espressione* | *SelectSQL* | `null` | `default` ⟩ }
[`where` *Condizione*]

Il comando di `update` permette di aggiornare uno o più attributi delle righe di *NomeTabella* che soddisfano l'eventuale *Condizione*. Se il comando non presenta la clausola `where`, come al solito si suppone che la condizione sia soddisfatta e si esegue la modifica su tutte le righe. Il nuovo valore cui viene posto l'attributo può essere:

1. il risultato della valutazione di un'espressione sugli attributi della tabella, che può anche far riferimento al valore corrente dell'attributo che verrà modificato dal comando;
2. il risultato di una generica interrogazione SQL;
3. il valore nullo;
4. il valore di default per il dominio.

Il comando:

```
update Dipendente
       set Stipendio = StipendioBase + 5
       where Matricola = 'M2047'
```

opera su una singola riga, aggiornando lo stipendio del dipendente con matricola M2047, mentre l'esempio successivo opera su un insieme di righe:

```
update Impiegato
       set Stipendio = Stipendio * 1.1
       where Dipart = 'Amministrazione'
```

Il comando aumenta del 10% lo stipendio di tutti gli impiegati che lavorano in Amministrazione. L'operatore di assegnamento = ha un comportamento analogo a quello dei normali linguaggi di programmazione, per cui `Stipendio` sul lato destro dell'operatore rappresenta il vecchio valore dell'attributo, valutato per ogni riga su cui deve essere applicato l'aggiornamento. Il risultato dell'espressione diventa il nuovo valore dell'attributo.

La natura *set-oriented* di SQL presenta alcune particolarità di cui bisogna tenere conto quando si scrivono comandi di aggiornamento. Supponiamo che si vogliano modificare gli stipendi dei dipendenti, aumentando del 10% gli stipendi inferiori a 30, e del 15% gli stipendi superiori. Un modo per aggiornare in questo modo la base di dati consiste nell'eseguire questo comando:

```
update Impiegato
       set Stipendio = Stipendio * 1.1
       where Stipendio <= 30

update Impiegato
       set Stipendio = Stipendio * 1.15
       where Stipendio > 30
```

Il problema di questa soluzione è che se consideriamo un dipendente con uno stipendio iniziale di 30, questo soddisferà la condizione del primo comando di aggiornamento, per cui l'attributo `Stipendio` verrà posto pari a 33. Ma a questo punto la riga soddisferà anche le condizioni del secondo comando di aggiornamento, per cui lo stipendio sarà di nuovo modificato. Il risultato finale è che per questa riga l'aumento complessivo risulta del 26,5%, violando quindi i requisiti di partenza.

Il problema ha origine nel carattere *set-oriented* di SQL. Con un linguaggio *tuple-oriented* sarebbe possibile analizzare le righe una a una e applicare o l'una o l'altra delle modifiche a seconda del valore dello stipendio. In questo caso una semplice soluzione consiste nell'invertire l'ordine di esecuzione dei due comandi, aumentando prima gli stipendi superiori e poi i rimanenti. In casi più complicati la soluzione può introdurre degli aggiornamenti intermedi, fare uso del costrutto `case` (Paragrafo 5.2.2) o cambiare completamente approccio e scrivere un programma in un tradizionale linguaggio di programmazione di alto livello, realizzando all'interno del programma, per esempio tramite l'uso di cursori, gli aggiornamenti desiderati (descriveremo il funzionamento dei cursori nel Paragrafo 10.1.1).

4.5 Esempi riepilogativi

1. Dato il seguente schema relazionale che descrive il calendario di una manifestazione sportiva a squadre nazionali:

STADIO(Nome,Citta,Capienza)
INCONTRO(NomeStadio,Data,Ora,Squadra1,Squadra2)
NAZIONALE(Paese,Continente,Categoria)

Esprimere in SQL le seguenti interrogazioni:

(a) Estrarre i nomi degli stadi in cui non gioca nessuna nazionale europea.
Soluzione:

```
select Nome
from Stadio
where Nome not in
      (select NomeStadio
       from Incontro
       where (Squadra1 in
                 (select Paese
                  from Nazionale
                  where Continente = 'Europa' ))
             or
             (Squadra2 in
                (select Paese
                 from Nazionale
                 where Continente = 'Europa' )))
```

(b) Esprimere l'interrogazione in algebra relazionale, in calcolo e in Datalog.
Soluzione:

i. Algebra relazionale:

$$\pi_{\text{Nome}}(\text{STADIO}) - \pi_{\text{NomeStadio}}((\pi_{\text{Paese}}(\sigma_{\text{Continente}='Europa'}\text{NAZIONALE})) \bowtie_{\text{Squadra1}=\text{Paese}\vee\text{Squadra2}=\text{Paese}} (\pi_{\text{NomeStadio,Squadra1,Squadra2}}(\text{INCONTRO})))$$

ii. Calcolo relazionale:

$$\{ s.\text{Nome} \mid s(\text{STADIO}) \mid \neg(\exists i(\text{INCONTRO})(\exists n(\text{NAZIONALE}) (i.\text{NomeStadio} = s.\text{Nome} \wedge n.\text{Continente} = 'Europa' \wedge (i.\text{Squadra1} = n.\text{Paese} \vee i.\text{Squadra2} = n.\text{Paese}))))\}$$

iii. Datalog:

STADIOCONEUROPEA(NomeStadio : n) ←
INCONTRO(NomeStadio : n, Data : d, Ora : o,
Squadra1 : s_1, Squadra2 : s_2),
NAZIONALE(Paese : s_1, Continente : c, Categoria : ct),
c = 'Europa'

STADIOCONEUROPEA(NomeStadio : n) ←
INCONTRO(NomeStadio : n, Data : d, Ora : o,
Squadra1 : s_1, Squadra2 : s_2),
NAZIONALE(Paese : s_2, Continente : c, Categoria : ct),
c = 'Europa'

?STADIO(Nome : n, Citta : c, Capienza : cp),
NOT STADIOCONEUROPEA(NomeStadio : n)

(c) Estrarre la capienza complessiva degli stadi in cui si giocano le partite che hanno come prima squadra una nazione sudamericana (nota: ai fini della valutazione della capienza complessiva, si sommino le capienze associate a ciascuna gara, anche se più gare si svolgono nello stesso stadio).
Soluzione:

```
select sum(Capienza)
from Stadio join Incontro on Nome = NomeStadio
where Squadra1 in select Paese
                  from Nazionale
                  where Continente = 'Sudamerica'
```

2. Dato il seguente schema relazionale:

MOTO(Targa,Cilindrata,Marca,Nazione,Tasse)
PROPRIETARIO(Nome,Targa)

Scrivere in SQL le interrogazioni seguenti:

(a) Estrarre i nomi dei proprietari di solo moto giapponesi di almeno due marche diverse.

i. Prima soluzione:

```
select Nome
from Proprietario join Moto
     on Proprietario.Targa=Moto.Targa
where Nome not in
            (select Nome
             from Proprietario join Moto on
                  Proprietario.Targa=Moto.Targa
             where Nazione <> 'Giappone')
group by Nome
having count(distinct Marca) >= 2
```

ii. Seconda soluzione:

```
select P1.Nome
from Proprietario P1, Moto M1,
     Proprietario P2, Moto M2
where P1.Nome not in
                (select Nome
                 from Proprietario join Moto on
                      Proprietario.Targa=Moto.Targa
                 where Nazione <> 'Giappone') and
        P1.Targa = M1.Targa and
        P2.Targa = M2.Targa and
        P1.Nome = P2.Nome and
        M1.Marca <> M2.Marca
```

(b) Rappresentare la query in algebra relazionale.
Soluzione:

$$\pi_{\text{Nome}}(\,(\text{PROPRIETARIO} \bowtie \text{MOTO}) \bowtie_{\text{Marca}\neq\text{Marca2}\wedge\text{Nome}=\text{Nome2}} (\rho_{\text{Nome2}\leftarrow\text{Nome}}(\text{PROPRIETARIO}) \bowtie \rho_{\text{Marca2}\leftarrow\text{Marca}}(\text{MOTO}))) - \pi_{\text{Nome}}(\text{PROPRIETARIO} \bowtie \sigma_{\text{Nazione}\neq\text{'Giappone'}}\text{MOTO})$$

Note bibliografiche

SQL è stato inizialmente descritto da Chamberlin *et al.* in [25] e [26]. Studi sistematici del linguaggio SQL, relativamente alle tecniche di ottimizzazione e al significato delle query, sono contenuti in [21] e [51].

La descrizione ufficiale dello standard SQL può essere ottenuta dall'Organizzazione internazionale degli standard ISO. Tali documenti sono però di costo elevato e di lettura non molto agevole. Su SQL esistono un gran numero di libri, tra i quali il libro di Cannan e Otten [17] (di cui esiste la traduzione italiana) e il libro di Melton e Simon [60]. Eisenberg e Melton [39] discutono il processo di standardizzazione in generale e con riferimenti specifici all'area delle basi di dati. Melton e Simon [61] presentano le principali caratteristiche di SQL-3.

Spesso i manuali che accompagnano i sistemi relazionali commerciali sono fatti con molta cura e possono costituire un ottimo punto di riferimento. Tra l'altro questi manuali sono indispensabili per conoscere quali funzionalità di SQL sono state effettivamente implementate nel particolare sistema.

Esercizi

Soluzioni sul sito http://www.ateneonline.it/atzeni

4.1 Ordinare i seguenti domini in base al valore massimo rappresentabile, supponendo che `integer` abbia una rappresentazione a 32 bit e `smallint` a 16 bit: `numeric (12,4)`, `decimal (10)`, `decimal (9)`, `integer`, `smallint`, `decimal (6,1)`.

4.2 Definire un attributo che permetta di rappresentare stringhe di lunghezza massima pari a 256 caratteri, su cui non sono ammessi valori nulli e con valore di default "sconosciuto".

4.3 Dare le definizioni SQL delle tre tabelle FONDISTA(Nome, Nazione, Età), GAREGGIA(NomeFondista, NomeGara, Piazzamento) e GARA(Nome, Luogo, Nazione, Lunghezza), rappresentando in particolare i vincoli di foreign key della tabella GAREGGIA.

4.4 Dare le definizioni SQL delle tabelle AUTORE(Nome,Cognome, DataNascita, Nazionalità), LIBRO(TitoloLibro, NomeAutore, CognomeAutore, Lingua). Per il vincolo di *foreign key* specificare una politica di `cascade` sulle cancellazione e di `set null` sulle modifiche.

4.5 Dato lo schema dell'esercizio precedente, spiegare cosa può capitare con l'esecuzione dei seguenti comandi di aggiornamento:

```
delete from Autore
       where Cognome = 'Rossi'
update Libro set Nome = 'Umberto'
       where Cognome = 'Eco'
insert into Autore(Nome,Cognome)
       values('Antonio','Bianchi')
update Autore set Nome = 'Italo'
       where Cognome = 'Calvino'
```

4.6 Date le definizioni:

```
create domain Dominio integer default 10
create table Tabella(Attributo Dominio default 5)
```

indicare cosa avviene in seguito ai comandi:

```
alter table Tabella
      alter column Attributo drop default
alter domain Dominio drop default
drop domain Dominio
```

4.7 Con riferimento a una relazione PROFESSORI(CF, Nome, Età, Qualifica), scrivere le interrogazioni SQL che calcolano l'età media dei professori di ciascuna qualifica, nei due casi seguenti:

1. se l'età non è nota si usa per essa il valore nullo
2. se l'età non è nota si usa per essa il valore 0

4.8 Spiegare perché in SQL è previsto (e necessario) un operatore di unione mentre in molte versioni non esistono gli operatori di intersezione e differenza.

4.9 Considerare le relazioni IMPIEGATI (Matricola, Nome, Stipendio, Direttore) e DIPARTIMENTI (Codice, Direttore) e le due interrogazioni seguenti, specificare se e in quali casi esse possono produrre risultati diversi:

```
select avg(Stipendio)
from Impiegato
where Direttore in (select Direttore
                    from   Dipartimento)

select avg(Stipendio)
from Impiegato I, Dipartimento D
where I.Direttore = D.Direttore
```

4.10 Si consideri una base di dati sulle relazioni:

$$R_1(\underline{A}, B, C)$$
$$R_2(\underline{D}, \underline{E}, F)$$

Facendo riferimento a una versione dell'SQL che non prevede la differenza (parole chiave `EXCEPT` e `MINUS`) e che permette l'uso dei confronti nella nidificazione solo su singoli attributi (e quindi non ammette condizioni del tipo ... `(A,B) IN SELECT C,D FROM` ...), scrivere interrogazioni in SQL equivalenti alle seguenti espressioni dell'algebra relazionale:

$$\pi_{BC}(\sigma_{C>10}(R_1))$$

$$\pi_{B}(R_1 \bowtie_{C=D} \sigma_{F=2}(R_2))$$

$$\pi_{AB}(R_1) - \pi_{AB}(R_1 \bowtie_{C=D} R_2))$$

4.11 Con riferimento alla base di dati nell'Esercizio 4.10 scrivere espressioni dell'algebra relazionale equivalenti alle seguenti interrogazioni SQL:

```
1. select distinct A , B
   from R1, R2
   where C = D and E > 100
2. select distinct A , B
   from R1 X1
   where not exists
   (select *
   from R1 Y1, R2
   where Y1.C = D and X1.A = Y1.A and F>10)
```

4.12 Con riferimento alla base di dati nell'Esercizio 4.10, indicare, per ciascuna delle seguenti interrogazioni, se la parola chiave `distinct` è necessaria.

1. l'interrogazione 1 nell'Esercizio 4.11
2. l'interrogazione 2 nell'Esercizio 4.11
3.
```
select distinct A , B
from R1, R2
where B = D and C = E
```
4.
```
select distinct B , C
from R1, R2
where B = D and C = E
```

4.13 Con riferimento a una base di dati sullo schema R_1(A,B,C), R_2(A,B,C), R_3(C,D,E) considerare l'espressione dell'algebra relazionale $\pi_{AE}((R_1 \cup R_2) \bowtie R_3)$ e scrivere

un'espressione SQL a essa equivalente senza utilizzare il join esplicito (cioè la parola chiave JOIN) né viste.

4.14 Dato il seguente schema:

AEROPORTO(Città,Nazione,NumPiste)
VOLO(IdVolo,GiornoSett,CittaPart,OraPart,CittaArr,OraArr,TipoAereo)
AEREO(TipoAereo,NumPasseggeri,QtaMerci)

scrivere le interrogazioni SQL che permettono di determinare:

1. le città con un aeroporto di cui non è noto il numero di piste;
2. le nazioni da cui parte e arriva il volo con codice AZ274;
3. i tipi di aereo usati nei voli che partono da Torino;
4. i tipi di aereo e il corrispondente numero di passeggeri per i tipi di aereo usati nei voli che partono da Torino. Se la descrizione dell'aereo non è disponibile, visualizzare solamente il tipo;
5. le città da cui partono voli internazionali;
6. le città da cui partono voli diretti a Bologna, ordinate alfabeticamente;
7. il numero di voli internazionali che partono il giovedì da Napoli;
8. il numero di voli internazionali che partono ogni settimana da città italiane (farlo in due modi, facendo comparire o meno nel risultato gli aeroporti senza voli internazionali);
9. le città francesi da cui partono più di venti voli alla settimana diretti in Italia;
10. gli aeroporti italiani che hanno solo voli interni. Rappresentare questa interrogazione in quattro modi: (i) con operatori insiemistici, (ii) con un'interrogazione nidificata con l'operatore `not in`, (iii) con un'interrogazione nidificata con l'operatore `not exists`, (iv) con l'outer join e l'operatore di conteggio. Esprimere l'interrogazione pure in algebra relazionale;
11. le città che sono servite dall'aereo caratterizzato dal massimo numero di passeggeri.

4.15 Dato il seguente schema:

DISCO(NroSerie,TitoloAlbum,Anno,Prezzo)
CONTIENE(NroSerieDisco,CodiceReg,NroProgr)
ESECUZIONE(CodiceReg,TitoloCanz,Anno)
AUTORE(Nome,TitoloCanzone)
CANTANTE(NomeCantante,CodiceReg)

formulare le interrogazioni SQL che permettono di determinare:

1. i cantautori (persone che hanno scritto e cantato la stessa canzone) il cui nome inizia per 'D';
2. i titoli dei dischi che contengono canzoni di cui non si conosce l'anno di registrazione;
3. i pezzi del disco con numero di serie 78574, ordinati per numero progressivo, con indicazione degli interpreti per i pezzi che hanno associato un cantante;
4. gli autori e i cantanti puri, ovvero autori che non hanno mai registrato una canzone e cantanti che non hanno mai scritto una canzone;
5. i cantanti del disco che contiene il maggior numero di canzoni;
6. gli autori solisti di "collezioni di successi" (dischi in cui tutte le canzoni sono di un solo cantante e in cui almeno tre registrazioni sono di anni precedenti la pubblicazione del disco);
7. i cantanti che non hanno mai registrato una canzone come solisti;

8. i cantanti che non hanno mai inciso un disco in cui comparissero come unici cantanti;
9. i cantanti che hanno sempre registrato canzoni come solisti.

4.16 Considerare la base di dati relazionale definita per mezzo delle seguenti istruzioni:

```
create table Studenti (
  Matricola numeric not null primary key,
  Cognome char(20) not null,
  Nome char(20) not null,
  DataNascita date not null
);


create table Esami (
  CodiceCorso numeric not null,
  Studente numeric not null
        references Studenti(Matricola),
  Data date not null,
  Voto numeric not null,
  primary key (CodiceCorso, Studente, Data)
);
```

Si supponga che vengano registrati anche gli esami non superati, con voti inferiori al 18.
Formulare in SQL:

1. l'interrogazione che trova gli studenti che non hanno superato esami;
2. l'interrogazione che trova gli studenti che hanno riportato in almeno un esame un voto più alto di Archimede Pitagorico;
3. l'interrogazione che trova i nomi degli studenti che hanno superato almeno due esami;
4. l'interrogazione che trova, per ogni studente, il numero di esami superati e la relativa media;

4.17 Considerare la seguente base di dati relazionale:

NEGOZI(IDNegozio, Nome, Città)
PRODOTTI(CodProdotto, NomeProdotto , Marca)
LISTINO(Negozio, Prodotto, Prezzo)

con vincoli di integrità referenziale fra Negozio e la relazione NEGOZI fra Prodotto e la relazione PRODOTTI.
Fare riferimento a una versione dell'SQL che non prevede la differenza (parole chiave `except` e `minus`) e che permette l'uso dei confronti nella nidificazione solo su singoli attributi (quindi sono ammesse condizioni del tipo ... `A in select C from` ... ma non del tipo ... `(A,B) in select C, D from` ...

- Formulare in SQL l'interrogazione che fornisce nome e città dei negozi che vendono prodotti della marca XYZ
- Formulare in SQL l'interrogazione che trova, per ciascun prodotto, la città in cui viene venduto al prezzo più basso
- Formulare in SQL l'interrogazione che trova i prodotti che vengono venduti in una sola città.

4.18 Dare una sequenza di comandi di aggiornamento che modifichi l'attributo Stipendio della tabella IMPIEGATO, aumentando del 10% gli stipendi inferiori a 30 e diminuendo del 5% gli stipendi superiori a 30.

5 SQL: caratteristiche evolute

Il capitolo continua la presentazione delle caratteristiche del linguaggio SQL, mostrandone alcuni aspetti evoluti. Completeremo la descrizione dei servizi per la definizione dei dati, mostrando l'uso del linguaggio di interrogazione per questo scopo (Paragrafo 5.1). Illustreremo poi l'uso di funzioni scalari (Paragrafo 5.2).

Il linguaggio SQL non si limita alla definizione di semplici query o comandi di modifica, ma permette la definizione di componenti più estese, quali le procedure e i trigger. Tramite queste funzionalità è possibile aggregare diverse azioni SQL ed estendere l'insieme di servizi del sistema; in generale, queste estensioni rendono SQL analogo a un normale linguaggio di programmazione, permettendo in alcuni casi di realizzare completamente l'applicazione all'interno della base di dati. In questo capitolo presenteremo quindi l'estensione di SQL che permette la definizione di procedure (Paragrafo 5.3) e trigger (Paragrafo 5.4). Descriveremo poi i comandi per il controllo dell'accesso ai dati (Paragrafo 5.5) e infine introdurremo il concetto di transazione (Paragrafo 5.6).

5.1 Caratteristiche evolute di definizione dei dati

Dopo aver descritto nel capitolo precedente i comandi di base per la definizione dei dati e la scrittura delle interrogazioni in SQL, completiamo ora la rassegna dei componenti di uno schema. Descriviamo quindi la clausola `check`, le asserzioni e le primitive per la definizione di viste.

5.1.1 Vincoli di integrità generici

Abbiamo visto che SQL permette di specificare un certo insieme di vincoli sugli attributi e sulle tabelle, soddisfacendo le esigenze di verifica dell'integrità dei dati più basilari. Per specificare vincoli più complessi, SQL-2 offre la clausola `check`, con la seguente sintassi:

`check` (*Condizione*)

Le condizioni ammissibili sono le stesse che possono apparire come argomento della clausola `where` di una interrogazione SQL. La condizione deve essere sempre verificata affinché la base di dati sia corretta. In questo modo è possibile specificare tutti i vincoli intrarelazionali descritti nel Paragrafo 4.2.6, e anche di più, poiché la condizione può far riferimento ad altri attributi della relazione.

Una dimostrazione della potenza del costrutto consiste nel mostrare come tutti i vincoli predefiniti possano essere descritti con la clausola check. Per questo possiamo riprendere la definizione dello schema della tabella IMPIEGATO che è stata data nel Paragrafo 4.2.7 (la prima delle tre):

```
create table Impiegato
(Matricola character(6)
           check (Matricola is not null and
                  1 = (select count(*)
                       from Impiegato I
                       where Matricola=I.Matricola)),
 Cognome   character(20) check (Cognome is not null),
 Nome      character(20) check (Nome is not null and
                  2 > (select count(*)
                       from Impiegato I
                       where Nome = I.Nome
                       and Cognome = I.Cognome)),
 Dipart    character(15) check (Dipart in
                       (select NomeDip
                        from Dipartimento))
)
```

Confrontando questa specifica con quella che faceva uso dei vincoli predefiniti, possiamo fare diverse osservazioni. In primo luogo, i vincoli predefiniti permettono una rappresentazione molto più compatta e leggibile; per esempio il vincolo di chiave ha bisogno di una rappresentazione abbastanza complicata, che fa uso dell'operatore aggregato count. Si nota anche che utilizzando la clausola check si perde la possibilità di associare ai vincoli una politica di reazione alle violazioni. Infine, quando i vincoli sono espressi mediante i costrutti predefiniti, il sistema li può riconoscere immediatamente e spesso può riuscire a gestirli in modo più efficiente.

Per apprezzare il potere espressivo della clausola check si può esprimere un vincolo che richiede che un impiegato abbia un manager del proprio dipartimento, a meno che la matricola non inizi con la cifra 1. Per far ciò, si deve estendere la definizione della tabella IMPIEGATO con le dichiarazioni seguenti:

```
Superiore character(6),
check (Matricola like '1%' or
       Dipart = (select Dipart
                 from Impiegato I
                 where I.Matricola = Superiore))
```

5.1.2 Asserzioni

Grazie alla clausola check è possibile definire anche un ulteriore componente dello schema di una base di dati, le *asserzioni*. Le asserzioni, introdotte in SQL-2, rappresentano dei vincoli che non sono associati a un attributo o a una tabella in particolare, bensì appartengono direttamente allo schema.

Mediante le asserzioni è possibile esprimere tutti i vincoli che abbiamo specificato fin qui nella definizione delle tabelle. Le asserzioni permettono inoltre di esprimere vincoli che non sarebbero altrimenti definibili: vincoli che coinvolgono più tabelle o che richiedono che una tabella abbia una cardinalità minima. Le asserzioni possiedono un nome, tramite il quale possono essere eliminate esplicitamente dallo schema con l'istruzione `drop` (Paragrafo 4.2.8).

La sintassi per la definizione delle asserzioni è:

`create assertion` *NomeAsserzione* `check` (*Condizione*)

Un'asserzione può per esempio imporre che nella tabella IMPIEGATO sia sempre presente almeno una riga:

```
create assertion AlmenoUnImpiegato
        check (1 <= (select count(*)
                     from impiegato))
```

Ogni vincolo d'integrità, definito tramite `check` o tramite asserzione, è associato a una politica di controllo che specifica se il vincolo è immediato o differito. I vincoli immediati sono verificati immediatamente dopo ogni modifica della base di dati, mentre i vincoli differiti sono verificati solo al termine dell'esecuzione di una serie di operazioni (che costituisce una transazione, si veda il Paragrafo 5.6).

Il controllo differito viene utilizzato per gestire casi in cui non è possibile costruire uno stato consistente della base di dati con una singola modifica. L'esempio classico è costituito da una coppia di vincoli di integrità incrociati. Supponiamo che la tabella IMPIEGATO presenti nel proprio schema un attributo Dipart con vincolo *not null* associato a un vincolo di integrità referenziale verso la tabella DIPARTIMENTO, e la tabella DIPARTIMENTO presenti a sua volta un attributo Direttore *not null* associato a un vincolo di integrità referenziale verso la tabella IMPIEGATO. A questo punto, se entrambi i vincoli fossero immediati non sarebbe possibile modificare lo stato iniziale vuoto delle due tabelle, in quanto ogni singolo comando di inserimento di tuple non rispetterebbe il vincolo di integrità referenziale. Il modo differito permette di gestire agevolmente questa situazione.

Quando un vincolo immediato non è soddisfatto, l'operazione di modifica che ha causato la violazione è stata appena eseguita e il sistema può "disfarla"; questo modo di procedere è chiamato *rollback parziale*. Tutti i vincoli predefiniti, introdotti nel Paragrafo 4.2.6 (*not null*, *unique* e *primary key*) e nel Paragrafo 4.2.7 (*foreign key*) sono per default verificati in modo immediato e la loro violazione causa un rollback parziale. Quando invece si rileva una violazione di un vincolo differito al termine della transazione, non c'è modo di individuare l'operazione che ha causato la violazione, e perciò diventa necessario disfare l'intera sequenza di operazioni che costituiscono la transazione; in questo caso si esegue un *rollback*. *Grazie a questi meccanismi, l'esecuzione di un comando di modifica dell'istanza di una base di dati che soddisfa tutti i vincoli, immediati e differiti, produrrà sempre un'istanza della base di dati che pure soddisfa tutti i vincoli (si dice anche che lo stato della base di dati è* consistente*)*. È possibile cambiare il tipo di controllo associato ai vincoli nell'ambito di una transazione, assegnando la modalità immediata o differita. Ciò avviene

tramite i comandi `set constraints [ NomeVincoli | all ] immediate` e `set constraints [ NomeVincoli | all ] deferred`, che modificano la modalità di controllo dei vincoli nominati, o di tutti i vincoli se si usa l'opzione `all`, limitatamente alla transazione che ha invocato il comando.

5.1.3 Viste

Nel Capitolo 3 sono state introdotte le viste, ovvero tabelle "virtuali" il cui contenuto dipende dal contenuto delle altre tabelle di una base di dati. Le viste vengono definite in SQL associando un nome e una lista di attributi al risultato dell'esecuzione di una interrogazione. Nell'interrogazione che definisce la vista possono comparire anche altre viste. Si definisce una vista utilizzando il comando:

```
create view NomeVista [ ( ListaAttributi ) ] as SelectSQL
[ with [ local | cascaded ] check option ]
```

L'interrogazione SQL deve restituire un insieme di attributi compatibile con gli attributi nello schema della vista; l'ordine nella clausola `select` deve corrispondere all'ordine degli attributi nello schema. Si può per esempio definire una vista IMPIEGATIAMMIN che contiene tutti gli impiegati del dipartimento Amministrazione con uno stipendio superiore a 10:

```
create view ImpiegatiAmmin(Matricola, Nome,
                           Cognome, Stipendio) as
select Matricola, Nome, Cognome, Stipendio
from Impiegato
where Dipart = 'Amministrazione' and
      Stipendio > 10
```

Costruiamo quindi una vista IMPIEGATIAMMINPOVERI definita a partire dalla vista IMPIEGATIAMMIN, che conterrà gli impiegati amministrativi con uno stipendio compreso tra 10 e 50:

```
create view ImpiegatiAmminPoveri as
select *
from ImpiegatiAmmin
where Stipendio < 50
with check option
```

Su certe viste è permesso effettuare operazioni di modifica, che verranno tradotte negli opportuni comandi di modifica al livello delle tabelle di base da cui la vista dipende. Come è già stato accennato nel Paragrafo 3.1.9, non è sempre possibile determinare un modo univoco in cui la modifica sulla vista possa essere riportata sulle tabelle di base; si incontrano problemi soprattutto quando la vista è definita tramite un join tra più tabelle. Lo standard SQL permette che una vista sia aggiornabile solo quando una sola riga di ciascuna tabella di base corrisponde a una riga della vista.

I sistemi commerciali tipicamente considerano una vista aggiornabile solo se è definita su una sola tabella; qualche sistema richiede anche che l'insieme di attributi della vista contenga almeno una chiave primaria della tabella base. La clausola `check option` può essere utilizzata solo nel contesto di questa categoria ristretta di viste. Essa specifica che sono ammessi aggiornamenti solo sulle righe della vista, e che dopo ogni modifica tutte le righe devono continuare ad appartenere alla vista. La "sparizione" di righe da una vista a seguito di una modifica può per esempio capitare se si assegna a un attributo della vista un valore che rende falso uno dei predicati di selezione. Nel caso in cui una vista sia definita in termini di altre viste, l'opzione `local` o `cascaded` specifica se il controllo sul fatto che le righe vengono escluse dalla vista debba essere effettuato solo all'ultimo livello (per cui si controlla solo che la modifica non faccia violare la condizione della vista più esterna) o se deve essere propagato a tutti i livelli di definizione (per cui si controlla che le righe su cui si apportano le modifiche non scompaiano dalla vista, a causa della violazione di una qualsiasi delle condizioni di selezione delle viste coinvolte); l'opzione di default è quella di `cascaded`.

Dato che la vista IMPIEGATIAMMINPOVERI è stata definita con `check option`, ogni comando di aggiornamento fatto sulla vista, per poter essere propagato, non deve eliminare righe dalla vista. Un assegnamento a **Stipendio** del valore 8 non è accettato con la presente definizione della vista, ma sarebbe accettato qualora la `check option` fosse stata definita come `local`. Una modifica dell'attributo **Stipendio** di una riga della vista per assegnare il valore 60 non sarebbe accettato neanche con l'opzione `local`.

5.1.4 Le viste per la scrittura di interrogazioni

Le viste in SQL possono anche servire per formulare delle interrogazioni che non sarebbero altrimenti esprimibili, aumentando il potere espressivo del linguaggio. Mediante la definizione di opportune viste, è possibile definire in SQL interrogazioni che richiedono di utilizzare diversi operatori aggregati in cascata, o che fanno un uso sofisticato dell'operatore di unione. In generale, le viste possono essere considerate uno strumento che permette di estendere la possibilità di nidificare le interrogazioni.

Per esempio, si vuole determinare qual è il dipartimento che spende il massimo in stipendi. A questo scopo, si definisce una vista che verrà utilizzata dalla successiva interrogazione:

```
create view BudgetStipendi(Dip,TotaleStipendi) as
select Dipart, sum(Stipendio)
from Impiegato
group by Dipart
```

Interrogazione 51: estrarre il dipartimento avente il valore massimo della somma degli stipendi.

```
select Dip
from BudgetStipendi
where TotaleStipendi = (select max(TotaleStipendi)
                        from BudgetStipendi)
```

La definizione della vista BUDGETSTIPENDI costruisce una tabella in cui compare una riga per ogni dipartimento. L'attributo Dip corrisponde all'attributo Dipart di IMPIEGATO e contiene il nome del dipartimento, mentre il secondo attributo TotaleStipendi contiene il risultato della valutazione della somma degli stipendi di tutti gli impiegati facenti capo a quel dipartimento.

Un altro modo per formulare la stessa interrogazione è il seguente:

Interrogazione 52:

```
select Dipart
from Impiegato
group by Dipart
having sum(Stipendio) >= all (select sum(Stipendio)
                              from Impiegato
                              group by Dipart)
```

Questa soluzione può non essere riconosciuta da qualche interprete SQL, il quale può imporre la restrizione che la condizione retta dalla clausola `having` sia una condizione semplice di confronto con un attributo o una costante, e non il risultato dell'esecuzione di una interrogazione nidificata. Vediamo un altro esempio d'uso di viste per la costruzione di query complesse:

```
create view DipartUffici(NomeDip,NroUffici) as
select Dipart, count(distinct Ufficio)
from Impiegato
group by Dipart
```

Interrogazione 53: estrarre il numero medio di uffici per ogni dipartimento.

```
select avg(NroUffici)
from DipartUffici
```

Si potrebbe pensare di esprimere la stessa interrogazione nel seguente modo:

Interrogazione 54:

```
select avg(count(distinct Ufficio))
from Impiegato
group by Dipart
```

L'interrogazione è però scorretta, in quanto la sintassi SQL non permette di combinare in cascata la valutazione di diversi operatori aggregati. Il problema di fondo è che la valutazione dei due diversi operatori avviene a diversi livelli di aggregazione, mentre è ammessa una sola occorrenza della clausola `group by` per ogni interrogazione.

5.1.5 Esempi riepilogativi d'uso delle viste

Riprendendo gli schemi utilizzati negli esercizi riepilogativi del capitolo precedente (Paragrafo 4.5), illustriamo l'uso delle viste nella formulazione di query.
Esempio 1: si ha il seguente schema relazionale che descrive il calendario di una manifestazione sportiva a squadre nazionali:

STADIO(Nome,Citta,Capienza)
INCONTRO(NomeStadio,Data,Ora,Squadra1,Squadra2)
NAZIONALE(Paese,Continente,Categoria)

Estrarre la città in cui si trova lo stadio in cui la squadra italiana gioca più partite. (Illustriamo due alternative.)

1. Con una vista apposita:

```
create view StadiItalia(NomeStadio,NroPart) as
   select NomeStadio, count(*)
   from Incontro
   where Squadra1 = 'Italia' or
         Squadra2 = 'Italia'
   group by NomeStadio

select Città
from Stadio
where NomeStadio in
      (select NomeStadio
       from StadiItalia
       where NroPart =
          (select max(NroPart)
           from StadiItalia))
```

2. Con una vista più generale:

```
create view Stadi(NomeStadio,Squadra,NroPart) as
  select NomeStadio,Paese,
         count(distinct Data,Ora)
  from Incontro, Nazionale
  where (Squadra1 = Paese or Squadra2 = Paese)
  group by NomeStadio, Paese
select Città
from Stadio
where NomeStadio in
      (select NomeStadio
```

```
from Stadi
where Squadra = 'Italia' and
      NroPart =
          (select max(NroPart)
           from Stadi
           where Squadra = 'Italia'))
```

Esempio 2: si ha il seguente schema relazionale:

MOTO(Targa,Cilindrata,Marca,Nazione,Tasse)
PROPRIETARIO(Nome,Targa)

Estrarre per ogni cliente le tasse che devono essere pagate per tutte le moto possedute, ipotizzando che se vi sono più proprietari per una moto, l'ammontare delle tasse viene equamente diviso tra i proprietari.

Soluzione:

```
create view TasseInd(Targa,Tassa) as
   select Targa, Tasse/count(*)
   from Moto join Proprietario
         on Moto.Targa = Proprietario.Targa
   group by Targa, Tasse

select Nome, sum(Tassa)
from Proprietario join TasseInd
       on Proprietario.Targa = TasseInd.Targa
group by Nome
```

5.1.6 Viste ricorsive in SQL-3

La sintassi SQL-2 non ammette dipendenze ricorsive, né immediate (definendo una vista in termini di se stessa), né transitive (ovvero situazioni in cui una vista V_1 è definita usando una vista V_2, V_2 usando V_3 e così via, infine V_n è definita usando V_1).

SQL-3 offre invece il supporto per le viste ricorsive, utilizzando una struttura di definizione che usa come modello formale di riferimento il linguaggio Datalog presentato nel Paragrafo 3.3. Non trattiamo questo argomento in modo esaustivo e per quanto riguarda l'analisi di come la ricorsione può essere gestita nelle query facciamo riferimento a quanto detto a proposito di Datalog. Ci limitiamo a mostrare un esempio d'uso della sintassi SQL-3.

Si supponga di disporre di una tabella IMPIEGATO(Matr,Nome,Cognome, Dipart,Superiore) che memorizza i superiori diretti di tutti gli impiegati. Supponiamo ora di voler conoscere i superiori, i superiori dei superiori, e tutti gli altri superiori indiretti dell'impiegato Mario Rossi. È ben noto che questa interrogazione è esprimibile in Datalog, mentre non può essere espressa né in algebra relazionale né in

SQL-2 perché, intuitivamente, richiederebbe di effettuare un numero non prevedibile a priori di join della tabella IMPIEGATO con se stessa. L'interrogazione si può invece esprimere in SQL-3 mediante una vista ricorsiva.

Interrogazione 55: estrarre i superiori diretti o indiretti dell'impiegato Mario Rossi.

```
with recursive Responsabile(Matr,Superiore) as
 ((select Matr, Superiore
   from Impiegato)
  union
  (select Impiegato.Matr, Responsabile.Superiore
   from Impiegato, Responsabile
   where Impiegato.Superiore = Responsabile.Matr))
select Nome, Cognome, Responsabile.Superiore
from Impiegato join Responsabile
       on (Impiegato.Matr = Responsabile.Matr)
where Nome = 'Mario' and Cognome = 'Rossi'
```

In questa istruzione, la clausola `with` definisce la vista RESPONSABILE che viene costruita ricorsivamente a partire dalla tabella IMPIEGATO. In particolare, la costruzione coinvolge una interrogazione di base non ricorsiva (definizione di base) e una interrogazione che esprime un join tra le tabelle IMPIEGATO e RESPONSABILE (definizione ricorsiva). La vista ricorsiva RESPONSABILE viene quindi utilizzata nell'ambito della query che definisce come punto di partenza l'impiegato Mario Rossi.

5.2 Funzioni scalari

Oltre alle funzioni aggregate che abbiamo già visto, SQL mette a disposizione diverse funzioni scalari, che possono essere usate all'interno delle espressioni del linguaggio. Le funzioni ricevono come argomento una o più espressioni, che restituiscono valori di un dominio elementare in corrispondenza di ogni tupla su cui viene valutata la query; le funzioni a loro volta restituiscono un valore semplice per ogni diversa tupla.

5.2.1 Famiglie di funzioni

SQL prevede alcune famiglie di funzioni. I sistemi spesso arricchiscono l'insieme di funzioni di ogni famiglia. Mostriamo le famiglie previste da SQL-2, facendo riferimento per le estensioni a quanto offerto dal sistema Postgres, un DBMS open-source particolarmente interessante, descritto in un documento disponibile sul sito del libro.

- Funzioni temporali: sono servizi di utilità per la gestione di informazioni temporali. SQL-2 prevede funzioni con nome `current_date`, `current_time`, `current_timestamp`, che restituiscono, per il relativo dominio, il valore dell'orologio del sistema nell'istante in cui il comando viene eseguito; la funzione `extract` restituisce una componente specifica di un dominio temporale (`year`,

month ecc.). I sistemi possono offrire funzioni ulteriori, come per esempio age, che restituisce l'intervallo di differenza tra una data e l'istante corrente.

- Funzioni di manipolazione di stringhe: si applicano a espressioni che rappresentano stringhe di caratteri e permettono di trasformare il loro contenuto. SQL-2 definisce diverse funzioni in questa famiglia, tra cui char_length (restituisce la lunghezza della stringa), lower (converte la stringa in caratteri minuscoli), upper (converte in maiuscolo) e substring (restituisce parte della stringa, usando parametri numerici per identificare la posizione di inizio e la lunghezza della sottostringa).
- Funzioni di conversione di dominio: la funzione cast permette di convertire un valore in un dominio nella sua rappresentazione in un altro dominio; per esempio, cast (Data as char(10)) converte un valore del dominio date nella sua rappresentazione testuale. Non tutte le conversioni sono ammesse, in quanto in alcuni casi non esistono regole di conversione standard; per esempio, non è possibile convertire un valore del dominio real in un valore del dominio date.
- Funzioni condizionali: descriviamo le funzioni di questa famiglia nel prossimo paragrafo.

Vi sono poi altre famiglie di funzioni che non fanno parte di SQL-2, ma che rappresentano servizi offerti dalle diverse implementazioni di SQL.

- Funzioni per la formattazione dell'output: servono per controllare l'aspetto del risultato della query: l'indentazione, la dimensione di ogni campo, e altre caratteristiche del formato di rappresentazione.
- Funzioni matematiche: si applicano a espressioni numeriche e restituiscono normalmente valori numerici (per esempio, abs per calcolare il valore assoluto, sqrt per la radice quadrata ecc.).
- Funzioni di accesso ai servizi del sistema operativo: permettono di accedere ai servizi dell'ambiente ospite, comandando dall'interno dell'ambiente SQL l'esecuzione di comandi arbitrari.

5.2.2 Funzioni condizionali

Tra le diverse famiglie di funzioni offerte da SQL-2, consideriamo con particolare attenzione la famiglia delle funzioni condizionali, che contiene le funzioni coalesce, nullif e case. Queste funzioni non estendono il potere espressivo del linguaggio, ma permettono di realizzare comandi SQL in modo più compatto e facile da comprendere, evitando per esempio di costruire interrogazioni composte da unioni di tante interrogazioni più semplici.

Coalesce La funzione coalesce ammette come argomento una sequenza di espressioni e restituisce il primo valore non nullo. La funzione può quindi essere usata per convertire valori nulli in valori definiti dal programmatore.

Interrogazione 56: estrarre i nomi, i cognomi e i dipartimenti cui afferiscono gli impiegati, usando la stringa "Ignoto" nel caso in cui non si conosca il dipartimento.

```
select Nome, Cognome, coalesce(Dipart,'Ignoto')
from Impiegato
```

Nullif La funzione `nullif` richiede come argomento una espressione e un valore costante; se l'espressione è pari al valore costante, la funzione restituisce il valore nullo, altrimenti restituisce il valore dell'espressione.

Interrogazione 57: estrarre i nomi, i cognomi e i dipartimenti cui afferiscono gli impiegati, restituendo il valore nullo per il dipartimento quando l'attributo Dipart possiede il valore "Ignoto".

```
select Nome, Cognome, nullif(Dipart,'Ignoto')
from Impiegato
```

L'esempio mostra come la funzione `nullif` svolga un compito opposto a quello della funzione `coalesce`.

Case La funzione `case` permette di specificare strutture condizionali, il cui risultato dipende dalla valutazione del contenuto delle tabelle. La sintassi ammette due diverse varianti:

```
case Espressione
     when Valore then EsprRisultato
     { when Valore then EsprRisultato }
     [ else EsprRisultato ]
     end

case when Condizione then Espressione
     { when Condizione then Espressione }
     [ else Espressione ]
end
```

La prima forma restituisce risultati diversi a seconda del valore di una specifica espressione (tipicamente, il valore di un attributo). Assumiamo per esempio di avere una tabella contenente alcune informazioni su veicoli, avente schema:

VEICOLO(Targa, Tipo, Anno, KWatt, Lunghezza, NAssi)

Supponiamo ora di voler calcolare le tasse di circolazione dei veicoli immatricolati dopo il 1975, sulla base di un tariffario che fa riferimento al tipo di veicolo. Una possibile soluzione è la seguente, nella quale il valore viene calcolato sulla base dei valori che compaiono nella colonna Tipo.

Interrogazione 58: estrarre l'ammontare delle tasse annuali per un veicolo.

```
select Targa,
 case Tipo
  when 'Auto' then 2.58 * KWatt
  when 'Moto' then (22.00 + 1.00 * KWatt)
  else null
 end as Tassa
from Veicolo
where Anno > 1975;
```

La seconda forma del costrutto `case` invece ammette la valutazione di predicati SQL generici. Mostriamo un esempio di applicazione di questa forma nel contesto di un'operazione di aggiornamento. La seguente istruzione SQL specifica una modifica dello stipendio di un impiegato, sulla base dei valori assunti dalle colonne Dipart e Ufficio.

```
update Impiegato
set Stipendio =
 case
  when (Dipart = 'Amministrazione' and Ufficio = 10)
     then Stipendio * 1.1
  when (Dipart = 'Amministrazione' and Ufficio <> 10)
     then Stipendio * 1.2
  when Dipart = 'Produzione'
     then Stipendio * 1.15
  else Stipendio
 end
```

Se non si facesse uso della funzione `case`, non sarebbe possibile effettuare la medesima operazione con una sola istruzione.

5.3 Procedure

Lo standard SQL-2 prevede la definizione di procedure, anche dette *stored procedures* per il fatto che normalmente vengono memorizzate all'interno della base di dati come parti dello schema. Come accade nei linguaggi di programmazione, le procedure permettono di associare un nome a un'istruzione SQL, con la possibilità di specificare dei parametri da utilizzare per lo scambio di informazioni con la procedura. I vantaggi sono un aumento della comprensibilità del programma, una più facile manutenibilità, e, nel caso delle procedure SQL, la possibilità di ottenere in diversi casi un sensibile incremento delle prestazioni. Una volta che la procedura è definita, essa è utilizzabile come se facesse parte dell'insieme dei comandi SQL predefiniti. Consideriamo come primo esempio la procedura SQL che aggiorna il nome della città di un dipartimento:

```
procedure AssegnaCitta(:Dip varchar(20),
                       :Citta varchar(20))
```

```
update Dipartimento
set Città = :Citta
where Nome = :Dip;
```

La procedura può essere invocata avendo cura di associare un valore ai parametri. Nell'esempio si mostra una invocazione della procedura all'interno di un programma C, che possiede le due variabili `:NomeDip` e `:NomeCitta`:

```
$ AssegnaCitta(:NomeDip,:NomeCitta)
```

Lo standard SQL-2 non tratta la scrittura di procedure complesse, ma si limita a specificare la definizione di procedure composte da un singolo comando SQL. Molti sistemi rimuovono questa limitazione, andando incontro alle esigenze delle applicazioni.

Le estensioni procedurali proposte dai diversi sistemi differiscono molto tra di loro: vi sono sistemi che permettono solamente di associare a ogni procedura una sequenza di comandi, altri che invece permettono l'utilizzo di strutture di controllo, dichiarazioni di variabili locali e l'invocazione di programmi esterni. In ogni caso, l'uso di queste funzioni è fuori dallo standard e rende non portabile il codice SQL generato. SQL-3 estende questo aspetto del linguaggio e fornisce una ricca sintassi per la definizione di procedure; fino a che, però, SQL-3 non si diffonderà, bisognerà utilizzare i servizi effettivamente disponibili sui sistemi, preventivando uno sforzo aggiuntivo nel caso in cui si debba adattare la propria applicazione a un altro ambiente.

Il seguente esempio mostra una procedura costituita dalla sequenza di due istruzioni SQL. La procedura permette di assegnare all'attributo Città il valore `:NuovaCittà`, per tutte le righe di DIPARTIMENTO e IMPIEGATO in cui l'attributo vale `:VecchiaCitta`.

```
procedure CambiaCittaATutti(:NuovaCitta varchar(20),
                            :VecchiaCitta varchar(20))
begin
  update Dipartimento
  set Città = :NuovaCitta
  where Città = :VecchiaCitta;
  update Impiegato
  set Città = :NuovaCitta
  where Città = :VecchiaCitta;
end;
```

Una delle estensioni normalmente fornite dagli attuali sistemi relazionali è la struttura di controllo *if-then-else*, che permette di esprimere esecuzioni condizionali e può essere usata per rilevare condizioni eccezionali. Mostriamo un esempio in cui si definisce una procedura che permette di porre a `:NuovaCitta` il valore dell'attributo Città per tutte le righe di DIPARTIMENTO con nome `:NomeDip`; se non si trova un dipartimento da modificare, si inserisce un elemento in ERRORIDIP.

```
procedure CambiaCittaADip(:NomeDip varchar(20),
                          :NuovaCitta varchar(20))
   if not exists(select *
                 from Dipartimento
                 where Nome = :NomeDip)
      insert into ErroriDip values(:NomeDip)
   else
      update Dipartimento
      set Città = :NuovaCitta
      where Nome = :NomeDip;
   end if;
end;
```

Come è già stato accennato, vi sono sistemi commerciali che offrono un insieme di estensioni procedurali di SQL molto ampio; in effetti, tali estensioni sono spesso in grado di rendere il linguaggio *computazionalmente completo*, ovvero con lo stesso potere espressivo di un normale linguaggio di programmazione. Esiste quindi la possibilità di scrivere una intera applicazione con questo SQL esteso; tuttavia, è molto raro che questa sia la soluzione migliore, poiché normalmente il sistema relazionale è ottimizzato solo per l'accesso ai dati.

Vediamo infine un esempio di programma scritto in PL/SQL, l'estensione procedurale del sistema relazionale Oracle Server, per dare un'idea del livello di sofisticazione offerto da questi sistemi.

```
procedure Addebita(CodConto char(5),
                   Prelievo integer) is
   TroppoScoperto exception;
   AmmontarePrec integer;
   NuovoAmmontare integer;
   Limite integer;
begin
   select Ammontare, Scoperto
     into AmmontarePrec, Limite
     from ContoCorrente
     where CodiceConto = CodConto
     for update of Ammontare;
   NuovoAmmontare := AmmontarePrec - Prelievo;
   if NuovoAmmontare > Limite then
     update ContoCorrente
       set Ammontare = NuovoAmmontare
       where CodiceConto = CodConto;
   else
     insert into TransazioniOltreScoperto
       values(CodConto,Prelievo,sysdate);
   end if;
end Addebita;
```

L'esempio mostra una procedura che preleva l'ammontare `Prelievo` dal conto con codice `CodConto` se sul conto è presente una copertura sufficiente. La procedura fa uso di variabili locali (`AmmontarePrec`, `NuovoAmmontare` e `Limite`) e sfrutta la struttura di controllo *if-then-else*.

5.4 Trigger e basi di dati attive

SQL fornisce un costrutto estremamente potente, il *trigger*, per rendere la base di dati in grado di reagire ad eventi definiti dall'amministratore tramite l'esecuzione di opportune azioni.

Una base di dati con tale capacità si dice *attiva*: essa dispone di un sottosistema integrato per definire e gestire regole di produzione (dette anche regole attive). Le regole seguono il cosiddetto paradigma *Evento-Condizione-Azione*: ciascuna regola reagisce ad alcuni eventi (normalmente modifiche della base di dati), valuta una condizione e, in base al valore di verità della condizione, esegue una reazione. L'esecuzione delle regole avviene sotto il controllo di un sotto-sistema autonomo, detto *processore delle regole* (*rule engine*), che tiene traccia degli eventi e manda in esecuzione le regole in base a proprie politiche; in questo modo, si determina un alternarsi tra l'esecuzione delle transazioni, lanciate dagli utenti, e delle regole, lanciate dal sistema; si dice che il sistema risultante ha un *comportamento reattivo*, che si differenzia dal tipico comportamento passivo di una base di dati priva di regole attive.

Quando una base di dati ha un comportamento reattivo, una parte dell'applicazione normalmente codificata mediante i programmi può essere espressa tramite regole attive. Come vedremo, le regole attive possono per esempio gestire vincoli di integrità, calcolare dati derivati e gestire eccezioni, oltre a codificare vere e proprie "regole aziendali". Questo strumento aggiunge all'indipendenza delle basi di dati, discussa nel primo capitolo, una nuova dimensione, detta *indipendenza della conoscenza*: la conoscenza di tipo reattivo viene sottratta ai programmi applicativi e codificata sotto forma di regole attive. Ricordiamo che le basi di dati offrono le due dimensioni di indipendenza fisica (un programma non deve conoscere l'organizzazione fisica dei dati) e logica (un programma può vedere i dati tramite opportuni schemi esterni o view). Il vantaggio introdotto dalla nuova dimensione di indipendenza della conoscenza è che la logica applicativa relativa al comportamento reattivo viene definita una volta per tutte sotto forma di regole, che fanno parte dello schema (tramite il DDL) e vengono condivise da tutte le applicazioni, invece che essere replicata in tutti i programmi; modifiche alle elaborazioni di tipo reattivo possono essere gestite semplicemente cambiando le regole attive, senza dover modificare le applicazioni.

Purtroppo, i trigger sono stati standardizzati solo in SQL-3. Questo ritardo nella standardizzazione è la causa di una notevole difformità nei comportamenti dei trigger, che sono stati introdotti nei sistemi commerciali prima della standardizzazione (molti sistemi relazionali includono i trigger dalla fine degli anni Ottanta). D'altra parte, lo standard di riferimento SQL-3 rischia di rimanere disatteso, almeno per quanto riguarda i principali prodotti, in quanto è improbabile che i sistemi vogliano adeguarsi a esso, introducendo incompatibità rispetto alle applicazioni che già fanno uso di trigger. Pertanto, daremo innanzitutto una descrizione generale, che si adatta abbastanza

bene a qualunque sistema di tipo relazionale, per poi precisare esattamente la sintassi e il comportamento dei trigger in base allo standard SQL-3, e presenteremo poi due specifici sistemi relazionali, IBM DB2 e Oracle. DB2 ha maggiormente influenzato la specifica dello standard e quindi ne riflette le specifiche in modo abbastanza fedele, mentre Oracle si discosta da esso per alcuni aspetti sintattici e semantici.

5.4.1 Definizione e uso dei trigger in SQL-3

La creazione dei trigger fa parte del *Data Definition Language* (DDL); i trigger possono anche essere cancellati e in taluni sistemi attivati e disattivati dinamicamente. I trigger sono basati sul paradigma *Evento-Condizione-Azione* (ECA):

- gli eventi sono primitive per la manipolazione dei dati, secondo il modello SQL (`insert`, `delete`, `update`);
- la condizione (che può talvolta mancare) è un predicato booleano, espresso in SQL;
- l'azione è una sequenza di primitive SQL generiche, talvolta arricchite da un linguaggio di programmazione integrato disponibile nell'ambito di uno specifico prodotto (per esempio, PL/SQL in Oracle).

In genere i trigger fanno riferimento a una tabella, detta *target*, in quanto rispondono a eventi relativi a tale tabella.

Il paradigma ECA ha un comportamento semplice e intuitivo: *quando* si verifica l'evento, *se* la condizione è soddisfatta, *allora* viene svolta l'azione. Si dice che un trigger è *attivato* da uno dei suoi eventi, viene *valutato* durante la verifica della sua condizione e viene *eseguito* quando, a seguito della valutazione positiva, viene posta in esecuzione la parte azione. Tuttavia, vi sono differenze significative nel modo in cui i sistemi definiscono attivazione, valutazione ed esecuzione dei trigger.

I trigger relazionali hanno due livelli di granularità, detti di tupla (*row-level*) e di primitiva (*statement-level*). Nel primo caso, l'attivazione avviene per ogni tupla coinvolta nell'operazione; si ha cioè un comportamento orientato alle singole istanze. Nel secondo caso, l'attivazione avviene una sola volta per ogni primitiva SQL facendo riferimento a tutte le tuple coinvolte dalla primitiva, con un comportamento orientato agli insiemi.

Inoltre, i trigger possono avere la modalità *immediata* oppure *differita*. Quando i trigger hanno modalità immediata, la loro valutazione in genere avviene immediatamente dopo l'evento che li ha attivati (opzione *after*); più raramente, la valutazione del trigger deve precedere logicamente l'evento cui si riferisce (opzione *before*). Invece la valutazione differita dei trigger avviene alla fine della transazione, a seguito di un comando di `commit work`.

È possibile che i trigger si riattivino l'uno con l'altro; ciò accade quando l'azione di un trigger è anche l'evento di un altro trigger. In tal caso, si dice che i trigger sono *in cascata*. È anche possibile che i trigger si attivino l'un l'altro in modo infinito, generando situazioni di non terminazione; affronteremo questo problema nel Paragrafo 5.4.4.

Sintassi dei trigger Ogni trigger è attivato da un solo evento, che può essere una qualunque primitiva di modifica dei dati in SQL. L'attivazione dei trigger in SQL-3 è sempre immediata, prima oppure dopo l'evento di riferimento. I trigger hanno due livelli di granularità, di tupla e di primitiva. La sintassi dell'istruzione di creazione dei trigger è la seguente:

```
create trigger NomeTrigger
        Modo Evento on TabellaTarget
        [ referencing Referenza ]
        [ for each Livello ]
        [ when ( PredicatoSQL ) ]
        StatementProceduraleSQL
```

ove il *Modo* è `before` oppure `after`, l'*Evento* è `insert`, `delete`, oppure `update` (eventualmente riferito a una specifica colonna), il *Livello* è `row` (tupla) oppure `statement` (primitiva). La clausola opzionale *Referenza* consente di introdurre dei nomi di variabili. Se il livello è `row`, le variabili si riferiscono alle tuple che subiscono la modifica, introdotte dalla clausola:

`old as` *VarTuplaOld* | `new as` *VarTuplaNew*

Chiaramente, la variabile `old` fa riferimento alla tupla nello stato precedente alla modifica, mentre la variabile `new` fa riferimento alla tupla nello stato prodotto da essa.

Se il livello è `statement`, le variabili si riferiscono alle porzioni della tabella target che subiscono la modifica, anche in questo caso rispettivamente nello stato precedente e successivo alla modifica, e sono introdotte dalla clausola:

`old_table as` *VarTabellaOld* | `new_table as` *VarTabellaNew*

Le variabili `new`, `old`, `new_table` e `old_table` sono implicitamente definite e quindi utilizzabili nel predicato oppure nella parte procedurale della regola, mentre le clausole `referencing` consentono l'introduzione di variabili denominate in modo diverso. Nel caso l'evento sia un inserimento sono definite solo le variabili `new` o `new_table`; nel caso di cancellazione sono definite solo le variabili `old` o `old_table`.

La granularità è espressa tramite la clausola opzionale `for each` *Livello*; se la clausola è omessa la granularità è implicitamente a livello di primitiva. Il predicato è un qualunque predicato esprimibile in SQL che però deve essere, a seconda del livello, un predicato semplice (se la granularità è di tupla) oppure aggregato (se la granularità è di primitiva).

Lo statement procedurale in SQL che conclude la sintassi comprende uno o più comandi SQL, che costituiscono la parte reattiva della regola. Se sono presenti molti comandi, essi devono essere contenuti all'interno delle parole chiave `begin atomic` ed `end`; il termine `atomic` ricorda che tutti i comandi del trigger devono andare a buon fine oppure essere tutti disfatti, ripristinando lo stato precedente allo statement SQL che ne ha causato la attivazione.

Prima di addentrarci nella semantica dei trigger, che presenta qualche elemento di difficoltà, vediamo due semplici esempi, che si riferiscono ad una tabella `Impiegato` con attributi `ImpNum` e `Stipendio`. Un tipico trigger attivato prima di un evento, detto di tipo *before*, è il seguente:

```
create trigger LimitaAumenti1
before update of Stipendio on Impiegato
for each row
when (new.Stipendio > old.Stipendio * 1.2)
set new.Stipendio = old.Stipendio * 1.2
```

Si noti l'uso di un assegnamento (istruzione `set`) con il quale è possibile modificare il valore di una variabile che denota la tupla attualmente sottoposta a modifica; quando il trigger è di tipo *before*, tale assegnamento avviene prima che la modifica della base di dati abbia luogo (e quindi, per esempio, porti a violazioni di vincoli di integrità).

Un comportamento simile, ma non identico, è ottenuto dal seguente trigger di tipo *after*:

```
create trigger LimitaAumenti2
after update of Stipendio on Impiegato
for each row
when (new.Stipendio > old.Stipendio * 1.2)
update Impiegato
set new.Stipendio = old.Stipendio * 1.2
where ImpNum = new.ImpNum
```

Si noti che in questo caso il trigger scatta dopo la modifica di stipendio, che viene svolta in ogni caso, e quindi può violare vincoli di integrità; se la condizione (che utilizza sia lo stato precedente sia lo stato successivo alla modifica) è vera, scatta la parte procedurale del trigger, e cioè un update che ripristina, per la specifica tupla su cui scatta la condizione, il valore di stipendio consentito dalla regola.

Questi due semplici esempi hanno illustrato la principale differenza fra trigger di tipo *before* e *after*: i primi sono usati per "condizionare" i valori usati da una operazione di modifica, i secondi per "reagire" ad una modifica tramite altre operazioni, che in genere annullano effetti indesiderati.

Comportamento dei trigger Illustriamo ora il comportamento di un sistema che esegue transazioni e regole attivate da esse. Il sistema opera in un contesto transazionale in cui l'esecuzione delle regole avviene all'interno della transazione che le scatena; in caso di fallimento di un'operazione di modifica o di una regola *tutti* gli effetti prodotti vengono annullati. Nel caso di rollback parziale di una istruzione SQL vengono annullati gli effetti di quella istruzione e di tutte le regole attivate da essa, sia in modo diretto sia, come vedremo, in modo indiretto.

Iniziamo chiarendo una limitazione posta alle azioni dei trigger di tipo *before*: essi possono solo modificare i valori assegnati alle variabili `new`, ma non possono contenere comandi DML che provochino una modifica dello stato della base di dati;

come conseguenza, essi non possono attivare altri trigger. Il sistema deve garantire un comportamento in cui l'effetto dei trigger *before* sia precedente alla esecuzione della primitiva che li attiva; i trigger *before* possono però richiedere la valutazione anticipata dei valori prodotti dalla primitiva, che vengono memorizzati in strutture dati temporanee. Nell'esempio precedente, viene valutato il valore `new.Stipendio`.

Vari trigger a diversi livelli di granularità possono fare riferimento allo stesso evento; essi vengono considerati in base a un ordinamento gestito dal sistema, che tiene conto del loro tempo di creazione (trigger creati prima hanno maggior priorità). Quindi, trigger a livello di tupla e di primitiva possono essere ordinati in modo arbitrario tra loro. Un trigger a livello di tupla viene eseguito iterativamente su tutte le tuple coinvolte nell'operazione di modifica che lo ha attivato; si noti che nei sistemi relazionali non è normalmente noto l'ordine con cui il sistema opera sulle tuple soggette ad uno stesso comando SQL, in quanto tale ordine dipende dal "piano" deciso dall'ottimizzatore. Se una azione di un trigger a livello di tupla contiene molte primitive SQL, esse vengono tutte eseguite per una tupla prima di passare alla tupla successiva.

Particolare attenzione va posta relativamente alla possibilità di attivazione ricorsiva dei trigger e alla valutazione congiunta di trigger e vincoli di integrità, in particolare quelli di tipo referenziale che sono associati a una azione compensatrice. Prendiamo in considerazione una azione di modifica S svolta sulla tabella R nel contesto di una generica transazione. Se S attiva qualche trigger, viene inizialmente svolta la valutazione ed esecuzione dei trigger *before*, che possono causare modifiche ai valori `new`; vengono successivamente svolte le azioni legate al ripristino dell'integrità referenziale legate ad S; queste azioni possono causare azioni a catena (quando nel vincolo è presente l'opzione `cascade`) che possono a loro volta causare l'attivazione di molti trigger sia di tipo *before* (che vengono valutati senza causare ulteriori modifiche della base di dati) sia di tipo *after* (che si aggiungono ai trigger *after* attivati da S). Si ottiene così un insieme $I(S)$ di trigger di tipo *after* attivati direttamente o indirettamente da S.

Quando l'esecuzione di uno statement S' presente nella parte procedurale di uno dei trigger di $I(S)$ provoca l'attivazione di altri trigger, lo stato di esecuzione dell'algoritmo relativo a S viene salvato e il sistema reagisce, in modo ricorsivo, alla modifica S', eseguendo i corrispondenti trigger *before* e *after* come descritto in precedenza per S. Al termine dell'esecuzione scatenata da S', lo stato relativo a S viene ripristinato e l'esecuzione viene ripresa dal punto in cui era stata sospesa. Se durante l'esecuzione viene sollevata una eccezione o si incorre in un errore, tutte le modifiche eseguite a partire dalla primitiva transazionale che innesca l'esecuzione dei trigger, compresa la primitiva stessa, vengono disfatte; viene cioè garantito un *rollback parziale* della primitiva e di tutte le azioni causate dai trigger.

5.4.2 Definizione e uso dei trigger in DB2

Lo standard SQL-3 si è fortemente ispirato a DB2, a causa della significativa presenza dei progettisti di DB2 nell'ambito del gruppo di standardizzazione. Possiamo perciò considerare DB2 come la principale realizzazione dello standard SQL-3. L'unica differenza sostanziale è l'uso, in DB2, di statement procedurali disponibili nell'ambito

del sistema e che non necessariamente fanno riferimento allo standard SQL-3. Per questo motivo, diamo nel seguito alcuni esempi di trigger DB2 che chiariscono anche la semantica di SQL-3 descritta nel paragrafo precedente.

Si consideri una base di dati contenente le tabelle PARTE, DISTRIBUTORE e AUDIT; la tabella PARTE ha per chiave principale l'attributo PartNum e tre altri attributi: Fornitore, Città e Costo; un vincolo di integrità referenziale è presente nella tabella PARTE e fa riferimento alla tabella DISTRIBUTORE:

```
foreign key (Fornitore)
 references Distributore
 on delete set null
```

Consideriamo i seguenti trigger:

- FORNITOREUNICO è un trigger *before* che impedisce di modificare l'attributo Fornitore se non ponendolo al valore *null* (in tutti gli altri casi, solleva una eccezione che forza un rollback della primitiva).
- AUDITPARTE è un trigger *after* che registra nella tabella AUDIT il numero di tuple modificate nella tabella PARTE.

```
create trigger FornitoreUnico
before update of Fornitore on Parte
referencing new as N
for each row
when (N.Fornitore is not null)
 signal sqlstate '70005'
   ('Non si cambia il fornitore')

create trigger AuditParte
after update on Parte
referencing old_table as OT
for each statement
insert into Audit
 values(user, current date,
        (select count(*) from OT))
```

Per esempio, la cancellazione dalla tabella DISTRIBUTORE di tutti i fornitori di "Como" fa sì che il vincolo di integrità referenziale sia violato. A questo punto, la politica di gestione delle violazioni del vincolo di integrità provoca la modifica al valore *null* di tutte le tuple della tabella PARTE rimaste orfane a seguito delle cancellazioni. Ciò attiva i due trigger FORNITOREUNICO e AUDITPARTE; il primo è un trigger *before* che viene quindi considerato per primo. La sua valutazione, tupla per tupla, avviene logicamente prima della modifica, però avendo a disposizione il valore N che caratterizza la variazione; quindi, tale valore viene trovato uguale a *null*, e la condizione risulta falsa. Infine, viene considerato ed eseguito il trigger AUDITPARTE, che inserisce nella tabella AUDIT un'unica tupla contenente il codice utente, la data corrente e il numero di tuple modificate.

5.4.3 Definizione e uso dei trigger in Oracle

I trigger di Oracle sono stati sviluppati precedentemente allo standard SQL-3; rispetto a essi presentano alcune differenze sintattiche e semantiche. Vediamo innanzitutto le caratteristiche del comando per creare trigger, per poi discuterne il comportamento e un esempio di applicazione.

Sintassi dei trigger La sintassi per la creazione dei trigger in Oracle è la seguente:

```
create trigger NomeTrigger
        modo evento {, evento }
        on TabellaTarget
        [ [ referencing referenza
          for each row
          [ when ( PredicatoSQL ) ] ]
        BloccoPL/SQL
```

ove il *modo* è `before` oppure `after`, l'*evento* è `insert`, `delete`, oppure `update` (talvolta riferito a una specifica colonna). La clausola *referenza* consente di introdurre altri nomi di variabili rispetto ai default `old` e `new`, tramite la sintassi:

```
old as VariabileOld | new as VariabileNew
```

Discutiamo ora le differenze sintattiche rispetto a SQL-3. Ogni trigger controlla una qualunque combinazione delle tre primitive di DML (inserimento, cancellazione e modifica) sulla tabella target, mentre in SQL-3 ogni trigger è attivato da un solo evento. La granularità del trigger è determinata dalla clausola `for each row`, che si aggiunge nel caso di granularità a livello di tupla mentre viene omessa nel caso di granularità a livello di primitiva. La condizione può essere presente solo nei trigger con granularità a livello di tupla e consiste in un semplice predicato sulla tupla corrente; nei trigger con granularità a livello di primitiva, tuttavia, è possibile introdurre strutture di controllo nella parte azione, scritta nel linguaggio PL/SQL, che estende SQL con costrutti tipici di un linguaggio di programmazione. La parte azione non può contenere istruzioni DDL o comandi transazionali.

I riferimenti allo stato precedente e successivo a una operazione di modifica sono possibili solo se un trigger è *row-level*, limitatamente alla tupla che viene modificata; nel caso di inserimento è definito solo lo stato successivo e nel caso di cancellazione è definito solo lo stato precedente. Le variabili `old` e `new` sono implicitamente disponibili per indicare, rispettivamente, la tupla nello stato precedente e successivo a una operazione; altri nomi di variabili possono essere introdotti dalla clausola `referencing`.

Comportamento dei trigger I trigger in Oracle sono immediati e prevedono sia l'opzione *before* o *after* sia la granularità a livello di tupla o di primitiva; perciò, combinando le due granularità e le due modalità, si ottengono per ogni evento quattro combinazioni:

```
before row
before statement
after row
after statement
```

L'esecuzione di una primitiva di `insert`, `delete` o `update` in SQL è inframmezzata dalla esecuzione dei trigger che vengono da essa attivati, secondo il seguente schema:

1. Si eseguono i trigger `before statement`.
2. Per ogni tupla della tabella target coinvolta nella primitiva:
   (a) si esegue il trigger `before row`;
   (b) si applica la primitiva alla tupla e si eseguono i test relativi all'integrità che possono essere verificati tupla per tupla;
   (c) si eseguono i trigger `after row`.
3. Si eseguono i test relativi all'integrità che devono essere verificati sull'intera tabella.
4. Si eseguono i trigger `after statement`.

Le azioni svolte dai trigger possono causare l'attivazione di altri trigger; in tal caso, l'esecuzione del trigger corrente è sospesa e vengono considerati gli altri trigger attivati, applicando l'algoritmo illustrato nel Paragrafo 5.4.1. Il massimo numero di trigger *in cascata* (cioè, successivamente attivati secondo questo schema) è 32; raggiunta questa soglia, il sistema ipotizza una situazione di esecuzione infinita e sospende l'esecuzione, sollevando una specifica eccezione.

Esempio di esecuzione Illustriamo il comportamento dei trigger in Oracle alle prese con un classico problema di gestione di scorte. Il trigger RIORDINO, illustrato di seguito, è usato per generare automaticamente un nuovo ordine (tramite inserimento di una tupla nella tabella ORDINIPENDENTI) ogni qual volta la quantità disponibile QtaDisp di una particolare parte nella tabella MAGAZZINO scende al di sotto di una specifica soglia di riordino (QtaSoglia):

```
create trigger Riordino
after update of QtaDisp on Magazzino
when (new.QtaDisp < new.QtaSoglia)
for each row
 declare
  X number;
 begin
  select count(*) into X
   from OrdiniPendenti
   where Parte = new.Parte;
```

```
  if X = 0
  then
   insert into OrdiniPendenti
    values(new.Parte,new.QtaRiord,sysdate)
  end if;
end;
```

Questo trigger ha una granularità a livello di tupla e viene considerato immediatamente dopo ogni modifica all'attributo QtaDisp. La condizione viene valutata tupla per tupla, confrontando i valori degli attributi QtaDisp e QtaSoglia dopo ogni modifica al valore dell'attributo QtaDisp; l'azione è un programma scritto in PL/SQL, in cui viene inizialmente dichiarata una variabile numerica X, che successivamente memorizza il numero di ordini già emessi relativi alla parte considerata. Se tale numero è zero, un ordine viene emesso inserendo una tupla nella tabella ORDINIPENDENTI che contiene il numero della parte, la quantità di riordino QtaRiord (supposta fissa) e la data corrente. I valori della tupla cui fa riferimento l'esecuzione del trigger vengono acceduti tramite l'uso della variabile di correlazione new.

Si assuma che la tabella MAGAZZINO abbia lo stato iniziale presentato in Figura 5.1 e si assuma la tabella ORDINIPENDENTI inizialmente vuota. Si consideri poi la seguente transazione, attivata il 10/10/2013:

```
T1: update Magazzino
 set QtaDisp = QtaDisp - 70
 where Part = 1
```

Questa transazione causa la attivazione, valutazione ed esecuzione del trigger RIORDINO, portando all'inserimento nella tabella ORDINIPENDENTI della tupla: (1,100,10-10-2013). Si supponga successivamente eseguita la transazione:

```
T2: update Magazzino
 set QtaDisp = QtaDisp - 60
 where Part <= 3
```

Il trigger viene così eseguito relativamente a tutte le parti, e la condizione è verificata per le parti 1 e 3; tuttavia, l'azione relativa alla parte 1 non ha effetto. Quindi, l'esecuzione del trigger comporta l'inserimento in ORDINIPENDENTI della sola tupla: (3, 120, 10-10-2013).

MAGAZZINO

Parte	QtaDisp	QtaSoglia	QtaRiord
1	200	150	100
2	780	500	200
3	450	400	120

Figura 5.1 Stato iniziale della tabella MAGAZZINO.

5.4.4 Caratteristiche evolute e proprietà delle regole attive

Rispetto alle caratteristiche di base dei trigger relazionali, viste in precedenza, alcuni sistemi e prototipi evoluti di basi di dati attive hanno varie caratteristiche interessanti, che aumentano il potere espressivo delle regole attive.

- Per quanto concerne gli eventi, essi possono includere anche eventi *temporali* o *applicativi*; i primi consentono di esprimere eventi come per esempio "ogni venerdì sera" oppure "alle 15:45 del 7/5/2000"; gli ultimi vengono esplicitamente attivati dai programmi degli utenti.
- L'attivazione dei trigger può dipendere non solo da un evento o da un insieme di eventi con una semplice interpretazione disgiuntiva, ma anche da generiche *espressioni booleane di eventi* costruite a partire da operatori più complessi, quali le precedenze tra eventi e la congiunzione di eventi.
- Oltre alle clausole `before` e `after` esiste anche una clausola `instead of`; quando la condizione della corrispondente regola è vera, l'azione viene eseguita *al posto* dell'evento. Questa clausola deve essere usata con attenzione, in quanto può dar luogo a una semantica assai poco intuitiva (per esempio, "quando si modifica lo stipendio del dipendente X, modificare invece lo stipendio del dipendente Y"). D'altra parte, la clausola `instead of` rappresenta uno strumento molto interessante per risolvere in alcuni sistemi il problema dell'aggiornamento delle viste, consentendo di creare trigger che reagiscono a comandi di modifica delle viste; questi trigger possono quindi sostituire alla richiesta di modifica sulla vista una opportuna azione sulle tabelle di base, risolvendo una delle criticità che si presentano quando si usano le viste per gestire l'evoluzione degli schemi relazionali.
- La valutazione e l'esecuzione delle regole può essere *distaccata* (*detached*); in tal caso, valutazione o esecuzione avvengono nel contesto di un'altra transazione, che può essere completamente autonoma oppure può coordinarsi con la transazione in cui si è verificato l'evento tramite sofisticati meccanismi di dipendenza reciproca.
- I conflitti fra regole attivate dallo stesso evento (o comunque presenti in un insieme di regole attivate, detto *conflict set*) possono venire risolti da *priorità esplicite*, definite cioè direttamente dall'utente all'atto della creazione delle regole, espresse sia come un ordinamento parziale (tramite relazioni di precedenza tra regole), sia come un ordinamento totale (tramite priorità numeriche). Normalmente, le priorità esplicite sostituiscono meccanismi di priorità implicitamente presenti nei sistemi.
- Le regole possono essere organizzate in *gruppi* e ciascun gruppo può essere separatamente *attivato* e *disattivato*.

Progettare ciascuna regola attiva non è difficile, una volta che vengono ben individuati il suo evento, la sua condizione e la sua azione. Tuttavia, comprendere il comportamento collettivo delle regole attive è assai più complesso, perché la loro interazione è spesso subdola e non facilmente comprensibile. Per questo motivo, il problema principale nel progetto delle basi di dati attive consiste nel comprendere il comportamento di insiemi complessi di regole. Le principali proprietà di tali regole sono la terminazione, la confluenza e la osservabilità deterministica:

- Un insieme di regole garantisce la *terminazione* quando, per ogni transazione che scatena l'esecuzione delle regole, tale esecuzione termina producendo uno stato finale; rientra in questo caso anche la possibilità di un abort locale o globale.
- Un insieme di regole garantisce la *confluenza* quando, per ogni transazione che scatena l'esecuzione delle regole, tale esecuzione termina producendo un unico stato finale, che non dipende dall'ordine di esecuzione delle regole non esplicitamente prioritizzate.
- Un insieme di regole garantisce il *determinismo delle osservazioni* quando, per ogni transazione che scatena l'esecuzione delle regole, tale esecuzione è confluente e tutte le azioni visibili svolte dalle regole (inclusi gli invii di messaggi agli utenti) sono identiche e prodotte nello stesso ordine.

Queste proprietà non sono tutte importanti o desiderabili allo stesso modo; in particolare, la terminazione è una proprietà essenziale, in quanto si vuole evitare che transazioni attivate da utenti portino il sistema in condizione di esecuzione infinita a causa di regole definite, normalmente, dall'amministratore della base di dati; l'utente avrebbe infatti grosse difficoltà nel comprendere la situazione e porvi rimedio. Invece, confluenza e determinismo delle osservazioni possono essere meno importanti, specie in presenza di varie soluzioni equivalenti di uno stesso problema applicativo.

Il processo di *analisi delle regole* consente di verificare, a tempo di creazione, che le proprietà succitate valgono per uno specifico insieme di regole. In particolare, uno strumento essenziale per verificare la terminazione di un insieme di regole è il cosiddetto *grafo di attivazione*, che rappresenta le interazioni fra regole. Il grafo è costituito facendo corrispondere ogni regola a un nodo, e tracciando un arco da una regola R_1 a una regola R_2 quando l'azione di R_1 contiene una primitiva che coincide con uno degli eventi di R_2. Un'esecuzione può non terminare solo in presenza di cicli nel grafo di attivazione, che possono corrispondere a sequenze infinite di esecuzioni di regole; un esempio di grafo di attivazione ciclico è illustrato in Figura 5.2.

Sistemi con molte regole attive sono spesso ciclici; tuttavia solo alcuni cicli corrispondono realmente a situazioni critiche; la maggior parte dei cicli è normalmente "innocua". Infatti, il grafo di attivazione indica solo una situazione sintattica di

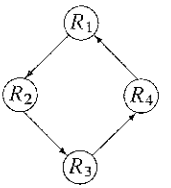

Figura 5.2 Grafo di attivazione ciclico.

possibile mutua interazione; se le regole sono scritte bene, tale interazione ha però termine.

Consideriamo per esempio la regola CONTROLLASTIPENDI (scritta in DB2), che realizza una politica "conservatrice" di controllo degli stipendi; essa riduce lo stipendio di tutti gli impiegati quando la media degli stipendi supera una certa soglia:

```
create trigger ControllaStipendi
after update of Stipendio on Impiegato
then update Impiegato
 set Stipendio = 0.9 * Stipendio
 where (select avg(Stipendio) from Impiegato) > 100
```

Il grafo di attivazione relativo a questa regola ha solo un nodo e un cappio (detto anche *auto-anello*, ovvero un arco che entra ed esce dallo stesso nodo); il grafo quindi presenta un ciclo, che indica la possibilità che la regola si riattivi. D'altra parte, qualunque sia la transazione iniziale, l'esecuzione della regola prima o poi termina, in quanto la regola riduce progressivamente gli stipendi fino a farli rientrare al di sotto della soglia; a quel punto, la condizione è falsa. Tuttavia, una regola leggermente diversa pone problemi:

```
create trigger ControllaStipendi2
after update of Stipendio on Impiegato
then update Impiegato
 set Stipendio = 1.1 * Stipendio
 where (select avg(Stipendio) from Impiegato) > 100
```

Il grafo di attivazione associato a questa regola non cambia; però, se la regola viene eseguita una volta, verrà eseguita un numero infinito di volte, causando non-terminazione, in quanto l'operazione eseguita dalla regola non è in grado di rendere la sua condizione falsa.

Questo esempio mostra che i cicli danno solamente delle "indicazioni" di possibili cause di non-terminazione. Un'analisi sofisticata dei cicli, che può essere solo in parte automatizzata in quanto il problema della terminazione è indecidibile, è in grado di portare a concludere che un ciclo è innocuo, oppure è in grado di suggerire modifiche alle regole che lo compongono in modo da garantirne la terminazione.

5.4.5 Applicazioni delle basi di dati attive

Le regole attive rispondono a esigenze applicative di tipo diverso. Le applicazioni più classiche delle regole attive sono *interne* alla base di dati: il gestore delle regole attive opera come sottosistema della base di dati per implementare alcune sue funzionalità; in tal caso, i trigger sono generati dal sistema e talvolta non visibili agli utenti. La caratteristica tipica di queste applicazioni è la possibilità di dare una *specifica dichiarativa* da cui derivare, in tutto o in parte, il codice delle regole attive. Le principali funzionalità che possono essere affidate alle regole attive di tipo interno comprendono la gestione di vincoli di integrità di struttura predefinita, il computo di dati derivati,

e la gestione di dati replicati; altre funzionalità includono la gestione di versioni, la gestione della privatezza e sicurezza dei dati, il logging delle azioni e la registrazione degli eventi.

Altre regole, classificate come *esterne*, esprimono conoscenza di tipo applicativo che sfugge a schemi rigidi predefiniti. Queste regole vengono anche denominate *regole aziendali* (*business rules*) in quanto esprimono le strategie di un'azienda nel perseguire i propri scopi primari. Nel caso delle regole aziendali non esistono però tecniche fisse per derivare le regole a partire dalle specifiche, quindi ciascun problema applicativo deve essere affrontato separatamente. Nel seguito vediamo brevemente la gestione dell'integrità referenziale, la principale applicazione interna delle basi di dati attive, e alcune regole aziendali.

Gestione dell'integrità referenziale La gestione di un qualunque vincolo di integrità tramite regole attive richiede innanzitutto che il vincolo sia espresso sotto forma di predicato SQL. Il predicato corrisponderà alla parte *condizione* di una o più regole attive associate al vincolo; si noti, però, che il predicato deve essere negato nella regola, in modo che la considerazione della regola produca un valore vero quando il vincolo viene effettivamente violato. Successivamente il progettista si concentrerà sugli eventi che possono causare una violazione del vincolo; essi contribuiscono alla parte *eventi* delle regole attive. Infine, il progettista dovrà decidere quale azione svolgere a seguito della violazione del vincolo; per esempio, potrà forzare il rollback della transazione, oppure il rollback parziale della primitiva che ha causato la violazione del vincolo, oppure infine potrà svolgere una *azione compensatrice* che corregge la violazione del vincolo. Viene così costruita la parte *azione* della regola attiva.

Illustriamo questo approccio generale tramite un esempio di gestione dell'integrità referenziale, un vincolo tra i più classici. Si noti, tuttavia, che l'integrità referenziale viene normalmente gestita con metodi *ad hoc*.

Riprendiamo il semplice vincolo di integrità referenziale discusso nel Paragrafo 4.2.7, relativo alle due tabelle IMPIEGATO e DIPARTIMENTO; il vincolo indica che l'attributo Dipart di IMPIEGATO è una `foreign key` rispetto all'attributo NomeDip di DIPARTIMENTO, tramite la seguente clausola inserita nella definizione della tabella IMPIEGATO, che svolge il ruolo di tabella *interna*:

```
foreign key(Dipart) references Dipartimento(NomeDip)
 on delete set null,
 on update cascade
```

Le operazioni che possono violare questo vincolo sono:

- `insert into Impiegato;`
- `delete from Dipartimento;`
- `update to Impiegato.Dipart;`
- `update to Dipartimento.NomeDip.`

La condizione può essere espressa come un predicato relativo alle tuple di IMPIEGATO, che imponga per ogni impiegato l'esistenza di un dipartimento di appartenenza:

```
IMPIEGATO: exists (select * from Dipartimento
                   where NomeDip = Impiegato.Dipart)
```

Si noti che questo predicato indica una proprietà che deve essere *vera* per tutti gli impiegati, ma in una regola attiva siamo interessati alle violazioni del vincolo. Useremo quindi per la condizione delle regole attive la sua *negazione*:

```
IMPIEGATO: not exists (select * from Dipartimento
                       where NomeDip = Impiegato.Dipart)
```

Il predicato può essere espresso in forma negata anche relativamente alle tuple di DIPARTIMENTO; in tal caso, il vincolo è violato se esiste qualche impiegato privo di dipartimento:

```
DIPARTIMENTO: exists (select * from Impiegato
                      where Dipart not in
                            (select NomeDip
                             from Dipartimento))
```

Occorre poi costruire quattro regole attive: due di loro reagiscono a ogni inserimento su IMPIEGATO o modifica dell'attributo Dipart, annullando gli effetti delle due primitive se esse violano il vincolo. Rammentiamo infatti che le operazioni sulla tabella *interna*, secondo la semantica dei vincoli di integrità referenziale, debbono essere impedite.

La prima delle due regole è codificata dal seguente trigger in DB2:

```
create trigger DipRef1
after insert on Impiegato
for each row
when (not exists
 (select * from Dipartimento
  where NomeDip = new.Dipart))
signal sqlstate '70006'
    ('impiegato senza dipartimento');
```

La seconda regola ha, rispetto alla prima, solo una parte evento differente:

```
create trigger DipRef2
after update of Dipart on Impiegato
for each row
when (not exists
 (select * from Dipartimento
  where NomeDip = new.Dipart))
signal sqlstate '70006'
    ('impiegato senza dipartimento');
```

La terza regola reagisce alla cancellazione di tuple da DIPARTIMENTO ponendo a *null* il valore dell'attributo Dipart delle tuple coinvolte:

```
create trigger DipRef3
after delete on Dipartimento
for each row
when (exists
  (select * from Impiegato
   where Dipart = old.NomeDip))
update Impiegato
 set Dipart = null
 where Dipart = old.NomeDip
```

Si noti che la condizione è semplificata e ottimizzata rispetto a quanto visto in precedenza; essa individua come critici quegli impiegati il cui dipartimento coincide con un dipartimento cancellato dalla primitiva di `delete`. Infine, si noti che la condizione potrebbe addirittura essere omessa, perché l'azione è automaticamente estesa a tutte e sole le tuple selezionate dalla condizione.

La quarta regola reagisce alle modifiche dell'attributo **NomeDip** riproducendo le stesse modifiche sull'attributo **Dipart** (tenendo conto che esso costituisce una chiave per la tabella IMPIEGATO):

```
create trigger DipRef4
after update of Dipartimento on NomeDip
for each row
when (exists
  (select * from Impiegato
   where Dipart = old.NomeDip))
update Impiegato
 set Dipart = new.NomeDip
 where Dipart = old.NomeDip
```

Si noti che anche in questo caso la condizione è ottimizzata e potrebbe essere omessa.

Regole aziendali Le regole aziendali esprimono le strategie di una azienda nel perseguire i propri scopi primari. Esempi sono le regole che descrivono acquisto e cessione di titoli in base alle fluttuazioni del mercato, le regole per la gestione di una rete di trasporti o di energia, oppure anche, come visto nel paragrafo 5.4.3, le regole per la gestione del magazzino in base alla variazione delle quantità giacenti. Alcune di queste regole sono semplici *allertatori* (*alerters*) che si limitano nella parte azione ad emettere messaggi e avvisi, lasciando agli utenti la gestione delle situazioni anomale.

Le regole aziendali, come si vedrà nel Capitolo 6, possono anche essere usate per esprimere vincoli sullo schema. Queste regole possono essere classificate come regole di integrità o di derivazione. Le regole di integrità sono predicati che esprimono condizioni che devono essere vere. Nei sistemi commerciali che lo consentono, possono essere programmate tramite le clausole `check` o le *asserzioni*; molti sistemi però introducono restrizioni ai predicati esprimibili con queste clausole, di fatto limitandone la genericità. Inoltre, l'uso di questi costrutti costringe anche ad adottare,

quale politica di reazione alla violazione del vincolo, quella presente nello standard, mentre spesso la reazione desiderata è differente. Le regole attive sono quindi adatte per specificare e implementare vincoli e reazioni veramente "generici".

Vediamo come è possibile programmare, tramite una regola attiva, la regola aziendale RV2 che introdurremo nel Capitolo 6, qui anticipata:

(RV2) *un impiegato* **non deve** *avere uno stipendio maggiore del direttore del dipartimento al quale afferisce.*

Supponiamo che siano disponibili due tabelle IMPIEGATO e DIPARTIMENTO; assumiamo che ImpNum sia la chiave principale di IMPIEGATO, DipNum di DIPARTIMENTO, e assumiamo in IMPIEGATO gli attributi Mgr, Stipendio e DipNum, in DIPARTIMENTO l'attributo Direttore. Le operazioni che possono violare il vincolo sono le modifiche dello stipendio degli impiegati nel loro doppio ruolo di dipendente e di manager e l'inserimento di un nuovo impiegato. Supponiamo che fra queste la modifica realmente critica sia l'aumento di stipendio dato a un generico impiegato. Supponiamo inoltre che la politica di reazione sia di bloccare la modifica, segnalandone il motivo. Queste scelte sono realizzate dal seguente trigger, scritto secondo la sintassi di DB2.

```
create trigger StipendioEccessivo
after update on Stipendio of Impiegato
for each row
when new.Stipendio >
 select Stipendio
 from Impiegato
 where ImpNum = new.Mgr
 and ImpNum in
 (select Direttore
  from Dipartimento
  where DipNum = new.DipNum)
signal sqlstate '70005'
     ('Stipendio troppo elevato')
```

Le regole relative alla gestione dei magazzini o dei fornitori, illustrate nei paragrafi 5.4.2 e 5.4.3, possono essere considerate anch'esse regole aziendali.

Le regole aziendali sono particolarmente vantaggiose quando esprimono le politiche reattive a livello di schema (valide quindi per tutte le applicazioni) perché ne consentono una specifica univoca e centralizzata. Ciò consente di ottenere la proprietà di *indipendenza della conoscenza*, discussa nel paragrafo introduttivo di questo capitolo.

5.5 Controllo dell'accesso

La presenza di meccanismi di protezione dei dati riveste grande rilevanza in molte applicazioni. Uno dei compiti più importanti di un amministratore di basi di dati consiste nello scegliere e implementare opportune politiche di controllo di accesso. SQL

riconosce l'importanza di questo aspetto e un insieme delle istruzioni del linguaggio è dedicato a questo obiettivo.

SQL prevede innanzitutto che ogni utente sia identificato in modo univoco dal sistema. L'identificazione dell'utente può sfruttare le funzionalità del sistema operativo (per cui a un utente della base di dati corrisponde un utente del sistema) o essere indipendente. I sistemi commerciali più sofisticati offrono una gestione indipendente. con una propria procedura d'identificazione, per cui a un utente del sistema possono corrispondere più utenti della base di dati e viceversa.

5.5.1 Risorse e privilegi

Le risorse che il sistema protegge sono normalmente tabelle e viste, con la possibilità di specificare singoli attributi all'interno di esse. Il modello di controllo dell'accesso di SQL permette comunque di proteggere un qualsiasi componente dello schema (domini, procedure ecc.).

Di regola l'utente che crea la risorsa ne è il proprietario ed è autorizzato a compiere su di essa qualsiasi operazione. Un sistema in cui solo i proprietari delle risorse fossero autorizzati a farne uso sarebbe di limitata utilità, come lo sarebbe un sistema in cui tutti gli utenti fossero in grado di utilizzare in qualsiasi modo ogni risorsa. SQL offre invece dei meccanismi di gestione flessibili, mediante i quali è possibile specificare quali sono le risorse cui devono accedere gli utenti e quali sono invece le risorse che devono essere mantenute private. Il sistema basa il controllo di accesso su un concetto di *privilegio*. Gli utenti possiedono dei privilegi di accesso alle risorse del sistema.

Ogni privilegio è caratterizzato dai seguenti parametri:

1. la risorsa cui si riferisce;
2. l'utente che concede il privilegio;
3. l'utente che riceve il privilegio;
4. l'azione che viene permessa sulla risorsa;
5. se il privilegio può essere trasmesso o meno ad altri utenti.

Quando una risorsa viene creata, il sistema concede automaticamente tutti i privilegi su tale risorsa al creatore. Esiste inoltre un utente predefinito, `_system`, che rappresenta il database administrator, il quale possiede tutti i privilegi su tutte le risorse.

I privilegi disponibili sono i seguenti.

- `insert`: permette di inserire un nuovo oggetto nella risorsa (si può applicare solo alle tabelle e alle viste).
- `update`: permette di aggiornare il valore di un oggetto (vale per le tabelle, le viste e gli attributi).
- `delete`: permette di rimuovere oggetti dalla risorsa (vale solo per le tabelle e le viste).
- `select`: permette di leggere la risorsa, ovvero utilizzarla nell'ambito di una interrogazione (vale per le tabelle, le viste e gli attributi).
- `references`: permette che venga fatto un riferimento a una risorsa nell'ambito della definizione dello schema di una tabella. Può essere associato solo a tabelle e a specifici attributi. Con il privilegio di `references` (per esempio su una

tabella DIPARTIMENTO, di proprietà di Paolo) l'utente cui è concesso il privilegio (per esempio, Stefano) può definire un vincolo di `foreign key` (per esempio, sulla sua tabella IMPIEGATO), richiedendo che un attributo della propria tabella abbia valori contenuti tra le chiavi della tabella referenziata. A questo punto, se Stefano specifica sul vincolo una politica di reazione di tipo `no action`, a Paolo può essere impedito di cancellare o modificare delle righe della propria tabella DIPARTIMENTO se il comando di aggiornamento rende scorretto il contenuto di IMPIEGATO. Perciò, la concessione del privilegio di `references` può limitare la possibilità di modificare la risorsa.

- `usage`: permette che venga usata la risorsa, per esempio nell'ambito della definizione dello schema di una tabella, ma solo per risorse come i domini.

Il privilegio di effettuare un `drop` o un `alter` di un oggetto non può essere concesso, ma rimane di competenza del creatore dell'oggetto stesso. I privilegi vengono concessi o revocati tramite le istruzioni `grant` e `revoke`.

5.5.2 Comandi per concedere e revocare privilegi

La sintassi del comando `grant` è la seguente:

`grant` *Privilegi* `on` *Risorsa* `to` *Utenti* [`with grant option`]

Il comando permette di concedere i *Privilegi* sulla *Risorsa* agli *Utenti*. Per esempio, il comando:

```
grant select on Dipartimento to Stefano
```

concede all'utente Stefano il privilegio di `select` sulla tabella DIPARTIMENTO. La clausola `with grant option` specifica se deve essere concesso a Stefano anche il privilegio di propagare il privilegio ad altri utenti. È possibile usare al posto dei privilegi la parola chiave `all privileges`, che identifica tutti i privilegi che l'utente può concedere sulla particolare risorsa. Così il comando:

```
grant all privileges on Impiegato to Paolo, Riccardo
```

concede sulla tabella IMPIEGATO agli utenti Paolo e Riccardo tutti i privilegi che possono essere concessi da chi esegue il comando.

Il comando `revoke` fa invece l'inverso: sottrae a un utente i privilegi che gli erano stati concessi:

`revoke` *Privilegi* `on` *Risorsa* `from` *Utenti* [`restrict` | `cascade`]

Tra i privilegi che possono essere rimossi, oltre a quelli che possono comparire come argomento del comando di `grant`, vi è pure il privilegio `grant option`, derivante dall'uso dell'opzione `with grant option`.

L'unico utente che può sottrarre privilegi a un altro utente è l'utente che aveva concesso i privilegi in primo luogo. Il comando di `revoke` può eliminare tutti i privilegi

che erano stati concessi, o limitarsi a revocarne un sottoinsieme. L'opzione `restrict` è il valore di default e specifica che il comando non deve essere eseguito qualora la revoca dei privilegi all'utente comporti qualche altra revoca di privilegi, come può capitare quando l'utente ha ricevuto i privilegi con la `grant option` e ha propagato il privilegio ad altri utenti, o come capita quando il privilegio che si vuole revocare è stato usato per la definizione di una vista o di una tabella dell'utente. Con l'opzione `restrict` in una situazione di questo tipo viene segnalato un errore. Con l'opzione di `cascade`, invece, si forza l'esecuzione del comando; così tutti i privilegi che erano stati propagati vengono revocati e tutti gli elementi della base di dati che erano stati costruiti sfruttando questi privilegi vengono rimossi. Si noti che l'opzione `cascade` può generare anche in questo caso una reazione a catena, per cui, per ogni elemento che viene rimosso, vengono anche rimossi tutti gli oggetti che hanno una qualche relazione di dipendenza da esso; come in altri casi bisogna prestare molta attenzione per evitare che un semplice comando produca modifiche estese e non desiderate sulla base di dati.

Non è solo il comando di `revoke` a poter generare delle reazioni a catena: anche quello di `grant` può esibire un comportamento analogo. Può infatti capitare che un utente abbia ricevuto un privilegio su una tabella che gli ha permesso di creare delle viste che fanno riferimento a questa tabella, tramite per esempio un privilegio di `select`. Qualora all'utente vengano concessi ulteriori privilegi sulla tabella, questi privilegi vengono automaticamente concessi sulle viste (e ricorsivamente sulle viste costruite a partire dalle viste).

5.5.3 I ruoli in SQL-3

SQL-3 ha introdotto una novità significativa nell'ambito del controllo dell'accesso, proponendo un modello di controllo dell'accesso basato sui ruoli (*Role-Based Access Control*, RBAC). Questo modello, pur mantenendo il supporto per il tradizionale approccio che associa direttamente i privilegi agli utenti, introduce un meccanismo che disaccoppia l'attribuzione di un insieme di privilegi agli utenti dalla loro attivazione.

In SQL-3 è possibile creare un ruolo tramite un opportuno comando `create role` *NomeRuolo*. Il ruolo si comporta come una sorta di contenitore di privilegi, che vengono attribuiti a esso tramite il comando di `grant` visto prima. Il comando di `grant` viene inoltre utilizzato per concedere agli utenti la possibilità di ricoprire un certo ruolo, beneficiando dei privilegi a esso associati. Per fruire però dei privilegi è necessario che l'utente invochi un esplicito comando `set role` *NomeRuolo*. In ogni istante un utente dispone quindi dei privilegi che gli sono stati attribuiti direttamente e dei privilegi associati al ruolo che è stato esplicitamente attivato.

Questo approccio rappresenta una realizzazione significativa di un modello di controllo dell'accesso flessibile. La motivazione principale del modello è di rispettare il principio del "minimo privilegio", il quale prescrive che per garantire un buon comportamento in termini di sicurezza è bene che ogni utente disponga esclusivamente dell'insieme dei privilegi che sono necessari per svolgere il proprio compito. Questo principio motiva innanzitutto l'introduzione di un modello di controllo dell'accesso a granularità fine come quello tradizionale di SQL, che permette di isolare le risorse effettive cui un utente ha diritto di accedere. Il modello a ruoli tiene inoltre conto che ciascun utente ha in momenti diversi la necessità di svolgere diverse funzioni; tramite il ruolo diven-

ta possibile variare dinamicamente l'insieme di privilegi attivati, disponendo in ogni momento dell'insieme minimo di privilegi necessari per una certa attività.

Un altro vantaggio significativo dei ruoli è la semplificazione dell'attività di amministrazione dei privilegi. L'introduzione di un nuovo utente nel sistema può essere gestita abilitando con pochi comandi i ruoli che l'utente deve poter attivare, senza dover ripetere per il nuovo identificatore di utente tutti i comandi di `grant` che descrivono la collezione di privilegi raccolti nei ruoli.

5.6 Transazioni

Una *transazione* identifica una unità elementare di lavoro svolta da una applicazione, cui si vogliono associare particolari caratteristiche di correttezza, robustezza e isolamento. In particolare, nel nostro contesto, il concetto è rilevante soprattutto con riferimento alle operazioni che modificano il contenuto della base di dati. Un sistema che mette a disposizione un meccanismo per la definizione e l'esecuzione di transazioni con le caratteristiche suddette viene detto *sistema transazionale*.

Nella parte conclusiva di questo Capitolo, cominciamo ad introdurre i concetti fondamentali della gestione delle transazioni, per poi approfondire l'argomento nel Capitolo 12.

5.6.1 Specifica delle transazioni: commit e rollback

Una transazione può essere definita sintatticamente: ogni transazione, quale che sia il linguaggio di programmazione in cui essa è scritta, è specificata racchiudendo la sequenza di operazioni che la compongono all'interno di una coppia di istruzioni che ne specificano l'inizio e la conclusione. Per esempio, in SQL una transazione che trasferisce 10 unità da un conto corrente (numero 42177) a un altro (numero 12202) può venire specificata nel modo seguente

```
start transaction;
update ContoCorrente
  set Saldo = Saldo + 10
  where NumConto = 12202;
update ContoCorrente
  set Saldo = Saldo - 10
  where NumConto = 42177;
commit work;
```

In particolare, l'istruzione `commit work` (in alcuni sistemi semplicemente `commit` o con `work` opzionale) specifica il fatto che si richiede una conclusione positiva della transazione e che quindi tutti gli aggiornamenti debbono essere salvati nella base di dati. In programmi più articolati, che prevedano anche strutture di controllo, è possibile avere casi in cui, dopo avere effettuato alcune operazioni, le condizioni verificate portano il programma stesso a stabilire che gli aggiornamenti debbono essere annullati. Allo scopo, è disponibile un'altra istruzione, la `rollback work` (o semplicemente `rollback`). Nell'esempio precedente, si potrebbe fare tale scelta se il saldo

del conto corrente da cui si preleva risultasse negativo, nel modo seguente, usando una sintassi intuitiva per le strutture di controllo:[1]

```
start transaction;
update ContoCorrente
  set Saldo = Saldo + 10
  where NumConto = 12202;
update ContoCorrente
  set Saldo = Saldo - 10
  where NumConto = 42177;
select Saldo into A
  from ContoCorrente
  where NumConto = 42177;
if A >= 0
  then commit work;
  else rollback work;
```

Vale la pena notare che, in alcuni contesti, per esempio gli ambienti interattivi, il sistema assume che ciascuna istruzione da sola costituisca una transazione; si usa talvolta il termine *modalità autocommit* per fare riferimento a questa caratteristica. Non sono in tal caso necessarie né la `start transaction` né la `commit`.

Poiché in generale le transazioni possono essere specificate nell'ambito di programmi di ogni tipo, anche con strutture di controllo articolate, è utile introdurre il concetto di transazione *ben formata* (a tempo di esecuzione) che prevede un inizio `start transaction` (o in alcuni sistemi `begin transaction`) e una fine con un'ideale `end transaction`, nel cui corso viene eseguito uno solo dei due comandi `commit` o `rollback` e in cui non avvengono operazioni di accesso e/o modifica alla base di dati successive all'esecuzione del comando di commit o rollback. In alcune interfacce transazionali, una coppia di comandi `end transaction`, `start transaction` viene immediatamente e implicitamente eseguita dopo ogni commit o rollback, in modo da rendere tutte le computazioni transazioni ben formate. In tal caso, la `start transaction` diventa opzionale o addirittura non è prevista. Nel seguito, assumeremo che tutti i programmi per la modifica del contenuto di un DBMS siano transazioni ben formate.

5.6.2 Proprietà acide delle transazioni

Tutto il codice che viene eseguito all'interno di una transazione gode di proprietà particolari, le cosiddette *proprietà acide* delle transazioni: *atomicità*, *consistenza*, *isolamento* e *persistenza* (il termine è un acronimo derivante dall'inglese, ove ACID denota le iniziali di: "Atomicity, Consistency, Isolation, Durability").

[1] In effetti, in questo caso, sarebbe opportuno non effettuare per niente l'operazione, ma procediamo in questo modo per illustrare il concetto con semplicità.

Atomicità L'*atomicità* rappresenta il fatto che una transazione è un'unità *indivisibile* di esecuzione; o vengono resi visibili tutti gli effetti di una transazione, oppure la transazione non deve avere alcun effetto sulla base di dati, con un approccio "tutto o niente". In pratica, non è possibile lasciare la base di dati in uno stato intermedio attraversato durante l'elaborazione della transazione.

L'atomicità ha conseguenze significative sul piano operativo. Se durante l'esecuzione delle operazioni si verifica un errore e una delle operazioni di lettura o modifica della base di dati non può essere portata a compimento, allora il sistema deve essere in grado di ricostruire la situazione esistente all'inizio della transazione, *disfacendo* il lavoro svolto dalle istruzioni eseguite fino a quel momento (operazione di `undo`). Viceversa, dopo l'esecuzione del commit, il sistema deve assicurare che la transazione lasci la base di dati nel suo stato finale; ciò può comportare di dover *rifare* il lavoro svolto (operazione di `redo`). In questo modo, la corretta effettuazione dell'operazione di commit fissa il momento, atomico e indivisibile, in cui la transazione "va a buon fine"; prima di tale operazione, qualunque guasto provoca la eliminazione di tutti gli effetti della transazione, che ripristina lo stato iniziale.

Quando viene eseguito il comando `rollback work`, la situazione è simile a un "*suicidio*" autonomamente deciso nell'ambito della transazione. Viceversa, il sistema può decidere che la transazione non può essere portata a corretto compimento e "*uccidere*" la transazione. Infine, varie transazioni possono essere "*uccise*" a seguito di un guasto del sistema. In entrambe le situazioni (suicidio od omicidio), i meccanismi che realizzano l'abort di una transazione utilizzano le stesse strutture dati e talvolta gli stessi algoritmi. In genere, ci aspettiamo che le applicazioni siano scritte bene e che perciò la maggioranza delle transazioni vadano a buon fine e terminino con un commit; solo in casi sporadici legati a malfunzionamenti o situazioni impreviste le transazioni terminano con un abort.

Consistenza La *consistenza* richiede che l'esecuzione della transazione non violi i vincoli di integrità definiti sulla base di dati. Quando il sistema rileva che una transazione sta violando uno dei vincoli, per esempio che si sta inserendo una tupla con un campo chiave avente un valore già presente nella tabella, il sistema interviene per annullare la transazione o per correggere la violazione del vincolo.

La verifica di vincoli di integrità di tipo *immediato* può essere fatta nel corso della transazione, rimuovendo gli effetti della specifica istruzione di manipolazione dei dati che causa la violazione del vincolo, senza imporre un abort alle transazioni. Invece, la verifica di vincoli di integrità di tipo *differito* deve essere effettuata alla conclusione della transazione, dopo che l'utente ha richiesto il commit. Si noti che in questo secondo caso, se il vincolo è violato, l'istruzione `commit work` non va a buon fine, e gli effetti della transazione vengono annullati "in extremis", cioè poco prima di produrre e rendere visibile lo stato finale della base di dati, poiché questo stato sarebbe inconsistente.

Isolamento L'*isolamento* richiede che l'esecuzione di una transazione sia indipendente dalla contemporanea esecuzione di altre transazioni. In particolare, si richiede che il risultato dell'esecuzione concorrente di un insieme di transazioni sia analogo al

risultato che le stesse transazioni otterrebbero qualora ciascuna di esse fosse eseguita da sola.

L'isolamento si pone come obiettivo anche di rendere l'esito di ciascuna transazione indipendente da tutte le altre; si vuole cioè impedire che l'esecuzione di un rollback di una transazione causi l'esecuzione del rollback di altre transazioni, eventualmente generando una reazione a catena (effetto *domino*).

Persistenza La *persistenza* invece richiede che l'effetto di una transazione che ha eseguito il commit correttamente non venga più perso. In pratica, una base di dati deve garantire che nessun dato venga perso per nessun motivo; si pensi al valore dell'informazione contenuta in una base di dati quando essa rappresenta un'operazione su un conto corrente bancario.

Sottolineiamo che è responsabilità del DBMS assicurare il soddisfacimento delle proprietà acide, mentre l'utente si limita a specificare, per ciascuna transazione, l'inizio, le operazioni e la fine.

È opportuno osservare che la definizione di transazione data in questo paragrafo è diversa dal concetto di transazione che può avere un utente. Per il sistema, una transazione è una unità di esecuzione caratterizzata da proprietà acide; per l'utente, una transazione è spesso identificata con ogni interazione col sistema, caratterizzata da una iniziale immissione di dati cui fa seguito una risposta da parte del sistema. Spesso le due nozioni coincidono, ma altre volte una transazione di sistema incapsula varie transazioni d'utente, oppure una transazione d'utente incapsula varie transazioni di sistema.

Nel Capitolo 10, quando parleremo dello sviluppo di applicazioni che interagiscono con basi di dati, tratteremo diverse modalità per l'uso delle transazioni da parte di un'applicazione. Nel Capitolo 12 approfondiremo invece gli aspetti tecnologici della gestione delle transazioni, mostrando cioè come i DBMS garantiscono le proprietà acide.

Note bibliografiche

Per quanto riguarda le caratteristiche evolute di SQL, valgono gli stessi riferimenti che sono stati forniti nel capitolo precedente. Anche in questo caso si può consigliare di consultare i manuali che accompagnano i sistemi relazionali commerciali, i quali oggigiorno sono quasi sempre disponibili per la consultazione in Internet sui siti dei produttori, con funzioni di ricerca efficaci che consentono in breve tempo di risolvere dubbi sulla sintassi di un comando o sull'insieme di opzioni effettivamente riconosciute dall'interprete SQL di uno specifico sistema.

Per quanto riguarda le basi di dati attive, la ricerca in questo campo ha visto il momento di massima produttività tra il 1985 e il 1995, portando alla definizione di numerosi prototipi e dello standard SQL-3. Il libro *Active Database Systems*, di Widom e Ceri [84], contiene una descrizione completa dei principali risultati prodotti in questo periodo. Anche una delle sei parti di cui è composto il libro *Introduction to Advanced Database Systems* [87], è dedicata alle basi di dati attive. Lo standard SQL-3, oltre che nei documenti ufficiali, è descritto nell'articolo [53] e in [61]. La descrizione dei trigger

disponibili in Oracle Server e in IBM DB2 è tratta dai loro manuali, disponibili liberamente sui siti dei produttori; DB2 è anche descritto nell'articolo [35]. La possibilità di dare una specifica dichiarativa e successivamente derivare le regole attive è stata introdotta nell'articolo [36]. Una metodologia di progettazione di basi di dati che fa ampio uso delle regole attive e anche della impostazione a oggetti è descritta nel libro *Designing Database Applications with Objects and Rules*, di Ceri e Fraternali [20].

Per quanto riguarda il controllo dell'accesso nelle basi di dati, il testo [18] è dedicato totalmente all'argomento. Per quanto riguarda il concetto di transazione, rimandiamo al Capitolo 12 e alle sue note bibliografiche.

Esercizi

Soluzioni sul sito http://www.ate.uniroma2.it/atzeni

5.1 Definire sulla tabella IMPIEGATO il vincolo che il dipartimento Amministrazione abbia meno di 100 dipendenti, con uno stipendio medio superiore a 40.

5.2 Definire (con una opportuna notazione) su una relazione PAGHE (Matricola, StipLordo, Ritenute, StipNetto, OK) un vincolo che imponga che il valore di OK è:

- zero se StipNetto è pari alla differenza fra StipLordo e Ritenute
- uno altrimenti.

5.3 Definire a livello di schema il vincolo che il massimo degli stipendi degli impiegati di dipartimenti con sede a Firenze sia minore dello stipendio di tutti gli impiegati del dipartimento Direzione.

5.4 Indicare quali delle seguenti affermazioni sono vere.

1. Nei sistemi relazionali le viste possono essere utili al fine di rendere più semplice la scrittura delle interrogazioni.
2. Nei sistemi relazionali le viste possono essere utili al fine di rendere più efficienti le interrogazioni.
3. Nei sistemi relazionali le viste introducono ridondanze.

5.5 Dato il seguente schema:

AEROPORTO(Città,Nazione,NumPiste)
VOLO(IdVolo,GiornoSett,CittaPart,OraPart,CittàArr,OraArr,TipoAereo)
AEREO(TipoAereo,NumPasseggeri,QtaMerci)

scrivere, facendo uso di una vista, l'interrogazione SQL che permette di determinare il massimo numero di passeggeri che possono arrivare in un aeroporto italiano dalla Francia di giovedì (se vi sono più voli, si devono sommare i passeggeri).

5.6 Definire una vista che mostra per ogni dipartimento il valore medio degli stipendi superiori alla media.

5.7 Dato il seguente schema relazionale:

- DIPENDENTE(CodiceFiscale, Cognome, Nome)
- PROFESSORE(CodiceFiscale, Qualifica, Anzianità,Facoltà) con vincolo di integrità referenziale tra CodiceFiscale e la relazione DIPENDENTE e fra Facoltà e la relazione FACOLTÀ

- FACOLTÀ(Codice, Nome, Indirizzo)
- CORSODISTUDIO(Codice, Nome, Facoltà, Presidente) con vincolo di integrità referenziale tra Facoltà e la relazione FACOLTÀ a fra Presidente e la relazione PROFESSORE
- COLLABORAZIONE(CorsoDiStudio, Facoltà, Professore, Tipo) con vincolo di integrità referenziale fra CorsodiStudio, Facoltà e la relazione CORSODISTUDIO e fra Professore e la relazione PROFESSORE
- CORSO(Codice, Materia, Docente, Semestre) con vincolo di integrità referenziale fra Materia e la relazione MATERIA e fra Docente e la relazione PROFESSORE
- MATERIA(Sigla, Nome) formulare le interrogazioni in SQL:
  1. mostrare i professori, con codice fiscale, cognome, cognome, qualifica, anzianità e nome della eventuale facoltà di afferenza (per i professori che non afferiscono ad alcuna facoltà dovrà comparire il valore nullo);
  2. trovare cognome e qualifica dei professori che afferiscono alla stessa facoltà di un professore chiamato Mario Bruni di qualifica "ordinario";
  3. trovare i codici delle facoltà cui non afferisce alcun professore con cognome Bruni e qualifica "ordinario".

5.8 Considerare la base di dati relazionale definita per mezzo delle seguenti istruzioni (è lo schema già visto nell'Esercizio 4.16):

```
create table Studenti (
        Matricola numeric not null primary key,
        Cognome char(20) not null,
        Nome char(20) not null,
        DataNascita date not null
);
create table Esami (
        CodiceCorso numeric not null,
        Studente numeric not null
            references Studenti(Matricola),
        Data date not null,
        Voto numeric not null,
        primary key (CodiceCorso, Studente, Data)
);
```

Formulare in SQL:

1. l'interrogazione che trova lo studente con la media più alta.

5.9 Considerare la seguente base di dati relazionale:

VENDITE(NumeroScontrino, Data)
CLIENTI(Codice, Cognome, Età)
DETTAGLIVENDITE(NumeroScontrino, Riga, Prodotto, Importo, Cliente)

con valori nulli ammessi sull'attributo Cliente e con vincoli di integrità referenziale fra NumeroScontrino e la relazione VENDITE e fra Cliente e la relazione CLIENTI;
formulare in SQL:

- l'interrogazione che restituisce i prodotti acquistati in ciascuna data (che mostra cioè le coppie $< p, d >$ tali che il prodotto p è stato acquistato nella data d;

- l'interrogazione che restituisce i prodotti che sono stati acquistati in due date diverse;
- la vista VENDITECONTOTALE(NumeroScontrino, Totale), che riporta, per ogni scontrino l'importo totale (ottenuto come somma degli importi dei prodotti riportati sullo scontrino).

5.10 Considerare la seguente base di dati relazionale:

- PERSONE(FC, Cognome, Nome, Età)
- IMMOBILI(Codice, Via, NumeroCivico, Città, Valore)
- PROPRIETÀ(Persona, Immobile, Percentuale) con vincolo di di integrità referenziale fra Immobile e la relazione PERSONE e fra Immobile e la relazione IMMOBILI

Nota: l'attributo Percentuale indica la percentuale di proprietà.
Definire in SQL:

- la vista definita per mezzo della seguente espressione dell'algebra relazionale: Vista = Immobili $\bowtie_{Codice=Immobile}$ Proprietà
- l'interrogazione che fornisce codici fiscali, nome e cognome delle persone che posseggono un solo immobile e lo posseggono al 100%
- l'interrogazione che fornisce, per ciascuna persona, il codice fiscale, il nome, il cognome e il valore complessivo degli immobili di sua proprietà (dove il valore è la somma dei valori ciascuno pesato con la percentuale di proprietà: se Tizio possiede un immobile di valore 150 al 100% e uno di valore 200 al 50%, allora il valore complessivo sarà (150 x 100)/100 + (200 x 50)/100 = 250).

5.11 Si supponga di avere le tabelle:

MAGAZZINO(Prodotto,QtaDisp,Soglia,QtaRiordino)
ORDINEINCORSO(Prodotto,Qta)

Scrivere una procedura SQL che realizza il prelievo dal magazzino accettando due parametri, il prodotto *Prod* e la quantità da prelevare *QtaPrelievo*. La procedura deve verificare inizialmente che *QtaPrelievo* sia inferiore al valore di QtaDisp per il prodotto indicato. *QtaPrelievo* viene quindi sottratta al valore di QtaDisp. A questo punto la procedura verifica se per il prodotto QtaDisp risulta minore di Soglia, senza che in ORDINEINCORSO compaia già una tupla relativa al prodotto prelevato; se si, viene inserito un nuovo elemento nella tabella ORDINEINCORSO, con i valori di *Prod* e del corrispondente attributo QtaRiordino.

5.12 Dato lo schema relazionale:

IMPIEGATO(Nome, Stipendio, Dipnum)
DIPARTIMENTO(Dipnum, NomeManager)

definire le seguenti regole attive in DB2 e in Oracle:

1. una regola che, quando un dipartimento è cancellato, mette ad un valore di default (99) il valore di Dipnum degli impiegati appartenenti a quel dipartimento;
2. una regola che cancella tutti gli impiegati appartenenti a un dipartimento quando quest'ultimo è cancellato;
3. una regola che, ogni qual volta lo stipendio di un impiegato supera lo stipendio del suo manager, pone tale stipendio uguale allo stipendio del manager;

4. una regola che, ogni qual volta vengono modificati gli stipendi, verifica che non vi siano dipartimenti in cui lo stipendio medio cresce più del tre per cento, e in tal caso annulla la modifica.

5.13 Riferendosi alla base di dati dell'esercizio precedente, definire in DB2 e in Oracle un trigger R_1 che, quando è cancellato un impiegato che svolge il ruolo di manager di un dipartimento, cancella quel dipartimento e tutti i suoi dipendenti. Definire inoltre un trigger R_2 che, ogni qual volta vengono modificati gli stipendi, verifica la loro media, e se essa supera 50.000 cancella tutti gli impiegati il cui stipendio è stato modificato e attualmente supera 80.000. Si consideri poi uno stato di base di dati con sei impiegati: Giovanna, Maria, Andrea, Giuseppe, Sandro e Carla, in cui:

- Giovanna è manager del dipartimento 1, in cui lavorano Giovanna, Maria e Giuseppe;
- Maria è manager del dipartimento 2, in cui lavora Andrea;
- Giuseppe è manager del dipartimento 3, in cui lavorano Sandro e Carla.

Si assuma infine una transazione che cancella l'impiegata Giovanna e modifica gli stipendi in modo tale che la loro media ecceda 50.000 e lo stipendio di Maria dopo le modifiche ecceda 80.000. Descrivere l'operato dei trigger.

5.14 Mostrare il grafo di attivazione delle seguenti regole, descritte in forma sintetica, e discutere quindi la loro terminazione:

- r_1:
  - event: $Update(B)$;
  - condition: $B = 1$;
  - action: $A = 0$, $B = 1$
- r_2:
  - event: $Update(C)$;
  - condition: $B = 1$;
  - action: $A = 1$, $B = 0$
- r_3:
  - event: $Update(A)$;
  - condition: $A = 1$;
  - action: $A = 0$, $B = 1$, $C = 1$.

5.15 Dato lo schema relazionale:

DOTTORANDO(Nome, Disciplina, Relatore)
PROFESSORE(Nome, Disciplina)
CORSO(Titolo, Professore)
ESAMI(NomeStud,TitoloCorso)

descrivere in DB2 e in Oracle i trigger che gestiscono i seguenti vincoli di integrità (*business rules*):

1. ogni dottorando deve lavorare nella stessa area del suo relatore;
2. ogni dottorando deve aver sostenuto almeno 3 corsi nell'area del suo relatore;
3. ogni dottorando deve aver sostenuto l'esame del corso di cui è responsabile il suo relatore.

5.16 Tramite la definizione di una vista, permettere all'utente "Carlo" di accedere al contenuto di IMPIEGATO, escludendo l'attributo Stipendio.

5.17 Descrivere l'effetto delle seguenti istruzioni: quali autorizzazioni sono presenti dopo ciascuna istruzione? (Ciascuna linea è preceduta dal nome dell'utente che esegue il comando.)

```
Stefano:  grant select on Tabella to Paolo, Riccardo
                                  with grant option
Paolo:    grant select on Tabella to Piero
Riccardo: grant select on Tabella to Piero
                                  with grant option
Stefano:  revoke select on Tabella from Paolo
                cascade
Piero:    grant select on Tabella to Paolo
Stefano:  revoke select on Tabella from Riccardo
                cascade
```

Metodologie e modelli per il progetto

Nei capitoli precedenti sono state analizzate le modalità di descrizione (modelli) e manipolazione (linguaggi) di una base di dati, supponendo che la base di dati con la quale interagire esistesse già. Incominceremo adesso ad affrontare il problema che esiste a monte, quello di progettare una base di dati a partire dai suoi requisiti. Progettare una base di dati significa definirne struttura, caratteristiche e contenuto. Si tratta, come è facile immaginare, di un processo nel quale bisogna prendere molte decisioni delicate e l'uso di opportune metodologie è indispensabile per la realizzazione di un prodotto di alta qualità.

In questo capitolo introduttivo, affrontiamo il problema della progettazione di basi di dati da un punto di vista generale e proponiamo alcuni strumenti di lavoro. In particolare, forniremo, nel Paragrafo 6.1, un inquadramento generale nel contesto dello sviluppo dei sistemi informativi e presenteremo una metodologia di progettazione che si è largamente diffusa nell'ambito delle basi di dati. Nel Paragrafo 6.2 illustreremo invece il modello *Entità-Relazione*, che fornisce al progettista un valido strumento per produrre una rappresentazione dei dati, detta *schema concettuale*, sulla quale l'intera metodologia si fonda.

La metodologia di riferimento è articolata in tre fasi: la *progettazione concettuale*, la *progettazione logica* e la *progettazione fisica*. Ognuna di queste fasi verrà presentata in dettaglio nei capitoli successivi a questo. La seconda parte del libro verrà completata da un capitolo che descrive la *normalizzazione*, una importante tecnica di analisi di qualità per schemi di basi di dati.

6.1 Introduzione alla progettazione

6.1.1 Il ciclo di vita dei sistemi informativi

La progettazione di una base di dati costituisce solo una delle componenti del processo di sviluppo di un sistema informativo complesso e va quindi inquadrata in un contesto più ampio, quello del *ciclo di vita* dei sistemi informativi.

Come descritto in Figura 6.1, il ciclo di vita di un sistema informativo comprende, generalmente, le seguenti attività.

- **Studio di fattibilità.** Serve a definire, in maniera per quanto possibile precisa, i costi delle varie alternative possibili e a stabilire le priorità di realizzazione delle varie componenti del sistema.
- **Raccolta e analisi dei requisiti.** Consiste nell'individuazione e nello studio delle proprietà e delle funzionalità che il sistema informativo dovrà avere. Questa fase richiede un'interazione con gli utenti del sistema e produce una descrizione completa, ma generalmente informale, dei dati coinvolti (anche in termini di

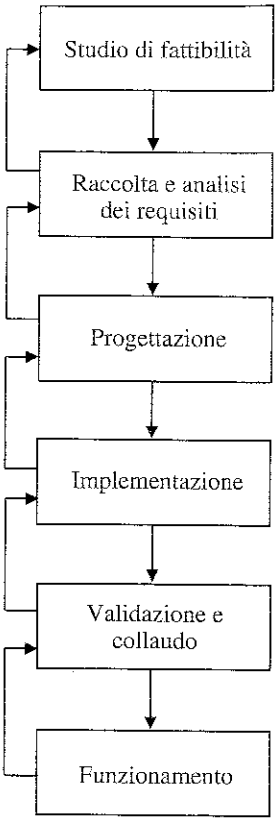

Figura 6.1 Ciclo di vita di un sistema informativo.

previsione sul carico applicativo) e delle operazioni su di essi (anche in termini di previsione sulla loro frequenza). Vengono inoltre stabiliti i requisiti software e hardware del sistema informativo.

- **Progettazione**. Si divide generalmente in *progettazione dei dati* e *progettazione delle applicazioni*. Nella prima si individua la struttura e l'organizzazione che i dati dovranno avere, nell'altra si definiscono le caratteristiche dei programmi applicativi. Le due attività sono complementari e possono procedere in parallelo o in cascata. Le descrizioni dei dati e delle applicazioni prodotte in questa fase sono formali e fanno riferimento a specifici modelli.

- **Implementazione**. Consiste nella realizzazione del sistema informativo secondo la struttura e le caratteristiche definite nella fase di progettazione. Viene costruita e popolata la base di dati e viene prodotto il codice dei programmi.
- **Validazione e collaudo**. Serve a verificare il corretto funzionamento e la qualità del sistema informativo. La sperimentazione deve prevedere, per quanto possibile, tutte le condizioni operative.
- **Funzionamento**. In questa fase il sistema informativo diventa operativo ed esegue i compiti per i quali era stato originariamente progettato. Se non si verificano malfunzionamenti o revisioni delle funzionalità del sistema, questa attività richiede solo operazioni di gestione e manutenzione.

Va precisato che, come indicato graficamente in Figura 6.1, il processo non è quasi mai strettamente sequenziale in quanto spesso, durante l'esecuzione di una delle attività citate, bisogna rivedere decisioni prese nell'attività precedente. Quello che si ottiene è proprio un "ciclo" di operazioni. Inoltre, si aggiunge talvolta alle attività citate quella di *prototipizzazione*, che consiste nell'uso di specifici strumenti software per la realizzazione rapida di una versione semplificata del sistema informativo, con la quale sperimentare le sue funzionalità. La verifica del prototipo può portare a una modifica dei requisiti e una eventuale revisione del progetto.

Le basi di dati costituiscono in effetti solo una delle componenti di un sistema informativo che tipicamente include anche i programmi applicativi, le interfacce con l'utente e altri programmi di servizio. Comunque, il ruolo centrale che i dati hanno in un sistema informativo giustifica ampiamente uno studio autonomo relativo alla progettazione delle basi di dati. Ci interesseremo perciò solo agli aspetti dello sviluppo dei sistemi informativi che riguardano da vicino il progetto delle basi di dati, rimandando a testi sull'ingegneria del software lo studio di tutte le altre attività connesse. In particolare, focalizzeremo la nostra attenzione sulla terza fase del ciclo di vita riportato in Figura 6.1, facendo riferimento alla progettazione dei dati e discutendo anche alcuni aspetti della relativa attività di raccolta e analisi dei requisiti che la precede. Questa maniera di procedere è peraltro coerente con l'approccio allo sviluppo dei sistemi informativi *basato sui dati*, in cui l'attenzione è centrata sui dati e sulle loro proprietà. Questo approccio prevede prima la progettazione della base di dati e, successivamente, la realizzazione delle applicazioni che la utilizzano.

6.1.2 Metodologie di progettazione e basi di dati

Un aspetto che vale la pena di precisare è che cosa si intende per *metodologia di progettazione* e quali sono le proprietà che una metodologia deve garantire. In buona sostanza, una metodologia di progettazione consiste in:

- una *decomposizione* dell'intera attività di progetto in passi successivi indipendenti tra loro;
- una serie di *strategie* da seguire nei vari passi e alcuni *criteri* per la scelta in caso di alternative;
- alcuni *modelli di riferimento* per descrivere i dati di ingresso e uscita delle varie fasi.

Le proprietà che una metodologia deve garantire sono principalmente:

- la *generalità* rispetto alle applicazioni e ai sistemi in gioco (e quindi la possibilità di utilizzo indipendentemente dal problema allo studio e dagli strumenti a disposizione);
- la *qualità del prodotto* in termini di correttezza, completezza ed efficienza rispetto alle risorse impiegate;
- la *facilità d'uso* delle strategie e dei modelli di riferimento.

Nell'ambito delle basi di dati, si è consolidata negli anni una metodologia di progetto che ha dato prova di soddisfare pienamente le proprietà descritte. Tale metodologia è articolata in tre fasi principali da effettuare in cascata (Figura 6.2) e si fonda su un principio dell'ingegneria semplice ma molto efficace: separare in maniera netta le decisioni relative a "cosa" rappresentare in una base di dati (prima fase), da quelle relative a "come" farlo (seconda e terza fase).

- **Progettazione concettuale**. Il suo scopo è quello di rappresentare le specifiche informali della realtà di interesse in termini di una descrizione formale e completa, ma indipendente dai criteri di rappresentazione utilizzati nei sistemi di gestione di basi di dati. Il prodotto di questa fase viene chiamato *schema concettuale* e fa riferimento a un *modello concettuale* dei dati. Come abbiamo accennato nel Paragrafo 1.3, i modelli concettuali ci consentono di descrivere l'organizzazione dei dati a un alto livello di astrazione, senza tenere conto degli aspetti implementativi. In questa fase infatti, il progettista deve cercare di rappresentare il *contenuto informativo* della base di dati, senza preoccuparsi né delle modalità con le quali queste informazioni verranno codificate in un sistema reale, né dell'efficienza dei programmi che faranno uso di queste informazioni.
- **Progettazione logica**. Consiste nella traduzione dello schema concettuale definito nella fase precedente, in termini del modello di rappresentazione dei dati adottato dal sistema di gestione di base di dati a disposizione. Il prodotto di questa fase viene denominato *schema logico* della base di dati e fa riferimento a un *modello logico* dei dati. Come noto, un modello logico ci consente di descrivere i dati secondo una rappresentazione ancora indipendente da dettagli fisici, ma concreta perché disponibile nei sistemi di gestione di base di dati. In questa fase, le scelte progettuali si basano, tra l'altro, su criteri di ottimizzazione delle operazioni da effettuare sui dati. Si fa comunemente uso anche di tecniche formali di verifica della qualità dello schema logico ottenuto. Nel caso del modello relazionale dei dati, la tecnica comunemente utilizzata è quella della *normalizzazione*.
- **Progettazione fisica**. In questa fase lo schema logico viene completato con la specifica dei parametri fisici di memorizzazione dei dati (organizzazione dei file e degli indici). Il prodotto di questa fase viene denominato *schema fisico* e fa riferimento a un *modello fisico* dei dati. Tale modello dipende dallo specifico sistema di gestione di basi di dati scelto e si basa sui criteri di organizzazione fisica dei dati in quel sistema.

Vediamo ora in che maniera i requisiti della base di dati vengono utilizzati nelle varie fasi della progettazione. È bene qui fare una distinzione tra *specifiche sui dati*,

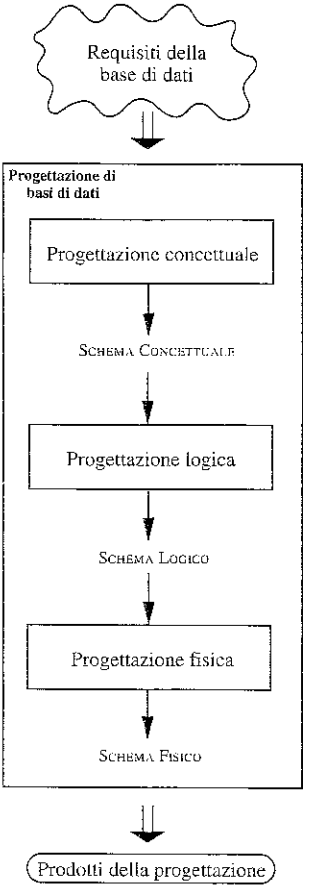

Figura 6.2 Le fasi della progettazione di una base di dati.

che riguardano il contenuto della base di dati, e *specifiche sulle operazioni*, che riguardano l'uso che utenti e applicazioni fanno della base di dati. Nella progettazione concettuale si fa uso soprattutto delle specifiche sui dati mentre le specifiche sulle operazioni servono solo a verificare che lo schema concettuale sia completo, contenga cioè le informazioni necessarie per eseguire tutte le operazioni previste. Nella progettazione logica lo schema concettuale in ingresso riassume le specifiche sui dati,

mentre le specifiche sulle operazioni si utilizzano, insieme alle previsioni sul carico applicativo, per ottenere uno schema logico che renda tali operazioni eseguibili in maniera efficiente. In questa fase bisogna anche conoscere il modello logico adottato ma non è ancora necessario conoscere il particolare DBMS scelto (solo la categoria cui appartiene). Infine, nella progettazione fisica si fa uso dello schema logico e delle specifiche sulle operazioni per ottimizzare le prestazioni del sistema. In questa fase bisogna anche tenere conto delle caratteristiche del particolare sistema di gestione di basi di dati utilizzato.

Il risultato della progettazione di una base di dati non è solo lo schema fisico, ma è costituito anche dallo schema concettuale e dallo schema logico. Lo schema concettuale fornisce infatti una rappresentazione della base di dati di alto livello, che può essere molto utile a scopo documentativo, mentre lo schema logico fornisce una descrizione concreta del contenuto della base di dati che, prescindendo dagli aspetti implementativi, è il riferimento per le operazioni di interrogazione e aggiornamento.

In Figura 6.3 vengono mostrati i prodotti delle varie fasi nel caso della progettazione di una base di dati relazionale basata sull'uso del più diffuso modello concettuale dei dati, il modello Entità-Relazione. A partire da requisiti rappresentati da documenti e moduli di vario genere, acquisiti anche attraverso l'interazione con gli utenti, viene costruito uno schema Entità-Relazione (rappresentato da un diagramma) che descrive a livello concettuale la base di dati. Questa rappresentazione viene poi tradotta in uno schema relazionale, costituito da una collezione di tabelle. Infine, i dati vengono descritti da un punto di vista fisico (tipo e dimensioni dei campi) e vengono specificate strutture ausiliarie, come gli indici, per l'accesso efficiente ai dati.

Nei prossimi capitoli affronteremo in maniera dettagliata i vari passi della progettazione di basi di dati secondo la decomposizione di Figura 6.2 e con riferimento ai modelli usati nella Figura 6.3. Prima di cominciare presenteremo, nel prossimo paragrafo, il modello Entità–Relazione, che si è ormai affermato come standard di riferimento nelle metodologie di progetto di basi di dati e negli strumenti di ausilio alla progettazione di sistemi informativi. La fase di progettazione concettuale che discuteremo nel prossimo capitolo si fonda su questo modello concettuale. Tratteremo successivamente la progettazione logica con riferimento al modello relazionale, che rimane a tutt'oggi il modello dei dati più diffuso nei sistemi di gestione di basi di dati.

6.2 Il modello Entità-Relazione

Il modello Entità-Relazione (nel seguito utilizzeremo spesso per questo termine l'abbreviazione E-R) è un modello *concettuale* di dati e, come tale, fornisce una serie di strutture, dette *costrutti*, atte a descrivere la realtà di interesse in una maniera facile da comprendere e che prescinde dai criteri di organizzazione dei dati nei calcolatori. Questi costrutti vengono utilizzati per definire *schemi* che descrivono l'organizzazione e la struttura delle *occorrenze*[1] dei dati, ovvero, dei valori assunti dai dati al variare

[1] In genere si utilizza il termine *istanza* invece di *occorrenza*, ma noi qui preferiamo occorrenza per non generare confusione con il concetto di istanza (insieme di tuple) utilizzato nel modello relazionale.

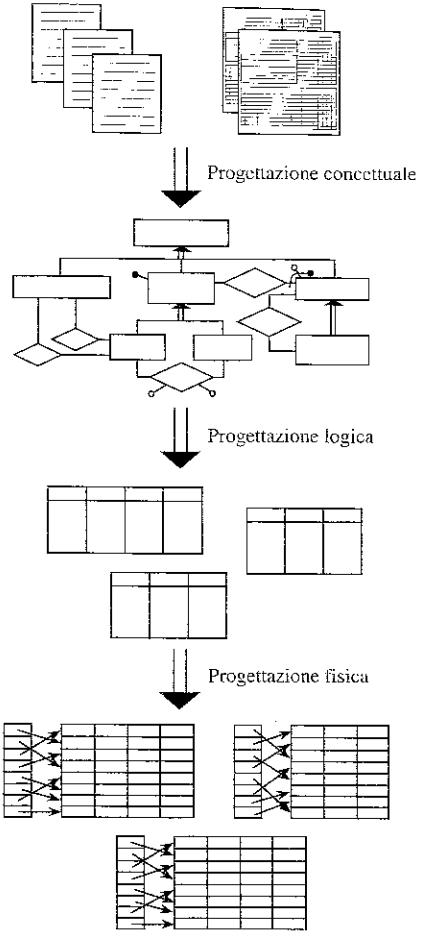

Figura 6.3 I prodotti delle varie fasi del progetto di una base di dati relazionale con il modello Entità-Relazione.

del tempo. Nella tabella in Figura 6.4 vengono elencati tutti i costrutti che il modello E-R mette a disposizione: si può osservare che, per ogni costrutto, esiste una relativa rappresentazione grafica. Come vedremo, questa rappresentazione ci consente di definire uno schema E-R mediante un diagramma che ne semplifica l'interpretazione.

Costrutti	Rappresentazione grafica
Entità	
Relazione	
Attributo semplice	
Attributo composto	
Cardinalità di relazione	(m_1,M_1) (m_2,M_2)
Cardinalità di attributo	(m,M)
Identificatore interno	
Identificatore esterno	
Generalizzazione	
Sottoinsieme	

Figura 6.4 I costrutti del modello E-R e la loro rappresentazione grafica.

6.2.1 I costrutti principali del modello

Cominciamo ad analizzare i costrutti principali di questo modello: le entità, le relazioni e gli attributi.

Entità Rappresentano classi di oggetti (per esempio, fatti, cose, persone) che hanno proprietà comuni ed esistenza "autonoma" ai fini dell'applicazione di interesse:

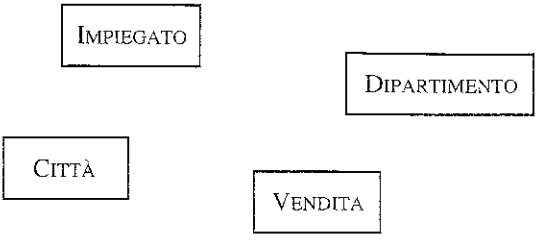

Figura 6.5 Esempi di entità nel modello E-R.

CITTÀ, DIPARTIMENTO, IMPIEGATO, ACQUISTO e VENDITA sono esempi di entità di un'applicazione aziendale. Una occorrenza di un'entità è un oggetto della classe che l'entità rappresenta. Le città di Roma, Milano e Palermo sono esempi di occorrenze dell'entità CITTÀ, gli impiegati Marini e Ferrari sono invece esempi di occorrenze dell'entità IMPIEGATO. Si osservi che una occorrenza di entità non è un valore che identifica un oggetto (per esempio, il cognome dell'impiegato o il suo codice fiscale) ma è l'oggetto stesso (l'impiegato "in carne e ossa"). Una interessante conseguenza di questo fatto è che una occorrenza di entità ha un'esistenza (e un'identità) indipendente dalle proprietà a esso associate (un impiegato esiste indipendentemente dal fatto di avere un nome, un cognome, una età ecc.). In questo il modello E-R presenta una marcata differenza rispetto al modello relazionale nel quale, come abbiamo visto nel Capitolo 2, non possiamo rappresentare un oggetto senza conoscere alcune sue proprietà (un impiegato viene rappresentato da una tupla contenente il nome, il cognome, l'età ecc.).

In uno schema, ogni entità ha un nome che la identifica univocamente e viene rappresentata graficamente mediante un rettangolo con il nome dell'entità all'interno. La Figura 6.5 riporta alcuni esempi di entità.

Relazioni (o associazioni)[2] Rappresentano legami logici, significativi per l'applicazione di interesse, tra due o più entità. RESIDENZA è un esempio di relazione che può sussistere tra le entità CITTÀ e IMPIEGATO mentre ESAME è un esempio di relazione che può sussistere tra le entità STUDENTE e CORSO. Una occorrenza di relazione è un'ennupla (coppia nel caso più frequente di relazione binaria) costituita da occorrenze di entità, una per ciascuna delle entità coinvolte. La coppia di oggetti composta dall'impiegato Ferrari e dalla città di Bologna, oppure la coppia di oggetti composta dall'impiegato Marini e dalla città di Firenze, sono esempi di occorrenze della relazione RESIDENZA. Esempi di occorrenze della relazione ESAME tra le en-

[2]In questo capitolo verrà usato prevalentemente il termine *relazione*. Negli altri capitoli si userà invece il termine *associazione* in casi di possibile ambiguità (per esempio con le relazioni del modello relazionale).

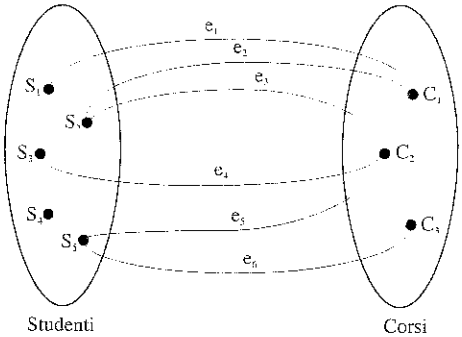

Figura 6.6 Esempio di occorrenze della relazione ESAME.

tità STUDENTE e CORSO sono le coppie e_1, e_2, e_3, e_4, e_5 ed e_6 riportate in Figura 6.6, nella quale vengono raffigurate anche le occorrenze delle entità coinvolte.

In uno schema E-R, ogni relazione ha un nome che la identifica univocamente e viene rappresentata graficamente mediante un rombo, con il nome della relazione all'interno, e da linee che connettono la relazione con ciascuna delle sue componenti. La Figura 6.7 riporta esempi di schema con relazioni tra entità. Si osservi che possono esistere relazioni diverse che coinvolgono le stesse entità, come le relazioni RESIDENZA e SEDE DI LAVORO tra le entità IMPIEGATO e CITTÀ. Nella scelta dei nomi di relazione è preferibile utilizzare sostantivi invece che verbi, in maniera da non indurre ad assegnare un "verso" alla relazione. Per esempio, SEDE DI LAVORO è da preferire a LAVORA IN.

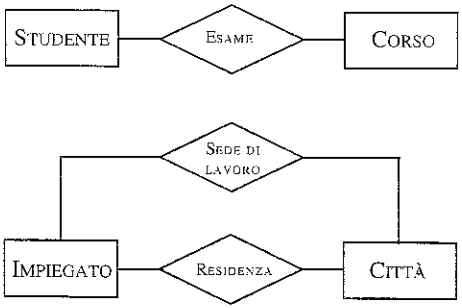

Figura 6.7 Esempi di relazioni tra entità nel modello E-R.

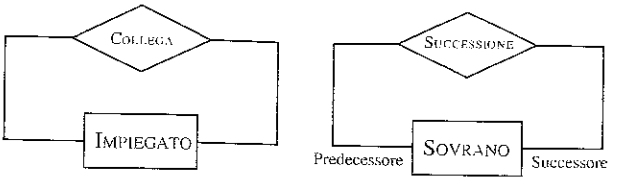

Figura 6.8 Esempi di relazioni ricorsive nel modello E-R.

Un aspetto molto importante delle associazioni è il seguente: come risulta evidente osservando la Figura 6.6, l'insieme delle occorrenze di una relazione del modello E-R è, a tutti gli effetti, una relazione matematica tra le occorrenze delle entità coinvolte, ossia, è un sottoinsieme del loro prodotto cartesiano. Questo significa che tra le occorrenze di una relazione del modello E-R non ci possono essere ennuple ripetute. Questo aspetto ha importanti conseguenze: per esempio, la relazione ESAME in Figura 6.7 non è in grado di descrivere il fatto che un certo studente ha sostenuto più volte lo stesso esame (perché questo produrrebbe ennuple identiche). In tal caso, anche l'esame va rappresentato con una entità collegata mediante relazioni alle entità STUDENTE e CORSO.

È anche possibile avere relazioni *ricorsive*, ovvero relazioni tra una entità e se stessa. Per esempio, in Figura 6.8 la relazione COLLEGA sull'entità IMPIEGATO connette coppie di impiegati che lavorano insieme, mentre la relazione SUCCESSIONE sull'entità SOVRANO associa a ogni sovrano di una dinastia il suo immediato successore. Va osservato che, a differenza della prima relazione, la relazione SUCCESSIONE non è simmetrica. In questo caso è necessario stabilire i due *ruoli* che l'entità coinvolta gioca nella relazione. Questo può essere fatto associando degli identificatori (nel nostro caso Successore e Predecessore) alle linee uscenti dalla relazione ricorsiva.

È possibile infine avere relazioni n-arie, relazioni cioè che coinvolgono più di due entità. Un esempio viene mostrato in Figura 6.9: la relazione FORNITURA tra le tre entità FORNITORE, PRODOTTO e DIPARTIMENTO descrive il fatto che un fornitore rifornisce un dipartimento di un certo prodotto.

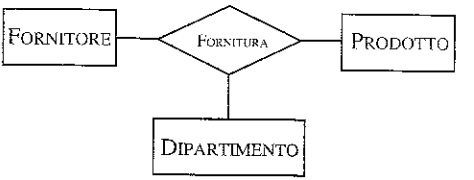

Figura 6.9 Esempio di relazione ternaria nel modello E-R.

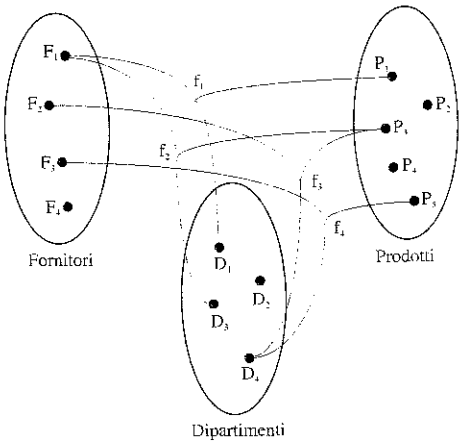

Figura 6.10 Esempio di occorrenze della relazione FORNITURA.

Un possibile insieme di occorrenze di questa relazione potrebbe stabilire che la ditta Pinto fornisce stampanti al dipartimento Vendite e calcolatori al dipartimento Sviluppo, mentre la ditta Sami fornisce calcolatori al dipartimento Ricerca e fotocopiatrici al dipartimento Vendite. Un esempio grafico di possibili occorrenze della relazione FORNITURA è riportato in Figura 6.10 (triple f_1, f_2, f_3 e f_4). Nella figura sono riportate anche le occorrenze delle entità coinvolte.

Attributi Descrivono le proprietà elementari di entità o relazioni che sono di interesse ai fini dell'applicazione. Per esempio, Cognome, Stipendio ed Età sono possibili attributi dell'entità IMPIEGATO, mentre Data e Voto lo sono per la relazione ESAME tra STUDENTE e CORSO. Un attributo associa a ciascuna occorrenza di entità (o di relazione) un valore appartenente a un insieme, detto *dominio*, che contiene i valori ammissibili per l'attributo. Per esempio, l'attributo Cognome dell'entità IMPIEGATO può avere come dominio l'insieme delle stringhe di 20 caratteri, mentre l'attributo Età può avere come dominio gli interi compresi tra 18 e 65. In Figura 6.11 viene mostrato come vengono rappresentati graficamente gli attributi. I domini non vengono riportati nello schema, ma sono generalmente descritti nella documentazione associata.

Può risultare comodo, qualche volta, raggruppare attributi di una medesima entità o relazione che presentano affinità nel loro significato o uso: l'insieme di attributi che si ottiene in questa maniera viene detto *attributo composto*. Possiamo, per esempio, raggruppare gli attributi Via, Numero civico e CAP dell'entità PERSONA per formare l'attributo composto Indirizzo. La rappresentazione grafica di un attributo

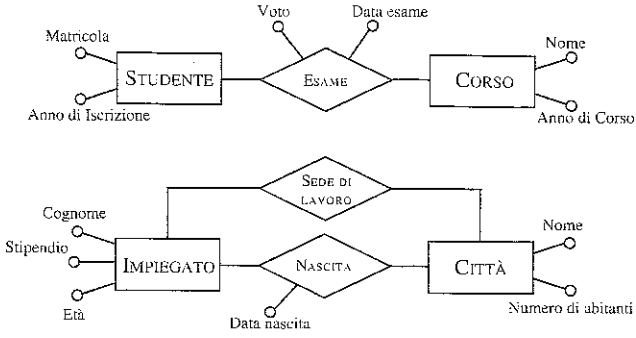

Figura 6.11 Schemi E-R con relazioni, entità e attributi.

composto viene mostrata nell'esempio in Figura 6.12. Per ridurre la complessità degli schemi, gli attributi composti verranno usati raramente nel seguito preferendo usare, per quanto possibile, attributi atomici.

Costruzione di schemi con i costrutti di base I tre costrutti del modello Entità-Relazione visti fino a questo momento ci consentono già di costruire schemi per descrivere realtà di una certa complessità. Si consideri per esempio lo schema E-R riportato in Figura 6.13. Si comprende facilmente che questo schema rappresenta alcune informazioni di carattere organizzativo relative a una azienda con diverse sedi. Partendo dall'entità SEDE e procedendo in senso antiorario si può vedere che una sede dell'azienda è dislocata in una certa città e ha un certo indirizzo (attributi Città e Indirizzo). Ogni sede è organizzata in dipartimenti (relazione COMPOSIZIONE) e ogni dipartimento ha un nome e un numero di telefono (entità DIPARTIMENTO e relativi attributi). A questi dipartimenti afferiscono, a partire da una certa data, gli impiegati dell'azienda (relazione AFFERENZA e relativo attributo) e ci sono impiegati che dirigono tali dipartimenti (relazione DIREZIONE). Per gli impiegati vengono rappresentati il cognome, lo stipendio, l'età e un codice che serve a identificarli (en-

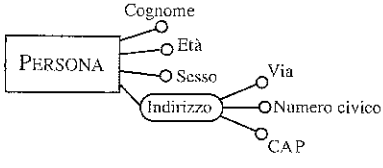

Figura 6.12 Un esempio di entità con attributo composto.

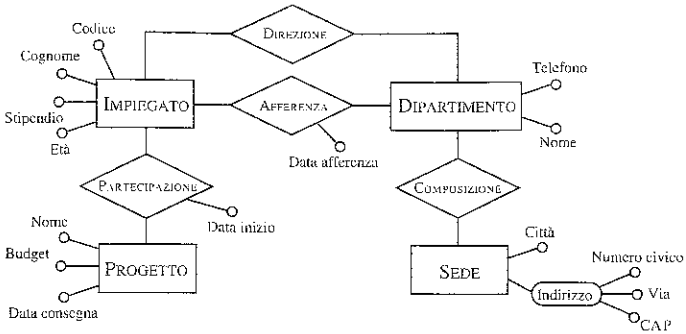

Figura 6.13 Uno schema Entità-Relazione.

tità IMPIEGATO e relativi attributi). Gli impiegati lavorano su progetti a partire da una certa data (relazione PARTECIPAZIONE e relativo attributo). Ogni progetto ha un nome, un budget e una data di consegna (entità PROGETTO e relativi attributi).

6.2.2 Altri costrutti del modello

Esaminiamo ora i rimanenti costrutti del modello E-R: le cardinalità delle relazioni e degli attributi, gli identificatori delle entità e le generalizzazioni. Come vedremo, solo l'ultimo è un costrutto "nuovo"; gli altri costituiscono, in realtà, dei *vincoli di integrità* su costrutti già visti, cioè proprietà che occorrenze di entità e di relazioni devono soddisfare per poter essere considerate "valide".

Cardinalità delle relazioni Vengono specificate per ciascuna partecipazione di entità a una relazione e descrivono il numero minimo e massimo di occorrenze di relazione a cui una occorrenza dell'entità può partecipare. Dicono quindi quante volte, in una relazione tra entità, un'occorrenza di una di queste entità può essere legata a occorrenze delle altre entità coinvolte. Per esempio, se in una relazione ASSEGNAMENTO tra le entità IMPIEGATO e INCARICO specifichiamo per la prima entità una cardinalità minima pari a uno e una cardinalità massima pari a cinque, vogliamo indicare che un impiegato può partecipare a un minimo di una occorrenza e a un massimo di cinque occorrenze della relazione ASSEGNAMENTO. In altre parole, vogliamo dire che, nella nostra applicazione a un impiegato deve essere assegnato almeno un incarico ma non più di cinque. Se per l'entità INCARICO specifichiamo una cardinalità minima pari a zero e una cardinalità massima pari a 50 imponiamo che un certo incarico può partecipare o a nessuna occorrenza oppure a 50 occorrenze al massimo della relazione ASSEGNAMENTO. Quindi, un certo incarico può non essere assegnato a nessun impiegato oppure può essere assegnato a un numero di impiegati

Figura 6.14 Cardinalità di una relazione nel modello E-R.

inferiore o uguale a 50. In uno schema E-R, le cardinalità minima e massima delle partecipazioni di entità a relazioni si specificano tra parentesi, come descritto in Figura 6.14.

In linea di principio è possibile assegnare un qualunque intero non negativo a una cardinalità di una relazione con l'unico vincolo che la cardinalità minima deve essere minore o uguale della cardinalità massima. In realtà, nella maggior parte dei casi, è sufficiente utilizzare solo tre valori: zero, uno e il simbolo N (che indica genericamente un intero maggiore di uno). In particolare:

- per la cardinalità minima, zero o uno; nel primo caso si dice che la partecipazione dell'entità relativa è *opzionale*, nel secondo si dice che la partecipazione è *obbligatoria*;
- per la cardinalità massima, uno o molti (N); nel primo caso la partecipazione dell'entità relativa può essere vista come una funzione (parziale se la cardinalità minima vale zero) che associa a una occorrenza dell'entità una sola occorrenza (o nessuna) dell'altra entità che partecipa alla relazione; nel secondo c'è invece un'associazione con un numero arbitrario di occorrenze dell'altra entità.

Se analizziamo la Figura 6.6 possiamo concludere che l'entità STUDENTE partecipa alla relazione ESAME con cardinalità pari a (0,N), in quanto ci sono studenti che non partecipano a nessuna occorrenza della relazione (lo studente S_4), altri che partecipano a più di una occorrenza della relazione (per esempio lo studente S_2 che partecipa a e_2 ed e_3).

In Figura 6.15 sono riportati diversi casi di cardinalità per relazioni del modello E-R. Per esempio, le cardinalità della relazione RESIDENZA ci dicono che ogni persona può essere residente in una e una sola città, mentre ogni città può non aver residenti oppure ha, in generale, molti residenti.

Osservando le cardinalità massime, è possibile classificare le relazioni binarie in base al tipo di corrispondenza che viene stabilita tra le occorrenze delle entità coinvolte. Le relazioni aventi cardinalità massima pari a uno per entrambe le entità coinvolte, come la relazione VENDITA in Figura 6.15, definiscono una corrispondenza uno a uno tra le occorrenze di tali entità e vengono quindi denominate *relazioni uno a uno*. In maniera analoga, le relazioni aventi un'entità con cardinalità massima pari a uno e l'altra con cardinalità massima pari a N, come la relazione RESIDENZA in Figura 6.15, sono denominate *relazioni uno a molti*. Infine, le relazioni aventi cardinalità massima pari a N per entrambe le entità coinvolte, come la relazione PRENOTAZIONE in Figura 6.15, vengono denominate *relazioni molti a molti*.

Per le cardinalità minime va detto invece che il caso di partecipazione obbligatoria per tutte le entità coinvolte è piuttosto raro, perché quando si aggiunge una

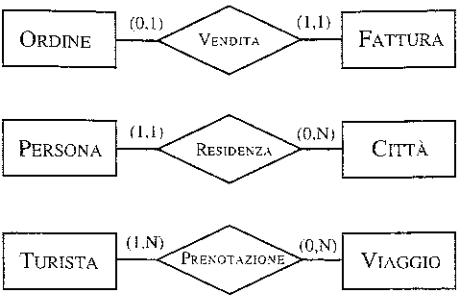

Figura 6.15 Esempi di cardinalità di relazioni.

nuova occorrenza di entità, molto spesso non sono note (o addirittura non esistono) le corrispondenti occorrenze delle entità a essa collegate. Per esempio, relativamente al primo schema in Figura 6.15, quando si riceve un nuovo ordine, non esiste ancora una fattura a esso relativa e non è quindi possibile costruire una occorrenza della relazione VENDITA che contiene il nuovo ordine.

Negli esempi visti fino a questo momento abbiamo fatto riferimento solo a relazioni binarie. C'è da dire in effetti che nelle relazioni n-arie le entità coinvolte partecipano quasi sempre con cardinalità massima pari a N. Un esempio concreto viene fornito dalla relazione ternaria FORNITURA di Figura 6.9: come si può osservare dalla Figura 6.10, esistono esempi di occorrenze di ciascuna delle entità coinvolte (F_1, P_3 e D_4) che partecipano a più occorrenze di tale relazione. Nel caso in cui un'entità partecipa a una relazione n-aria con cardinalità massima pari a uno, significa che ogni sua occorrenza può essere legata a una sola occorrenza della relazione, e quindi a un'unica ennupla di occorrenze delle altre entità coinvolte nella relazione. Questo significa che è possibile (e risulta a volte più naturale) eliminare la relazione n-aria e legare direttamente tale entità con le altre entità, mediante delle relazioni binarie di tipo uno a molti. Riprenderemo questo argomento informalmente nel Paragrafo 7.2.2 e, in maniera più sistematica, nel Capitolo 9 dedicato alla normalizzazione dove forniremo dei criteri di analisi più precisi.

Cardinalità degli attributi Possono essere specificate per gli attributi di entità o relazioni e descrivono il numero minimo e massimo di valori dell'attributo associati a ogni occorrenza di entità o relazione. Nella maggior parte dei casi, la cardinalità di un attributo è pari a (1,1) e viene omessa. In questi casi l'attributo rappresenta sostanzialmente una funzione che associa a ogni occorrenza di entità un solo valore dell'attributo. Il valore per un certo attributo può essere però nullo (con le medesime accezioni introdotte nel Paragrafo 2.2.3 per il modello relazionale), oppure possono esistere diversi valori di un certo attributo per una occorrenza di entità. Queste situazioni si possono rappresentare associando all'attributo in questione una cardinalità

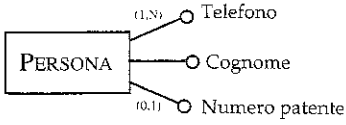

Figura 6.16 Esempio di attributi di entità con cardinalità.

minima pari a zero nel primo caso, e una cardinalità massima pari a molti (N), nel secondo. In Figura 6.16 viene presentato un esempio di entità con attributi dotati di cardinalità. Sulla base delle cardinalità risulta che una persona ha uno e un solo cognome, può avere o non avere un numero di patente, ma se ne ha uno è unico, e ha almeno un recapito telefonico, ma ne può avere, in generale, più di uno.

In maniera simile alle partecipazioni delle occorrenze di entità alle relazioni, diremo che un attributo con cardinalità minima pari a zero è *opzionale* per la relativa entità o relazione, mentre è *obbligatorio* se la cardinalità minima è pari a uno. Diremo infine che un attributo è *multivalore* se la sua cardinalità massima è pari a N. Come discusso nel Capitolo 2, in molte situazioni reali accade che certe informazioni non sono disponibili, ed è quindi utile avere la possibilità di specificare attributi opzionali. Gli attributi multivalore vanno invece utilizzati con maggiore cautela, perché essi rappresentano situazioni che possono essere modellate, in alcune occasioni, con entità a sé, legate da relazioni uno a molti (o molti a molti) con l'entità cui si riferiscono. Per fare un esempio concreto, si potrebbe pensare di aggiungere un attributo multivalore Titolo di studio all'entità PERSONA di Figura 6.16, perché una persona può avere più titoli di studio. Il titolo di studio è però un concetto condiviso da molte persone: può risultare quindi più naturale modellarlo con una entità a parte legata con l'entità PERSONA da una relazione molti a molti. Rimandiamo comunque questo discorso al Paragrafo 7.2, nel quale forniremo dei criteri per la scelta del costrutto più adatto alla rappresentazione di un certo concetto del mondo reale.

Identificatori delle entità Vengono specificati per ciascuna entità di uno schema e descrivono i concetti (attributi e/o entità) dello schema che permettono di identificare in maniera univoca le occorrenze delle entità. In molti casi, uno o più attributi di una entità sono sufficienti a individuare un identificatore: si parla in questo caso di identificatore *interno* (detto anche *chiave*). Per esempio, un identificatore interno per l'entità AUTOMOBILE con attributi Modello, Targa e Colore è l'attributo Targa, in quanto non possono esistere due automobili con la stessa targa e quindi due occorrenze della entità AUTOMOBILE con gli stessi valori sull'attributo Targa. Alla stessa maniera, un identificatore interno per l'entità PERSONA con attributi Nome, Cognome, Indirizzo e Data di Nascita può essere l'insieme degli attributi Nome, Cognome e Data di Nascita, avendo assunto che nella nostra applicazione non esistono due persone aventi, contemporaneamente, lo stesso nome, lo stesso cognome e la stessa data di nascita. In Figura 6.17 viene mostrata la simbologia usata per rap-

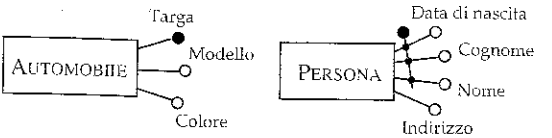

Figura 6.17 Esempi di identificatori interni di entità.

presentare gli identificatori interni in uno schema E-R. Si osservi la diversa notazione usata per indicare identificatori composti da un solo attributo e identificatori composti da più attributi.

Alcune volte però gli attributi di un'entità non sono sufficienti a identificare univocamente le sue occorrenze. Si consideri per esempio l'entità STUDENTE nello schema in Figura 6.18. Può sembrare a prima vista che l'attributo Matricola possa essere un identificatore per tale entità, ma ciò non è vero: lo schema descrive infatti studenti iscritti a varie università e due studenti iscritti a università diverse possono avere lo stesso numero di matricola. In questo caso, per identificare univocamente uno studente serve, oltre al numero di matricola, anche la relativa università. Quindi, un identificatore corretto per l'entità STUDENTE in questo schema è costituito dall'attributo Matricola e dall'entità UNIVERSITÀ. Va osservato che questa identificazione è resa possibile dalla relazione uno a molti tra le entità UNIVERSITÀ e STUDENTE, che associa a ogni studente una e una sola università. Se questa relazione non esistesse, l'identificazione univoca attraverso un'altra entità non sarebbe possibile. Quindi, un'entità E può essere identificata da altre entità solo se tali entità sono coinvolte in una relazione a cui E partecipa con cardinalità (1,1). Nei casi in cui l'identificazione di un'entità è ottenuta utilizzando altre entità si parla di identificatore *esterno*. La rappresentazione diagrammatica di un identificatore esterno è riportata nell'esempio di Figura 6.18.

Sulla base di quanto detto sulle identificazioni, è possibile fare alcune considerazioni generali:

- un identificatore può coinvolgere uno o più attributi, ognuno dei quali deve avere cardinalità (1,1);

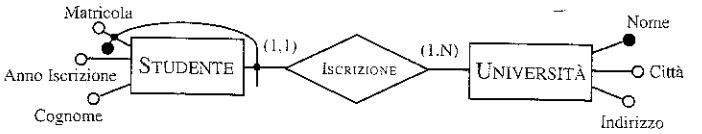

Figura 6.18 Esempio di identificatore esterno di entità.

- un'identificazione esterna può coinvolgere una o più entità, ognuna delle quali deve essere membro di una relazione alla quale l'entità da identificare partecipa con cardinalità (1,1);
- un'identificazione esterna può coinvolgere un'entità che è a sua volta identificata esternamente, purché non vengano generati, in questa maniera, cicli di identificazioni esterne;
- ogni entità deve avere almeno un identificatore (interno o esterno), ma ne può avere in generale più di uno; nel caso di più identificatori, gli attributi e le entità coinvolte in alcune identificazioni (tranne una) possono essere opzionali (cardinalità minima uguale a zero).

Possiamo a questo punto riesaminare lo schema presentato in Figura 6.13 introducendo cardinalità e identificatori. Lo schema risultante viene proposto in Figura 6.19.

Si può osservare che il nome di una città identifica una sede dell'azienda: questo vuol dire che non c'è più di una sede nella stessa città. Un dipartimento è invece identificato dal nome e dalla sede di cui fa parte (dalle cardinalità si evince che una sede ha diversi dipartimenti ma ogni dipartimento fa parte di una sola sede). Un dipartimento ha almeno un numero di telefono, ma può averne più di uno. Un impiegato (identificato da un codice) può afferire a un solo dipartimento (ma può accadere che non afferisca a nessun dipartimento, per esempio se appena assunto) e può dirigere o zero o un dipartimento. Viceversa, ogni dipartimento ha un solo direttore e uno o più impiegati. Sui progetti (identificati univocamente dal loro nome) lavorano diversi impiegati (almeno uno) e ogni impiegato lavora in generale su più progetti (ma può accadere che non lavori a nessun progetto). Infine, la data di scadenza di un progetto può non essere nota.

Generalizzazioni Rappresentano legami logici tra un'entità E, detta entità *genitore*, e una o più entità $E_1, \ldots, E_n$, dette entità *figlie*, di cui E è più generale, nel senso

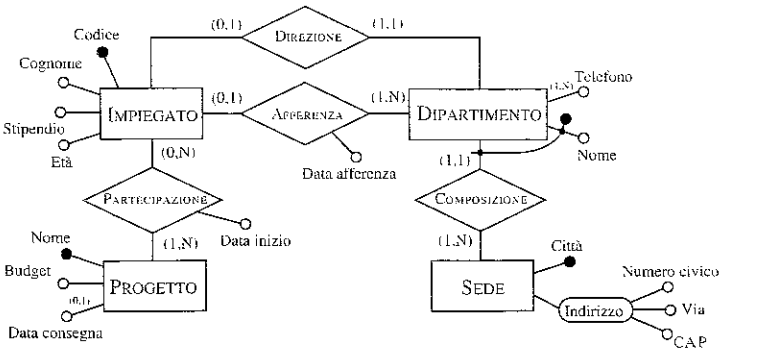

Figura 6.19 Lo schema di Figura 6.13 completato con identificatori e cardinalità.

che le comprende come caso particolare. Si dice in questo caso che E è *generalizzazione* di $E_1, \ldots, E_n$ e che le entità $E_1, \ldots, E_n$ sono *specializzazioni* dell'entità E. Per esempio, l'entità PERSONA è una generalizzazione delle entità UOMO e DONNA, mentre PROFESSIONISTA è una generalizzazione delle entità INGEGNERE, MEDICO e AVVOCATO. Per contro, le entità UOMO e DONNA sono specializzazioni dell'entità PERSONA.

Tra le entità coinvolte in una generalizzazione valgono le seguenti proprietà generali.

- Ogni occorrenza di un'entità figlia è anche una occorrenza dell'entità genitore. Per esempio, una occorrenza dell'entità AVVOCATO è anche una occorrenza dell'entità PROFESSIONISTA.
- Ogni proprietà dell'entità genitore (attributi, identificatori, relazioni e altre generalizzazioni) è anche una proprietà delle entità figlie. Per esempio, se l'entità PERSONA ha attributi Cognome ed Età, anche le entità UOMO e DONNA possiedono questi attributi. Inoltre, l'identificatore di PERSONA è un identificatore valido anche per le entità UOMO e DONNA. Questa proprietà delle generalizzazioni è nota sotto il nome di *ereditarietà*.

Le generalizzazioni vengono rappresentate graficamente mediante delle frecce che congiungono le entità figlie con l'entità genitore, come mostrato negli esempi in Figura 6.20. Si osservi che, per le entità figlie, le proprietà ereditate non vanno rappresentate esplicitamente.

Le generalizzazioni possono essere classificate sulla base di due proprietà tra loro ortogonali.

- Una generalizzazione è *totale* se ogni occorrenza dell'entità genitore è una occorrenza di almeno una delle entità figlie, altrimenti è *parziale*.
- Una generalizzazione è *esclusiva* se ogni occorrenza dell'entità genitore è al più un'occorrenza di una delle entità figlie, altrimenti è *sovrapposta*.

La generalizzazione tra PERSONA, UOMO e DONNA in Figura 6.20 è, per esempio, totale (gli uomini e le donne costituiscono "tutte" le persone) ed esclusiva (una perso-

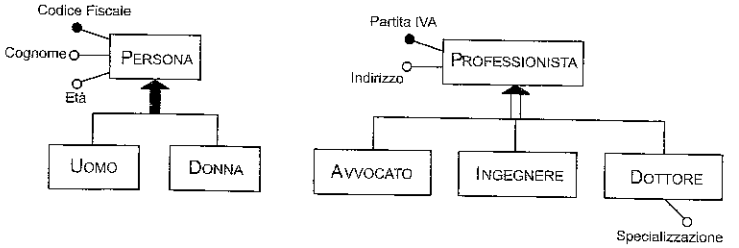

Figura 6.20 Esempi di generalizzazioni tra entità.

na è o uomo o donna). Una generalizzazione tra l'entità PROFESSIONISTA e le entità INGEGNERE e DOTTORE è invece parziale ed esclusiva, perché assumiamo che ciascun professionista abbia una sola professione principale e che vi siano altre professioni oltre a queste tre. Tra l'entità PERSONA e le entità STUDENTE e LAVORATORE esiste infine una generalizzazione parziale e sovrapposta, perché esistono studenti che sono anche lavoratori.

Questo ultimo esempio ci suggerisce che, in realtà, le generalizzazioni sovrapposte possono essere facilmente trasformate in generalizzazioni esclusive aggiungendo una o più entità figlie, per rappresentare i concetti che costituiscono le "intersezioni" delle entità che si sovrappongono. Nel caso degli studenti e dei lavoratori è sufficiente aggiungere l'entità STUDENTELAVORATORE per ottenere una generalizzazione esclusiva. Quindi, sebbene sia importante saper distinguere questi tipi di generalizzazioni, assumeremo nel seguito, senza sostanziale perdita di generalità, che le generalizzazioni sono sempre esclusive. Come vedremo nel prossimo capitolo, questa scelta rende peraltro più semplice la traduzione verso il modello relazionale. Per quanto riguarda invece le generalizzazioni totali, queste vengono in genere rappresentate disegnando la freccia con tratto pieno (si vedano gli esempi in Figura 6.20 e in Figura 6.21). Questa notazione comunque non verrà sempre rispettata, ma se ne farà uso solo quando strettamente necessario.

In generale, una stessa entità può essere coinvolta in più generalizzazioni diverse. Possono esserci inoltre generalizzazioni su più livelli: si parla in questo caso di *gerarchia* di generalizzazioni. Infine, una generalizzazione può avere una sola entità figlia: si parla in questo caso di *sottoinsieme*. In Figura 6.21 viene mostrata una gerarchia di generalizzazioni. Il legame che esiste tra l'entità RESPONSABILE DI PROGETTO e PROGETTISTA è un esempio di sottoinsieme.

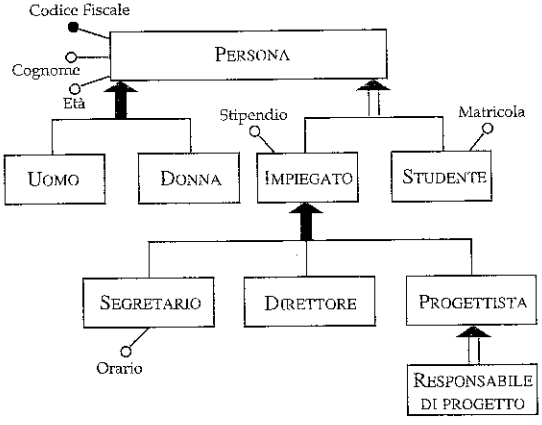

Figura 6.21 Gerarchia di generalizzazioni tra entità.

6.2.3 Panoramica finale sul Modello E-R

Abbiamo visto come il modello Entità-Relazione ci mette a disposizione alcuni strumenti, detti costrutti, per descrivere i dati di una applicazione e rappresentarli in una forma grafica facilmente comprensibile.

Tutti i costrutti del modello E-R visti vengono illustrati nello schema in Figura 6.22 che fornisce al tempo stesso un esempio di schema E-R e una descrizione (semplificata) del modello E-R stesso. Analizziamo ora questo schema e consideriamo questa esplorazione come esercizio di "lettura" di uno schema E-R. Si tratta in effetti di un'attività di cui è bene impratichirsi, perché frequente nell'analisi e nella manutenzione di sistemi informativi esistenti.

Si può osservare che il modello è composto da una serie di costrutti di cui due sono considerati di base: l'entità e la relazione. Un'entità può partecipare a zero o a diverse relazioni, mentre una relazione coinvolge due o più entità. La partecipazione di una entità a una relazione ha una cardinalità minima e una massima (per semplicità qui non consideriamo le cardinalità come costrutti veri e propri ma proprietà della partecipazione di entità a relazioni). Gli altri costrutti del modello sono gli attributi e le generalizzazioni. Un attributo ha un nome, una cardinalità minima e massima, e appartiene a un costrutto di base, cioè a un'entità o a una relazione (per la proprietà delle generalizzazioni infatti, la relazione APPARTENENZA viene ereditata dalle entità figlie). Un sottoinsieme degli attributi sono gli attributi composti, che si compongono di uno o più attributi. Una generalizzazione ha esattamente una entità genitore e una (nel caso di sottoinsiemi) o molte entità figlie. Un'entità può essere genitore e figlia di diverse generalizzazioni (o anche di nessuna). Si osservi infine che un costrutto di base è identificato univocamente dal suo nome (è essenziale infatti non utilizzare in uno schema lo stesso nome per concetti diversi), mentre un attributo è identificato dal suo nome e dal costrutto a cui è associato (come indicato dall'identificazione

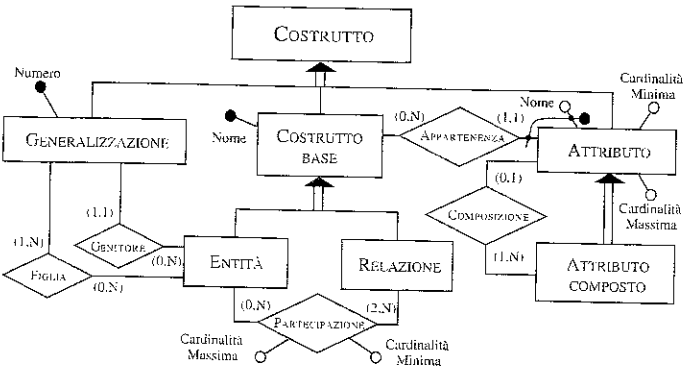

Figura 6.22 Descrizione del modello E-R con il modello E-R.

esterna). Ci possono essere cioè attributi con lo stesso nome ma devono appartenere a relazioni o entità diverse (si veda per esempio l'attributo Nome in Figura 6.19). Le generalizzazioni non possiedono in genere nomi e, per identificarle, assumiamo qui che siano numerate.

Esistono infine altri vincoli sull'uso dei costrutti che non si possono esprimere sullo schema. Per esempio, il fatto che le gerarchie di generalizzazione non possono contenere cicli, oppure il fatto che una cardinalità minima non può essere maggiore della corrispondente cardinalità massima. Il problema della documentazione di vincoli non esprimibili con il modello E-R verrà discusso diffusamente nel prossimo paragrafo.

Concludiamo il paragrafo con una considerazione di natura generale. Abbiamo detto più volte che gli schemi E-R costituiscono utili strumenti nell'attività di progettazione di basi di dati. In realtà tali schemi, fornendo rappresentazioni astratte dei dati di una applicazione, possono essere utilizzati con profitto anche per attività non strettamente legate alla progettazione.

Si possono fare a tale riguardo diversi esempi:

- gli schemi E-R possono essere utilizzati a scopo documentativo, poiché sono facilmente comprensibili anche da non specialisti di basi di dati;
- gli schemi E-R possono essere utilizzati per descrivere i dati di un sistema informativo già esistente (per esempio per integrarlo con altri) e, nel caso di sistema costituito da diversi sottosistemi, c'è il vantaggio di poter rappresentare le varie componenti con un linguaggio astratto e quindi unificante;
- gli schemi E-R possono essere utilizzati per comprendere, in caso di modifica dei requisiti di una applicazione, su quali porzioni del sistema si deve operare e in cosa consistono le modifiche da effettuare.

6.3 Documentazione di schemi E-R

Abbiamo visto come il modello Entità-Relazione fornisca strumenti di modellazione molto espressivi che ci permettono di descrivere, con efficacia e facilità, situazioni anche molto complesse. Uno schema E-R però non è quasi mai sufficiente, da solo, a rappresentare nel dettaglio tutti gli aspetti di un'applicazione, per varie ragioni. Innanzitutto, in uno schema E-R compaiono solo i nomi dei vari concetti in esso presenti ma questo può essere insufficiente per comprenderne il significato. Se riprendiamo per esempio lo schema in Figura 6.19, può risultare non chiaro se l'entità PROGETTO fa riferimento a progetti interni all'azienda oppure a progetti esterni, ai quali l'azienda partecipa. Nel caso di schemi particolarmente complessi può accadere inoltre di non riuscire a rappresentare in maniera comprensibile ed esaustiva i vari concetti. Con riferimento all'esempio di Figura 6.19, sarebbe per esempio difficile rappresentare altri attributi per l'entità IMPIEGATO, senza inficiare la leggibilità dello schema. Risulta infine, in certi casi, addirittura impossibile rappresentare alcune proprietà dei dati attraverso i costrutti che il modello E-R mette a disposizione. Consideriamo per esempio ancora lo schema in Figura 6.19 e supponiamo che nella nostra azienda un impiegato possa essere direttore solo del dipartimento a cui afferisce: questa proprietà non può

essere espressa direttamente sullo schema perchè fa riferimento a due concetti indipendenti (direzione e afferenza) descritti da due relazioni e non esistono costrutti del modello che ci permettono di correlare due relazioni. Un altro esempio di proprietà non esprimibile direttamente da costrutti del modello E-R è il fatto che un impiegato non può avere uno stipendio maggiore del direttore del dipartimento al quale afferisce. Si osservi che queste proprietà corrispondono a vincoli di integrità sui dati. È stato infatti osservato che mentre il modello E-R è sufficientemente espressivo per rappresentare dati, risulta meno adatto a rappresentare vincoli complessi su di essi.

In conclusione, risulta indispensabile corredare ogni schema E-R con una documentazione di supporto, che possa servire a facilitare l'interpretazione dello schema stesso e a descrivere proprietà dei dati rappresentati che non possono essere espressi direttamente dai costrutti del modello. Nei prossimi paragrafi descriveremo quindi strutture e tecniche atte a documentare uno schema E-R. Queste strutture non vanno intese come nuovi costrutti di rappresentazione, ma semplici strumenti, peraltro non formali, atti a completare e arricchire la descrizione dei dati di un'applicazione fatta con un modello concettuale. Vanno quindi considerate come strumenti di supporto all'analisi concettuale ma non possono certamente sostituirsi a essa.

6.3.1 Regole aziendali

Uno degli strumenti più usati dagli analisti di sistemi informativi per la descrizione di proprietà di un'applicazione che non si riesce a rappresentare direttamente con modelli concettuali è quello delle *regole aziendali* o, per usare una più nota terminologia inglese, delle *business rules*. Questa accezione deriva dal fatto che, nella maggior parte dei casi, quello che si vuole esprimere è proprio una "regola" del particolare dominio applicativo che stiamo considerando. Riprendendo l'esempio appena fatto, il fatto che un impiegato non può guadagnare più del proprio direttore costituisce, appunto, una possibile regola dell'azienda.

In effetti il termine regola aziendale viene spesso utilizzato dagli analisti con un'accezione più ampia, per indicare una qualunque informazione che definisce o vincola qualche aspetto di un'applicazione. In particolare, in base a una classificazione piuttosto consolidata, una regola aziendale può essere:

1. la *descrizione di un concetto* rilevante per l'applicazione, ovvero la definizione precisa di un'entità, di un'attributo o di una relazione del modello E-R;
2. un *vincolo di integrità* sui dati dell'applicazione, sia esso la documentazione di un vincolo espresso con qualche costrutto del modello E-R (per esempio le cardinalità di una relazione) o la descrizione di un vincolo non esprimibile direttamente con i costrutti del modello;
3. una *derivazione*, ovvero un concetto che può essere ottenuto, attraverso un'inferenza o un calcolo aritmetico, da altri concetti dello schema (per esempio un attributo Costo il cui valore può essere ottenuto dalla somma degli attributi Costo Netto e Tasse).

Per le regole del primo tipo è chiaramente impossibile definire una sintassi precisa e si fa in genere ricorso a frasi in linguaggio naturale. Come verrà descritto nel paragrafo che segue, queste regole vengono tipicamente rappresentate sotto forma di

glossari, raggruppando le descrizioni in maniera opportuna (per esempio, per entità e per relazione).

Le regole che descrivono vincoli di integrità e derivazioni sono invece più adatte a definizioni formali e, in effetti, sono state proposte nella letteratura sintassi più o meno complesse. Dato però che non esistono standardizzazioni e che ogni formalismo scelto rischia di non essere sufficientemente espressivo, faremo ricorso ancora a definizioni in linguaggio naturale, avendo però cura di strutturare in maniera adeguata tali definizioni.

In particolare, le regole che descrivono vincoli di integrità possono essere espresse sotto forma di *asserzioni*, ovvero affermazioni che devono essere sempre verificate nella nostra base di dati. Per motivi di chiarezza e per favorirne la costruzione, tali affermazioni devono essere "atomiche", non possono cioè essere decomposte in frasi che costituiscono esse stesse delle asserzioni. Inoltre, poiché vengono usate per documentare uno schema E-R, le asserzioni vanno enunciate in maniera dichiarativa, in una forma cioè che non suggerisca un metodo per soddisfarle. Questo è infatti un problema realizzativo e pertanto non pertinente alla rappresentazione concettuale. Quindi notazioni del tipo "*se <condizione> allora <azione>*" non sono adatte a esprimere regole aziendali, quando queste documentano uno schema E-R. Una struttura predefinita per enunciare regole aziendali sotto forma di asserzioni potrebbe essere invece la seguente:

< *concetto* > `deve/non deve` < *espressione su concetti* >

dove i concetti citati possono corrispondere o a concetti che compaiono nello schema E-R a cui si fa riferimento, oppure a concetti derivabili da essi. Per esempio, riprendendo gli esempi già citati per lo schema in Figura 6.19, regole aziendali che esprimono vincoli di integrità possono essere le seguenti (RV sta per regola di vincolo):

(RV1) *il direttore di un dipartimento* `deve` *afferire a tale dipartimento*;
(RV2) *un impiegato* `non deve` *avere uno stipendio maggiore del direttore del dipartimento al quale afferisce*;
(RV3) *un dipartimento con sede a Roma* `deve` *essere diretto da un impiegato con più di dieci anni di anzianità*.

Si noti come concetti quali "direttore di dipartimento" e "impiegato con più di dieci anni di anzianità" non sono rappresentati direttamente sullo schema, ma possono comunque essere derivati da esso.

Consideriamo ora le regole aziendali che esprimono derivazioni. Queste regole possono essere espresse specificando le operazioni (aritmetiche o di altro genere) che permettono di ottenere il concetto derivato. Una possibile struttura è quindi:

< *concetto* > `si ottiene` < *operazione su concetti* >

Per esempio, se nello schema in Figura 6.19 l'entità DIPARTIMENTO avesse un attributo Numero Impiegati, ci potrebbe essere una regola del tipo:

(RD1) *il numero degli impiegati di un dipartimento* `si ottiene` *contando gli impiegati che vi afferiscono*.

dove RD sta per regola di derivazione.

Abbiamo detto che le regole aziendali costituiscono una forma di documentazione di uno schema concettuale. Quando lo schema concettuale viene tradotto in una base di dati (fasi di progettazione logica e fisica) le regole aziendali non descrittive (quelle cioè che esprimono vincoli o derivazioni) vanno ovviamente codificate per garantire la consistenza dei dati rispetto alle proprietà che esse rappresentano. A tale riguardo, possiamo dire che per implementare le regole aziendali è possibile seguire diversi approcci:

- fare uso di clausole del linguaggio SQL all'atto della definizione dello schema logico di una base di dati, mediante vincoli predefiniti, vincoli generici o asserzioni (come descritto nel Capitolo 4);
- mediante triggers o *regole attive*, una tecnologia che viene ampiamente descritta nel secondo volume;
- con opportune procedure scritte in qualche linguaggio di programmazione.

6.3.2 Tecniche di documentazione

Abbiamo detto che uno schema E-R va corredato con una documentazione di supporto, per facilitare l'interpretazione dello schema stesso e per descrivere proprietà dei dati che non possono essere espresse direttamente dai costrutti del modello. Abbiamo visto inoltre che questa documentazione può essere espressa in termini di regole aziendali. Vediamo ora in quale forma è possibile produrre questa documentazione, facendo riferimento a un caso concreto.

La documentazione dei vari concetti rappresentati in uno schema, ovvero le regole aziendali di tipo descrittivo, può essere prodotta facendo uso di un *dizionario dei dati*. Esso è composto da due tabelle: la prima descrive le entità dello schema con il nome, una definizione informale in linguaggio naturale, l'elenco di tutti gli attributi (con eventuali descrizioni associate) e i possibili identificatori. L'altra tabella descrive le relazioni con il nome, una loro descrizione informale, l'elenco degli attributi (con eventuali descrizioni) e l'elenco delle entità coinvolte insieme alla loro cardinalità di partecipazione. Un esempio di dizionario dei dati per lo schema E-R in Figura 6.19 è riportato in Figura 6.23. Si osservi come il dizionario possa servire a documentare con semplicità anche alcuni vincoli sui dati e quindi altre forme di regole aziendali. Come già accennato, l'uso del dizionario dei dati è particolarmente importante nei casi in cui lo schema è complesso (molti concetti collegati in maniera articolata) e risulta pesante specificare direttamente sullo schema tutti gli attributi di entità e relazioni.

Per quel che riguarda le altre regole aziendali, si può far ricorso ancora a una tabella, nella quale vengono elencate le varie regole, specificando di volta in volta la loro tipologia. Tali regole possono essere espresse secondo le modalità suggerite nel paragrafo precedente, possibilmente facendo esplicito riferimento ai concetti dello schema. Ricordiamo che risulta importante rappresentare tutte le regole che descrivono vincoli non espressi dallo schema, ma risulta a volte utile rappresentare anche regole che documentano vincoli già espressi nello schema.

Un esempio di documentazione di questo tipo per lo schema in Figura 6.19 viene riportata in Figura 6.24.

Entità	Descrizione	Attributi	Identificatore
Impiegato	Impiegato che lavora nell'azienda.	Codice, Cognome, Stipendio, Età	Codice
Progetto	Progetti aziendali sui quali lavorano gli impiegati.	Nome, Budget, Data consegna	Nome
Dipartimento	Dipartimenti delle sedi dell'azienda.	Telefono, Nome	Nome, Sede
Sede	Sede dell'azienda in una certa città.	Città, Indirizzo (Numero, Via e CAP)	Città

Relazione	Descrizione	Entità Coinvolte	Attributi
Direzione	Associa un dipartimento al suo direttore.	Impiegato (0,1), Dipartimento (1,1)	
Afferenza	Associa un impiegato al suo dipartimento.	Impiegato (0,1), Dipartimento (1,N)	Data afferenza
Partecipazione	Associa agli impiegati i progetti sui quali lavorano.	Impiegato (0,N), Progetto (1,N)	Data inizio
Composizione	Associa una sede ai dipartimenti di cui è composta.	Dipartimento (1,1), Sede (1,N)	

Figura 6.23 Il dizionario dei dati per lo schema in Figura 6.19.

Regole di vincolo
(RV1) Il direttore di un dipartimento deve afferire a tale dipartimento.
(RV2) Un impiegato non deve avere uno stipendio maggiore del direttore del dipartimento al quale afferisce.
(RV3) Un dipartimento con sede a Roma deve essere diretto da un impiegato con più di dieci anni di anzianità.
(RV4) Un impiegato che non afferisce a nessun dipartimento non deve partecipare a nessun progetto.
Regole di derivazione
(RD1) Il budget di un progetto si ottiene moltiplicando per 3 la somma degli stipendi degli impiegati che vi partecipano.

Figura 6.24 Regole aziendali per lo schema in Figura 6.19.

6.4 Modellazione dei dati in UML

UML (Unified Modeling Language) è un linguaggio grafico per la modellazione di applicazioni software basate sulla programmazione orientata agli oggetti che, negli ultimi anni, si è rapidamente affermato nell'ambito dell'ingegneria del software. Si tratta di un formalismo molto ricco che consente di rappresentare attraverso una serie di diagrammi tutti gli aspetti di un'applicazione software: dati, operazioni, processi e architetture.

Per il suo successo, UML viene talvolta utilizzato, in alternativa al modello Entità-Relazione, per la rappresentazione concettuale di una base di dati. In particolare vengono utilizzati, a questo scopo, i *diagrammi delle classi* che descrivono le classi di oggetti di interesse per l'applicazione e le relazioni che intercorrono tra di esse. In effetti, molti costrutti del modello Entità-Relazione sono riconducibili a nozioni usate nei diagrammi delle classi. Inoltre, in base al principio di *incapsulamento* della programmazione orientata agli oggetti che prevede una stretta correlazione tra dati e operazioni, con questi diagrammi è possibile rappresentare oltre agli aspetti "strutturali" dell'applicazione, cioè i dati sui quali opera, anche quelli "comportamentali", ovvero le procedure associate ai dati.

L'uso di un diagramma delle classi UML per rappresentare una base di dati permette di collocarsi in un contesto metodologico più ampio, nel quale possiamo descrivere aspetti dell'applicazione che il modello Entità-Relazione, da solo, non consente di rappresentare. D'altro canto, va detto che alcuni costrutti del modello Entità-Relazione che sono rilevanti nella modellazione di una base di dati (per esempio gli identificatori esterni) non sono previsti in UML. Si adottano in questi casi delle notazioni non standard che richiedono quindi il preventivo accordo dei progettisti sull'interpretazione dei simboli usati. Ne consegue che la progettazione di una base di dati con UML è possibile ma può presentare alcune difficoltà.

Tutto questo non deve sorprendere in quanto il modello E-R è stato appositamente ideato per la modellazione concettuale dei dati e i costrutti che presenta sono funzionali a questo uso. Il diagramma delle classi UML è stato invece ideato per il progetto di una applicazione software, in cui le classi sono viste più come meccanismi per organizzare procedure che come contenitori di dati. Deve inoltre restare chiaro che, indipendentemente dalla notazione grafica adottata, non bisogna confondere il diagramma delle classi di un'applicazione con lo schema concettuale della base di dati dell'applicazione stessa. Si tratta infatti di componenti diverse del progetto complessivo che sono certamente correlate ma che ma non possono, per loro natura, coincidere.

Vista comunque la frequente adozione dei diagrammi delle classi UML anche per rappresentare schemi concettuali di dati, vedremo nei prossimi paragrafi come questo sia possibile, rimandando a testi di ingegneria del software l'approfondimento dell'uso del linguaggio per altri scopi.

Resta inteso che l'uso di UML cambia la notazione diagrammatica ma non l'approccio alla progettazione di una base di dati: la metodologia introdotta nel Paragrafo 6.1.2 (e approfondita nei prossimi capitoli) rimane quella di riferimento.

6.4.1 Panoramica su UML

UML è stato proposto a metà degli anni Novanta con l'intento di unificare alcuni formalismi preesistenti per la modellazione orientata agli oggetti. Successivamente, tale linguaggio è stato standardizzato sotto l'egida dell'Object Management Group (OMG), un consorzio industriale non-profit che si occupa di favorire l'interoperabilità tra applicazioni software sviluppate da aziende diverse attraverso la definizione di specifiche concordate dai suoi membri. Oggi, UML si è universalmente imposto come linguaggio di riferimento per la modellazione e la documentazione di applicazioni software.

UML offre diversi tipi di diagrammi che, corredati da una opportuna descrizione testuale della loro semantica, servono a rappresentare i molteplici aspetti di un'applicazione software o, per usare una terminologia dell'ingegneria del software, "viste" secondo prospettive diverse della medesima applicazione. Questi diagrammi compongono quello che viene denominato il *modello dell'applicazione*. Va subito chiarito che questa terminologia è diversa da quella adottata nel mondo delle basi di dati, nel quale il concetto corrispondente, cioè la descrizione dell'organizzazione di una base di dati, è detto *schema*. Il *modello di dati* adottato per la rappresentazione dello schema della base di dati (Paragrafo 1.3) corrisponde invece al concetto di *metamodello* nell'ambito dell'ingegneria del software. UML viene infatti considerato un metamodello per la descrizione di modelli di applicazioni software. Per non generare confusione tra tutti questi concetti, nel seguito utilizzeremo solo il termine, piuttosto intuitivo, di *diagramma*.

Nella versione corrente, UML prevede i seguenti diagrammi principali:

- il diagramma delle classi, che illustra le caratteristiche statiche e dinamiche delle componenti (dette appunto *classi*) di un'applicazione software e le relazioni (dette in UML *associazioni*) intercorrenti tra di esse;
- il diagramma degli oggetti, che fornisce una rappresentazione delle possibili istanze delle classi (gli *oggetti*) e dei collegamenti tra di esse;
- il diagramma dei casi d'uso, che descrive le modalità di utilizzo del sistema da parte degli *attori* (persone o sistemi che interagiscono con esso) e l'interazione tra attori e sistema;
- il diagramma di sequenza, che descrive l'ordinamento temporale di messaggi (invocazione di procedure dette *metodi* in UML) scambiati tra i diversi oggetti dell'applicazione;
- il diagramma di comunicazione (detto anche di collaborazione), che, come quello di sequenza, descrive lo scambio di messaggi tra gli oggetti, ma con una notazione e una prospettiva diverse;
- il diagramma delle attività, che descrive il comportamento dinamico di un processo che fa parte dell'applicazione attraverso flussi di attività da svolgere;
- il diagramma degli stati, che illustra il ciclo di vita di un oggetto dell'applicazione attraverso gli stati che esso può assumere;
- il diagramma dei componenti, che rappresenta come le componenti fisiche del sistema (file, eseguibili, librerie, moduli) sono organizzate e quali sono le loro dipendenze;

- il diagramma di distribuzione dei componenti, che illustra la dislocazione dei nodi hardware del sistema e delle associazioni esistenti tra di essi.

Nel prossimo paragrafo porremo l'attenzione sui diagrammi delle classi che, nati per descrivere le classi di oggetti che compongono un'applicazione software, si prestano, con opportuni accorgimenti, anche alla descrizione dello schema concettuale di una base di dati.

6.4.2 Rappresentazione di dati con i diagrammi delle classi

I diagrammi delle classi UML offrono un formalismo molto ricco per descrivere i molteplici aspetti delle componenti di un'applicazione software basata su oggetti. Senza pretendere di analizzare nel dettaglio questo formalismo, ci soffermiamo sui costrutti che si possono usare per descrivere, a livello concettuale, una base di dati.

Classi Sono le componenti principali dei diagrammi delle classi e corrispondono in buona sostanza alle entità del modello E-R. Come si intuisce dai semplici esempi riportati in Figura 6.25, una classe viene rappresentata in UML da un rettangolo contenente, in alto, il nome della classe e, al suo interno, gli attributi a essa associati.

C'è da aggiungere che, a differenza del modello E-R, per una classe è possibile specificare nel riquadro in basso (lasciato vuoto in figura) del rettangolo anche i relativi *metodi*, ovvero le operazioni ammissibili su oggetti della classe secondo il già citato principio di incapsulamento della programmazione orientata agli oggetti. Tale principio suggerisce di descrivere i dati *insieme* alle operazioni da svolgere su di essi. Per esempio, si potrebbe associare alla classe `Impiegato` in Figura 6.25 il metodo `SetStipendio()`, che assegna a un impiegato un certo stipendio. Non è invece possibile definire attributi composti.

In UML è possibile associare agli attributi i rispettivi domini (interi, reali, stringhe ecc.) e diverse altre proprietà, alcune delle quali verranno menzionate più avanti (molteplicità e vincoli). Le altre non sono significative nella modellazione concettuale dei dati e non verranno perciò approfondite. Citiamo solo il fatto che, nei diagrammi delle classi, i simboli +, –, #, che spesso precedono il nome di un attributo o di un metodo, indicano la loro *visibilità*, ovvero se possono essere acceduti o meno da oggetti di altre classi. Questo aspetto non è però rilevante in una rappresentazione concettuale di dati.

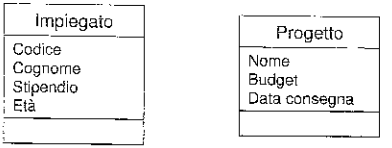

Figura 6.25 Rappresentazione di classi in UML.

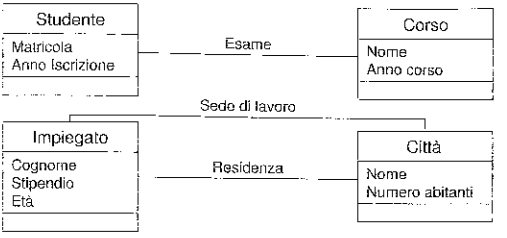

Figura 6.26 Associazioni binarie in UML.

Associazioni Corrispondono alle relazioni del modello E-R e vengono rappresentate come indicato negli esempi in Figura 6.26.

Si può osservare che le associazioni binarie si rappresentano con semplici linee che congiungono le classi coinvolte. Il nome della relazione viene generalmente posto sulla linea, ma questo non è obbligatorio perché in UML possono esistere associazioni senza nome.

Come per le relazioni del modello E-R, si possono definire più associazioni tra le medesime classi (si vedano per esempio le due associazioni esistenti tra le classi `Impiegato` e `Città` in Figura 6.26) ed è anche possibile associare ruoli alle classi coinvolte in un'associazione. Non è invece possibile assegnare attributi alle associazioni. Per far questo, si fa uso delle cosiddette *classi di associazione* che descrivono proprietà di un'associazione e vengono collegate, mediante una linea tratteggiata, all'associazione da descrivere. La classe `Esame` in Figura 6.27 è un esempio di classe di associazione che usiamo per rappresentare gli attributi `Voto` e `Data` dell'associazione tra la classe `Studente` e la classe `Corso`. Si osservi che in questo caso non è necessario assegnare un nome all'associazione.

Finora abbiamo visto solo esempi di associazioni binarie. Se l'associazione è n-aria, si adotta la stessa notazione grafica del modello E-R: l'associazione viene rappresentata da un rombo e da linee che congiungono il rombo con le classi che partecipano all'associazione. Un esempio viene proposto in Figura 6.28 nella qua-

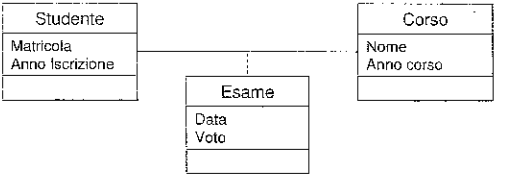

Figura 6.27 Una classe di associazione in UML.

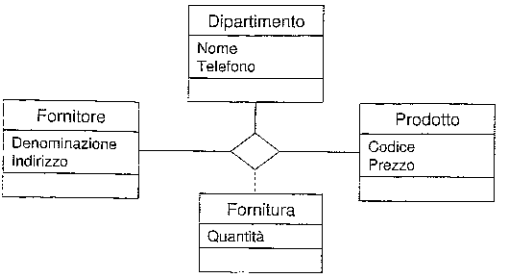

Figura 6.28 Una associazione ternaria in UML.

le si rappresenta una relazione ternaria tra le classi `Fornitore`, `Prodotto` e `Dipartimento`. Anche in questo caso, facciamo uso di una classe di associazione per assegnare attributi all'associazione tra queste classi.

C'è da dire che le associazioni n-arie si usano molto di rado nei diagrammi delle classi e, quando si incontrano, viene sempre suggerito di *reificarle*[3], ovvero di trasformare l'associazione in una classe legata alle classi originarie con associazioni binarie. Per esempio, la reificazione dell'associazione in Figura 6.28 produce lo schema riportato in Figura 6.29. Nel seguito della trattazione, considereremo quindi solo associazioni binarie.

Per le associazioni è possibile specificare una serie di proprietà, non tutte rilevanti nella progettazione concettuale dei dati. Per esempio, si può indicare con una freccia un verso privilegiato di *navigabilità* di un'associazione. Uno strumento interessante è invece la possibilità di specificare associazioni che sono *aggregazioni* di concetti, associazioni cioè che definiscono una relazione tra un concetto composito

[3]Dal latino *res* "cosa", cioè far diventare una cosa, un oggetto.

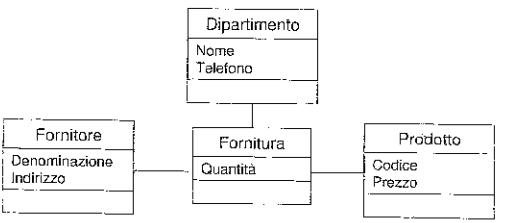

Figura 6.29 Una associazione ternaria reificata in UML.

e uno o più concetti che ne costituiscono una sua parte. Tali associazioni si indicano in UML con una linea avente un rombo attaccato alla classe che rappresenta il concetto "aggregante"; dall'altro capo della linea c'è una classe che costituisce una sua "parte". Esempi di associazioni di questo tipo sono quelle tra `Team` e `Tecnico` e tra `Azienda` e `Filiale` nei diagramma delle classi in Figura 6.30. La prima ci dice che un tecnico fa parte di un team, la seconda che una filiale è parte di un'azienda.

Il rombo si lascia in bianco se un oggetto della classe "parte" può esistere senza dover appartenere a un oggetto della classe "aggregante", altrimenti viene annerito e l'aggregazione viene chiamata *composizione*. Negli esempi in Figura 6.30 si assume che un tecnico può essere rappresentato indipendentemente dal team di cui fa parte (aggregazione semplice), mentre una filale non può essere rappresentata senza specificare l'azienda di cui fa parte (composizione).

Molteplicità Con un diagramma delle classi è possibile specificare alcuni importanti vincoli di integrità sui dati. In particolare, è possibile indicare le cardinalità di partecipazione (qui denominate *molteplicità*) delle classi alle associazioni, secondo le medesime modalità delle cardinalità del modello E-R, ovvero come coppia di valori che specificano la cardinalità minima e massima di partecipazione di un oggetto della classe all'associazione. Le convenzioni adottate nei due formalismi sono però diverse. Innanzitutto la cardinalità minima viene separata dalla massima non da una virgola ma da due punti (per esempio, una possibile cardinalità è 0..1). Inoltre, la cardinalità "molti" viene rappresentata dal simbolo $*$. Quando si specifica solo $*$ si intende $0..*$, ovvero $(0, N)$, mentre con un semplice 1 si denota la coppia di cardinalità 1..1. Quest'ultima cardinalità viene considerata quella di *default* per le classi, con l'unica eccezione che, nelle aggregazioni, la cardinalità di default per la classe "aggregante" è $*$ (cioè $0..*$). Le cardinalità di default possono essere omesse nel diagramma. La differenza però più importante a livello di notazione è che, in un'associazione binaria, le cardinalità di partecipazione minima e massima di una classe non vengono riportate accanto alla classe stessa, ma accanto all'altra classe che partecipa all'associazione. In altre parole, rispetto a uno schema E-R, le cardinalità delle associazioni binarie nei diagrammi delle classi risultano invertite. Esempi di uso di molteplicità

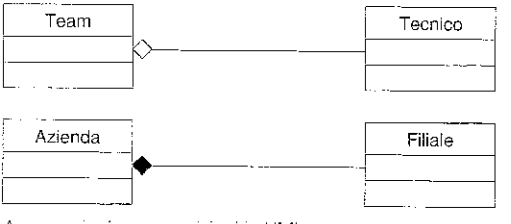

Figura 6.30 Aggregazioni e composizioni in UML.

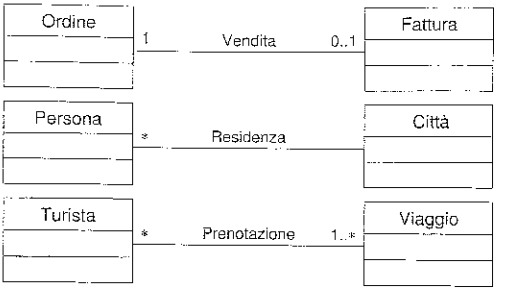

Figura 6.31 Associazioni con molteplicità in UML.

in associazioni vengono riportate in Figura 6.31, nella quale sono ripresi gli esempi discussi a pagina 206 e rappresentati nel modello E-R in Figura 6.15.

In questa figura la molteplicità 0..1 nell'associazione `Vendita` tra le classi `Ordine` e `Fattura` indica che un ordine può avere una fattura associata o nessuna, mentre la molteplicità 1 indica che una fattura ha uno e un solo ordine associato. L'assenza di molteplicità nell'associazione `Residenza` tra `Persona` e `Città`, dalla parte della classe `Città`, sottintende 1..1, cioé il fatto che una persona è residente esattamente in una città. La molteplicità * indica invece che ogni città ha molti residenti ma può non averne nessuno. Si verifichi confrontando la Figura 6.31 con la Figura 6.15 come le molteplicità sono collocate in posizione invertita rispetto alla corrispondente notazione adottata nel modello E-R.

Usando la medesima sintassi, è possibile associare molteplicità anche agli attributi delle classi.

Identificatori In UML non esiste una notazione per esprimere identificatori di classi. Questo in realtà non deve sorprendere perché, secondo il paradigma di orientazione agli oggetti, ogni oggetto è dotato implicitamente di un identificatore (detto, appunto, identificatore di oggetto) che ne consente l'identificazione univoca e non ha quindi bisogno di identificazioni esplicite. Siccome però gli identificatori sono indispensabili nel modello relazionale, nella modellazione di dati tesa alla realizzazione di una base di dati relazionale è utile denotare attributi che possono essere usati per questo scopo. Non esiste una notazione standard, ma una soluzione ragionevole consiste nel far uso di un costrutto chiamato *vincolo utente*. In UML si possono definire vincoli d'integrità su associazioni e su attributi specificandoli tra parentesi graffe vicino all'elemento oggetto del vincolo. Esistono una serie di vincoli predefiniti, nessuno dei quali però riconducibile al concetto di identificatore, per i motivi sopra citati. È però possibile definire liberamente vincoli propri (detti, appunto, vincoli utente). Abbiamo usato questo strumento nei diagrammi in Figura 6.32 dove un identificatore costituito da un solo attributo viene specificato con il vincolo utente `{id}`, mentre

Figura 6.32 Identificatori in UML.

un identificatore costituito da più attributi viene denotato associando questo vincolo a tutti gli attributi che lo compongono (quindi `{id}` denota la partecipazione di un attributo a un identificatore). Ovviamente con queste notazioni è possibile specificare un solo identificatore per classe.

Per quel che riguarda invece le identificazioni esterne, dato che sintatticamente non è possibile usare un vincolo, è pratica comune ricorrere a uno *stereotipo*. Gli stereotipi si usano in UML per estendere i costrutti base quando si vuole modellare un concetto ma non riusciamo a farlo con gli elementi base del linguaggio. In genere gli stereotipi fanno riferimento a qualche elemento base di UML dal quale si possono ottenere per estensione. È però possibile definire anche stereotipi personalizzati. Gli stereotipi vengono indicati da un nome racchiuso tra i simboli << e >>. Nel diagramma UML in Figura 6.33 è stato usato lo stereotipo `<<identificante>>` per indicare che l'associazione tra `Studente` e `Università` è, appunto, identificante in quanto insieme all'attributo `Matricola` identifica uno studente. È possibile confrontare questo schema con l'analogo schema E-R riportato in Figura 6.18.

Si osservi che in questo caso non ci sono ambiguità sulla classe da identificare. Nel caso ci fossero, grazie alla flessibilità del concetto di stereotipo, sarà sufficiente aggiungere un chiarimento nel nome associato allo stereotipo.

Ribadiamo il fatto che le soluzioni suggerite per gli identificatori non sono standardizzate ed è quindi possibile incontrare notazioni diverse. Per esempio, in alcuni strumenti CASE gli identificatori interni vengono specificati tramite un vincolo denominato `{PK}` (Primary Key).

Generalizzazioni Esiste in UML la possibilità di definire generalizzazioni, con modalità molto simili a quelle del modello E-R. Per esempio, la Figura 6.34 ripro-

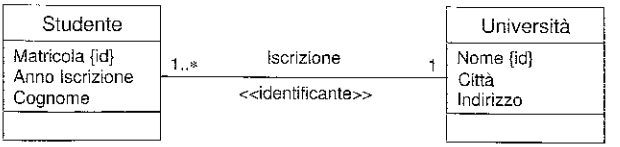

Figura 6.33 Identificatore esterno in UML.

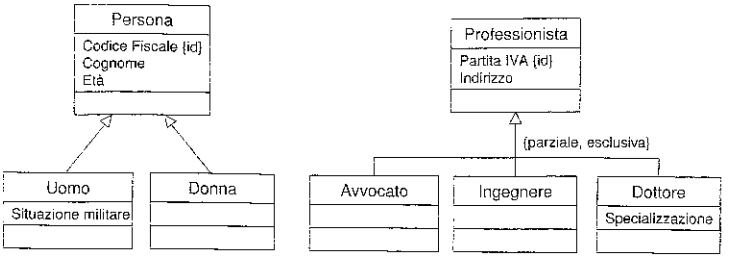

Figura 6.34 Generalizzazioni in UML.

pone, in termini di UML, gli schemi presentati in Figura 6.20. Come viene mostrato nell'esempio di sinistra, le classi figlie della generalizzazione vengono tipicamente collegate con frecce separate alla classe genitore, ma è ammesso unire tali linee come nell'esempio di destra, come avviene nelle generalizzazioni del modello E-R.

Eventuali proprietà delle generalizzazioni possono essere rappresentate con vincoli, usando la medesima sintassi descritta per gli attributi. In particolare, come avviene nell'esempio in Figura 6.34, possiamo indicare se la generalizzazione è totale oppure parziale e se è esclusiva oppure sovrapposta.

Per concludere, citiamo l'interessante possibilità di documentare un diagramma UML con l'uso di *note*, che consistono in semplici commenti testuali. Le note vengono riportate sul diagramma stesso in un rettangolo con l'angolo superiore destro ripiegato. È possibile associare una nota a un particolare elemento del diagramma legandola a esso con una linea tratteggiata, oppure a nessuno in particolare.

Come esempio finale, in Figura 6.35 viene riproposto, sotto forma di diagramma delle classi UML, lo schema E-R in Figura 6.19.

Il diagramma descrive informazioni di carattere organizzativo relative a una azienda con diverse sedi. In base a quanto è stato detto sui diagrammi delle classi si può osservare che una sede dell'azienda (rappresentata dalla classe `Sede`) è identificata dalla città ed è composta da una serie di dipartimenti (associazione `Composizione`) che non possono essere definiti al di fuori di una sede (simbolo di composizione). Ogni dipartimento è identificato dal nome e dalla sede di appartenenza (tramite un identificatore esterno) e possiede diversi numeri di telefono (come indicato dalla molteplicità associata all'attributo `Telefono`). A questi dipartimenti afferiscono, a partire da una certa data, uno o più impiegati (classe di associazione `Afferenza` e relativa molteplicità) e un impiegato li dirige (associazione `Direzione` e relativa molteplicità). Per gli impiegati vengono rappresentati il cognome, lo stipendio, l'età e un codice che serve a identificarli (classe `Impiegato` e relativi attributi). Gli impiegati lavorano su zero o più progetti a partire da una certa data (classe di associazione `Partecipazione` e relativo attributo). Ogni progetto ha un nome, un budget e una data di consegna che può essere non specificata (classe `Progetto` e relativi attributi e molteplicità). Una nota associata alla classe `Progetto` ne descrive il significato.

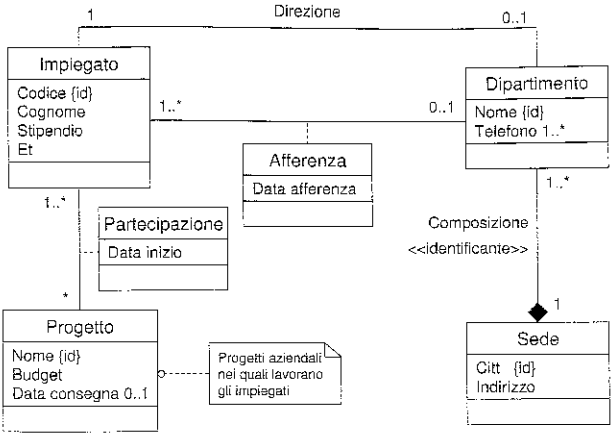

Figura 6.35 Il diagramma delle classi UML..

Da questa breve presentazione dovrebbe essere chiaro come i diagrammi delle classi UML, pensati per un uso diverso, possano essere, pur con qualche difficoltà, adattati alla descrizione del progetto concettuale dei dati. Ciò può essere particolarmente utile quando si dispone solo di strumenti di progetto CASE basati su UML o quando si vuole realizzare una stretta integrazione tra la descrizione concettuale della base di dati e il progetto delle classi della propria applicazione. Citiamo infine il fatto che UML viene talvolta usato anche per descrivere schemi logici di basi di dati. Un esempio pratico di rappresentazione di uno schema relazionale in UML verrà illustrato nel prossimo Paragrafo 8.6.

Note bibliografiche

Esistono molti libri sull'ingegneria del software che descrivono in maniera dettagliata tutte le fasi dello sviluppo di un sistema informativo. Tra questi citiamo quello di Ghezzi *et al.* [47], quello di Pressman [65] e quello di Sommerville [73].

L'organizzazione del processo di progettazione di una base di dati in quattro fasi (analisi dei requisiti, progettazione concettuale, progettazione logica e progettazione fisica) è stata proposta da Lum *et al.* [54] come risultato di un workshop tenuto nel 1979. Un trattamento dettagliato della progettazione concettuale e logica è offerto dal libro in italiano di Batini *et al.* [9] e da quello in inglese di Batini, Ceri e Navathe [8]. Altre letture interessanti sono i testi di Mannila e Raiha [58], Teorey [75] e Wiederhold [85]. Alcuni di questi libri includono un descrizione dettagliata del modello Entità-Relazione. La progettazione di basi di dati viene discussa anche nei testi di ElMasri e Navathe [41] e di Ramakrishnan [66].

Il modello Entità-Relazione è di solito attribuito a Chen [28], che nel 1976 presentò una versione semplificata rispetto a quella presentata in questo capitolo, riprendendo e sistematizzando concetti già discussi nella letteratura. Successivamente sono state proposte diverse estensioni di questo modello, le più importanti delle quali sono state incluse nella nostra trattazione. Il costrutto di generalizzazione è stato introdotto da Smith e Smith [72]. Un libro che presenta, oltre al modello E-R, altri modelli di dati è quello di Tsichritzis e Lochovski [78]. Infine, in un articolo di rassegna molto interessante, Hull e King hanno confrontato diversi modelli concettuali [49].

Le regole aziendali sono trattate diffusamente nel libro di Fleming e von Halle [42].

Esistono infine molti libri che trattano diffusamente di UML. In particolare esiste una collana della Addison-Wesley curata dagli ideatori di UML (Grady Booch, Ivar Jacobson e Jim Rumbaugh) che raccoglie i riferimenti principali su UML e sulla metodologia di sviluppo a esso associata (il cosiddetto "Processo Unificato" o RUP). Tra i testi di questa collana citiamo la guida utente di UML degli stessi Booch, Jacobson e Rumbaugh [14]. Un libro più snello che introduce gli aspetti essenziali di UML è quello di Fowler [43]. Infine, un ottimo testo sull'uso di UML nella progettazione di applicazioni software è quello di Larman [55].

Esercizi

Soluzioni sul sito http://www.ceriasoftware.it/atzeni

6.1 Considerare lo schema E-R in Figura 6.36: lo schema rappresenta varie proprietà di uomini e donne.

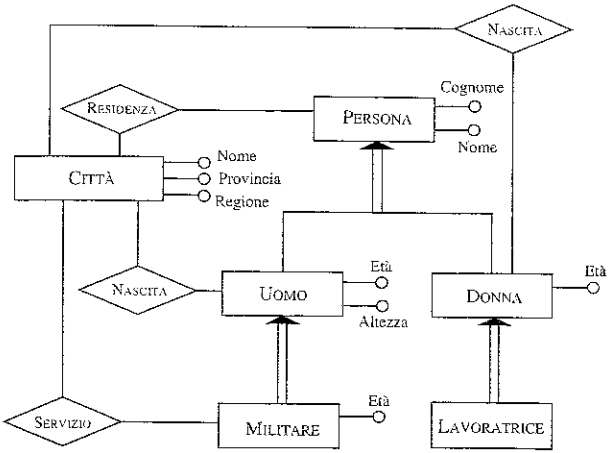

Figura 6.36 Schema E-R per l'Esercizio 6.1.

- Correggere lo schema tenendo conto delle proprietà fondamentali delle generalizzazioni.
- Lo schema rappresenta solo le lavoratrici donne; modificare lo schema rappresentando ora tutti i lavoratori, uomini e donne.
- Tra le proprietà delle città, l'attributo Regione può essere visto anche come un attributo del concetto PROVINCIA. Ristrutturare lo schema in tal senso.

6.2 Aggiungere le cardinalità minime e massime allo schema prodotto nell'Esercizio 6.1 e gli identificatori principali. Dire se esistono dei vincoli di integrità sullo schema che non possono essere espressi con il modello Entità-Relazione.

6.3 Rappresentare le seguenti realtà utilizzando i costrutti del modello Entità-Relazione e introducendo solo le informazioni specificate.

- In un giardino zoologico ci sono degli animali appartenenti a una specie e aventi una certa età; ogni specie è localizzata in un settore (avente un nome) dello zoo.
- Una agenzia di noleggio di autovetture ha un parco macchine, ognuna delle quali ha una targa, un colore e fa parte di un categoria; per ogni categoria c'è una tariffa di noleggio.
- Una casa discografica produce dischi aventi un codice e un titolo; ogni disco è inciso da uno o più cantanti, ognuno dei quali ha un nome e un indirizzo; qualche cantante ha un nome d'arte.

6.4 Completare i frammenti di schema prodotti nell'esercizio precedente con ulteriori informazioni, basandosi sulle proprie conoscenze o facendo delle ipotesi sulle rispettive realtà di interesse.

6.5 Rappresentare le seguenti classi di oggetti facendo uso, dove opportuno, del costrutto di generalizzazione del modello Entità-Relazione. Indicare, nei vari casi, gli attributi delle varie entità e il tipo di generalizzazione, risolvendo i casi di sovrapposizione.

- Gli impiegati di una azienda si dividono in dirigenti, programmatori, analisti, capi progetto e segretari. Ci sono analisti che sono anche programmatori. I capi progetto devono essere dirigenti. Gli impiegati hanno un codice, un nome e un cognome. Ogni categoria di impiegato ha un proprio stipendio base. Ogni impiegato, tranne i dirigenti, ha un orario di lavoro.
- Una compagnia aerea offre voli che possiedono un numero che identifica la tratta (per esempio, Roma–Milano), una data (25 marzo 2010), un orario di partenza (ore 8:00) e uno di arrivo (ore 9:00), un aeroporto di partenza e uno di destinazione. Ci sono voli nazionali e internazionali. I voli internazionali possono avere uno o più scali. Dei voli passati è di interesse l'orario reale di partenza e di arrivo (per esempio, con riferimento al volo suddetto, ore 8:05 e 9:07), di quelli futuri è di interesse il numero di posti disponibili.
- Una casa automobilistica produce veicoli che possono essere automobili, motocicli, camion e trattori. I veicoli sono identificati da un numero di telaio e hanno un nome (per esempio, Punto), una cilindrata e un colore. Le automobili si suddividono in utilitarie (lunghezza sotto i due metri e mezzo) e familiari (lunghezza sopra i due metri e mezzo). Vengono anche classificate in base alla cilindrata: piccola (fino a 1200 cc), media (da 1200 cc a 2000 cc) e grossa cilindrata (sopra i 2000 cc). I motocicli si suddividono in motorini (cilindrata sotto i 125 cc) e moto (cilindrata sopra i 125 cc). I camion hanno un peso e possono avere un rimorchio.

6.6 Si consideri lo schema Entità-Relazione in Figura 6.37. Descrivere le informazioni che esso rappresenta utilizzando il linguaggio naturale.

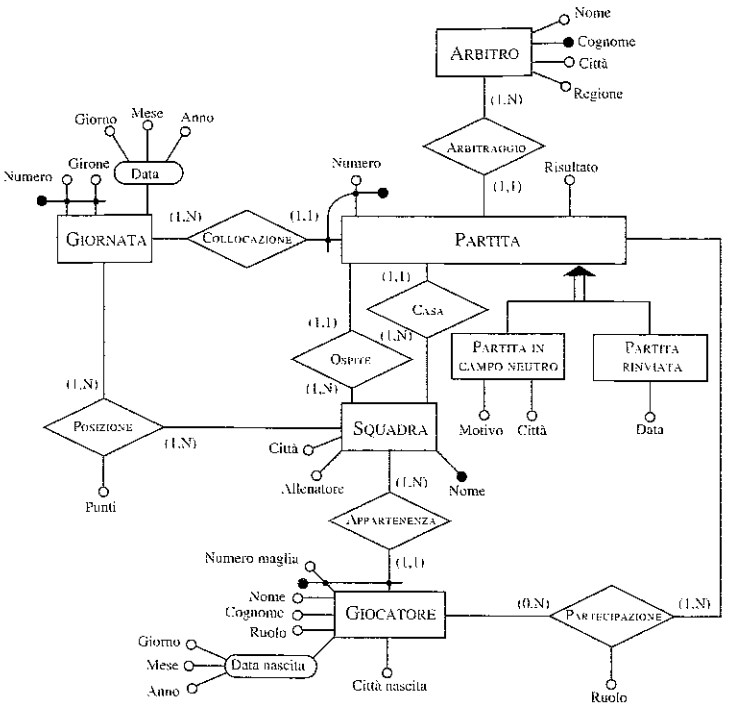

Figura 6.37 Schema E-R per l'Esercizio 6.6.

6.7 Tradurre in regole aziendali le seguenti proprietà sui concetti dello schema di Figura 6.37.

- in una squadra non ci possono essere più di 5 giocatori che giocano nello stesso ruolo;
- una squadra guadagna 3 punti se vince, 1 se pareggia, 0 se perde;
- se una squadra gioca in casa una partita, allora è ospite nella partita successiva.

Produrre quindi una documentazione completa per tale schema.

6.8 Modificare lo schema Entità-Relazione in Figura 6.37 in maniera da descrivere anche i rapporti passati tra giocatori e squadre con data di inizio e fine del rapporto e il ruolo principale ricoperto da ogni giocatore in ogni squadra. È possibile che un giocatore abbia diversi rapporti con la stessa squadra in periodi diversi. Per i rapporti in corso si vuole conoscere la data di inizio.

6.9 In ciascuno dei seguenti casi, si fa riferimento a due o più entità definite in uno schema Entità-Relazione e a un concetto che le coinvolge. Specificare i relativi frammenti di schema, definendo i costrutti (una o più relazioni e, se necessario, ulteriori entità con il relativo identificatore) necessari a rappresentare il concetto, mantenendo le entità indicate e introducendo solo gli attributi richiesti esplicitamente.

- Entità: sport, nazione e superficie. Concetto: il fatto che uno sport si pratichi in una nazione su una certa superficie (per esempio, il tennis si gioca sull'erba in Inghilterra e in Australia, sulla terra rossa in Italia e in Francia, sul sintetico in USA, Italia e Francia; il calcio sull'erba in Italia, sul sintetico e sull'erba in USA, sull'erba in Inghilterra).
- Entità: studioso e dipartimento. Concetto: il fatto che lo studioso abbia tenuto seminari presso il dipartimento. Per ogni seminario è necessario rappresentare data, ora e titolo, con il vincolo che uno studioso non possa tenere più seminari nello stesso giorno.
- Entità: professionista e azienda. Concetto: il fatto che il professionista abbia svolto consulenze per l'azienda. È necessario rappresentare il numero di consulenze effettuate dal professionista per ciascuna azienda, con il relativo costo totale.

6.10 Si consideri una relazione ternaria che coinvolge le seguenti entità: IMPIEGATO, PROGETTO e CONSULENTE. Indicare in quali dei seguenti casi (e, in caso affermativo, come) è opportuno sostituire a tale relazione due (o tre) relazioni binarie.

1. Ogni impiegato è coinvolto in zero o più progetti e interagisce con zero o più consulenti. Ogni consulente è coinvolto in zero o più progetti e interagisce con zero o più impiegati. Ogni progetto coinvolge uno o più impiegati e uno o più consulenti (che possono non interagire fra loro). Un impiegato e un consulente collaborano nell'ambito di un progetto se e solo se essi collaborano fra loro e sono entrambi coinvolti nel progetto.
2. Ogni impiegato è coinvolto in zero o più progetti, in ciascuno dei quali interagisce con uno o più consulenti (che possono essere diversi da progetto a progetto e che possono in generale essere un sottoinsieme dei consulenti coinvolti nel progetto). Ogni consulente è coinvolto in zero o più progetti, in ciascuno dei quali interagisce con uno o più impiegati (che possono essere diversi da progetto a progetto e che possono in generale essere un sottoinsieme degli impiegati coinvolti nel progetto). Ogni progetto coinvolge una o più coppie impiegato-consulente.
3. Ogni impiegato è coinvolto in zero o più progetti. Ogni consulente è coinvolto in zero o più progetti. Ogni progetto coinvolge uno o più impiegati e uno o più consulenti. Un impiegato e un consulente interagiscono se e solo se esiste almeno un progetto in cui siano entrambi coinvolti.

6.11 Modificare lo schema in Figura 6.38 (decomponendo la relazione e aggiungendo ulteriori entità, se necessario; indicare le cardinalità delle relazioni e eventuali necessità di identificatori esterni) tenendo conto delle seguenti specifiche:

- per ogni materia possono esistere più corsi, tenuti dallo stesso professore o da professori diversi;
- ogni corso è relativo a una e una sola materia;
- ogni professore tiene zero o più corsi;
- ogni corso ha uno e un solo professore ed è offerto a uno e un solo corso di studio;
- per ogni corso di studio esiste al più un corso di una data materia;
- tutti i corsi di una data materia hanno lo stesso libro di testo (uno e uno solo).

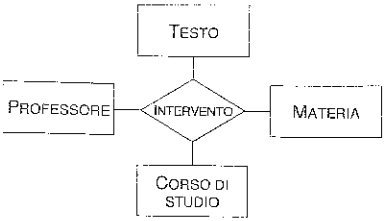

Figura 6.38 Schema per l'Esercizio 6.11.

6.12 Considerare ancora lo schema in Figura 6.38 e modificarlo (decomponendo la relazioni e aggiungendo ulteriori entità, se necessario; indicare le cardinalità delle relazioni e eventuali necessità di identificatori esterni) sulla base delle seguenti specifiche:

- per ogni materia possono esistere più corsi, tenuti dallo stesso professore o da professori diversi;
- ogni corso è relativo a una e una sola materia;

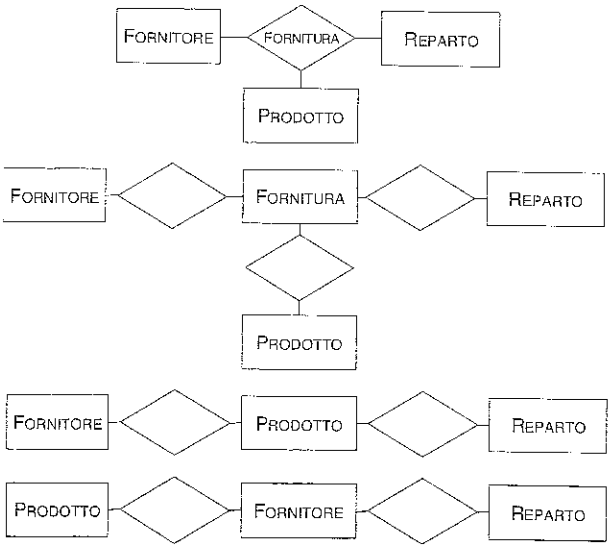

Figura 6.39 Schemi per l'Esercizio 6.13.

- ogni professore tiene zero o più corsi;
- ogni corso ha uno o più professori ed è offerto a uno e un solo corso di studio;
- per ogni corso di studio esiste al più un corso di una data materia;
- ogni corso ha uno e un solo libro di testo; i corsi di una data materia non hanno necessariamente lo stesso libro di testo.

6.13 Considerare gli schemi della Figura 6.39 e le seguenti specifiche. Individuare, per ciascuna specifica, lo schema che meglio la descrive, precisando le cardinalità delle relazioni e gli eventuali identificatori esterni delle entità, che potrebbero includere anche attributi.

1. Interessano le singole forniture di prodotti ai reparti, avvenute in date specifiche; per ogni data, c'è al più una fornitura di un certo prodotto a un certo reparto, con un solo fornitore (però in date diverse ci potrebbero essere altre forniture, di altri fornitori).
2. Interessano le singole forniture, avvenute in date specifiche; per ogni data, c'è al più una fornitura di un certo fornitore a un certo reparto, con un insieme di prodotti (specifico per quella data, e quindi potenzialmente diverso in altre date).
3. Ogni reparto utilizza un certo insieme di prodotti, ognuno dei quali ha uno e un solo fornitore e può essere utilizzato da più reparti.
4. Ogni reparto ha un insieme di fornitori e utilizza un insieme di prodotti; in generale, un fornitore potrebbe fornire alcuni prodotti a un reparto e altri prodotti ad altri reparto; un prodotto può essere fornito da più fornitori e utilizzato da diversi reparti.
5. Ogni fornitore dispone di un insieme di prodotti e può rifornire zero o più reparti; ogni reparto ha un insieme di fornitori e da ciascuno di essi può ricevere tutti i prodotti di cui esso dispone; ogni prodotto ha un solo fornitore.

6.14 Rappresentare lo schema Entità-Relazione in Figura 6.37 con un diagramma della classi UML.

La progettazione concettuale

La progettazione concettuale di una base di dati consiste nella costruzione di uno schema Entità-Relazione in grado di descrivere al meglio le specifiche sui dati di un'applicazione. Anche nel caso di applicazioni non particolarmente complesse, lo schema che si ottiene può contenere molti concetti correlati in una maniera piuttosto complicata. Ne consegue che la costruzione dello schema finale è, necessariamente, un processo graduale: lo schema concettuale viene progressivamente raffinato e arricchito attraverso una serie di trasformazioni ed eventuali correzioni. In questo capitolo verranno descritte le strategie che è possibile seguire in questo processo di sviluppo di uno schema concettuale.

Prima di iniziare a parlare di queste strategie, vale però la pena di spendere qualche parola sull'attività che precede la progettazione vera e propria: la raccolta e l'analisi dei requisiti. Questa fase infatti non è completamente separata da quella della progettazione, ma procede, in molti casi, parallelamente a essa. Possiamo infatti iniziare a costruire uno schema E-R quando non abbiamo ancora terminato di raccogliere e analizzare tutti i requisiti, per poi arricchirlo progressivamente man mano che le informazioni in nostro possesso aumentano.

Dopo aver discusso la fase di raccolta e analisi dei requisiti, presenteremo alcuni criteri di carattere generale per tradurre specifiche informali in concetti del modello Entità-Relazione. Successivamente, illustreremo le principali strategie di progettazione per poi analizzare le qualità che uno schema concettuale ben progettato deve possedere. Chiuderemo questo capitolo cercando di stabilire una metodologia generale di progettazione che tenga conto di tutti gli aspetti illustrati. Per spiegare meglio i vari concetti, faremo riferimento durante tutto il capitolo a un esempio applicativo, relativo alla progettazione di un'applicazione per la gestione dei dati di una società di formazione.

7.1 La raccolta e l'analisi dei requisiti

Va detto innanzitutto che il reperimento e l'analisi dei requisiti di un'applicazione sono attività difficilmente standardizzabili perché dipendono molto dall'applicazione con cui si ha a che fare. Vogliamo però parlare di alcune regole pratiche che è conveniente seguire in questa fase di sviluppo di una base di dati.

Per *raccolta dei requisiti* si intende la completa individuazione dei problemi che l'applicazione da realizzare deve risolvere e le caratteristiche che tale applicazione dovrà avere. Per caratteristiche del sistema si intendono sia gli aspetti statici (i dati) sia gli aspetti dinamici (le operazioni sui dati). I requisiti vengono inizialmente raccolti in specifiche espresse generalmente in linguaggio naturale e, per questo motivo, spesso ambigue e disorganizzate. L'*analisi dei requisiti* consiste nel chiarimento e nell'organizzazione delle specifiche dei requisiti. Si tratta ovviamente di attività fortemente interconnesse: l'attività di analisi inizia con i primi requisiti ottenuti per poi

procedere di pari passo con l'attività di raccolta. In molti casi è l'attività stessa di analisi dei requisiti che suggerisce successive attività di raccolta.

I requisiti di un'applicazione provengono, nella maggior parte dei casi, da fonti diverse. Le principali fonti di informazione sono, in genere, le seguenti.

- Gli *utenti dell'applicazione*. In questo caso le informazioni si acquisiscono mediante opportune interviste, anche ripetute, oppure attraverso una documentazione scritta che gli utenti possono aver predisposto appositamente per questo scopo.
- Tutta la *documentazione esistente* che ha qualche attinenza con il problema allo studio: moduli, regolamenti interni, procedure aziendali, normative. È richiesta, in questo caso, un'attività di raccolta e selezione che viene assistita dagli utenti, ma è a carico del progettista.
- Eventuali *realizzazioni preesistenti*, ovvero applicazioni che si devono rimpiazzare o che devono interagire in qualche maniera con il sistema da realizzare. La conoscenza delle caratteristiche di questi pacchetti software (tracciati record, maschere, algoritmi, documentazione associata) può fornirci importanti informazioni anche in relazione ai problemi esistenti che è necessario risolvere.

Risulta chiaro che, nella fase di acquisizione delle specifiche, gioca un importante ruolo l'interazione con gli utenti del sistema informativo. Durante questa interazione, può avvenire che utenti diversi forniscano informazioni diverse, spesso complementari ma qualche volta contraddittorie. In genere gli utenti a livello più alto possiedono una visione più ampia, ma meno dettagliata. Possono però indirizzare verso gli esperti dei singoli sottoproblemi.

Come criterio generale da seguire possiamo dire che, nel corso delle interviste, è opportuno effettuare con l'utente verifiche di comprensione e consistenza sulle informazioni che si stanno raccogliendo. Questo può essere fatto attraverso esempi (generali e relativi a casi limite) oppure richiedendo definizioni e classificazioni precise. È inoltre molto importante in questa fase cercare di individuare gli aspetti essenziali rispetto a quelli marginali e procedere per raffinamenti successivi. Partendo quindi dai principali aspetti del problema allo studio, dei quali si ha inizialmente una conoscenza solo parziale, si procede cercando di acquisire via via maggiori dettagli.

Come abbiamo già accennato, la specifica dei requisiti raccolti avviene spesso, almeno in prima battuta, facendo uso di descrizioni in linguaggio naturale. Sappiamo bene però che il linguaggio naturale è fonte di ambiguità e fraintendimenti. È molto importante quindi effettuare una profonda analisi del testo che descrive le specifiche per filtrare le eventuali inesattezze e i termini ambigui presenti. Per fissare alcune regole pratiche da seguire in questa attività faremo riferimento a un semplice esempio. Supponiamo di dover progettare una base di dati per una società di formazione e di aver raccolto, sulla base di alcune interviste fatte al personale di questa società, le specifiche dei dati espresse in linguaggio naturale riportate in Figura 7.1. Si noti che abbiamo acquisito in questa fase anche informazioni sul carico previsto dei dati a regime.

È facile rendersi conto che tale testo presenta un certo numero di ambiguità e imprecisioni. Per esempio si utilizzano i termini *partecipante* e *studente* per indicare lo stesso concetto. La stessa cosa accade per i termini *docente* e *professore* e per i termini *corso* e *seminario*.

Società di formazione

1 *Si vuole realizzare una base di dati per una società che eroga corsi, di*
2 *cui vogliamo rappresentare i dati dei partecipanti ai corsi e dei docenti.*
3 *Per i partecipanti (circa 5000), identificati da un codice, si vuole memo-*
4 *rizzare il codice fiscale, il cognome, l'età, il sesso, il luogo di nascita,*
5 *il nome dei loro attuali datori di lavoro, i posti dove hanno lavorato in*
6 *precedenza insieme al periodo, l'indirizzo e il numero di telefono, i corsi*
7 *che hanno frequentato (i corsi sono in tutto circa 200) e il giudizio finale.*
8 *Rappresentiamo anche i seminari che stanno attualmente frequentando*
9 *e, per ogni giorno, i luoghi e le ore dove sono tenute le lezioni. I corsi*
10 *hanno un codice, un titolo e possono avere varie edizioni con date di*
11 *inizio e fine e numero di partecipanti. Se gli studenti sono liberi pro-*
12 *fessionisti, vogliamo conoscere l'area di interesse e, se lo possiedono,*
13 *il titolo. Per quelli che lavorano alle dipendenze di altri, vogliamo co-*
14 *noscere invece il loro livello e la posizione ricoperta. Per gli insegnanti*
15 *(circa 300), rappresentiamo il cognome, l'età, il posto dove sono nati, il*
16 *nome del corso che insegnano, quelli che hanno insegnato nel passato e*
17 *quelli che possono insegnare. Rappresentiamo anche tutti i loro recapi-*
18 *ti telefonici. I docenti possono essere dipendenti interni della società o*
19 *collaboratori esterni.*

Figura 7.1 Esempio di requisiti espressi in linguaggio naturale.

Proviamo a fissare alcune regole generali per ottenere una specifica dei requisiti più precisa e senza ambiguità.

- **Scegliere il corretto livello di astrazione**. È bene evitare di utilizzare termini troppo generici o troppo specifici che rendono poco chiaro un concetto. Per esempio, nel nostro caso sono stati utilizzati i termini *titolo* (a riga 13), con riferimento ai partecipanti che sono liberi professionisti (che tra l'altro è utilizzato anche per indicare un concetto diverso a riga 10) e *giudizio* (riga 7), con riferimento alla valutazione dei corsi, che andrebbero specificati meglio (per esempio, come *titolo professionale* e *votazione in decimi*).
- **Standardizzare la struttura delle frasi**. Nella specifica di requisiti è preferibile utilizzare sempre lo stesso stile sintattico. Per esempio, "*per* <dato> *rappresentiamo* <insieme di proprietà>".
- **Evitare frasi contorte**. Le definizioni devono essere semplici e chiare. Per esempio, *lavoratori dipendenti* (o più semplicemente dipendenti) è da preferire a *quelli che lavorano alle dipendenze di altri* (riga 13).
- **Individuare sinonimi/omonimi e unificare i termini**. I *sinonimi* indicano termini diversi con lo stesso significato (per esempio, *docente* a riga 2 e *insegnante* a riga 14, oppure *partecipante* a riga 2 e *studente* a riga 11); gli *omonimi* indicano

termini uguali con diversi significati (per esempio *posto*, riferito a impiego a riga 5 e a città a riga 15, e *luogo*, riferito a città a riga 4 e ad aula a riga 9). Queste situazioni possono generare ambiguità e vanno chiarite: nel caso di sinonimi unificando i termini, nel caso di omonimi utilizzando termini diversi o specificandoli meglio.

- **Rendere esplicito il riferimento tra termini**. Può succedere che l'assenza di un contesto di riferimento renda alcuni concetti ambigui: in questi casi bisogna esplicitare il riferimento tra termini. Per esempio, nella riga 6, non è chiaro se i termini *indirizzo* e *numero di telefono* sono relativi ai partecipanti o ai loro datori di lavoro; inoltre a riga 13, nella frase *Per quelli che lavorano...*, si deve chiarire esplicitamente a chi ci stiamo riferendo (partecipanti, docenti?) per evitare confusione.
- **Costruire un glossario dei termini**. È molto utile, per la comprensione e la precisazione dei termini usati, definire un glossario che, per ogni termine, contenga: una breve descrizione, possibili sinonimi e altri termini contenuti nel glossario con i quali esiste un legame logico. Un breve glossario per la nostra applicazione è riportato in Figura 7.2.

Dopo aver individuato le varie ambiguità e le imprecisioni, esse vanno eliminate sostituendo i termini non corretti con termini più adeguati. In caso di dubbio, è necessario intervistare nuovamente colui che ha fornito il dato o consultare la documentazione relativa.

Vediamo quali sono le principali modifiche da apportare al nostro testo. Come già detto, *luogo* di nascita dei partecipanti (riga 4) è un omonimo del luogo in cui si tengono le lezioni e va sostituito da *città* di nascita, così come *posto* (riga 5) che va sostituito con *datore di lavoro*. Va poi chiarito che a riga 6 l'*indirizzo* e il *numero di*

Termine	Descrizione	Sinonimi	Collegamenti
Partecipante	Partecipante ai corsi. Può essere un dipendente o un professionista.	Studente	Corso, Datore
Docente	Docente dei corsi. Possono essere collaboratori esterni.	Insegnante	Corso
Corso	Corsi offerti. Possono avere varie edizioni.	Seminario	Docente, Partecipante
Datore	Datori di lavoro attuali e passati dei partecipanti ai corsi.	Posto	Partecipante

Figura 7.2 Un esempio di glossario dei termini.

telefono fanno riferimento ai datori di lavoro dei partecipanti. Il *giudizio* (riga 7) deve essere interpretato come *votazione in decimi*, mentre periodo (riga 6) va interpretato come *date di inizio e fine rapporto*. Bisogna inoltre specificare che i partecipanti frequentano o hanno frequentato specifiche *edizioni* di corsi. Per quanto riguarda gli altri termini che fanno riferimento ai corsi: *seminario* (riga 8) è un sinonimo e va sostituito da *edizione di corso*, *giorno* (riga 9), riferito alle lezioni, è troppo astratto, e va utilizzato *giorno della settimana* mentre *luogo* (riga 9) è un omonimo, che va sostituito da *aula*. Il termine *studente* (riga 11) va sostituito con *partecipante*. Per *titolo* (riga 13) di un partecipante che è libero professionista si intende il suo *titolo professionale*. Per quello che riguarda i docenti abbiamo che *insegnante* (riga 14) è sinonimo di *docente*, *posto* (riga 15) indica la *città* di nascita, il *nome* del corso che insegnano (riga 16) è un sinonimo di *titolo* del corso e il *recapito telefonico* (righe 17-18) è sinonimo di *numero di telefono*.

A questo punto possiamo riscrivere le nostre specifiche apportando le modifiche proposte. È molto utile, in questa fase, decomporre il testo in gruppi di frasi omogenee, relative cioè agli stessi concetti. Otteniamo così la strutturazione delle specifiche sui dati riportata in Figura 7.3.

Naturalmente, accanto alle specifiche sui dati vanno raccolte le specifiche sulle operazioni da effettuare su questi dati. Bisogna cercare di impiegare la medesima terminologia usata per i dati (possiamo per questo far riferimento al glossario dei termini) e informarci anche sulla frequenza con la quale le varie operazioni vengono eseguite. Come vedremo, la conoscenza di questa informazione sarà determinante nella fase di progettazione logica. Per la nostra applicazione, le operazioni sui dati potrebbero essere le seguenti.

Operazione 1: inserisci un nuovo partecipante indicando tutti i suoi dati (operazione da effettuare in media 40 volte al giorno).
Operazione 2: assegna un partecipante a una edizione di corso (circa 50 volte al giorno).
Operazione 3: inserisci un nuovo docente indicando tutti i suoi dati e i corsi che può insegnare (2 volte al giorno).
Operazione 4: assegna un docente abilitato a una edizione di un corso (15 volte al giorno);
Operazione 5: stampa tutte le informazioni sulle edizioni passate di un corso con titolo, orari lezioni e numero partecipanti (10 volte al giorno);
Operazione 6: stampa tutti i corsi offerti, con informazioni sui docenti che possono insegnarli (20 volte al giorno);
Operazione 7: per ogni docente, trova i partecipanti a tutti i corsi da lui/lei insegnati (5 volte a settimana);
Operazione 8: effettua una statistica su tutti i partecipanti a un corso con tutte le informazioni su di essi, sull'edizione alla quale hanno partecipato e sulla rispettiva votazione (10 volte al mese).

Dopo questa strutturazione dei requisiti, siamo pronti ad avviare la prima fase della progettazione che consiste nella costruzione di uno schema concettuale in grado di descrivere in maniera adeguata tutte le specifiche dei dati raccolte.

Frasi di carattere generale
Si vuole realizzare una base di dati per una società che eroga corsi, di cui vogliamo rappresentare i dati dei partecipanti ai corsi e dei docenti.

Frasi relative ai partecipanti
Per i partecipanti *(circa 5000), identificati da un codice, rappresentiamo il codice fiscale, il cognome, l'età, il sesso, la città di nascita, i nomi dei loro attuali datori di lavoro e di quelli precedenti (insieme alle date di inizio e fine rapporto), le edizioni dei corsi che stanno attualmente frequentando e quelli che hanno frequentato in passato, con la relativa votazione finale in decimi.*

Frasi relative ai datori di lavoro
Relativamente ai datori *di lavoro presenti e passati dei partecipanti, rappresentiamo il nome, l'indirizzo e il numero di telefono.*

Frasi relative ai corsi
Per i corsi *(circa 200), rappresentiamo il titolo e il codice, le varie edizioni con date di inizio e fine e, per ogni edizione, rappresentiamo il numero di partecipanti e il giorno della settimana, le aule e le ore dove si sono tenute le lezioni.*

Frasi relative a tipi specifici di partecipanti
Per i partecipanti che sono liberi professionisti, *rappresentiamo l'area di interesse e, se lo possiedono, il titolo professionale. Per i partecipanti che sono* dipendenti, *rappresentiamo invece il loro livello e la posizione ricoperta.*

Frasi relative ai docenti
Per i docenti *(circa 300), rappresentiamo il cognome, l'età, la città di nascita, tutti i numeri di telefono, il titolo del corso che insegnano, di quelli che hanno insegnato in passato e di quelli che possono insegnare. I docenti possono essere dipendenti interni della società di formazione o collaboratori esterni.*

Figura 7.3 Esempio di strutturazione dei requisiti.

7.2 Rappresentazione concettuale di dati

Prima di affrontare le metodologie di progetto, cerchiamo di stabilire alcune buone pratiche per una corretta rappresentazione concettuale dei dati. Inizieremo da alcuni criteri generali di rappresentazione per poi passare a una rassegna di alcuni classi-

ci *design pattern*, ossia soluzioni progettuali a problemi comuni della progettazione concettuale dei dati.

7.2.1 Criteri generali di rappresentazione

Va innanzitutto precisato che spesso non esiste una rappresentazione univoca di un insieme di specifiche, perché le stesse informazioni possono essere rappresentate in modi differenti e non comparabili. Comunque, quando ci si trova davanti a diverse possibilità, è utile avere delle indicazioni sulle scelte più opportune. Nel caso della progettazione concettuale conviene, in buona sostanza, seguire le "regole concettuali" del modello E-R.

- *Se un concetto ha proprietà significative e/o descrive classi di oggetti con esistenza autonoma, è opportuno rappresentarlo con una entità.* Per esempio, nel caso delle specifiche relative alla società di formazione viste nel paragrafo precedente, è naturale rappresentare il concetto di *docente* con un'entità, in quanto possiede diverse proprietà (cognome, età, città di nascita) e la sua esistenza è indipendente dagli altri concetti. Chiaramente, lo stesso discorso vale anche per concetti astratti come, per esempio, quello di *corso*.
- *Se un concetto ha una struttura semplice e non possiede proprietà rilevanti associate, è opportuno rappresentarlo con un attributo di un altro concetto a cui si riferisce.* Per esempio, nel caso della società di formazione, il concetto di età è certamente da rappresentare come attributo. In effetti anche il concetto di città, che può risultare in generale un concetto autonomo e strutturato, va rappresentato nella nostra applicazione con un attributo perché, oltre al nome, non è di interesse nessuna altra sua proprietà.
- *Se sono state individuate due (o più) entità e nei requisiti compare un concetto che le associa, questo concetto può essere rappresentato da una relazione.* Per esempio, nella nostra applicazione, il concetto di *partecipazione a un corso* è certamente rappresentabile da una relazione tra le entità che rappresentano i *partecipanti* e i *corsi*. È importante sottolineare il fatto che questo vale solo nel caso in cui il concetto in questione non abbia, esso stesso, le caratteristiche delle entità. Un esempio tipico è il concetto di *visita* relativo a pazienti e medici: è assai improbabile che questo concetto possa essere rappresentato con una relazione tra paziente e medico. Innanzitutto perché di una visita sono tipicamente di interesse diverse proprietà quali, per esempio, la data, l'orario e la diagnosi. Ma soprattutto perché, per poter rappresentare il fatto molto plausibile che lo stesso paziente può sostenere più visite con lo stesso medico, allora la visita deve essere per forza rappresentata con una entità collegata da relazioni uno a molti con le entità che rappresentano i pazienti e i medici.
- *Se uno o più concetti risultano essere casi particolari di un altro, è opportuno rappresentarli facendo uso di una generalizzazione.* Nella nostra applicazione, è evidente che i concetti di *professionista* e *dipendente* costituiscono dei casi particolari del concetto di *partecipante* ed è quindi indicato definire una generalizzazione tra le entità che rappresentano questi concetti.

I criteri visti hanno validità generale, sono cioè indipendenti dalla strategia di progettazione scelta. Come vedremo nel prossimo paragrafo infatti, in ogni strategia esiste prima o poi un momento in cui va presa la decisione sul costrutto da scegliere per rappresentare una certa specifica.

7.2.2 Pattern di progetto

Cominciamo da un caso semplice: quello in cui si individua nelle specifiche un concetto autonomo con proprietà associate, le chiare caratteristiche di una entità del modello E-R. Nel caso per esempio di un impiegato di cui sono di interesse un codice, il nome e l'azienda nel quale lavora, otteniamo il primo, semplice schema con una sola entità riportato in Figura 7.4.

È importante comprendere che, con questa soluzione, non stiamo rappresentando anche il concetto di azienda: qui l'azienda è solo un attributo, ovvero niente di più che una stringa che assegnamo a una occorrenza di impiegato. Per poter rappresentare esplicitamente il concetto di azienda dobbiamo reificare l'attributo, facendolo diventare un'entità. Otteniamo così lo schema in Figura 7.5.

Passiamo ora a dei semplici pattern che coinvolgono le relazioni. Un caso piuttosto frequente di uso di questo costrutto è quello in cui si vuole rappresentare il fatto che un'entità è *parte di* un'altra entità, come avviene negli schemi in Figura 7.6. Queste relazioni sono tipicamente uno a molti e si presentano in due forme. Nel primo caso, l'esistenza di una occorrenza dell'entità "parte" dipende dall'esistenza di una occorrenza dell'entità che la contiene (nell'esempio, la sala di un cinema multisala) e richiede un'identificazione esterna. Nel secondo, l'entità contenuta nell'altra (in questo esempio il tecnico di un team) ha esistenza autonoma, come indicato dalla partecipazione opzionale alla relazione.

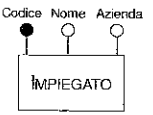

Figura 7.4 Un semplice pattern costituito da una sola entità.

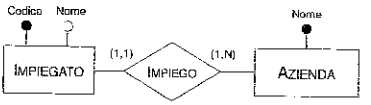

Figura 7.5 Reificazione dell'attributo Azienda in Figura 7.4.

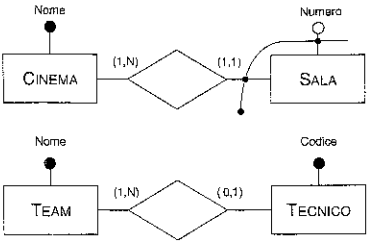

Figura 7.6 Relazioni di tipo "parte–di".

Un'altra situazione piuttosto comune è illustrata negli esempi in Figura 7.7 nei quali le occorrenze di un'entità della relazione sono *istanze di* occorrenze dell'altra entità.

Nel primo caso abbiamo un'entità che descrive il concetto astratto di volo presente sull'orario di una compagnia aerea, con un codice (per esempio AZ610), un'origine (per esempio Roma), una destinazione (per esempio New York) e un orario (per esempio 14:15), e un'altra entità che rappresenta il volo "reale", vale a dire l'istanza di un certo volo in un certo giorno (per esempio il volo AZ610 del 15/12/2013). È facile far confusione tra questi due concetti che però vanno tenuti ben distinti perché giocano ruoli diversi nell'applicazione. L'identificazione del volo reale avviene attraverso la data e, esternamente, il volo di cui è istanza (si assume quindi che lo stesso volo non possa essere ripetuto lo stesso giorno). Un caso analogo è l'altro schema in Figura 7.7 con il quale viene rappresentato il concetto di torneo sportivo (per esempio gli internazionali italiani di tennis) e una sua edizione (per esempio quella del 2014).

Consideriamo ora il caso in cui si utilizza una relazione, tipicamente molti a molti, per descrivere un concetto che lega altri due concetti, come avviene nell'esempio in Figura 7.8 nel quale l'esame è rappresentato da una relazione tra lo studente

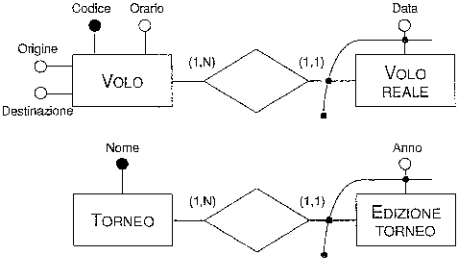

Figura 7.7 Relazione di tipo "istanza–di".

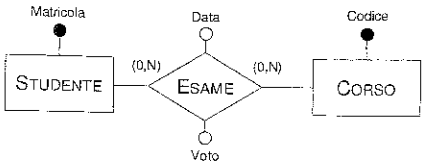

Figura 7.8 Relazione che rappresenta un concetto che lega altri concetti.

e il corso. Come già discusso nel Paragrafo 6.2.1, questa soluzione è valida solo se ogni studente può sostenere una sola volta un certo esame perché, per definizione, una occorrenza della relazione ESAME è un insieme di coppie studente–corso, senza duplicati.

Il fatto che questo concetto abbia degli attributi associati non cambia la situazione, ci suggerisce piuttosto che, soprattutto nel caso in cui uno studente può sostenere più volte lo stesso esame, la soluzione corretta è lo schema in Figura 7.9, nel quale abbiamo reificato la relazione ESAME di Figura 7.8 rappresentandola come entità. In questo caso, l'identificazione di un esame avviene attraverso lo studente, il corso e la data dell'esame.

Una soluzione alternativa che non richiede un'identificazione esterna complessa è costituita dallo schema in Figura 7.10, nel quale è stato introdotto un codice identificativo. Questa scelta semplifica le cose ma bisogna tenere conto del fatto che il codice è un concetto nuovo, non presente nelle specifiche e che quindi dovrà essere opportunamente gestito dal sistema informativo in via di sviluppo. Torneremo a parlare in termini generali di questo aspetto nel capitolo dedicato alla progettazione logica, quando affronteremo il problema della scelta degli identificatori nella traduzione verso il modello relazionale.

Lo schema in Figura 7.11 rappresenta un altro pattern piuttosto comune. Anche qui il concetto di partita può essere inizialmente visto inizialmente come una relazione ricorsiva sull'entità SQUADRA. Ma se, come spesso accade, in un torneo due squadre si incontrano più volte, è necessario reificare la relazione binaria e ottenere

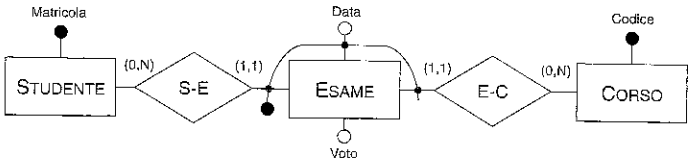

Figura 7.9 Reificazione della relazione in Figura 7.8.

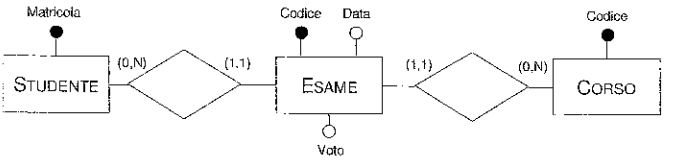

Figura 7.10 Introduzione di un codice nell'entità Esame in Figura 7.9.

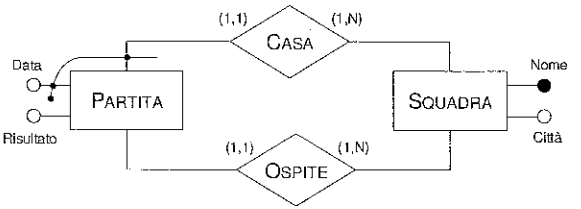

Figura 7.11 Reificazione di relazione ricorsiva.

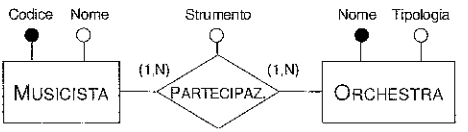

Figura 7.12 Relazione molti a molti con attributo.

lo schema in figura. L'identificazione dell'entità PARTITA coinvolge solo la data e la squadra che gioca in casa perché qui evidentemente si assume che una squadra non possa giocare due partite nello stesso giorno.

Consideriamo ora la relazione molti a molti in Figura 7.12 che rappresenta la partecipazione di un musicista a un'orchestra con un certo strumento. In base a quanto sopra esposto, se il musicista può suonare strumenti diversi ma suona, per ogni orchestra, sempre lo stesso strumento, lo schema è corretto ed è sufficiente una relazione con attributo Strumento. Il difetto di questo schema è semmai un altro e ha a che fare con quanto detto per lo schema in Figura 7.5: non stiamo rappresentando

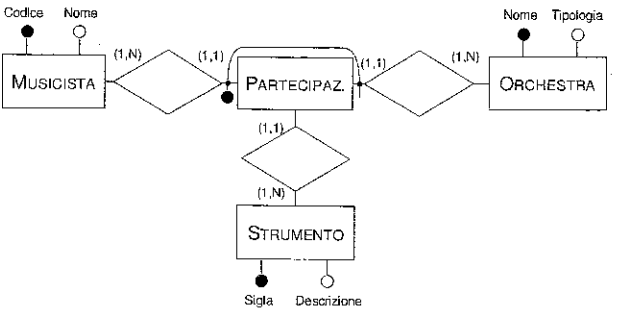

Figura 7.13 Reificazione di attributo di relazione.

esplicitamente il concetto di strumento, che qui è solo una stringa. Se lo strumento è un concetto rilevante per l'applicazione, dobbiamo reificare l'attributo della relazione e, per poterlo fare, dobbiamo reificare anche la relazione. Si ottiene in questo modo lo schema in Figura 7.13.

Passiamo ora ad alcuni pattern che coinvolgono le generalizzazioni. Un primo esempio di uso comune di questo costrutto è quello riportato in Figura 7.14, nel quale si vuole rappresentare un caso particolare di un altro, nell'esempio il sottoinsieme degli impiegati che sono dei manager. Si noti come sia possibile specializzare in questo modo i vari ruoli all'interno di un progetto (l'altra entità dello schema): la gestione è a carico solo dei manager.

In questo schema è ragionevole assumere che un manager può gestire solo un progetto al quale partecipa. Questo implica che che ogni coppia manager–progetto che compare tra le occorrenze della relazione GESTIONE deve comparire anche tra le occorrenze della relazione PARTECIPAZIONE. Questo vincolo però non può essere espresso direttamente sullo schema con un apposito costrutto e va quindi aggiunta una regola alla documentazione dello schema, come abbiamo descritto nel Paragrafo 6.3 del capitolo precedente.

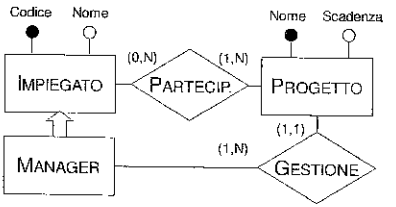

Figura 7.14 Caso particolare di entità.

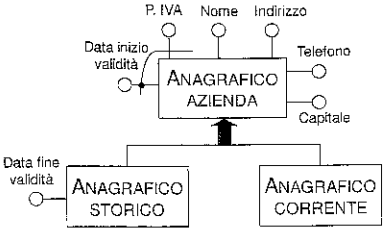

Figura 7.15 Storicizzazione di entità.

L'esempio appena presentato è un caso di sottoinsieme (la generalizzazione ha una sola entità figlia). Il pattern però può essere facilmente generalizzato al caso in cui ci siano più casi particolari da considerare: in questo caso la generalizzazione avrebbe più entità figlie, magari su più livelli, e le varie proprietà (attributi e partecipazione a relazioni) andrebbero distribuite, a seconda della loro specificità, tra le varie entità partecipanti alla generalizzazione.

Lo schema in Figura 7.15 mostra un altro caso di uso comune del costrutto di generalizzazione. Si tratta di uno schema nel quale si vuole gestire la "storicizzazione" di un concetto, nel caso particolare, di un'entità. Nell'esempio vogliamo memorizzare le informazioni correnti di un'azienda, tenendo però traccia dei dati che sono variati. Come suggerito dallo schema, una soluzione piuttosto efficace consiste nell'utilizzare allo scopo due entità con gli stessi attributi: una rappresenta il concetto di interesse con le informazioni aggiornate, l'altra lo "storico". Le proprietà di queste entità vengono messe a fattor comune mediante una generalizzazione la cui entità genitore rappresenta tutte le informazioni anagrafiche delle aziende, sia quelle correnti sia quelle passate. Vengono inoltre introdotti degli attributi per definire l'intervallo di validità dei dati (data inizio e data fine). L'identificazione si ottiene aggiungendo all'identificatore "naturale" (in questo caso, la partita iva) la data di inizio di validità delle informazioni, ovvero il momento in cui esse sono state introdotte: questo istante diventerà anche la data di fine validità delle informazioni che vengono soppiantate.

Un caso analogo è mostrato nello schema in Figura 7.16. In questo caso si vuole storicizzare un concetto rappresentato da una relazione tra entità, nell'esempio gli impieghi presenti e passati di una persona, in altre parole, il suo curriculum lavorativo. Come nel caso precedente, una possibile soluzione consiste nel rappresentare separatamente i dati correnti e i dati storici e introdurre opportuni attributi per specificare gli intervalli di validità delle informazioni. Notare le differenti cardinalità delle partecipazioni dell'entità PERSONA alle due relazioni. Un'analisi attenta di questo ultimo schema ci fa comprendere che, per i motivi già discussi in precedenza, qui non possiamo rappresentare il fatto che una persona possa aver lavorato, in periodi diversi, per la stessa azienda. Avremmo infatti in questo caso due occorrenze identiche della relazione IMPIEGO PASSATO. La soluzione in questo caso è, ancora una volta, la reificazione delle relazioni. Otteniamo così lo schema in Figura 7.17 che consente

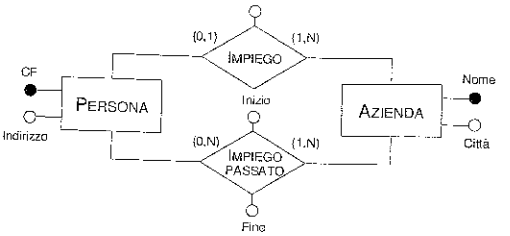

Figura 7.16 Storicizzazione di relazione.

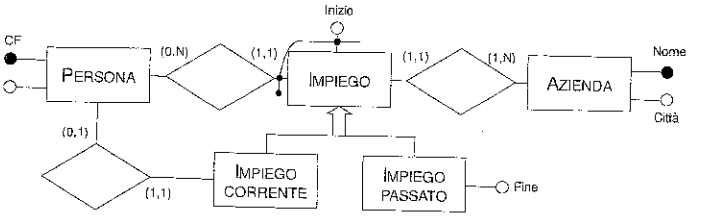

Figura 7.17 Reificazione delle relazioni in Figura 7.16.

di utilizzare una generalizzazione. Anche in questo caso risulta necessario l'inserimento di un vincolo esterno allo schema che impone che tutte le occorrenze della relazione tra PERSONA e IMPIEGO CORRENTE compaiano anche tra le occorrenze della relazione tra PERSONA e IMPIEGO.

L'ultimo esempio di uso comune del costrutto di generalizzazione è quello riportato in Figura 7.18. In questo schema vogliamo rappresentare il fatto che un certo concetto subisce una evoluzione nel tempo che può essere diversa per le diverse occorrenze del concetto. Nell'esempio abbiamo dei progetti che vengono proposti con l'obiettivo di ottenere un finanziamento. Solo alcuni di questi vengono accettati e, per questi, vanno aggiunte ulteriori informazioni quali la data di inizio ufficiale del progetto e il finanziamento effettivamente assegnato. Come si vede dallo schema in figura, il costrutto di generalizzazione si presta bene a modellare questa situazione.

Consideriamo infine la relazione ternaria in Figura 7.19. Come già accennato nel Capitolo 6.2.2, negli schemi E-R le relazioni ternarie si incontrano raramente e relazioni che coinvolgono più di tre entità sono fortemente sconsigliate, perché tipicamente cercano di rappresentare, con un unico costrutto, concetti tra loro indipendenti. Questo aspetto verrà chiarito maggiormente nel Capitolo 9 dedicato alla normalizzazione, una tecnica sistematica che consente di analizzare queste situazioni. Nell'esempio in figura è stata scelta correttamente una relazione ternaria perché si vuole modellare il caso in cui un operatore può effettuare operazioni che consistono in attività diverse svolte in

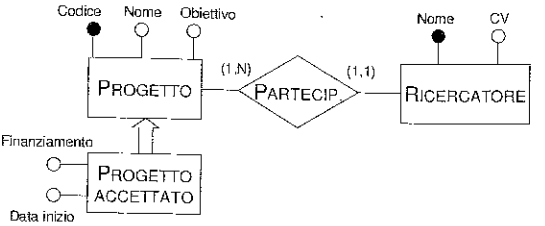

Figura 7.18 Evoluzione di un concetto.

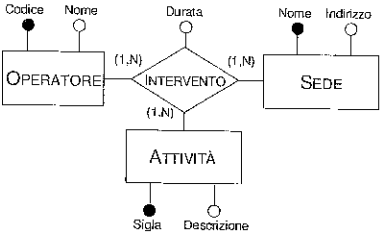

Figura 7.19 Relazione ternaria.

sedi diverse. Inoltre in ogni sede possono operare operatori diversi svolgendo attività diverse. Infine le attività possono essere svolte da operatori diversi e in sedi diverse.

Anche questa relazione, come tutte le altre, può essere reificata e questa operazione si rende necessaria quanto la realtà da modellare è diversa da quella appena descritta. In Figura 7.20 viene riportata la reificazione della relazione ternaria in Figura 7.19. Il nuovo schema modella esattamente la situazione dello schema originario perché la nuova entità risulta identificata da tutte le entità originarie. Cambiando opportunamente l'identificazione siamo però in grado di modellare con questo pattern altre situazioni per le quali la relazione ternaria non sarebbe corretta.

In particolare, se in ogni sede, ogni operatore svolge sempre la stessa attività, l'entità INTERVENTO sarebbe identificata solo dalle entità SEDE e OPERATORE. Se viceversa in ogni sede, ogni attività viene svolta sempre dallo stesso operatore, l'entità INTERVENTO sarebbe identificata solo dalle entità ATTIVITÀ e SEDE. Se infine ogni operatore svolge ogni attività in una sola sede, l'entità INTERVENTO sarebbe identificata solo dalle entità OPERATORE e SEDE.

Infine, lo schema in Figura 7.21 descrive nel modo migliore la situazione in cui la sola entità ATTIVITÀ è identificante, succede cioè che ogni attività viene svolta in una sola sede da un solo operatore.

In questo caso lo schema si semplifica perché il legame tra attività e la sede si può rappresentare separatamente da quello tra l'attività e l'operatore.

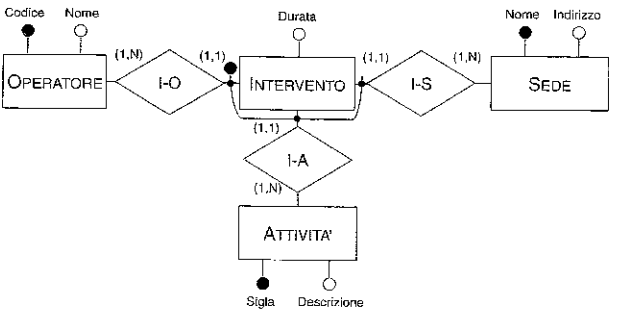

Figura 7.20 Reificazione della relazione ternaria in Figura 7.19.

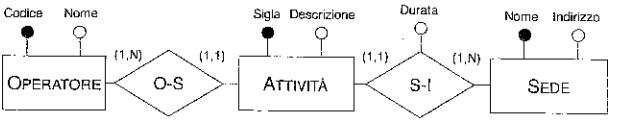

Figura 7.21 Semplificazione dello schema in Figura 7.20.

7.3 Strategie di progetto

Lo sviluppo di uno schema concettuale a partire dalle sue specifiche può essere considerato a tutti gli effetti un processo di ingegnerizzazione e, come tale, risultano a esso applicabili le strategie di progetto utilizzate anche in altre discipline. Vediamo quali sono queste strategie con specifico riferimento alla modellazione di una base di dati.

7.3.1 Strategia top-down

In questa strategia, lo schema concettuale viene prodotto mediante una serie di raffinamenti successivi a partire da uno schema iniziale che descrive tutte le specifiche con pochi concetti molto astratti. Lo schema viene poi via via raffinato mediante opportune trasformazioni che aumentano il dettaglio dei vari concetti presenti. Questo procedimento viene descritto graficamente in Figura 7.22 dove vengono rappresentati i diversi piani di raffinamento del processo: ognuno di questi piani contiene uno schema che descrive le medesime informazioni a un diverso livello di dettaglio. Con questa strategia quindi, tutti gli aspetti presenti nello schema finale sono presenti, in linea di principio, a ogni livello di raffinamento.

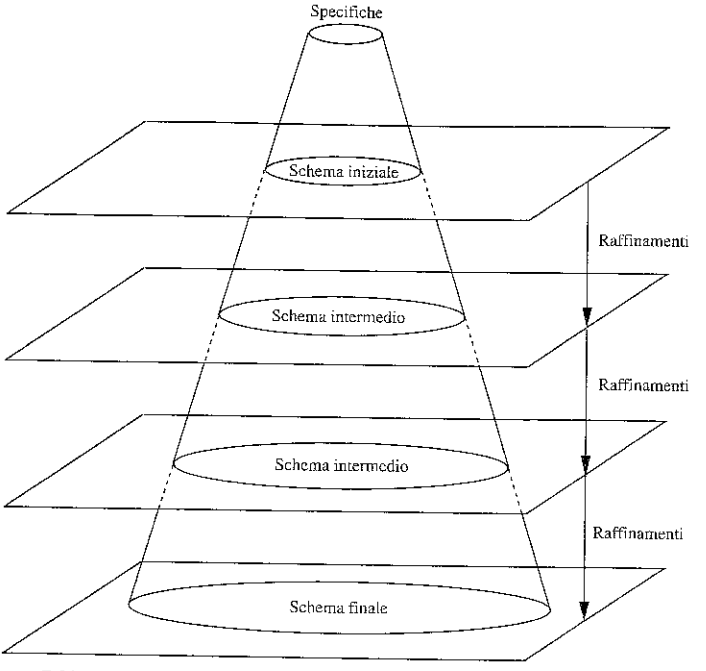

Figura 7.22 La strategia top-down.

Nel passaggio da un livello di raffinamento a un altro, lo schema viene modificato facendo uso di alcune trasformazioni elementari che vengono denominate *primitive di trasformazione top-down*.

Esempi di primitive di trasformazione top-down sono:

- la definizione degli attributi di una entità o di una relazione; per esempio, la specifica, per una entità PERSONA, di tutti gli attributi di interesse quali Codice Fiscale, Cognome, Età, Sesso e Città di nascita;
- la reificazione di un attributo o di una entità, che trasforma per esempio lo schema in Figura 7.4 nello schema in Figura 7.5 e lo schema in Figura 7.8 nello schema in Figura 7.9;
- la decomposizione di una relazione in due relazioni distinte; per esempio quella che consente di giungere, da uno schema con una sola relazione IMPIEGO tra le entità PERSONA e AZIENDA allo schema in Figura 7.16;
- la trasformazione di una entità in una gerarchia di generalizzazione che per esem-

pio consente di giungere allo schema di Figura 7.15 partendo da un'unica entità AZIENDA.

Il vantaggio della strategia top-down è che il progettista può descrivere inizialmente tutte le specifiche dei dati trascurandone i dettagli, per poi entrare nel merito di un concetto alla volta (si osservi infatti che le primitive di trasformazione agiscono su singoli concetti). Questo però è possibile solo quando si possiede, sin dall'inizio, una visione globale e astratta di *tutte* le componenti del sistema, ma ciò è estremamente difficile quando si ha a che fare con applicazioni di una certa complessità.

7.3.2 Strategia bottom-up

In questa strategia, le specifiche iniziali sono suddivise in componenti via via sempre più piccole, fino a quando queste componenti descrivono un frammento elementare della realtà di interesse. A questo punto, le varie componenti vengono rappresentate da semplici schemi concettuali che possono consistere anche in singoli concetti. I vari schemi così ottenuti vengono poi fusi fino a giungere, attraverso una completa integrazione di tutte le componenti, allo schema concettuale finale. Questo procedimento viene descritto graficamente in Figura 7.23, nella quale vengono rappresentate: la fase di decomposizione delle specifiche, la successiva fase di rappresentazione delle componenti di base e la fase finale d'integrazione degli schemi elementari. A differenza della strategia top-down, con questa strategia i vari concetti presenti nello schema finale vengono via via introdotti durante le varie fasi.

Anche in questo caso, lo schema finale si ottiene attraverso alcune trasformazioni elementari che vengono denominate *primitive di trasformazione bottom-up* che introducono in uno schema nuovi concetti non presenti precedentemente e in grado di descrivere aspetti della realtà di interesse che non erano ancora stati rappresentati.

Esempi di primitive di trasformazione bottom-up sono:

- l'introduzione di una nuova entità o di una relazione dall'analisi delle specifiche; per esempio, nell'applicazione relativa alla società di formazione, questo può accadere quando, nelle specifiche riportate a pagina 237, individuiamo l'entità PARTECIPANTE oppure quando individuiamo la relazione ABILITAZIONE tra le entità DOCENTE e CORSO;
- l'individuazione nelle specifiche di un legame tra diverse entità riconducibile a una generalizzazione; per esempio, con riferimento alle specifiche suddette, questo può accadere quando comprendiamo che l'entità DOCENTE è una generalizzazione delle entità INTERNO (dipendente della società di formazione) e COLLABORATORE;
- l'aggregazione di una serie di attributi in una entità o in una relazione; per esempio, quando dalle proprietà Matricola, Cognome, Data nascita, Città di nascita e Media esami si individua l'esistenza dell'entità STUDENTE.

Il vantaggio della strategia bottom-up è che si adatta a una decomposizione del problema in componenti più semplici, facilmente individuabili, il cui progetto può essere affrontato anche da progettisti diversi. È quindi un tipo di strategia che si presta bene a lavori svolti in collaborazione o suddivisi all'interno di un gruppo. Lo svantaggio di questa strategia è invece il fatto che richiede delle operazioni di integrazione di schemi

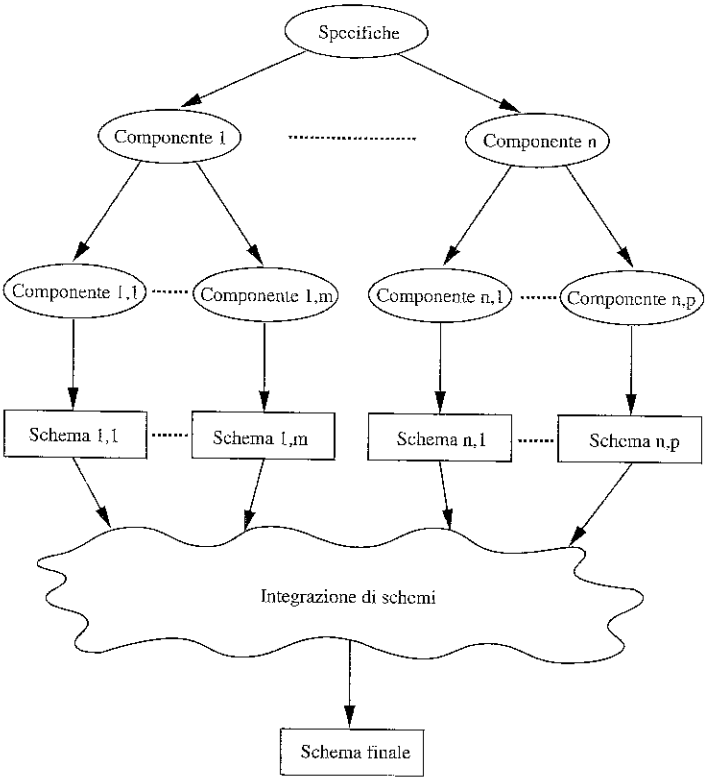

Figura 7.23 La strategia bottom-up.

concettuali diversi che, nel caso di schemi complessi, presentano quasi sempre grosse difficoltà.

7.3.3 Strategia inside-out

Questa strategia può essere vista come un caso particolare della strategia bottom-up. Si individuano inizialmente solo alcuni concetti importanti e poi si procede, a partire da questi, a "macchia d'olio". Si rappresentano cioè prima i concetti in relazione con i concetti iniziali, per poi muoversi verso quelli più lontani attraverso una "navigazione" tra le specifiche.

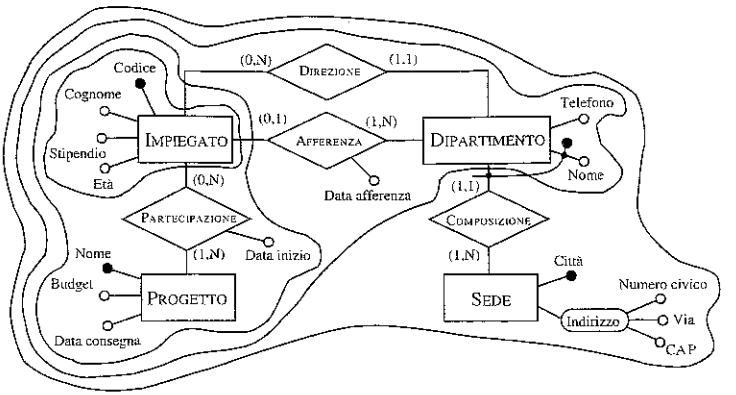

Figura 7.24 Un esempio di uso della strategia inside-out.

Un esempio di sviluppo inside-out di uno schema concettuale è mostrato in Figura 7.24 con riferimento a un esempio visto nel capitolo precedente. In questa figura le varie aree indicano un possibile sviluppo cronologico del progetto.

Si può osservare che è stata individuata inizialmente l'entità IMPIEGATO con i suoi attributi. A partire da questa entità sono state rappresentate la partecipazione degli impiegati ai progetti e tutte le proprietà dei progetti. Successivamente, sono state analizzate le correlazioni esistenti tra gli impiegati e i dipartimenti dell'azienda, individuando le relazioni DIREZIONE e AFFERENZA e l'entità DIPARTIMENTO con i relativi attributi. Infine, partendo da quest'ultima entità, sono state rappresentate le sedi dell'azienda (entità SEDE e relativi attributi) e l'appartenenza dei dipartimenti alle relative sedi (relazione COMPOSIZIONE). Si osservi che, nella penultima fase, non si poteva identificare l'entità DIPARTIMENTO (a meno di aggiungere altri attributi), perché è possibile avere dipartimenti con lo stesso nome in sedi diverse, ma, al passo successivo, è stato possibile identificare tale entità con l'attributo Nome e l'entità SEDE attraverso la relazione COMPOSIZIONE.

Questa strategia ha il vantaggio di non richiedere passi di integrazione. D'altro canto è necessario, di volta in volta, esaminare tutte le specifiche per individuare concetti non ancora rappresentati e descrivere i nuovi concetti nel dettaglio (cosa non sempre possibile come mostrato nell'esempio). Non è quindi possibile procedere per livelli di astrazione come avviene nella strategia top-down.

7.3.4 Strategia mista

La strategia mista cerca di combinare i vantaggi della strategia top-down con quelli della strategia bottom-up. Il progettista suddivide i requisiti in componenti separate,

come nella strategia bottom-up, ma allo stesso tempo definisce uno *schema scheletro* contenente, a livello astratto, i concetti principali dell'applicazione. Questo schema scheletro fornisce una visione unitaria, sia pure astratta, dell'intero progetto e favorisce le fasi di integrazione degli schemi sviluppati separatamente.

Come esempio, in Figura 7.25 viene riportato un possibile schema scheletro per la nostra applicazione relativa alla società di formazione. Da una semplice ispezione delle specifiche strutturate dei requisiti contenute nel Paragrafo 7.1, è quasi immediato individuare tre concetti principali che possono essere rappresentati in maniera naturale da entità: i *partecipanti*, i *corsi* e i *docenti*. Tra queste entità esistono delle relazioni che, a un primo livello di dettaglio, possiamo assumere siano descrizioni della *partecipazione* ai corsi da parte dei partecipanti e dell'attività didattica (la *docenza*) svolta dai docenti nei corsi. A questo punto possiamo procedere considerando, anche separatamente, questi concetti principali e proseguire per raffinamenti successivi (procedendo quindi in maniera top-down) oppure estendere lo schema (o il sottoschema) con concetti non ancora rappresentati (procedendo quindi in maniera bottom-up).

La strategia mista è probabilmente la più flessibile tra le strategie viste perché si adatta bene a esigenze contrapposte: quella di suddividere un problema complesso in sottoproblemi e quella di procedere per raffinamenti successivi. In effetti, questa strategia ingloba anche la strategia inside-out che, come abbiamo detto, è solo un caso particolare della strategia bottom-up. È infatti abbastanza naturale, durante uno sviluppo bottom-up di una sottocomponente del progetto, procedere a macchia d'olio per rappresentare le specifiche della nostra base di dati non ancora rappresentate.

C'è anche da dire che, in quasi tutti i casi pratici di una certa complessità, la strategia mista è l'unica che si può effettivamente adottare perché, come abbiamo detto all'inizio di questo capitolo, è spesso necessario cominciare la progettazione quando non sono ancora disponibili tutti i dati e, dei dati noti, abbiamo delle conoscenze a livelli di dettaglio non omogenei.

7.4 Qualità di uno schema concettuale

Nella costruzione di uno schema concettuale vanno comunque garantite alcune proprietà generali che uno schema concettuale di buona qualità deve possedere. Analizziamo le qualità più importanti e vediamo come è possibile verificare, durante la progettazione concettuale, queste qualità.

Figura 7.25 Schema scheletro per la società di formazione.

Correttezza Uno schema concettuale è *corretto* quando utilizza propriamente i costrutti messi a disposizione dal modello concettuale di riferimento. Come avviene nei linguaggi di programmazione, gli errori possono essere *sintattici* o *semantici*. I primi riguardano un uso non ammesso di costrutti come, per esempio, una generalizzazione tra relazioni invece che tra entità. I secondi riguardano invece un uso di costrutti che non rispetta la loro definizione. Per esempio, l'uso di una relazione per descrivere il fatto che una entità è specializzazione di un'altra. La correttezza di uno schema si può verificare per ispezione, confrontando i concetti presenti nello schema in via di costruzione con le specifiche e con le definizioni dei costrutti del modello concettuale usato.

Completezza Uno schema concettuale è *completo* quando rappresenta tutti i dati di interesse e quando tutte le operazioni possono essere eseguite a partire dai concetti descritti nello schema. La completezza di uno schema si può verificare controllando che tutte le specifiche sui dati siano rappresentate da qualche concetto presente nello schema che stiamo costruendo, e che tutti i concetti coinvolti in un'operazione presente nelle specifiche siano raggiungibili "navigando" attraverso lo schema.

Leggibilità Uno schema concettuale è *leggibile* quando rappresenta i requisiti in maniera naturale e facilmente comprensibile. Per garantire questa proprietà è necessario rendere lo schema autoesplicativo, per esempio, mediante una scelta opportuna dei nomi da dare ai concetti. La leggibilità dipende anche da criteri puramente estetici: la comprensione di uno schema è per esempio facilitata se tracciamo il relativo diagramma su una griglia nella quale i vari costrutti hanno le stesse dimensioni. Alcuni suggerimenti per rendere lo schema più leggibile sono i seguenti:

- disporre i costrutti su una griglia scegliendo come elementi centrali quelli con più legami (relazioni) con altri;
- tracciare solo linee perpendicolari e cercare di minimizzare le intersezioni;
- disporre le entità che sono genitori di generalizzazioni sopra le relative entità figlie.

La leggibilità di uno schema si può verificare facendo delle prove di comprensione con gli utenti.

Minimalità Uno schema è *minimale* quando tutte le specifiche sui dati sono rappresentate una sola volta nello schema. Uno schema quindi non è minimale quando esistono delle *ridondanze*, ovvero concetti che possono essere derivati da altri. Una possibile fonte di ridondanza in uno schema E-R è la presenza di cicli dovuta alla presenza di relazioni e/o generalizzazioni. A differenza delle altre proprietà comunque, non sempre una ridondanza è indesiderata, ma può nascere da precise scelte progettuali.[1] In ogni caso però, queste situazioni vanno documentate. La minimalità di uno schema si può verificare per ispezione, controllando se esistono concetti che possono

[1] Torneremo su questo punto quando affronteremo la progettazione logica.

essere eliminati dallo schema che stiamo costruendo senza inficiare la sua completezza. Per quanto detto, si deve prestare particolare attenzione ai cicli presenti nello schema.

Nel prossimo paragrafo vedremo come la verifica delle qualità di uno schema concettuale appena viste possa essere inglobata in una metodologia di progettazione generale.

7.5 Una metodologia generale

Cerchiamo di tirare le somme su quanto detto relativamente alla progettazione concettuale di basi di dati. Per quel che riguarda le strategie di progetto viste va precisato che, in pratica, non accade quasi mai che un progetto proceda *sempre* in maniera top-down o bottom-up. Indipendentemente dalla strategia scelta, nelle situazioni reali capita infatti di modificare lo schema in via di costruzione sia con trasformazioni che raffinano un concetto presente (e quindi tipicamente top-down) sia con trasformazioni che aggiungono un concetto non presente (e quindi tipicamente bottom-up). Presentiamo quindi una metodologia per la progettazione concettuale con il modello E-R con riferimento alla strategia mista che, come abbiamo detto, fa uso delle tecniche su cui si basano le altre e le comprende come caso particolare. La metodologia è composta dai seguenti passi.

1. **Analisi dei requisiti**.
   (a) Costruire un glossario dei termini.
   (b) Analizzare i requisiti ed eliminare le ambiguità presenti.
   (c) Raggruppare i requisiti in insiemi omogenei.
2. **Passo base**.
   (a) Individuare i concetti più rilevanti e rappresentarli in uno schema scheletro.
3. **Passo di decomposizione** (da effettuare se appropriato o necessario).
   (a) Effettuare una decomposizione dei requisiti con riferimento ai concetti presenti nello schema scheletro.
4. **Passo iterativo**: da ripetere, per tutti i sotto-schemi (se presenti), finché ogni specifica è stata rappresentata.
   (a) Raffinare i concetti presenti sulla base delle loro specifiche.
   (b) Aggiungere nuovi concetti allo schema per descrivere specifiche non ancora descritte.
5. **Passo di integrazione** (da effettuare se è stato eseguito il passo 3).
   (a) Integrare i vari sotto-schemi in uno schema generale facendo riferimento allo schema scheletro.
6. **Analisi di qualità**.
   (a) Verificare la correttezza dello schema ed eventualmente ristrutturare lo schema.
   (b) Verificare la completezza dello schema ed eventualmente ristrutturare lo schema.

(c) Verificare la minimalità, documentare le ridondanze ed eventualmente ristrutturare lo schema.
(d) Verificare la leggibilità dello schema ed eventualmente ristrutturare lo schema.

Si osservi che se il passo 3 e il passo 5 non vengono effettuati e nel passo 4 si procede solo mediante raffinamenti (azione (a)), abbiamo una strategia top-down pura. Viceversa, se il passo base non viene effettuato e nel passo 5 vengono solo aggiunti nuovi concetti, ci stiamo muovendo secondo la strategia bottom-up pura. Infine, nelle trasformazioni bottom-up, si può procedere a "macchia d'olio", cioè secondo la strategia inside-out.

Nella metodologia presentata, viene solo brevemente citata un'importante attività che dovrebbe in realtà accompagnare tutte le fasi di progetto: quella della documentazione degli schemi. Secondo quanto detto nel Paragrafo 6.3, anche questa attività può essere disciplinata facendo uso di strumenti opportuni. In particolare, è molto utile costruire, parallelamente allo sviluppo di uno schema, anche un dizionario dei dati che favorisca l'interpretazione dei vari concetti. Inoltre, possiamo far uso di regole aziendali per descrivere la presenza di ridondanze o requisiti dell'applicazione che non riusciamo a tradurre in costrutti del modello E-R.

Concludiamo questa presentazione con una breve riflessione sulla fase finale della metodologia presentata, quella dell'analisi della qualità del progetto. Innanzitutto va precisato che questa attività non va relegata al termine della progettazione, ma va effettuata, con regolarità, durante tutto lo sviluppo dello schema concettuale. Va infatti sottolineato che l'analisi di qualità costituisce un importante momento di verifica dello stato corrente del progetto nel quale è spesso necessario dover effettuare delle ristrutturazioni per rimediare a "errori" fatti nelle fasi precedenti. Bisogna porre, in questa fase, particolare attenzione a concetti dello schema aventi proprietà particolari: per esempio, entità senza attributi, insiemi di concetti che formano cicli, gerarchie di generalizzazioni troppo complesse o porzioni dello schema particolarmente contorte. Come accennato nel Paragrafo 7.4, non è detto che questa analisi porti necessariamente a delle ristrutturazioni, ma solo a una riorganizzazione dello schema che ne aumenti la leggibilità.

7.6 Un esempio di progettazione concettuale

Vediamo ora un esempio concreto e completo di progettazione concettuale che fa riferimento alla solita società di formazione. Abbiamo già eseguito per questo caso i compiti della prima fase della metodologia e abbiamo mostrato un possibile schema scheletro per questa applicazione in Figura 7.25. Con riferimento a questo schema, possiamo a questo punto decidere di analizzare separatamente le specifiche riguardanti i partecipanti, quelle riguardanti i corsi e quelle riguardanti i docenti e procedere a macchia d'olio per includere i concetti non presi in considerazione nello schema scheletro. Vedremo che, nella costruzione dei vari schemi, incontreremo in molti casi i pattern progettuali discussi nel Paragrafo 7.2.2.

Eseguiamo quindi il passo iterativo della metodologia generale, considerando prima i partecipanti. Tra questi si individuano facilmente due tipologie: i *professionisti* e i *dipendenti*. Questi concetti sono rappresentabili come entità figlie dell'entità

PARTECIPANTE iniziale: la generalizzazione che ne risulta è totale. A questo punto vanno rappresentati gli impieghi dei partecipanti. Questo può essere fatto introducendo innanzitutto un'entità DATORE visto che vanno rappresentate, per questo concetto, diverse proprietà. Analizzando poi le specifiche relative agli impieghi dei partecipanti, ci accorgiamo che vanno rappresentati due concetti distinti: i rapporti passati e quelli presenti. Possiamo allora introdurre due relazioni IMPIEGO PASSATO e IMPIEGO CORRENTE: la prima ha come attributi una data di inizio e una di fine rapporto e lega l'entità DATORE con l'entità PARTECIPANTE (perché anche i professionisti possono aver avuto, nel passato, un lavoro dipendente); la seconda ha solo una data di inizio rapporto e lega l'entità DATORE con l'entità DIPENDENTE. Aggiungendo gli attributi a entità e relazioni, le cardinalità alle relazioni e gli identificatori alle entità, si ottiene lo schema in Figura 7.26. Si osservi che l'entità PARTECIPANTE ha due identificatori: il codice interno dell'azienda e il codice fiscale. Si osservi inoltre che l'attributo Titolo professionale è opzionale, in quanto dalle specifiche si evince che questo dato può mancare.

Per quanto riguarda i docenti vanno distinti i dipendenti interni della società di formazione dai collaboratori esterni. Questo si può fare in maniera naturale con una generalizzazione totale di cui DOCENTE è l'entità genitore. Si possono quindi aggiungere gli attributi Cognome, Età, Città di nascita e Numero di telefono all'entità DOCENTE. Quest'ultimo è multivalore perché dalle specifiche risulta che i

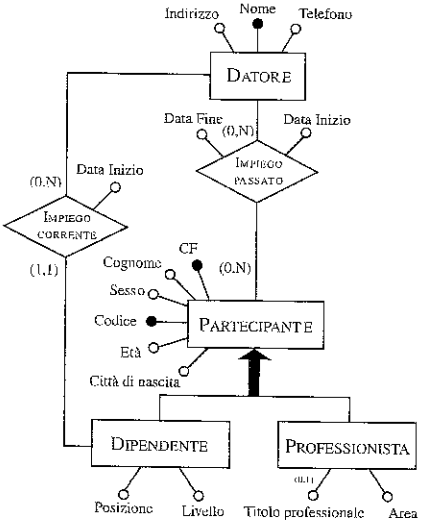

Figura 7.26 Il raffinamento di una porzione dello schema scheletro.

docenti possono avere più numeri di telefono e noi li vogliamo rappresentare tutti. A questo punto si può osservare che i vari attributi non forniscono un identificatore naturale per l'entità DOCENTE. In casi come questo si cerca di individuare un concetto che possa identificare l'entità anche se questo non rappresenta una specifica. Nel nostro caso, si può introdurre a questo scopo il codice fiscale del docente anche se questo dato non faceva parte delle specifiche iniziali. Alternativamente, si sarebbe potuto introdurre un codice da usare appositamente per questo scopo. Il sotto-schema risultante è riportato in Figura 7.27.

Passiamo ora all'analisi dell'entità CORSO. Vanno innanzitutto distinti due concetti legati tra loro, ma chiaramente distinti: il concetto astratto di corso (che ha un nome e un codice) dall'edizione di un corso che ha una data di inizio, una data di fine e un numero di partecipanti. Rappresentiamo questi due concetti con due entità distinte legate dalla relazione TIPOLOGIA. Vanno poi rappresentate le *lezioni* dei corsi, che possiamo descrivere con una entità legata alle edizioni dei corsi da una relazione COMPOSIZIONE. Aggiungiamo poi gli attributi, le cardinalità e gli identificatori. Per quel che riguarda gli identificatori assumiamo che una lezione sia identificata dall'aula, l'ora e il giorno (non è infatti possibile avere nello stesso giorno, nello stesso orario e nella stessa aula due lezioni diverse). Per le edizioni di corso assumiamo invece che non possono partire nello stesso giorno edizioni diverse dello stesso corso e quindi un identificatore per l'entità EDIZIONE DI CORSO è costituito dall'attributo Data inizio e dall'entità CORSO. Il sotto-schema risultante è riportato in Figura 7.28.

Lo schema finale si ottiene per integrazione degli schemi ottenuti fino a questo punto. Iniziamo con gli schemi relativi ai docenti e ai corsi rappresentati in Figura 7.27 e in Figura 7.28 rispettivamente: dallo schema scheletro si intuisce che il

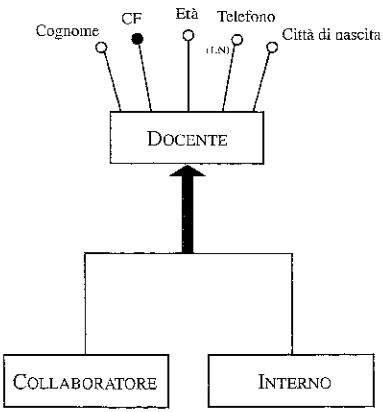

Figura 7.27 Il raffinamento di un'altra porzione dello schema scheletro.

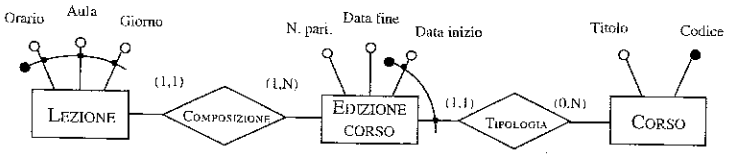

Figura 7.28 Il raffinamento di un'altra porzione dello schema scheletro.

collegamento avviene attraverso una relazione di docenza che va però raffinata. Dall'analisi delle specifiche non è difficile individuare tre tipi di correlazioni diverse tra i docenti e i corsi: le *docenze correnti*, quelle *passate* e l'*abilitazione* a insegnare un corso. Rappresentiamo questi legami con tre relazioni: le prime due collegano le entità DOCENTE ed EDIZIONE DI CORSO (perché un docente insegna o ha insegnato una specifica edizione di corso) mentre la terza collega l'entità DOCENTE e l'entità CORSO (perché un docente viene abilitato a insegnare un generico corso). Va ora integrato lo schema ottenuto con la porzione relativa ai partecipanti, riportata in Figura 7.26. Dallo schema scheletro si intuisce che, per fare questo, va stabilito il tipo di relazione che intercorre tra i corsi e i partecipanti. Se ne possono individuare due: le partecipazioni *correnti* e quelle *passate* che rappresentiamo con relazioni tra l'entità PARTECIPANTE e l'entità EDIZIONE DI CORSO. Di quelle passate è d'interesse la votazione che rappresentiamo con un attributo. Aggiungendo le varie cardinalità si ottiene lo schema finale riportato in Figura 7.29.

Va notato che abbiamo proceduto, in questo caso, decomponendo e poi integrando, ma, trattandosi di uno schema non molto complesso, avremmo potuto anche lavorare direttamente sullo schema scheletro procedendo per raffinamenti e integrazioni successive senza veri e propri passi di integrazione.

A questo punto, restano da verificare le proprietà dello schema così ottenuto. In particolare, la completezza si verifica ripercorrendo tutti i requisiti sui dati e sulle operazioni controllando che tutti i dati siano stati rappresentati e che tutte le operazioni possano essere eseguite mediante una navigazione sullo schema. Per citare un esempio, consideriamo l'operazione 7 che richiedeva l'elenco di tutti i partecipanti ai corsi insegnati da un docente. Questa operazione è in effetti eseguibile con riferimento allo schema in Figura 7.26 come segue: partiamo dall'entità DOCENTE, attraversiamo le relazioni DOCENZA PASSATA e DOCENZA CORRENTE, raggiungiamo l'entità EDIZIONE CORSO e da questa, tramite le relazioni PARTECIPAZIONE PASSATA e PARTECIPAZIONE CORRENTE, arriviamo infine all'entità PARTECIPANTE. Possiamo così ottenere, dato un docente, i partecipanti a tutti i corsi da lui/lei insegnati. Per quanto riguarda invece la minimalità, si può osservare che esiste nello schema una ridondanza: l'attributo **Numero di partecipanti** dell'entità EDIZIONE DI CORSO può essere infatti derivato, per una certa edizione, contando il numero di istanze dell'entità PARTECIPANTE che sono legate a questa edizione. Rimandiamo alla fase successiva, la progettazione logica, la decisione sul fatto di mantenere o eliminare tale ridondanza.

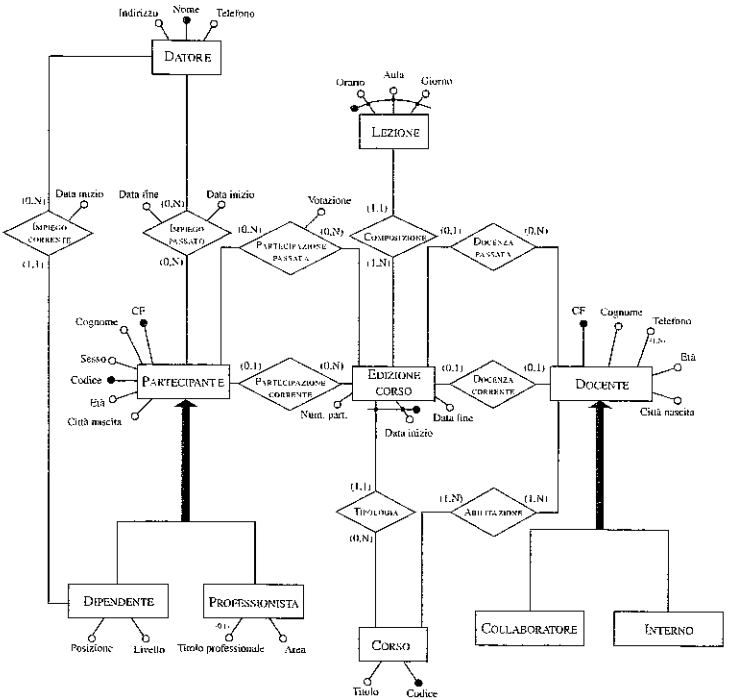

Figura 7.29 Lo schema E-R finale per la società di formazione.

Va infine ricordato che lo schema va corredato con un'opportuna documentazione che va prodotta parallelamente alla costruzione dello schema. In particolare, è importante descrivere, per esempio sotto forma di regole aziendali, eventuali vincoli non espressi direttamente dallo schema. Per esempio, il fatto che un docente può insegnare (o aver insegnato) un corso solo se è abilitato a farlo.

7.7 Strumenti CASE per la progettazione di basi di dati

La progettazione di basi di dati è un'attività complessa che è spesso difficile o addirittura impossibile svolgere manualmente. Questa attività può essere resa più produttiva facendo uso di programmi di editing dotati di interfacce grafiche per

gestire tabelle e diagrammi, ma esistono anche in commercio pacchetti applicativi dedicati proprio al progetto e allo sviluppo di basi di dati. Questi sistemi appartengono alla categoria degli strumenti CASE (Computer Aided Software Engineering) di ausilio all'ingegnerizzazione del software e forniscono un supporto a tutte le fasi principali dello sviluppo di una base di dati (progettazione concettuale, logica e fisica).

Le funzionalità offerte variano parecchio da un prodotto a un altro, ma esistono alcune componenti di base che, in forma più o meno esplicita, sono presenti in tutti i sistemi:

- un'*interfaccia grafica* con la quale è possibile manipolare direttamente schemi Entità-Relazione rappresentati in forma diagrammatica;
- un *dizionario dei dati* centralizzato che memorizza informazioni sui vari concetti dello schema (entità, attributi, relazioni, vincoli di integrità ecc.);
- una serie di *strumenti integrati* che eseguono, in maniera automatica o attraverso un'interazione con l'utente, compiti specifici della progettazione (layout automatico di diagrammi, verifiche di correttezza e di completezza, analisi di qualità di uno schema, produzione automatica di codice per la realizzazione della base di dati ecc.).

Molti sistemi sono integrabili direttamente con sistemi di gestione di basi di dati. Altri sistemi forniscono un supporto anche all'attività di analisi dei requisiti. Altri ancora mettono a disposizione anche delle librerie di progetti generici predefiniti o sviluppati precedentemente che possono essere utilizzati come punto di partenza per un nuovo progetto.

Per quel che riguarda specificatamente la progettazione concettuale, risulta generalmente possibile utilizzare le strategie proposte nei precedenti paragrafi anche quando si usano questi sistemi. Molti di essi permettono infatti di procedere in maniera top-down definendo solo parzialmente certi concetti dello schema per poi raffinarli successivamente. Per esempio, si può definire un'entità senza specificare attributi e identificatori. Altri sistemi consentono inoltre di definire e manipolare separatamente viste, ossia porzioni di uno schema di base, propagando automaticamente nello schema di base modifiche fatte sugli schemi derivati. Questo consente di procedere in maniera bottom-up. Un semplice esempio di prodotto della fase di progettazione concettuale fatta con uno strumento di questo genere è riportato in Figura 7.30.

Il sistema usato è un noto strumento per ambienti Microsoft Windows chiamato ER-Win. C'è da notare che questo sistema utilizza una notazione particolare per descrivere i costrutti del modello E-R, diversa da quella usata in questo capitolo, denominata IDEF1-X. In particolare, gli attributi vengono rappresentati direttamente dentro le entità, separando gli identificatori dagli altri attributi. Le linee rappresentano relazioni e particolari simboli sulle linee vengono usati per esprimere vincoli di cardinalità. Le generalizzazioni sono rappresentate da linee separate da un simbolo speciale (relazione tra IMPIEGATO e DIRETTORE). La notazione non permette di assegnare attributi alle relazioni.

Un altro esempio di uso di strumento CASE per il progetto di basi di dati viene riportato in Figura 7.31.

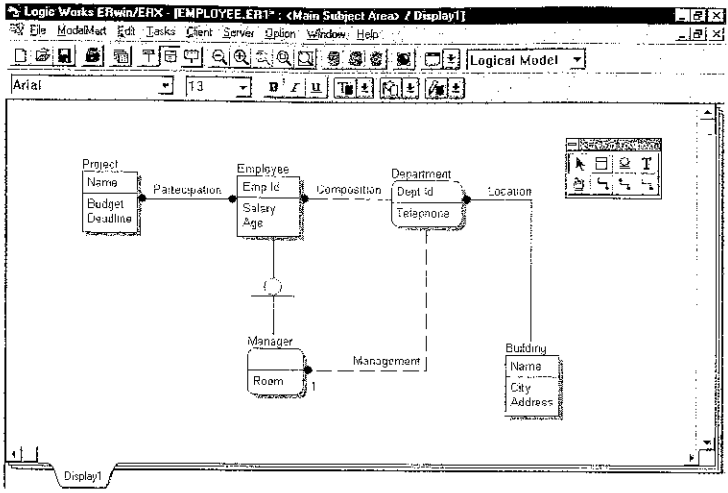

Figura 7.30 Progettazione concettuale fatta con ER-Win.

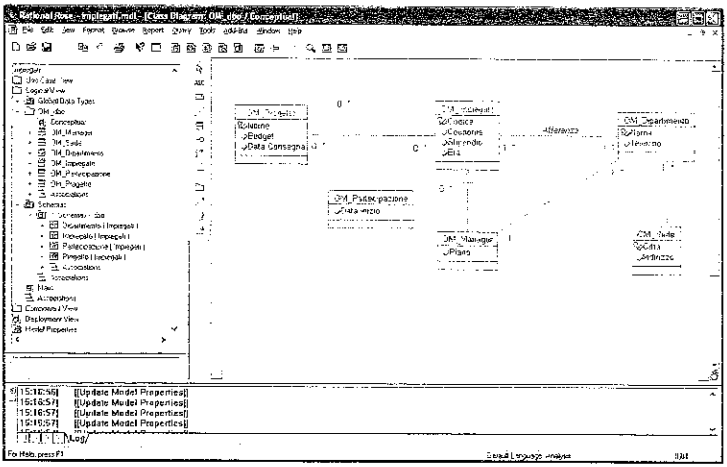

Figura 7.31 Progettazione concettuale fatta con Rational.

È stato usato in questo caso Rational, un noto strumento di ausilio alla progettazione del software che si basa sul linguaggio UML. In base a quanto abbiamo detto nel Paragrafo 6.4, questo linguaggio viene talvolta usato anche per la progettazione concettuale dei dati.

I due esempi fatti ci mostrano un problema classico che si deve affrontare quando si lavora con uno strumento CASE per progettare una basi di dati: non esistono di fatto standardizzazioni sulle notazioni usate, in quanto tutti i sistemi adottano un modello concettuale noto, ma di fatto in una versione personalizzata. È quindi spesso necessario uno sforzo del progettista per adattare le proprie conoscenze di modelli e metodologie alle caratteristiche del sistema scelto.

Note bibliografiche

La progettazione concettuale dei dati è affrontata in dettaglio nel libro in italiano di Batini *et al.* [9] e nel testo in inglese di Batini, Ceri e Navathe [8] che affronta anche, in maniera sistematica, il problema dell'integrazione di schemi. Esistono inoltre due testi molto interessanti sul progetto DATAID, che ha affrontato diversi aspetti legati al progetto di basi dati, tra cui la fase di raccolta e analisi dei requisiti [4, 19]. La nostra descrizione di questa fase si basa sui risultati di questo progetto.

Una rassegna approfondita scritta da David Reiner sugli strumenti CASE per il progetto di basi di dati è riportata nel Capitolo 15 del testo [8].

Come testi di esercizi sulla progettazione di basi di dati suggeriamo il testo di Cabibbo, Torlone e Batini [16], quello di Francalanci, Schreiber e Tanca [44] e quello di Maio e Rizzi [57].

Esercizi

Soluzioni sul sito http://www.ateneonline.it/atzeni

7.1 Si desidera automatizzare il sistema di prestiti di una biblioteca. Le specifiche del sistema, acquisite attraverso un'intervista con il bibliotecario, sono quelle riportate in Figura 7.32. Analizzare tali specifiche, filtrare le ambiguità presenti e poi raggrupparle in modo omogeneo. Prestare particolare attenzione alla differenza esistente tra il concetto di *libro* e di *copia* di libro. Individuare i collegamenti esistenti tra i vari gruppi di specifiche così ottenuti.

7.2 Rappresentare le specifiche dell'esercizio precedente (dopo la fase di riorganizzazione) con uno schema del modello Entità-Relazione.

7.3 Definire uno schema Entità-Relazione che descriva i dati di un'applicazione relativa a una catena di officine. Sono d'interesse le seguenti informazioni.

- Le officine, con nome (identificante), indirizzo e telefono.
- Le automobili, con targa (identificante) e modello (una stringa di caratteri senza ulteriore struttura) e proprietario.
- I clienti (proprietari di automobili), con codice fiscale, cognome, nome e telefono. Ogni cliente può essere proprietario di più automobili.

Biblioteche
I lettori che frequentano la biblioteca hanno una tessera su cui è scritto il nome e l'indirizzo ed effettuano richieste di prestito per i libri che sono catalogati nella biblioteca. I libri hanno un titolo, una lista di autori e possono esistere in diverse copie. Tutti i libri contenuti nella biblioteca sono identificati da un codice. A seguito di una richiesta, viene dapprima consultato l'archivio dei libri disponibili (cioè non in prestito). Se il libro è disponibile, si procede alla ricerca del volume negli scaffali; il testo viene poi classificato come in prestito. Acquisito il volume, viene consegnato al lettore, che procede alla consultazione. Terminata la consultazione, il libro viene restituito, reinserito in biblioteca e nuovamente classificato come disponibile. Per un prestito si tiene nota degli orari e delle date di acquisizione e di riconsegna.

Figura 7.32 Specifiche per l'Esercizio 7.1.

- Gli "interventi" di manutenzione, ognuno effettuato presso un'officina e con un numero progressivo (unico nell'ambito della singola officina), date di inizio e di fine, pezzi di ricambio utilizzati (con le rispettive quantità) e numero di ore di mano d'opera.
- I pezzi di ricambio, con codice, nome e costo unitario.

Indicare le cardinalità delle relazioni e (almeno) un identificatore per ciascuna entità.

7.4 Nella costruzione di uno schema concettuale di buona qualità vanno garantite alcune proprietà generali. Vanno memorizzate:

- informazioni sui cittadini nati nel comune e su quelli residenti in esso; ogni cittadino è identificato dal codice fiscale e ha cognome, nome, sesso e data di nascita; inoltre
  - per i nati nel comune, sono registrati anche gli estremi di registrazione (numero del registro e pagina);
  - per i nati in altri comuni, è registrato il comune di nascita;
- informazioni sulle famiglie residenti, ognuna delle quali ha uno e un solo capofamiglia e zero o più altri membri, per ognuno dei quali è indicato (con una sigla) il grado di parentela (coniuge, figlio, genitore, o altro); ogni cittadino residente appartiene a una e una sola famiglia; tutti i membri di una famiglia hanno lo stesso domicilio (via, numero civico, interno).

Cercare di procedere secondo la strategia inside-out. Al termine, verificare le qualità dello schema ottenuto.

7.5 Analizzare le specifiche relative a partite di un campionato riportate in Figura 7.33 e costruire un glossario dei termini a esse relativo.

7.6 Dopo aver riorganizzato in gruppi omogenei le specifiche dell'esercizio precedente, rappresentarle con il modello Entità-Relazione, procedendo in maniera top-down per livelli di astrazione successiva a partire da uno schema scheletro iniziale. Si osservi che lo schema in Figura 6.37 rappresenta una possibile soluzione di questo esercizio.

Campionato di calcio

Per ogni partita, descrivere il girone e la giornata in cui si è svolta, il numero progressivo nella giornata (per esempio prima partita, seconda partita ecc.), la data, con giorno, mese, anno, le squadre coinvolte nella partita, con nome, città della squadra e allenatore, e infine per ciascuna squadra se ha giocato in casa. Si vogliono conoscere i giocatori che giocano in ogni squadra con i loro nomi e cognomi, la loro data di nascita e il loro ruolo principale. Si vuole conoscere per ogni giornata, quanti punti ha ogni squadra. Si vogliono anche conoscere, per ogni partita, i giocatori che hanno giocato, i ruoli di ogni giocatore (i ruoli dei giocatori possono cambiare di partita in partita) e nome, cognome, città e regione di nascita dell'arbitro della partita. Distinguere le partite giocate regolarmente da quelle rinviate. Per quelle rinviate, rappresentare la data in cui si sono effettivamente giocate. Distinguere anche le partite giocate in una città diversa da quella della squadra ospitante; per queste si vuole rappresentare la città in cui si svolgono, nonché il motivo della variazione di sede. Dei giocatori interessa anche la città di nascita.

Figura 7.33 Specifiche per l'Esercizio 7.5.

7.7 Provare a rappresentare di nuovo le specifiche dell'Esercizio 7.5 con uno schema Entità-Relazione, procedendo però in maniera bottom-up: costruire frammenti di schema separati che descrivono le varie componenti omogenee delle specifiche e poi procedere per integrazione dei vari schemi. Confrontare il risultato con lo schema ottenuto nell'Esercizio 7.6.

7.8 Si vuole effettuare un'operazione di *reverse-engineering*, ovvero si vuole ricostruire, a partire da una base di dati relazionale, una sua rappresentazione concettuale con il modello Entità-Relazione. La base di dati è relativa a un'applicazione su treni e stazioni ferroviarie ed è composta dalle seguenti relazioni:

- STAZIONE(Codice,Nome,Città), con il vincolo di integrità referenziale fra l'attributo Città e la relazione CITTÀ;
- CITTÀ(Codice,Nome,Regione);
- TRATTA(Da,A,Distanza), con vincoli di integrità referenziale tra l'attributo Da e la relazione STAZIONE e tra l'attributo A e la relazione STAZIONE; questa relazione contiene tutte e sole le coppie di stazioni connesse da una linea in modo diretto (cioè senza stazioni intermedie);
- ORARIOTRENI(Numero,Da,A,OrarioDiPartenza, OrarioDiArrivo) con vincoli di integrità referenziale tra l'attributo Da e la relazione STAZIONE e tra l'attributo A e la relazione STAZIONE;
- TRATTETRENO(NumeroTreno,Da,A) con vincoli di integrità referenziale tra l'attributo NumeroTreno e la relazione ORARIOTRENI e tra gli attributi Da e A e la relazione TRATTA;
- ORARIOFERMATE(NumeroTreno,Stazione, Arrivo,Partenza) con vincoli di integrità referenziale tra l'attributo NumeroTreno e la relazione ORARIOTRENI e tra l'attributo Stazione e la relazione STAZIONE;

- TRENOREALE(Numero,Data, OrarioDiPartenza,OrarioDiArrivo) con il vincolo di integrità referenziale tra l'attributo Numero e la relazione ORARIOTRENI;
- FERMATEREALI(NumeroTreno,Data, Stazione,Arrivo,Partenza) con il vincolo di integrità referenziale tra gli attributi NumeroTreno e Stazione e la relazione ORARIOFERMATE.

Segnalare eventuali ridondanze. In particolare, qualora si tratti di relazioni derivate.

7.9 Definire uno schema Entità-Relazione che descriva i dati di un'applicazione relativa a un reparto ospedaliero. Sono d'interesse le seguenti informazioni.

- I pazienti, con codice fiscale, nome, cognome, data di nascita.
- I ricoveri dei pazienti, ognuno con data di inizio (identificante nell'ambito dei ricoveri di ciascun paziente) e medico curante; inoltre, per i ricoveri conclusi, la data di conclusione e la motivazione (dimissione, trasferimento ecc.), e, per i ricoveri in corso, il recapito di un parente (che si può assumere sia semplicemente una stringa).
- I medici, con un numero di matricola, cognome, nome e data di laurea.
- Le visite, con la data, l'ora, i medici visitanti, le medicine prescritte (con le relative quantità) e le malattie diagnosticate; ogni visita è identificata dal paziente coinvolto, dalla data e dall'ora.
- Per ogni medicina sono rilevanti un codice identificativo, un nome e un costo.
- Per ogni malattia sono rilevanti un codice identificativo e un nome.

7.10 Definire uno schema Entità-Relazione che descriva i dati di un'applicazione relativa all'archivio di un amministratore di condomini, secondo le seguenti specifiche (semplificate rispetto a molte realtà).

- Ogni condominio ha un nome (che lo identifica) e un indirizzo e comprende una o più *scale*, ognuna delle quali comprende un insieme di appartamenti.
- Se il condominio comprende più scale, a ogni scala sono associati:
  - un codice (es: scala "A") che la identifica insieme al nome del condominio;
  - un valore, detto *quota della scala*, che rappresenta, in millesimi, la frazione delle spese del condominio che sono complessivamente di competenza degli appartamenti compresi nella scala.
- Ogni appartamento è identificato, nel rispettivo condominio, dalla scala (se esiste) e da un numero (l'*interno*). A ogni appartamento è associata una quota (ancora espressa in millesimi) che indica la frazione delle spese (della scala) che sono di competenza dell'appartamento.
- Ogni appartamento ha un proprietario per il quale sono d'interesse il nome, il cognome, il codice fiscale e l'indirizzo al quale deve essere inviata la corrispondenza relativa all'appartamento. Ogni persona ha un solo codice fiscale, ma potendo essere proprietario di più appartamenti, potrebbe anche avere indirizzi diversi per appartamenti diversi. Di solito, anche chi è proprietario di molti appartamenti ha comunque solo uno o pochi indirizzi. In molti casi, l'indirizzo del proprietario coincide con quello del condominio.
- Per la parte contabile, è necessario tenere traccia delle spese sostenute dal condominio e dei pagamenti effettuati dai proprietari.
  - Ogni spesa è associata a un intero condominio, oppure a una scala o a un singolo appartamento.

- Ogni pagamento è relativo a uno e un solo appartamento.

Nella base di dati vengono mantenuti pagamenti e spese relativi all'esercizio finanziario in corso (di durata annuale) mentre gli esercizi precedenti vengono sintetizzati attraverso un singolo valore (il *saldo precedente*) per ciascun appartamento che indica il debito o il credito del proprietario. In ogni istante esiste un *saldo corrente* per ciascun appartamento, definito come somma algebrica del saldo precedente e dei pagamenti (positivi) e delle spese addebitate (negative).

Se e quando lo si ritiene opportuno, introdurre codici identificativi sintetici.

7.11 In Figura 7.34 è mostrata una schematizzazione dei programmi di una stagione dei diversi teatri di una città. Con riferimento a essa:

1. definire uno schema concettuale (nel modello ER) che descriva la realtà di interesse; limitarsi agli aspetti che vengono espressamente mostrati, introducendo tutt'al più, ove lo si ritenga necessario, opportuni codici identificativi; mostrare le cardinalità delle relazioni e gli identificatori delle entità;
2. progettare lo schema logico relazionale corrispondente allo schema concettuale definito al punto precedente, mostrando i nomi delle relazioni, quelli degli attributi e i vincoli di chiave e di integrità referenziale;
3. mostrare un'istanza della base di dati progettata al punto precedente, utilizzando i dati nell'esempio (o anche parte di essi, purché si riescano a mostrare gli aspetti significativi).

Osservazione: le risposte ai punti 2 e 3 richiedono lo studio del capitolo successivo, ma sono utili per verificare la correttezza della risposta al punto 1 e quindi la domanda viene proposta qui.

7.12 Mostrare lo schema concettuale di una base di dati per tornei di calcio, secondo le seguenti specifiche:

- i vari tornei hanno codice e nome;
- ogni torneo è composto da un certo numero di squadre e da una classifica che assegna a ogni squadra un punteggio;
- le squadre partecipano a un solo torneo e hanno un nome e una rosa di giocatori di cui registriamo il numero di maglia, il nome e la data di nascita;
- le partite si svolgono tra due squadre dello stesso torneo, in una certa data, in un certo stadio e hanno un risultato finale;
- si vogliono registrare i giocatori che giocano nelle varie partite e il ruolo ricoperto (che è lo stesso in una partita ma può variare in partite diverse).

Indicare gli eventuali vincoli di integrità che non è possibile rappresentare nello schema.

7.13 Estendere lo schema concettuale ottenuto in risposta alla domanda precedente, per tenere conto delle seguenti specifiche aggiuntive (mostrare separatamente i due frammenti di schema necessari per rappresentare le modifiche).

1. È di interesse rappresentare l'evoluzione temporale della classifica (una squadra può avere due punti un certo giorno e quattro in un altro).
2. I tornei si ripetono negli anni e ogni squadra partecipa a un torneo all'anno, con giocatori eventualmente diversi.

LA STAGIONE TEATRALE IN CITTÀ

Teatro Comunale
Via Roma, 25 Tel: 6555432

Prezzi:

	Prime	Sab e Dom	Feriale
Platea	85	70	40
Palchi	70	50	30
Loggione	30	25	15

Riduzioni:

- studenti 20%
- CRAL 10%

Spettacoli:

- **Così è (se vi pare) (1917)**
  L.Pirandello (1867-1936)
  dal 05.10.2005 al 21.11.2005
- **L'opera da tre soldi (1928)**
  B.Brecht (1967-1836)
  dal 25.11.2005 al 17.12.2005
- ...

Teatro Cittadino Piazza del municipio, 32 Tel: 6535455

Prezzi:

	Prime	Sabato sera	Domenica	Altri
Platea	90	70	60	50
Galleria	60	40	50	30

Riduzioni:

- studenti 20%
- insegnanti 20%
- gruppi 10%

Spettacoli:

- **Enrico IV (1921)**
  L.Pirandello (1867-1936)
  dal 6.10.2005 al 5.11.2005
- **Uno sguardo dal ponte (1955)**
  A.Miller (1915-2005)
  dal 7.11.2005 al 9.12.2005
- **Così è (se vi pare) (1917)**
  L.Pirandello (1867-1936)
  dal 5.01.2006 al 7.02.2006
- seguono altri spettacoli
  ...

Teatro Nuovo ...
...

Figura 7.34 Le informazioni da modellare per l'Esercizio 7.11.

7.14 Si consideri la seguente schematizzazione di alcune prenotazioni aeree:

```
Prenotazione N. 1270
   Passeggeri
   Mario  Rossi   (Cod.1230)   Tel.   06/45531123
   Lucia  Neri    (Cod.1231)   Tel.   06/64352134
   Piero  Rossi   (Cod.1232)
   Itinerario
      Da    A      Data        Ora    NumeroVolo  Aeromobile Classe
   1. FCO   LHR  11/03/2008   07:50    AZ024        A321       V
   2. LHR   MAN  11/03/2008   11:30    BA233        M80X       F
   3. LHR   FCO  18/03/2008   11:50    AZ175        A320       C

Prenotazione N.1343
   Passeggeri
   Giulio  Rossi   (Cod.1343)   Tel.   06/45521123
   Itinerario
      Da    A      Data        Ora    NumeroVolo  Aeromobile Classe
   1. FCO   LHR  12/04/2008   08:20    AZ024        A321       G
   2. LHR   FCO  21/04/2008   13:50    AZ175        A320       C

Prenotazione N.1777
   Passeggeri
   Mario  Rossi   (Cod.1230)   Tel.   06/45521123
   Itinerario
      Da    A      Data        Ora    NumeroVolo  Aeromobile Classe
   1. FCO   LHR  12/04/2008   08:20    AZ024        A321       G
   2. LHR   FCO  21/04/2008   13:50    AZ175        A320       C
```

Si tenga conto a riguardo delle seguenti precisazioni:

- per ogni passeggero esistono codice (identificativo), cognome, nome e numero di telefono che è opzionale ed è lo stesso in tutte le prenotazioni;
- le colonne Da e A contengono codici di aeroporti, per i quali sono memorizzati anche il nome e la città (per esempio, a "FCO" sono associati "Fiumicino" come nome e "Roma" come città);
- il numero del volo (per esempio "AZ024") è costituito dal codice della compagnia (per la quale interessa anche il nome; per esempio "AZ" è il codice della compagnia il cui nome è "Alitalia") e da un intero;
- un volo con un certo NumeroVolo ha sempre gli stessi aeroporti di partenza e di arrivo (Da e A) e lo stesso tipo di aeromobile (colonna Aeromobile), ma può avere orario diverso in date diverse; per il tipo di aeromobile al codice (mostrato nella scheda, per esempio "A321") è associato un nome (nell'esempio potrebbe essere "Airbus 321");
- la colonna Classe contiene un codice (della "classe di prenotazione") che, come si vede dai dati, è specificatamente associato a volo e prenotazione; per ogni valore di tale codice è memorizzata una descrizione.

Con riferimento alla corrispondente realtà definire uno schema concettuale che la descriva limitandosi agli aspetti che vengono citati e mostrando sia le cardinalità delle relazioni sia gli identificatori delle entità.

7.15 Mostrare lo schema concettuale per una base di dati per un programma di concerti, secondo le specifiche seguenti.

- Ogni concerto ha un codice, un titolo e una descrizione ed è composto da una sequenza (ordinata) di pezzi musicali.
- Ogni pezzo ha un codice, un titolo e un autore (con codice e nome); uno stesso pezzo può essere rappresentato in diversi concerti.
- Ogni concerto è eseguito da un'orchestra: ogni orchestra ha un nome, un direttore (del quale interessano solo nome e cognome) e un insieme di orchestrali.
- Ogni orchestrale ha una matricola (univoca nell'ambito della base di dati), nome e cognome, può partecipare a più orchestre, in ciascuna delle quali suona uno e un solo strumento, ma in orchestre diverse può suonare strumenti diversi.
- Ogni concerto è tenuto più volte, in giorni diversi, ma sempre nella stessa sala.
- Ogni sala ha un codice, un nome e una capienza.

7.16 In Figura 7.35 è mostrata una schematizzazione del catalogo dei viaggi di studio all'estero proposti da un operatore del settore. Con riferimento a essa definire uno schema concettuale (nel modello ER) che descriva la realtà d'interesse. Limitarsi agli aspetti che vengono espressamente mostrati, introducendo tutt'al più, ove lo si ritenga necessario, opportuni codici identificativi; mostrare le cardinalità delle relazioni e gli identificatori delle entità.

7.17 Definire uno schema E-R che descriva i dati di un'applicazione relativa alla gestione ed evasione degli ordini da parte di un'azienda, secondo le seguenti specifiche.

- L'azienda riceve gli ordini emessi dai clienti (ognuno dei quali ha numero di partita IVA, che identifica ragione sociale, indirizzo e percentuale di sconto). Ogni ordine ha un numero (attribuito dal cliente), indica il nome di un referente interno del cliente che può essere lo stesso per tutti gli ordini) e richiede uno o più prodotti, per ciascuno dei quali indica una quantità e una sede di destinazione (in quanto ciascun cliente può, anche nell'ambito di uno stesso ordine, richiedere che i vari prodotti siano consegnati in sedi diverse; per esempio: "tre calcolatori X386, due stampanti Z322 a via Roma 103 e due calcolatori X343 e una stampante Z320 a Corso Garibaldi 12". A ogni ordine viene assegnato, dall'azienda, all'atto della ricezione, un numero progressivo identificante. Ogni sede di destinazione viene rappresentata da un codice e un indirizzo e non ha correlazione formale con il cliente.
- Gli ordini vengono evasi attraverso consegne, ognuna delle quali è relativa a un unico cliente e un'unica sede di destinazione, ma può riferirsi a più ordini. Ogni ordine, a sua volta, è soddisfatto attraverso una o più consegne. Per ogni consegna sono rilevanti la data, l'ora e il numero di bolla di accompagnamento. L'azienda ha vari mezzi di trasporto (identificati ognuno da un codice e senza ulteriori proprietà di interesse), ognuno dei quali effettua al più un giro di consegne al giorno, per il quale è d'interesse l'ora di uscita dal magazzino.
- Ogni prodotto ha un codice identificante, un nome e un prezzo unitario. I prodotti si dividono in due categorie: inventariabili (per i quali ciascun esemplare ha un numero di matricola di cui si deve tenere traccia nell'ambito della consegna) e di consumo (per i quali è sufficiente far riferimento alle quantità).

Cambridge *Aeroporto*: Heathrow *Esame*: PET

King's College – 101 King's Street – Tel: +44 123 6667777
15 ore di lezione a settimana

Periodo	Prezzo
1/7-15/07/2010	1500
15/7-29/7/2010	1700
29/7-13/8/2010	1650

Sconto seconda quindicina 10%
Sconto gruppi 15%

Queen's College – 1021 Queen's Road – Tel: +44 123 7665433
20 ore di lezione a settimana

Periodo	Prezzo
5/7-19/07/2010	1400
19/7-2/8/2010	1600
...	...

Sconto seconda quindicina 10%
Sconto fratelli 8%

Oxford *Aeroporto*: Heathrow *Esame*: Trinity

Prince College — 1021 St.John's Road — Tel: +44 125 6765443
18 ore di lezione alla settimana

Periodo	Prezzo
4/7-18/07/2010	1200
...	...

Sconto gruppi 10%
seguono altri college

Stirling *Aeroporto*: Edimburgo *Esame*: Trinity
seguono altri college

---

seguono altre località

Informazioni generali, per tutte le località
Esami:
Pet: 30 euro
Trinity: 35 Euro

Voli:

	Heatrow	Edimburgo	Dublino
Roma	450	600	500
Milano	400	550	430
Palermo	550	700	650

Figura 7.35 Le informazioni da modellare per l'Esercizio 7.16.

- Quando un ordine è stato completamente evaso, viene emessa la fattura, che ha un numero progressivo, una data e un importo.

Indicare le cardinalità delle relazioni, (almeno) un identificatore per ciascuna entità e i vincoli non esprimibili per mezzo dello schema. Indicare se è necessario formulare delle ipotesi aggiuntive alle specifiche descritte, senza contraddirle. Limitare le relazioni ridondanti che secondo le specifiche potrebbero essere presenti: per esempio, è evidente che i prodotti sono associati agli ordini e alle consegne (compaiono su vari documenti, ordini, bolle e fatture) ma si richiede di rappresentare tutti i concetti di interesse senza ripeterli. Specificatamente si può pensare di associare i prodotti agli ordini solo inizialmente, per poi associarli alle consegne che, essendo comunque legate agli ordini stessi, permettono di ricostruire l'informazione originaria; per le stesse ragioni, è inopportuno associare i prodotti alle fatture (anche se sulla fattura sono elencati, ma è possibile ricostruire l'elenco per altra via).

7.18 Definire uno schema E-R che descriva informazioni relative a sale cinematografiche di una città, secondo le seguenti specifiche.

- Ogni cinema ha un nome che lo identifica univocamente, un indirizzo e un numero di telefono. Un cinema è organizzato in più sale, ognuna delle quali ha un codice che la distingue (nell'ambito del cinema) e un numero fissato di posti.
- Per ogni sala interessa la programmazione di una sola giornata (quella odierna, senza traccia di quelle passate e future) che consiste in un elenco di proiezioni di film (eventualmente anche diversi), ognuna delle quali ha un orario di inizio.
- Per ogni film si registrano il titolo, il genere (codice e nome descrittivo), la nazionalità (una semplice stringa) e il regista (con codice identificativo, nome, cognome e anno di nascita).

7.19 Estendere lo schema concettuale proposto in risposta alla domanda precedente per rappresentare anche le seguenti specifiche:

- i posti di ciascuna sala sono numerati;
- per ogni proiezione è possibile effettuare prenotazioni, ognuna delle quali ha un codice identificativo, un nominativo e un insieme di posti.

7.20 Mostrare lo schema concettuale di una base di dati per la gestione di articoli di una rivista scientifica secondo le seguenti specifiche.

- Gli articoli hanno un titolo, un sottotitolo, uno o più autori e un testo (una stringa molto grande, ma comunque gestibile).
- Gli autori hanno nome, cognome, e-mail e affiliazione (l'istituzione per la quale lavorano).
- Per ogni istituzione (degli autori) sono d'interesse il nome, l'indirizzo e la nazione.
- La rivista viene pubblicata un certo numero di volte in un anno. Le pubblicazioni di un anno vengono raccolte in un volume (a cui viene dato un titolo complessivo). Ogni pubblicazione ha un numero, unico nel rispettivo volume, una data di pubblicazione e una serie di articoli, per ognuno dei quali viene registrata la pagina di inizio e quella di fine.

7.21 Estendere lo schema concettuale ottenuto in risposta alla domanda precedente, per rappresentare l'attività di selezione degli articoli, sulla base delle seguenti specifiche aggiuntive.

- La rivista riceve proposte, per le quali sono d'interesse le stesse informazioni registrate per gli articoli, oltre che la data di presentazione.
- Ogni proposta viene revisionata da due o più esperti (per i quali sono d'interesse le stesse informazioni degli autori; si noti che gli autori possono essere esperti e viceversa, ma ovviamente un esperto non può revisionare una propria proposta), che assegnano alla proposta un punteggio tra 0 e 10 e forniscono un commento (un semplice testo).
- Le proposte che ricevono un punteggio medio superiore a 7 diventano articoli da pubblicare e hanno a quel punto una data di accettazione.

Indicare gli eventuali vincoli di integrità che non sia possibile rappresentare nello schema.

La progettazione logica

L'obiettivo della progettazione logica è quello di costruire uno schema logico in grado di descrivere, in maniera corretta ed efficiente, tutte le informazioni contenute nello schema Entità-Relazione prodotto nella fase di progettazione concettuale. Diciamo subito che non si tratta di una semplice traduzione da un modello a un altro perché, prima di passare allo schema logico, lo schema Entità-Relazione va ristrutturato per soddisfare due esigenze: quella di "semplificare" la traduzione e quella di "ottimizzare" il progetto. La semplificazione dello schema si rende necessaria perché non tutti i costrutti del modello Entità-Relazione hanno una traduzione naturale nei modelli logici. Per esempio, mentre un'entità può essere facilmente rappresentata da una relazione del modello relazionale (avente gli stessi attributi dell'entità), per le generalizzazioni esistono varie alternative. Inoltre, mentre la progettazione concettuale ha come obiettivo la rappresentazione accurata e naturale dei dati d'interesse dal punto di vista del significato che hanno nell'applicazione, la progettazione logica costituisce la base per l'effettiva realizzazione dell'applicazione e deve tenere conto, per quanto possibile, delle sue prestazioni: questa necessità può portare a una ristrutturazione dello schema concettuale che renda più efficiente l'esecuzione delle operazioni previste. Pertanto, è necessario prevedere sia un'attività di *riorganizzazione*, sia un'attività di *traduzione* (dal modello concettuale a quello logico). Nel resto di questo capitolo, dopo un breve inquadramento metodologico, presenteremo separatamente queste due attività. Come premessa parleremo degli strumenti e delle tecniche che si possono usare per analizzare le prestazioni di una base di dati facendo riferimento al suo schema concettuale.

8.1 Fasi della progettazione logica

Le attività principali della progettazione logica sono la riorganizzazione dello schema concettuale e la traduzione in un modello logico. Poiché la riorganizzazione può essere in buona misura discussa indipendentemente dal modello logico, è utile di solito articolare la progettazione logica in due fasi, come schematizzato in Figura 8.1.

- **Ristrutturazione dello schema Entità-Relazione**: è una fase indipendente dal modello logico scelto e si basa su criteri di ottimizzazione dello schema e di semplificazione della fase successiva.
- **Traduzione verso il modello logico**: fa riferimento a uno specifico modello logico (nel nostro caso il modello relazionale) e può includere una ulteriore ottimizzazione che si basa sulle caratteristiche del modello logico stesso.

I dati di ingresso della prima fase sono lo schema concettuale prodotto nella fase precedente e il *carico applicativo* previsto, in termini di dimensione dei dati e caratteristiche delle operazioni. Il risultato che si ottiene è uno schema E-R ristrutturato, che non è

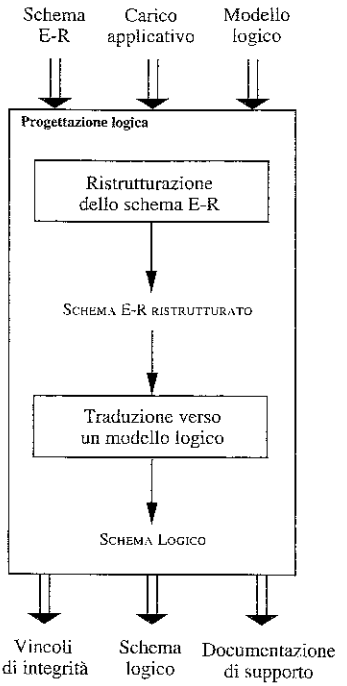

Figura 8.1 Progettazione logica di basi di dati.

più uno schema concettuale nel senso stretto del termine, in quanto costituisce una rappresentazione dei dati che tiene conto degli aspetti realizzativi. Questo schema e il modello logico scelto costituiscono i dati di ingresso della seconda fase, che produce lo schema logico della nostra base di dati. In questa seconda fase è possibile effettuare verifiche della qualità dello schema ed eventuali ulteriori ottimizzazioni mediante tecniche basate sulle caratteristiche del modello logico. La tecnica usata nell'ambito del modello relazionale (la *normalizzazione*) verrà studiata separatamente nel Capitolo 9. Lo schema logico finale, i vincoli di integrità definiti su di esso e la relativa documentazione, costituiscono i prodotti finali della progettazione logica.

8.2 Analisi delle prestazioni su schemi E-R

Abbiamo detto che uno schema E-R può essere modificato per ottimizzare alcuni *indici di prestazione* del progetto. Parliamo di indici di prestazione e non di prestazioni

perché, in realtà, le prestazioni di una base di dati non sono valutabili in maniera precisa in sede di progettazione logica, in quanto dipendenti anche da parametri fisici, dal sistema di gestione di basi di dati che verrà utilizzato e da altri fattori difficilmente prevedibili in questa fase. È comunque possibile, facendo uso di alcune schematizzazioni, effettuare studi di massima dei due parametri che generalmente regolano le prestazioni dei sistemi software:

- **costo di una operazione**: viene valutato in termini di numero di occorrenze di entità e associazioni[1] che mediamente vanno visitate per rispondere a una operazione sulla base di dati; questa schematizzazione è molto forte e, pur nelle semplici valutazioni che svilupperemo, sarà talvolta necessario riferirci a un criterio più fine;
- **occupazione di memoria**: viene valutato in termini dello spazio di memoria (misurato per esempio in numero di byte) necessario per memorizzare i dati descritti dallo schema.

Per studiare questi parametri abbiamo bisogno di conoscere, oltre allo schema, le seguenti informazioni.

- **Volume dei dati**. Vale a dire:
  - numero di occorrenze di ogni entità e associazione dello schema;
  - dimensioni di ciascun attributo (di entità o associazione).
- **Caratteristiche delle operazioni**. Vale a dire:
  - tipo dell'operazione (interattiva o *batch*);
  - frequenza (numero medio di esecuzioni in un certo intervallo di tempo);
  - dati coinvolti (entità e/o associazioni).

Per fare un esempio pratico, riprendiamo uno schema già incontrato che riportiamo, per comodità, in Figura 8.2. Trattandosi di uno schema riguardante dati sul personale di un'azienda, le operazioni possibili potrebbero essere quelle che seguono.

Operazione 1: assegna un impiegato a un progetto.
Operazione 2: trova i dati di un impiegato, del dipartimento nel quale lavora e dei progetti ai quali partecipa.
Operazione 3: trova i dati di tutti gli impiegati di un certo dipartimento.
Operazione 4: per ogni sede, trova i suoi dipartimenti con il cognome del direttore e l'elenco degli impiegati del dipartimento.

Sebbene un'analisi delle prestazioni che fa riferimento a un numero ristretto di operazioni può sembrare riduttiva rispetto al reale carico della base di dati, va notato che le operazioni sulle basi di dati seguono la cosiddetta regola "ottanta–venti". In base a questa regola, l'ottanta per cento del carico è generato dal venti per cento delle operazioni. Questo fatto ci consente di valutare adeguatamente il carico concentrandoci solo sulle operazioni principali previste.

[1] Per non generare confusione tra i due concetti, in tutto questo capitolo useremo sempre il termine *associazione*, per indicare una relazione del modello E-R, e *relazione*, per indicare una relazione del modello relazionale.

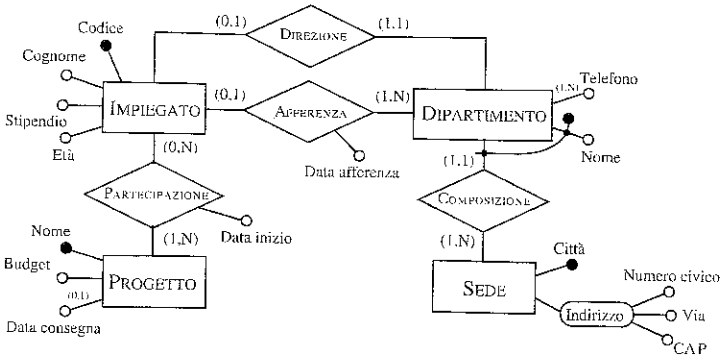

Figura 8.2 Uno schema E-R sul personale di un'azienda.

Il volume dei dati e le caratteristiche generali delle operazioni possono essere descritti facendo uso di tabelle come quelle in Figura 8.3. Nella *tavola dei volumi* vengono riportati tutti i concetti dello schema (entità e associazioni) con il volume previsto a regime. Nella *tavola delle operazioni* riportiamo, per ogni operazione, la frequenza prevista e un simbolo che indica se l'operazione è interattiva (I) o batch (B). Nella tavola dei volumi, il numero delle occorrenze delle associazioni dipende da due parametri: il numero di occorrenze delle entità coinvolte nelle associazioni e il numero (medio) di partecipazioni di una occorrenza di entità alle occorrenze di associazioni. Il secondo parametro dipende a sua volta dalle cardinalità delle asso-

Tavola dei volumi

Concetto	Tipo	Volume
Sede	E	10
Dipartimento	E	80
Impiegato	E	2000
Progetto	E	500
Composizione	R	80
Afferenza	R	1900
Direzione	R	80
Partecipazione	R	6000

Tavola delle operazioni

Operazione	Tipo	Frequenza
Op. 1	I	50 al giorno
Op. 2	I	100 al giorno
Op. 3	I	10 al giorno
Op. 4	B	2 a settimana

Figura 8.3 Esempio di tavole dei volumi e delle operazioni.

ciazioni. Per esempio, il numero di occorrenze dell'associazione COMPOSIZIONE è pari al numero dei dipartimenti, perché le cardinalità ci dicono che un dipartimento appartiene a una sola sede. Il numero di occorrenze dell'associazione AFFERENZA è invece poco meno del numero degli impiegati, perché dalle cardinalità si evince che ci sono impiegati che non afferiscono a nessun dipartimento. Infine, assumendo che un impiegato partecipa in media a tre progetti, abbiamo $2000 \times 3 = 6000$ occorrenze per l'associazione PARTECIPAZIONE (e quindi $6000/500 = 12$ impiegati in media su ogni progetto).

Per ogni operazione, possiamo inoltre descrivere graficamente i dati coinvolti con uno *schema di operazione* che consiste nel frammento dello schema E-R interessato dall'operazione, sul quale viene disegnato il "cammino logico" da percorrere per accedere alle informazioni d'interesse. Un esempio di schema di operazione viene proposto in Figura 8.4 con riferimento all'Operazione 2: per ottenere le informazioni d'interesse su un impiegato si parte dall'entità IMPIEGATO per accedere, attraverso l'associazione AFFERENZA, al suo dipartimento e, attraverso l'associazione PARTECIPAZIONE, ai progetti ai quali partecipa. Avendo a disposizione queste informazioni, è possibile fare una stima del costo di un'operazione sulla base di dati contando il numero di accessi alle occorrenze di entità e associazioni necessario per eseguire l'operazione. Consideriamo ancora l'Operazione 2: facendo riferimento allo schema di operazione dobbiamo innanzitutto accedere a una occorrenza dell'entità IMPIEGATO per accedere poi a una occorrenza dell'associazione AFFERENZA (infatti ogni impiegato afferisce al più a un dipartimento) e, attraverso questa, a una occorrenza dell'entità DIPARTIMENTO. Successivamente, per conoscere i dati dei progetti ai quali lavora, dobbiamo accedere a tre occorrenze dell'associazione PARTECIPAZIONE (perché abbiamo detto che in media un impiegato lavora su tre progetti) e, attraverso queste, a tre occorrenze dell'entità PROGETTO (per avere i dati sui progetti). Tutto questo può essere riassunto in una *tavola degli accessi* come quella riportata in Figura 8.5. Nell'ultima colonna di questa tabella viene riportato il tipo di accesso: L per

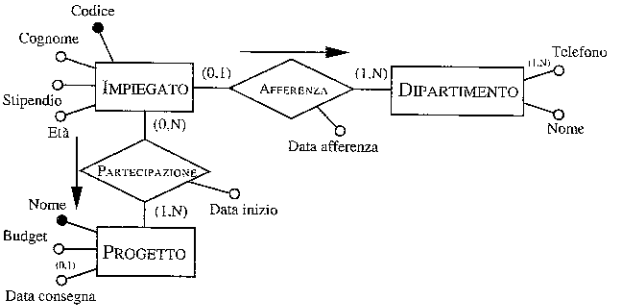

Figura 8.4 Esempio di schema di operazione.

Tavola degli accessi

Concetto	Costrutto	Accessi	Tipo
Impiegato	Entità	1	L
Afferenza	Relazione	1	L
Dipartimento	Entità	1	L
Partecipazione	Relazione	3	L
Progetto	Entità	3	L

Figura 8.5 Tavola degli accessi per l'operazione 2.

accesso in lettura e S per accesso in scrittura. Questa distinzione va fatta perché, generalmente, le operazioni di scrittura sono più onerose di quelle in lettura (in quanto devono essere eseguite in modo esclusivo e possono richiedere l'aggiornamento di *indici*, che sono strutture ausiliarie per l'accesso efficiente ai dati). Nel prossimo paragrafo vedremo come questi strumenti di analisi possono essere utilizzati per prendere delle decisioni durante la ristrutturazione di schemi Entità-Relazione.

8.3 Ristrutturazione di schemi E-R

La fase di ristrutturazione di uno schema Entità-Relazione si può suddividere in una serie di passi da effettuare in sequenza (Figura 8.6).

- **Analisi delle ridondanze**. Si decide se eliminare o mantenere eventuali ridondanze presenti nello schema.
- **Eliminazione delle generalizzazioni**. Tutte le generalizzazioni presenti nello schema vengono analizzate e sostituite da altri costrutti.
- **Partizionamento/accorpamento di entità e associazioni**. Si decide se è opportuno partizionare concetti dello schema (entità e/o associazioni) in più concetti o, viceversa, accorpare concetti separati in un unico concetto.
- **Scelta degli identificatori principali**. Si seleziona un identificatore per quelle entità che ne hanno più di uno.

Nel seguito del paragrafo, vedremo separatamente i vari passi di ristrutturazione attraverso degli esempi pratici.

8.3.1 Analisi delle ridondanze

Ricordiamo che una ridondanza in uno schema concettuale corrisponde alla presenza di un dato che può essere derivato (cioè ottenuto attraverso una serie di operazioni) da altri dati. In particolare, in uno schema Entità-Relazione si possono presentare varie forme di ridondanza. I casi più frequenti sono i seguenti.

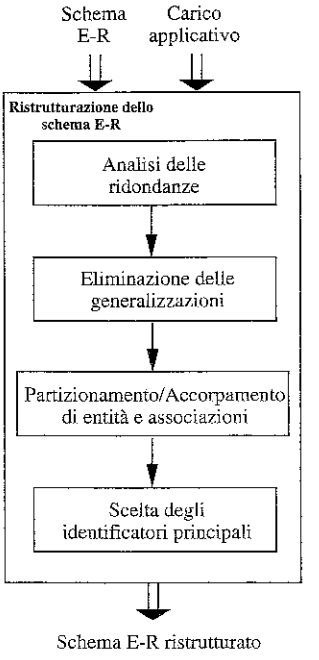

Figura 8.6 Fasi della ristrutturazione di uno schema E-R.

- Attributi derivabili, occorrenza per occorrenza, da altri attributi della stessa entità (o associazione). Per esempio, il primo schema in Figura 8.7 consiste in una entità FATTURA nella quale uno degli attributi è deducibile dagli altri attraverso una operazione di somma o differenza.
- Attributi derivabili da attributi di altre entità (o associazioni), di solito attraverso funzioni aggregative. Un esempio di ridondanza di questo tipo è presente nel secondo schema in Figura 8.7, nel quale l'attributo Importototale dell'entità ACQUISTO si può derivare, attraverso l'associazione COMPOSIZIONE, dall'attributo Prezzo dell'entità PRODOTTO, sommando i prezzi dei prodotti di cui un acquisto è composto.
- Attributi derivabili da operazioni di conteggio di occorrenze. Per esempio, nel terzo schema in Figura 8.7 l'attributo Numero di abitanti di una città può essere derivato contando le occorrenze della associazione RESIDENZA a cui tale città partecipa. Si tratta in effetti di una variante del caso precedente, che viene però discusso separatamente perché molto frequente in pratica.

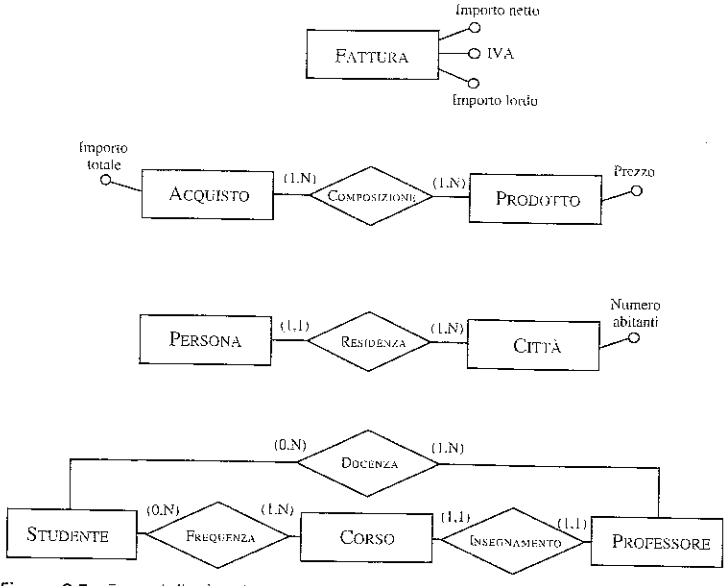

Figura 8.7 Esempi di schemi con ridondanze.

- Associazioni derivabili dalla composizione di altre associazioni in presenza di cicli. L'ultimo schema in Figura 8.7 contiene un esempio di ridondanza di questo tipo: l'associazione DOCENZA tra studenti e professori può essere infatti derivata dalle associazioni FREQUENZA e INSEGNAMENTO. Va comunque precisato che la presenza di cicli non genera necessariamente ridondanze. Se per esempio, al posto dell'associazione DOCENZA, ci fosse stata in questo schema una associazione TESI rappresentante il legame tra studente e relatori (concetto indipendente dal fatto che un professore è un docente dello studente) allora lo schema non sarebbe stato ridondante.

La presenza di un dato derivato presenta un vantaggio e alcuni svantaggi. Il vantaggio è una riduzione degli accessi necessari per calcolare il dato derivato, gli svantaggi sono una maggiore occupazione di memoria (che è comunque spesso un costo trascurabile) e la necessità di effettuare operazioni aggiuntive per mantenere il dato derivato aggiornato. La decisione di mantenere o eliminare una ridondanza va quindi presa confrontando costo di esecuzione delle operazioni che coinvolgono il dato ridondante e relativa occupazione di memoria, nei casi di presenza e assenza della ridondanza.

Vediamo, con un semplice esempio pratico, in che maniera gli strumenti di valutazione descritti nel paragrafo precedente possono essere usati per prendere una

Tavola dei volumi

Concetto	Tipo	Volume
Città	E	200
Persona	E	1 000 000
Residenza	R	1 000 000

Tavola delle operazioni

Operazione	Tipo	Frequenza
Op. 1	I	500 al giorno
Op. 2	I	2 al giorno

Figura 8.8 Tavole dei volumi e delle operazioni per lo schema in Figura 8.7 relativo a dati anagrafici.

decisione di questo tipo. Consideriamo lo schema su persone e città in Figura 8.7 e supponiamo che faccia riferimento a una applicazione anagrafica di una regione italiana per la quale sono definite le seguenti operazioni principali.

Operazione 1: memorizza una nuova persona con la relativa città di residenza.
Operazione 2: stampa tutti i dati di una città (incluso il numero di abitanti).

Supponiamo inoltre che per questa applicazione i dati di carico siano quelli riportati in Figura 8.8.

A questo punto proviamo a valutare gli indici di prestazione in caso di presenza del dato ridondante (attributo Numero abitanti nell'entità CITTÀ).

Assumendo che il numero degli abitanti di una città richieda 4 byte (ampiamente sufficienti per memorizzare interi di 7 cifre), abbiamo che il dato ridondante richiede $4 \times 200 = 800$ byte, ovvero meno di 1 kilobyte di memoria aggiuntiva. Passiamo ora alla stima del costo delle operazioni. Come descritto nella tavola degli accessi in Figura 8.9, l'Operazione 1 richiede un accesso in scrittura all'entità PERSONA (per memorizzare una nuova persona), un accesso in scrittura all'associazione RESIDENZA (per memorizzare una nuova coppia persona–città) e infine un accesso in lettura (per cercare la città d'interesse) e uno in scrittura (per incrementare di uno il numero degli abitanti di quella occorrenza) all'entità CITTÀ, il tutto ripetuto per 500 volte al giorno, per un totale di 1500 accessi in scrittura e 500 accessi in lettura. Il costo dell'operazione 2 è praticamente trascurabile perché richiede un solo accesso in lettura all'entità CITTÀ da ripetere due volte al giorno. Supponendo che un accesso in scrittura abbia un costo doppio rispetto a un accesso in lettura, abbiamo un totale di 3500 accessi al giorno in caso di presenza di dato ridondante.

Consideriamo ora il caso in cui il dato ridondante sia assente. Per l'Operazione 1 abbiamo bisogno di un accesso in scrittura all'entità PERSONA e un accesso in scrittura all'associazione RESIDENZA (non c'è infatti bisogno di accedere all'entità CITTÀ per aggiornare il dato derivato), per un totale di 1000 accessi in scrittura al giorno. Per l'Operazione 2 abbiamo invece bisogno di un accesso in lettura all'entità CITTÀ (per avere i dati della città), che possiamo trascurare, e di 5000 accessi in lettura all'associazione RESIDENZA in media (ottenuto dividendo il numero di persone per il numero di città) per calcolare il numero di abitanti di questa città, per un totale di 10 000 accessi in lettura al giorno. Contando doppi gli accessi in scrittura abbiamo

Tavole degli accessi in presenza di ridondanza

Operazione 1			
Concetto	**Costr.**	**Acc.**	**Tipo**
Persona	E	1	S
Residenza	R	1	S
Città	E	1	L
Città	E	1	S

Operazione 2			
Concetto	**Costr.**	**Acc.**	**Tipo**
Città	E	1	L

Tavole degli accessi in assenza di ridondanza

Operazione 1			
Concetto	**Costr.**	**Acc.**	**Tipo**
Persona	E	1	S
Residenza	R	1	S

Operazione 2			
Concetto	**Costr.**	**Acc.**	**Tipo**
Città	E	1	L
Residenza	R	5000	L

Figura 8.9 Tavole degli accessi per lo schema su dati anagrafici in Figura 8.7.

un totale di 12 000 accessi al giorno in caso di dato ridondante assente. Quindi, circa 8500 accessi giornalieri in più rispetto al caso di dato ridondante presente contro un risparmio di un solo kilobyte. Questo dipende dal fatto che gli accessi in lettura necessari per calcolare il dato derivato sono molti di più degli accessi in scrittura necessari per mantenerlo aggiornato. Possiamo quindi concludere che conviene, in questo caso, mantenere il dato ridondante.

8.3.2 Eliminazione delle generalizzazioni

Dato che i sistemi tradizionali per la gestione delle basi di dati non consentono di rappresentare direttamente una generalizzazione, risulta spesso necessario trasformare questo costrutto in altri costrutti del modello E-R per i quali esiste invece una implementazione naturale: le entità e le associazioni.

Per rappresentare una generalizzazione mediante entità e associazioni abbiamo essenzialmente tre alternative possibili. Per presentare queste alternative, faremo riferimento allo schema E-R generico in Figura 8.10 che contiene una generalizzazione e alcune associazioni tra entità.

I metodi per eliminare la generalizzazione di questo schema sono mostrati in Figura 8.11 e si ottengono attraverso le seguenti ristrutturazioni.

(1) **Accorpamento delle figlie della generalizzazione nel genitore**. Le entità E_1 ed E_2 vengono eliminate e le loro proprietà (attributi e partecipazioni ad associazioni e generalizzazioni) vengono aggiunte all'entità genitore E_0. A tale entità viene aggiunto un ulteriore attributo che serve a distinguere il "tipo" di una occorrenza di E_0, cioè se tale occorrenza apparteneva a E_1, a E_2 o, nel caso di generalizzazione non totale, a nessuna di esse. Se per esempio una generaliz-

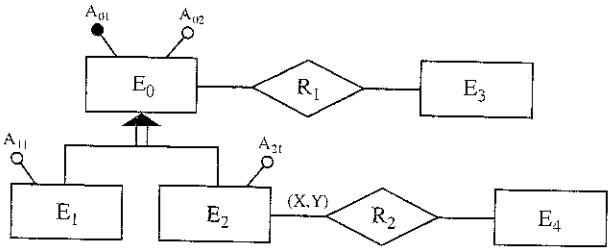

Figura 8.10 Esempio di schema con generalizzazione.

zazione tra l'entità PERSONA e le entità UOMO e DONNA viene ristrutturata in questo modo, all'entità PERSONA va aggiunto l'attributo Sesso per mantenere la distinzione tra le occorrenze di tale entità che la generalizzazione originaria rappresentava. Con riferimento al primo schema di Figura 8.11, si osservi che gli attributi A_{11} e A_{21} possono assumere valori nulli (perché non significativi) per alcune occorrenze di E_0 e che la relazione R_2 avrà, in ogni caso, una cardinalità minima pari a 0 sull'entità E_0 (perché le occorrenze di E_2 sono solo un sottoinsieme delle occorrenze di E_0).

(2) **Accorpamento del genitore della generalizzazione nelle figlie.** L'entità genitore E_0 viene eliminata e, per la proprietà dell'ereditarietà, i suoi attributi, il suo identificatore e le relazioni a cui tale entità partecipava, vengono aggiunti alle entità figlie E_1 ed E_2. Le relazioni R_{11} e R_{12} rappresentano rispettivamente la restrizione della relazione R_1 sulle occorrenze delle entità E_1 ed E_2. Se per esempio una generalizzazione tra l'entità PERSONA, avente Cognome ed Età come attributi e Codice Fiscale come identificatore, e le entità UOMO e DONNA viene ristrutturata in questo modo, alle entità UOMO e DONNA vanno aggiunti gli attributi Cognome ed Età e l'identificatore Codice Fiscale.

(3) **Sostituzione della generalizzazione con associazioni.** La generalizzazione si trasforma in due associazioni uno a uno che legano rispettivamente l'entità genitore con le entità figlie E_1 ed E_2. Non ci sono trasferimenti di attributi o associazioni e le entità E_1 ed E_2 sono identificate esternamente dall'entità E_0. Nello schema ottenuto vanno aggiunti però dei vincoli: ogni occorrenza di E_0 non può partecipare contemporaneamente a R_{G1} e R_{G2}; inoltre, se la generalizzazione è totale, ogni occorrenza di E_0 deve partecipare o a un'occorrenza di R_{G1} oppure a un'occorrenza di R_{G2}.

La scelta tra le varie alternative può essere fatta in maniera analoga a quanto fatto per i dati derivati, considerando vantaggi e svantaggi di ognuna delle scelte possibili relativamente alla occupazione di memoria e al costo delle operazioni coinvolte. È possibile comunque stabilire alcune regole di carattere generale.

- L'alternativa (1) è conveniente quando le operazioni non fanno molta distinzione

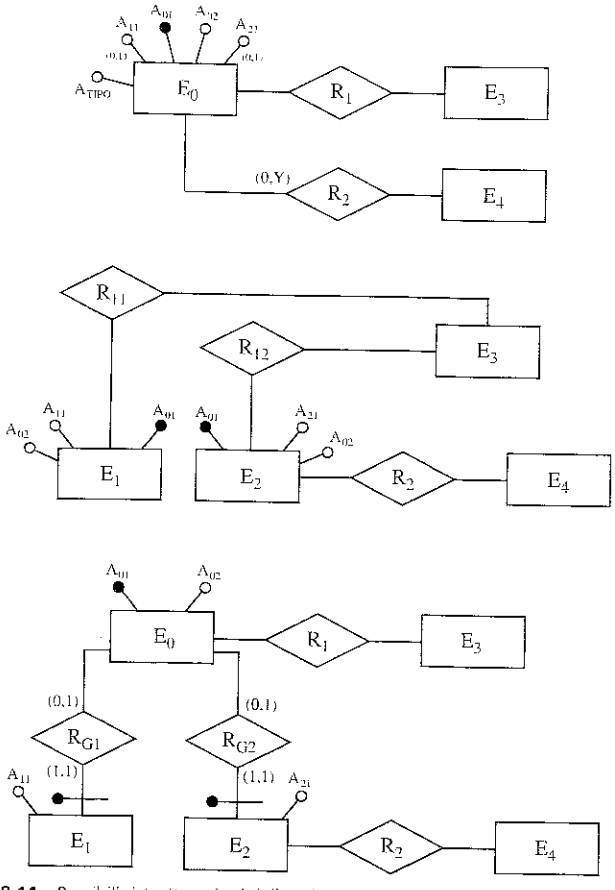

Figura 8.11 Possibili ristrutturazioni dello schema in Figura 8.10.

tra le occorrenze e tra gli attributi di E_0, E_1 ed E_2. In questo caso infatti, anche se abbiamo uno spreco di memoria per la presenza di valori nulli, la scelta ci assicura un numero minore di accessi rispetto alle altre nelle quali le occorrenze e gli attributi sono distribuiti tra le varie entità.

- L'alternativa (2) è possibile solo se la generalizzazione è totale, altrimenti le occorrenze di E_0 che non sono occorrenze nè di E_1 nè di E_2 non sarebbero rappresen-

tate. È conveniente quando ci sono operazioni che si riferiscono solo a occorrenze di E_1 oppure di E_2, e dunque fanno delle distinzioni tra tali entità. In questo caso abbiamo un risparmio di memoria rispetto alla scelta (1), perché, in linea di principio, gli attributi non assumono mai valori nulli. Inoltre, c'è una riduzione degli accessi rispetto alla scelta (3) perché non si deve visitare E_0 per accedere ad alcuni attributi di E_1 ed E_2.

- L'alternativa (3) è conveniente quando la generalizzazione non è totale (sebbene ciò non sia necessario) e ci sono operazioni che si riferiscono solo a occorrenze di E_1 (E_2) oppure di E_0, e dunque fanno delle distinzioni tra entità figlia ed entità genitore. In questo caso abbiamo un risparmio di memoria rispetto alla scelta (1), per l'assenza di valori nulli, ma c'è un incremento degli accessi per mantenere la consistenza delle occorrenze rispetto ai vincoli introdotti.

C'è un aspetto importante da chiarire rispetto a quanto detto. La ristrutturazione delle generalizzazioni è un tipico caso per il quale il semplice conteggio delle istanze e degli accessi non è sempre sufficiente per scegliere la migliore alternativa possibile. Infatti, da quanto detto, sembrerebbe che, sulla base di questi fattori, l'alternativa (3) non convenga quasi mai perché richiede molti più accessi a occorrenze delle altre per eseguire le operazioni sui dati. Questa ristrutturazione però ha il grosso vantaggio di generare entità con pochi attributi. Come vedremo, questo si traduce, a livello pratico, in strutture logiche (relazioni nel caso di sistemi per basi di dati relazionali) di piccole dimensioni per le quali un accesso fisico permette di recuperare molti dati (tuple) in una volta sola. In alcuni casi critici, va quindi effettuata una analisi più fine, che tiene conto di altri fattori quali le dimensioni dei domini degli attributi e la quantità di dati che è possibile recuperare con una sola operazione di accesso a memoria secondaria.

Le alternative viste non sono in effetti le uniche ammesse, ma è possibile effettuare ristrutturazioni che sono combinazioni delle tre trasformazioni presentate. Un esempio viene fornito in Figura 8.12, sempre con riferimento allo schema originale in Figura 8.10: in questo caso, in base a considerazioni analoghe a quelle discusse in precedenza, si è deciso di accorpare E_0 ed E_1 e di lasciare l'entità E_2 separata dalle altre. L'attributo A_{TIPO} è stato aggiunto per distinguere le occorrenze di E_0 da quelle di E_1.

Per quanto riguarda infine le generalizzazioni su più livelli, si può procedere analogamente analizzando una generalizzazione alla volta a partire dal fondo dell'intera gerarchia. In base a quanto detto, sono possibili diverse configurazioni, ottenibili per combinazione delle ristrutturazioni di base, sia a livello della singola generalizzazione sia lungo i vari livelli della gerarchia.

8.3.3 Partizionamento/accorpamento di concetti

Entità e associazioni in uno schema E-R possono essere partizionati o accorpati per garantire una maggior efficienza delle operazioni in base al seguente principio: gli accessi si riducono separando attributi di uno stesso concetto che vengono acceduti da operazioni diverse e raggruppando attributi di concetti diversi che vengono acceduti dalle medesime operazioni. Le stesse tecniche discusse per l'analisi delle generalizzazioni possono essere usate per prendere decisioni di questo tipo.

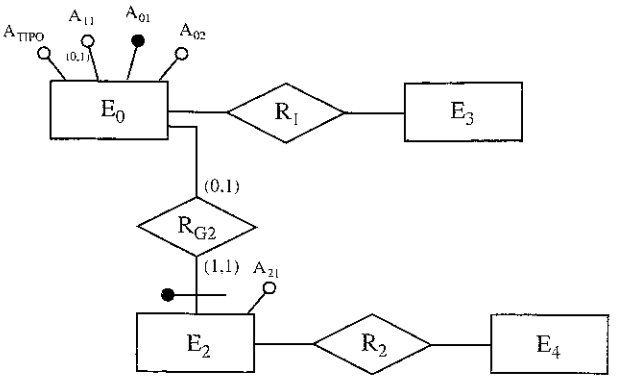

Figura 8.12 Possibile ristrutturazione dello schema in Figura 8.10.

Partizionamenti di entità Un esempio di partizionamento di entità viene mostrato in Figura 8.13: l'entità IMPIEGATO viene sostituita da due entità, collegate da un'associazione uno a uno, che descrivono rispettivamente i dati anagrafici degli impiegati e i dati relativi alla loro retribuzione. Questa ristrutturazione è conveniente se le operazioni che coinvolgono frequentemente l'entità originaria richiedono, per un impiegato, o solo informazioni di carattere anagrafico o solo informazioni relative alla sua retribuzione.

Un partizionamento di questo tipo è un esempio di *decomposizione verticale* di una entità, nel senso che si suddivide il concetto operando sui suoi attributi. È comunque possibile effettuare anche delle *decomposizioni orizzontali* nelle quali la suddivisione avviene sulle occorrenze dell'entità. Per esempio, per l'entità IMPIEGATO ci potrebbero essere alcune operazioni che riguardano soltanto gli *analisti* e altre che operano solo sui *venditori*. Anche in questo caso può convenire decomporre l'entità in due entità distinte ANALISTA e VENDITORE. In questo caso però le entità ottenute hanno gli stessi attributi dell'entità di partenza. È interessante osservare come una decomposizione orizzontale corrisponda all'introduzione di una generalizzazione a livello logico.

I partizionamenti orizzontali hanno un effetto collaterale: quello di dover duplicare tutte le associazioni a cui l'entità originaria partecipa. Questo fenomeno può avere delle ripercussioni negative sulle prestazioni del sistema. D'altra parte, i partizionamenti verticali generano entità con pochi attributi che possono essere tradotte in strutture logiche sulle quali, con un solo accesso, è possibile recuperare molti dati. Come per le generalizzazioni, anche in questo caso il semplice conteggio delle occorrenze e degli accessi, non è sempre sufficiente per scegliere la migliore alternativa possibile. Il problema del partizionamento dei dati viene ulteriormente discusso nel secondo volume, nel capitolo dedicato alle basi di dati distribuite.

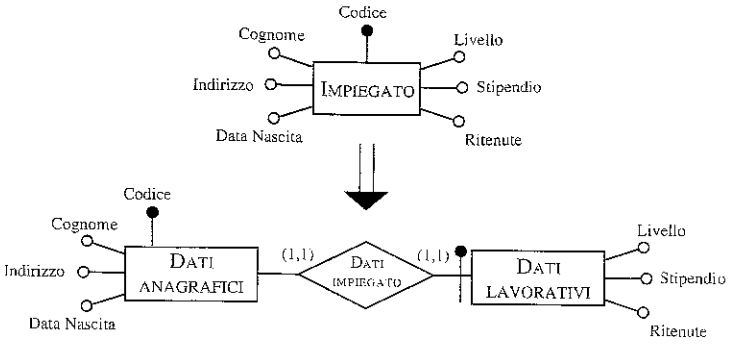

Figura 8.13 Esempio di partizionamento di entità.

Eliminazione di attributi multivalore Un particolare tipo di partizionamento che è opportuno trattare a parte è quello che riguarda l'eliminazione di attributi multivalore. Questa ristrutturazione si rende necessaria perché, come per le generalizzazioni, il modello relazionale non permette di rappresentare in maniera diretta questo tipo di attributo.

Il tipo di ristrutturazione necessario è piuttosto semplice e viene illustrato dall'esempio in Figura 8.14.

L'entità AGENZIA avente l'attributo multivalore Telefono viene partizionata in due entità: una entità con lo stesso nome e gli stessi attributi dell'entità originale ec-

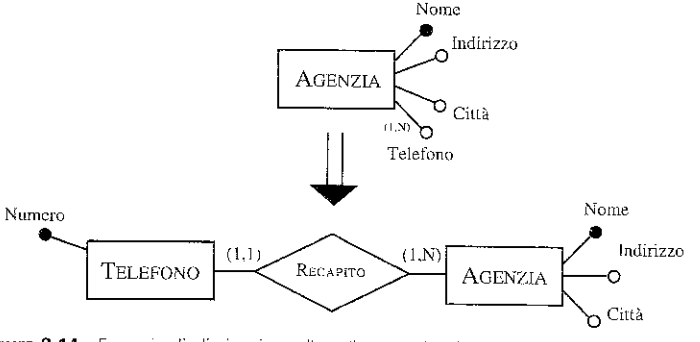

Figura 8.14 Esempio di eliminazione di attributo multivalore.

cetto l'attributo multivalore, e l'entità TELEFONO, con il solo attributo Numero, legata mediante un'associazione uno a molti con l'entità AGENZIA. Ovviamente, se l'attributo fosse stato anche opzionale, allora la cardinalità minima per l'entità AGENZIA nello schema risultato sarebbe stata pari a zero.

Accorpamento di entità L'accorpamento è l'operazione inversa del partizionamento. Un esempio di accorpamento di entità viene mostrato in Figura 8.15 nella quale le entità PERSONA e APPARTAMENTO, legate dall'associazione uno a uno INTESTAZIONE, vengono accorpate in un'unica entità contenente gli attributi di entrambi. Questa ristrutturazione può essere suggerita dal fatto che le operazioni più frequenti sull'entità PERSONA richiedono sempre i dati relativi all'appartamento che occupa e vogliamo quindi risparmiare gli accessi necessari per risalire a questi dati attraverso l'associazione che li lega. Un effetto collaterale di questa ristrutturazione è la possibile presenza di valori nulli dovuta al fatto che le cardinalità ci dicono che ci sono persone che non sono intestatari di nessun appartamento, e quindi non esistono per esse valori per gli attributi Indirizzo e Interno.

Gli accorpamenti si effettuano in genere su associazioni di tipo uno a uno, raramente su associazione uno a molti e praticamente mai su relazioni molti a molti. Questo perché gli accorpamenti di entità legate da un'associazione uno a molti o molti a molti generano ridondanze. In particolare, è facile verificare che si possono presentare ridondanze su attributi non chiave dell'entità che partecipava all'associazione originaria con una cardinalità massima pari a N. La presenza di ridondanze può essere comunque analizzata e discussa in maniera efficace con la tecnica della *normalizzazione*, che verrà presentata in dettaglio nel Capitolo 9 e a cui rimandiamo per ulteriori dettagli sull'argomento.

Altri tipi di partizionamento/accorpamento Abbiamo parlato finora di partizionamento e accorpamento di entità ma lo stesso discorso si può estendere alle asso-

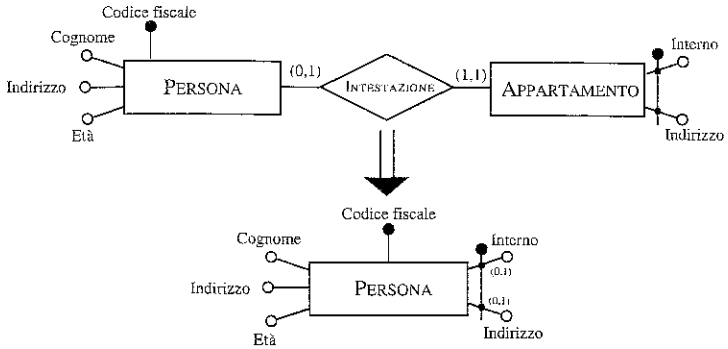

Figura 8.15 Esempio di accorpamento di entità.

ciazioni. Può convenire cioè, in alcuni casi, decomporre un'associazione tra due entità in due (o più) associazioni tra le medesime entità, per separare occorrenze dell'associazione originale accedute sempre separatamente e, viceversa, accorpare due (o più) associazioni tra le medesime entità (che si riferiscono però a due aspetti dello stesso concetto) in un'unica associazione, quando le relative occorrenze vengono sempre accedute contemporaneamente. Un esempio di partizionamento di associazioni viene fornito in Figura 8.16 nella quale vengono distinti i giocatori che compongono *attualmente* una squadra da quelli che ne facevano parte nel passato.

Prima di concludere questo paragrafo, vale la pena accennare al fatto che i problemi di partizione/accorpamento possono essere rinviati, in molti casi, alla fase di progettazione fisica. Diversi sistemi di gestione di basi di dati correnti permettono infatti di specificare *cluster* di strutture logiche (relazioni nei sistemi relazionali), ovvero raggruppamenti di dati, fatti a livello fisico, che permettono l'accesso rapido a dati distribuiti su strutture logiche separate.

8.3.4 Scelta degli identificatori principali

La scelta degli identificatori principali è essenziale nelle traduzioni verso il modello relazionale perché, come discusso nel Capitolo 2, in questo modello le chiavi vengono usate per stabilire legami tra dati in relazioni diverse. Inoltre, i sistemi di gestione

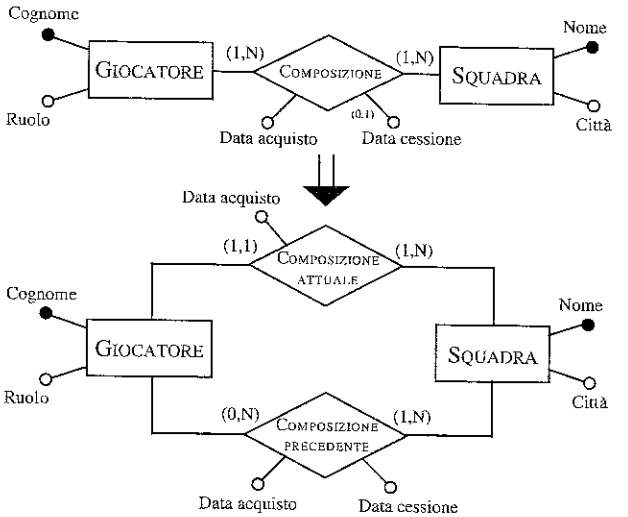

Figura 8.16 Esempio di partizionamento di associazione.

di basi di dati richiedono generalmente di specificare una *chiave primaria* sulla quale vengono costruite automaticamente delle strutture ausiliarie, dette *indici*, per il reperimento efficiente di dati. Quindi, nei casi in cui esistono entità per le quali sono stati specificati più identificatori, bisogna decidere quale di questi identificatori verrà utilizzato come chiave primaria.

I criteri di decisione per questa scelta sono i seguenti.

- Gli attributi con valori nulli non possono costituire identificatori principali. Tali attributi infatti non garantiscono l'accesso a tutte le occorrenze dell'entità corrispondente, come sottolineato quando abbiamo discusso le chiavi nel modello relazionale.
- Un identificatore composto da uno o da pochi attributi è da preferire a identificatori costituiti da molti attributi. Questo infatti garantisce che le strutture ausiliarie create per accedere ai dati (gli indici) siano di dimensioni ridotte, permette un risparmio di memoria nella realizzazione dei legami logici tra le varie relazioni e facilita le operazioni di join.
- Per gli stessi motivi del punto precedente un identificatore interno con pochi attributi è da preferire a un identificatore esterno, che magari coinvolge diverse entità. Infatti, come vedremo nel prossimo paragrafo, gli identificatori esterni vengono tradotti in chiavi che includono gli identificatori delle entità coinvolte nell'identificazione esterna: chiaramente in questa maniera si possono generare chiavi con molti attributi.
- Un identificatore che viene utilizzato da molte operazioni per accedere alle occorrenze di un'entità è da preferire rispetto agli altri. In questa maniera infatti tali operazioni possono essere eseguite efficientemente perché possono trarre vantaggio dagli indici creati automaticamente dal DBMS.

A questo punto, se nessuno degli identificatori candidati soddisfa tali requisiti, è possibile pensare di introdurre un ulteriore attributo all'entità: questo attributo conterrà valori speciali (detti *codici*) generati appositamente per identificare le occorrenze delle entità.

È comunque consigliabile tenere traccia in questa fase anche degli identificatori non selezionati come principali ma che vengono utilizzati da qualche operazione per accedere ai dati. Per questi identificatori è infatti possibile definire, in sede di progettazione fisica, degli *indici secondari*. Gli indici secondari consentono l'accesso efficiente ai dati e possono essere usati in alternativa agli indici definiti automaticamente sugli identificatori principali.

8.4 Traduzione verso il modello relazionale

La seconda fase della progettazione logica corrisponde a una traduzione tra modelli di dati diversi: a partire da uno schema E-R ristrutturato si costruisce uno schema logico *equivalente*, in grado cioè di rappresentare le medesime informazioni. Coerentemente con quanto detto nei paragrafi precedenti, facciamo riferimento a una versione semplificata del modello E-R, che non contiene generalizzazioni e attributi multivalore,

e nella quale ogni entità ha un solo identificatore. Studieremo inoltre la traduzione verso il modello relazionale.

Affrontiamo il problema della traduzione caso per caso, iniziando dal caso più generale (quello di entità legate da associazioni molti a molti) che ci suggerisce l'idea generale su cui si basa la metodologia di traduzione.

8.4.1 Entità e associazioni molti a molti

Consideriamo lo schema in Figura 8.17. La sua traduzione naturale nel modello relazionale prevede:

- per ogni entità, una relazione con lo stesso nome avente per attributi i medesimi attributi dell'entità e per chiave il suo identificatore;
- per l'associazione, una relazione con lo stesso nome avente per attributi gli attributi dell'associazione e gli identificatori delle entità coinvolte; tali identificatori formano la chiave della relazione.

Se gli attributi originali di entità o associazioni sono opzionali, i corrispondenti attributi di relazione possono assumere valori nulli.

Lo schema relazionale che si ottiene è quindi il seguente:

IMPIEGATO(Matricola, Cognome, Stipendio)
PROGETTO(Codice, Nome, Budget)
PARTECIPAZIONE(Matricola, Codice, DataInizio)

Per lo schema ottenuto esistono due vincoli di integrità referenziale tra gli attributi Matricola e Codice di PARTECIPAZIONE e gli omonimi attributi delle entità IMPIEGATO e PROGETTO.

Per rendere più comprensibile il significato dello schema è conveniente effettuare alcune ridenominazioni. Per esempio, nel nostro caso si può chiarire il contenuto della relazione PARTECIPAZIONE definendola come segue:

PARTECIPAZIONE(Impiegato, Progetto, DataInizio)

nella quale il dominio dell'attributo Impiegato è un insieme di matricole di impiegati e quello dell'attributo Progetto è un insieme di codici di progetti ed esistono vincoli di integrità referenziale tra questi attributi e, rispettivamente, l'attributo Matricola della relazione IMPIEGATO e l'attributo Codice della relazione PROGETTO.

Figura 8.17 Schema E-R con associazione molti a molti.

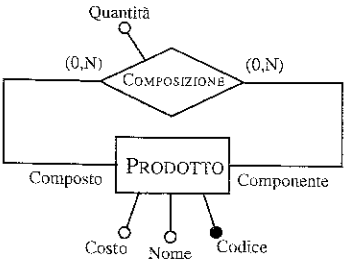

Figura 8.18 Schema E-R con associazione ricorsiva.

La ridenominazione è in effetti essenziale in alcuni casi. Per esempio, nel caso di associazioni ricorsive come quella in Figura 8.18.

Questo schema si traduce nelle due relazioni:

PRODOTTO(Codice, Nome, Costo)
COMPOSIZIONE(Composto, Componente, Quantità)

In questo schema, entrambi gli attributi Composto e Componente contengono codici di prodotti: il primo dei due ha il secondo come componente. Esiste quindi un vincolo di integrità referenziale tra questi attributi e l'attributo Codice della relazione PRODOTTO.

Le associazioni con più di due entità partecipanti si traducono in maniera analoga a quanto detto per le associazioni binarie. Per esempio, si consideri lo schema con una associazione ternaria riportato in Figura 8.19. Questo schema si traduce nelle seguenti tre relazioni:

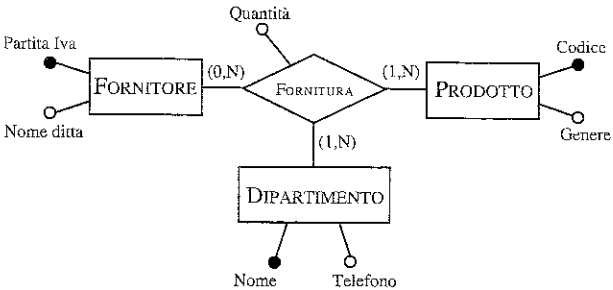

Figura 8.19 Schema E-R con associazione ternaria.

FORNITORE(PartitaIVA, NomeDitta)
PRODOTTO(Codice, Genere), DIPARTIMENTO(Nome, Telefono)
FORNITURA(Fornitore, Prodotto, Dipartimento, Quantità).

Per lo schema così ottenuto esistono i vincoli di integrità referenziale tra gli attributi Fornitore, Prodotto e Dipartimento della relazione FORNITURA e, rispettivamente, l'attributo PartitaIVA della relazione FORNITORE, l'attributo Codice della relazione PRODOTTO e l'attributo Nome della relazione DIPARTIMENTO.

In questo ultimo tipo di traduzione bisogna prestare attenzione ad alcuni casi particolari nei quali l'insieme delle chiavi delle relazioni che rappresentano le entità coinvolte costituisce in realtà una *superchiave* ridondante della relazione che rappresenta l'associazione dello schema E-R (esiste cioè un suo sottoinsieme proprio che è una chiave). Questo potrebbe accadere se, per esempio, nel caso dello schema di Figura 8.19, ci fosse un solo fornitore che fornisce un certo prodotto a un dipartimento. Si noti che le cardinalità sono ancora valide, perché tale fornitore può fornire diversi prodotti a questo o ad altri dipartimenti. In questo caso, la chiave della relazione FORNITURA, sarebbe costituita dai soli attributi Prodotto e Dipartimento perché, dato un prodotto e un dipartimento, il fornitore è univocamente determinato.

8.4.2 Associazioni uno a molti

Consideriamo lo schema con associazione uno a molti in Figura 8.20.

Secondo la regola vista per le associazioni molti a molti, la traduzione di questo schema dovrebbe essere la seguente:

GIOCATORE(Cognome, DataNascita, Ruolo)
SQUADRA(Nome, Città, ColoriSociali)
CONTRATTO(Giocatore, DataNascitaGiocatore, NomeSquadra, Ingaggio)

Va notato che, nella relazione CONTRATTO, la chiave è costituita solo dall'identificatore di GIOCATORE perché le cardinalità dell'associazione ci dicono che ogni giocatore ha un contratto con una sola squadra. A questo punto le relazioni GIOCATORE e CONTRATTO hanno la stessa chiave (il cognome e la data di nascita di un giocatore) ed è allora possibile fonderle in un'unica relazione (perché esiste una corrispondenza

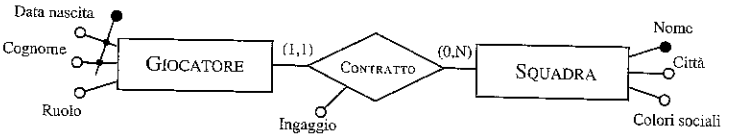

Figura 8.20 Schema E-R con associazione uno a molti.

biunivoca tra le rispettive occorrenze). È quindi preferibile, per lo schema in Figura 8.20, la traduzione che segue, nella quale la relazione GIOCATORE rappresenta sia l'entità relativa sia l'associazione dello schema E-R originale:

GIOCATORE(Cognome, DataNascita, Ruolo, NomeSquadra, Ingaggio)
SQUADRA(Nome, Città, ColoriSociali)

In questo schema, esiste ovviamente il vincolo di integrità referenziale tra l'attributo NomeSquadra della relazione GIOCATORE e l'attributo Nome della relazione SQUADRA.

Nel nostro esempio, la cardinalità minima dell'entità GIOCATORE è pari a uno. Nel caso in cui tale cardinalità fosse pari a zero (è possibile cioè avere giocatori che non hanno un contratto con una squadra) entrambe le alternative viste sono valide. Infatti, anche se nella seconda traduzione abbiamo un numero minore di relazioni, è possibile avere dei valori nulli nella relazione GIOCATORE sugli attributi NomeSquadra e Ingaggio, mentre, nella prima traduzione, questa eventualità non si può verificare.

Abbiamo accennato nel Paragrafo 6.2.2 che le associazioni n-arie sono quasi sempre di tipo molti a molti. Nel caso in cui un'entità partecipi a un'associazione ternaria con cardinalità massima pari a uno, la traduzione dell'associazione viene fatta in modo analogo a un'associazione binaria tra le altre entità. L'entità che partecipa all'associazione con cardinalità massima pari a uno, viene infatti tradotta in una relazione che contiene anche gli identificatori delle altre entità coinvolte nell'associazione (più eventuali attributi dell'associazione stessa) e non c'è più bisogno di rappresentare esplicitamente l'associazione di partenza. Per esempio, se l'entità PRODOTTO partecipasse all'associazione in Figura 8.19 con cardinalità pari a (1, 1) (e quindi, per ogni prodotto, esistesse un solo fornitore che lo fornisce e un solo dipartimento al quale viene fornito), allora lo schema si tradurrebbe come segue:

FORNITORE(PartitaIVA, NomeDitta)
DIPARTIMENTO(Nome, Telefono)
PRODOTTO(Codice, Genere, Fornitore, Dipartimento, Quantità).

nel quale esistono i vincoli di integrità referenziale tra l'attributo Fornitore della relazione PRODOTTO e l'attributo PartitaIVA della relazione FORNITORE, e tra l'attributo Dipartimento della relazione PRODOTTO e l'attributo Nome della relazione DIPARTIMENTO.

8.4.3 Entità con identificatore esterno

Le entità con identificatori esterni danno luogo a relazioni con chiavi che includono gli identificatori delle entità "identificanti". Consideriamo per esempio lo schema sugli studenti universitari riportato in Figura 8.21.

Lo schema relazionale corrispondente a questo schema è il seguente:

STUDENTE(Matricola, NomeUniversità, Cognome, AnnoIscrizione)
UNIVERSITÀ(Nome, Città, indirizzo)

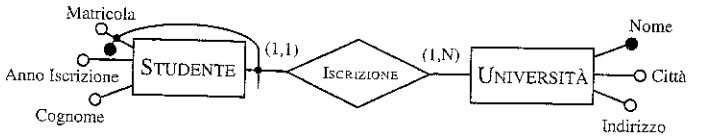

Figura 8.21 Schema E-R con identificatore esterno.

nel quale esiste il vincolo di integrità referenziale tra l'attributo NomeUniversità della relazione STUDENTE e l'attributo Nome della relazione UNIVERSITÀ.

Come si può vedere, rappresentando l'identificatore esterno si rappresenta direttamente anche l'associazione tra le due entità. Ricordiamo infatti che le entità identificate esternamente partecipano all'associazione sempre con una cardinalità minima e massima pari a uno. Questo tipo di traduzione è valido indipendentemente dalla cardinalità con cui l'altra entità partecipa all'associazione.

8.4.4 Associazioni uno a uno

Per le associazioni uno a uno ci sono, in genere, diverse possibilità di traduzione. Cominciamo a vedere le associazioni uno a uno con partecipazioni obbligatorie per entrambe le entità, come quella nello schema in Figura 8.22. Per questo tipo di associazioni abbiamo due possibilità simmetriche e ugualmente valide:

DIRETTORE(Codice, Cognome, Stipendio, DipartimentoDiretto, InizioDirezione)

DIPARTIMENTO(Nome, Telefono, Sede)

con il vincolo di integrità referenziale tra l'attributo DipartimentoDiretto della relazione DIRETTORE e l'attributo Nome della relazione DIPARTIMENTO, oppure:

DIRETTORE(Codice, Cognome, Stipendio)

DIPARTIMENTO(Nome, Telefono, Sede, Direttore, InizioDirezione)

per il quale esiste il vincolo di integrità referenziale tra l'attributo Direttore della relazione DIPARTIMENTO e l'attributo Codice della relazione DIRETTORE.

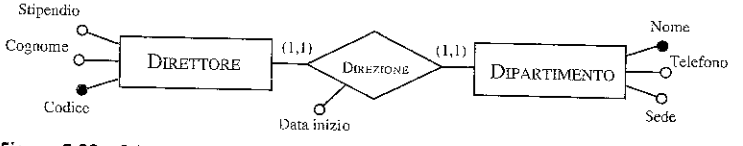

Figura 8.22 Schema E-R con associazione uno a uno.

È possibile quindi rappresentare l'associazione in una qualunque delle relazioni che rappresentano le due entità. Trattandosi di una relazione biunivoca tra le occorrenze delle entità, sembrerebbe possibile una ulteriore alternativa nella quale si rappresentano tutti i concetti in un'unica relazione contenente tutti gli attributi in gioco. Questa alternativa è però da escludere perché non dobbiamo dimenticarci che lo schema che stiamo traducendo è il risultato di una fase di ristrutturazione nella quale sono state effettuate precise scelte anche riguardo l'accorpamento e il partizionamento di entità. Questo significa che, se nello schema E-R ristrutturato abbiamo due entità collegate da una relazione uno a uno, vuol dire che abbiamo ritenuto conveniente tenere separati i due concetti ed è quindi inopportuno fonderli in sede di traduzione verso il modello relazionale.

Consideriamo ora il caso di associazione uno a uno con partecipazione opzionale per una sola entità, come quella nello schema in Figura 8.23. In questo caso abbiamo una soluzione preferibile rispetto alle altre:

IMPIEGATO(Codice, Cognome, Stipendio)
DIPARTIMENTO(Nome, Telefono, Sede, Direttore, InizioDirezione)

per la quale esiste il vincolo di integrità referenziale tra l'attributo Direttore della relazione DIPARTIMENTO e l'attributo Codice della relazione IMPIEGATO.

Questa alternativa è preferibile rispetto a quella in cui l'associazione viene rappresentata nella relazione IMPIEGATO mediante il nome del dipartimento diretto perché avremmo, per questo attributo, possibili valori nulli.

Consideriamo infine il caso in cui entrambe le entità hanno partecipazione opzionale come nel caso in cui, nello schema in Figura 8.23, possono esistere dipartimenti senza direttori (e quindi la cardinalità dell'entità DIPARTIMENTO diventa (0,1)). In questo caso esiste un'ulteriore possibilità che prevede tre relazioni separate:

IMPIEGATO(Codice, Cognome, Stipendio)
DIPARTIMENTO(Nome, Telefono, Sede)
DIREZIONE(Direttore, Dipartimento, DataInizioDirezione)

Su questo schema abbiamo due vincoli di integrità referenziale: uno tra l'attributo Direttore della relazione DIREZIONE e l'attributo Codice della relazione IMPIEGATO e l'altro tra l'attributo Dipartimento della relazione DIREZIONE e l'attributo Nome della relazione DIPARTIMENTO.

Figura 8.23 Schema E-R con associazione uno a uno.

Questa soluzione ha il vantaggio rispetto a quella in cui si accorpa la relazione DIREZIONE in una delle altre due relazioni (come nel caso precedente), di non presentare mai valori nulli sugli attributi che rappresentano l'associazione. Per contro, abbiamo bisogno di una relazione in più con un conseguente aumento della complessità della base di dati. Diciamo quindi che la soluzione con tre relazioni è da prendere in considerazione solo se il numero di occorrenze dell'associazione è molto basso rispetto alle occorrenze delle entità che partecipano all'associazione. In questo caso, c'è infatti il vantaggio di evitare la presenza di molti valori nulli.

8.4.5 Traduzioni di schemi complessi

Per vedere come procedere in un caso complesso, facciamo un esempio completo di traduzione con riferimento allo schema riportato in Figura 8.24.

In una prima fase, traduciamo ciascuna entità con una relazione. La traduzione delle entità dotate di identificatore interno è immediata:

E3(A31, A32) E4(A41, A42) E5(A51, A52)
E6 (A61, A62, A63)

Traduciamo ora le entità con le identificazioni esterne. Otteniamo le seguenti relazioni:

E1(A11, A51, A12) E2(A21, A11, A51, A22)

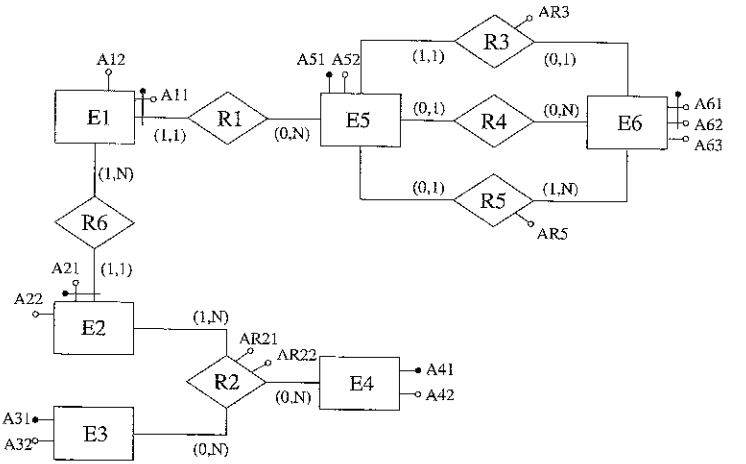

Figura 8.24 Uno schema E-R da tradurre.

Notare come E2 prende l'attributo A11 e, per la proprietà transitiva, anche l'attributo A51 che, insieme al primo, identifica E1. Alle relazioni prodotte vanno aggiunti anche alcuni vincoli di integrità referenziale (per esempio, sussiste un vincolo di integrità referenziale tra l'attributo A51 in E1 e l'attributo omonimo di E5).

Passiamo ora alla traduzione delle associazioni. Le associazioni R1 e R6 sono già state tradotte come conseguenza dell'identificazione esterna di E1 ed E2 rispettivamente. Assumiamo di aver deciso di ottenere un numero minimo di relazioni nello schema finale e cerchiamo quindi di accorpare quando possibile. Questo comporta una possibile presenza di valori nulli nelle istanze relazionali per tutti gli attributi introdotti traducendo relazioni dal lato di cardinalità (0, 1). Otteniamo le seguenti modifiche da effettuare allo schema iniziale:

- per tradurre R3, introduciamo con opportune ridenominazioni gli attributi che identificano E6 tra quelli di E5, nonché l'attributo AR3 proprio di R3; in pratica, introduciamo A61R3, A62R3 e AR3 in E5;
- analogamente per R4, introduciamo A61R4 e A62R4 in E5;
- analogamente per R5, introduciamo A61R5, A62R5 e AR5 in E5.

Si osservi che le ridenominazioni sono indispensabili per poter distinguere l'uso dello stesso attributo per rappresentare diverse associazioni (per esempio, A61R3 che rappresenta R3 e A61R4 che rappresenta R4). Infine, traduciamo l'unica associazione molti a molti:

R2(<u>A21</u>, <u>A11</u>, <u>A51</u>, <u>A31</u>, <u>A41</u>, AR21, AR22)

Lo schema relazionale ottenuto è il seguente:

E1(<u>A11</u>, <u>A51</u>, A12) E2(<u>A21</u>, <u>A11</u>, <u>A51</u>, A22)
E3(<u>A31</u>, A32) E4(<u>A41</u>, A42)
E5(<u>A51</u>, A52, A61R3, A62R3, AR3, A61R4, A62R4, A61R5, A62R5, AR5)
E6(<u>A61</u>, <u>A62</u>, A63) R2(<u>A21</u>, <u>A11</u>, <u>A51</u>, <u>A31</u>, <u>A41</u>, AR21, AR22)

Si osservi che abbiamo ottenuto relazioni (E2 e R2) con chiavi composte da molti attributi. In tali situazioni si può anche decidere di introdurre chiavi semplici (codici) o in questa stessa fase o precedentemente, nella fase di ristrutturazione, come discusso nel Paragrafo 8.3.4.

8.4.6 Tabelle riassuntive

Le traduzioni viste vengono riassunte nelle tabelle in Figura 8.25 e 8.26.

Per ogni tipo di configurazione di schema E-R, viene fornita una descrizione del caso e le traduzioni possibili. In queste tabelle i simboli X e Y indicano una qualunque tra le cardinalità ammesse, gli asterischi indicano la possibilità di avere valori nulli sugli attributi relativi e la sottolineatura tratteggiata indica una chiave alternativa a quella indicata da una sottolineatura piena.

Tipologia	Concetto iniziale	Risultati possibili
Associazione binaria molti a molti	E_1, A_{E11}, A_{E12}, (X,N), R, A_R, (X,N), E_2, A_{E21}, A_{E22}	$E_1(\underline{A_{E11}},A_{E12})$ $E_2(\underline{A_{E21}},A_{E22})$ $R(\underline{A_{E11},A_{E21}},A_R)$
Associazione ternaria molti a molti	E_1, A_{E11}, A_{E12}, (X,N), E_3, (X,N), R, A_R, (X,N), A_{E31}, A_{E32}, E_2, A_{E21}, A_{E22}	$E_1(\underline{A_{E11}},A_{E12})$ $E_2(\underline{A_{E21}},A_{E22})$ $E_3(\underline{A_{E31}},A_{E32})$ $R(\underline{A_{E11},A_{E21},A_{E31}},A_R)$
Associazione uno a molti con partecipazione obbligatoria	E_1, A_{E11}, A_{E12}, (1,1), R, A_R, (X,N), E_2, A_{E21}, A_{E22}	$E_1(\underline{A_{E11}},A_{E12},A_{E21},A_R)$ $E_2(\underline{A_{E21}},A_{E22})$
Associazione uno a molti con partecipazione opzionale	E_1, A_{E11}, A_{E12}, (0,1), R, A_R, (X,N), E_2, A_{E21}, A_{E22}	$E_1(\underline{A_{E11}},A_{E12})$ $E_2(\underline{A_{E21}},A_{E22})$ $R(\underline{A_{E11}},A_{E21},A_R)$ Oppure: $E_1(\underline{A_{E11}},A_{E12},\overset{*}{A_{E21}},\overset{*}{A_R})$ $E_2(\underline{A_{E21}},A_{E22})$
Associazione con identificatore esterno	E_1, A_{E11}, A_{E12}, (1,1), R, A_R, (X,Y), E_2, A_{E21}, A_{E22}	$E_1(\underline{A_{E12},A_{E21}},A_{E11},A_R)$ $E_2(\underline{A_{E21}},A_{E22})$

Figura 8.25 Traduzioni dal modello E-R al relazionale.

8.4.7 Documentazione di schemi logici

Come nel caso della progettazione concettuale, il risultato della progettazione logica non è costituito solo da un semplice schema di una base di dati ma anche da una documentazione a esso associata. Innanzitutto, buona parte della documentazione dello schema concettuale in ingresso alla fase di progettazione logica può essere ereditata dallo schema logico ottenuto come risultato di questa fase. In particolare, se i nomi dei concetti dello schema E-R sono stati riutilizzati per costruire lo schema relaziona-

Tipologia	Concetto iniziale	Risultati possibili
Associazione uno a uno con partecipazione obbligatoria per entrambe le entità	E_1 — A_{E11}, A_{E12}; (1,1); R — A_R; (1,1); E_2 — A_{E21}, A_{E22}	$E_1(\underline{A_{E11}}, A_{E12}, \underline{A_{E21}}, A_R)$ $E_2(\underline{A_{E21}}, A_{E22})$ Oppure: $E_2(\underline{A_{E21}}, A_{E22}, \underline{A_{E11}}, A_R)$ $E_1(\underline{A_{E11}}, A_{E12})$
Associazione uno a uno con partecipazione opzionale per una entità	E_1 — A_{E11}, A_{E12}; (1,1); R — A_R; (0,1); E_2 — A_{E21}, A_{E22}	$E_1(\underline{A_{E11}}, A_{E12}, \underline{A_{E21}}, A_R)$ $E_2(\underline{A_{E21}}, A_{E22})$
Associazione uno a uno con partecipazione opzionale per entrambe le entità	E_1 — A_{E11}, A_{E12}; (0,1); R — A_R; (0,1); E_2 — A_{E21}, A_{E22}	$E_1(\underline{A_{E11}}, A_{E12})$ $E_2(\underline{A_{E21}}, A_{E22}, A^*_{E11}, A^*_R)$ Oppure: $E_1(\underline{A_{E11}}, A_{E12}, A^*_{E21}, A^*_R)$ $E_2(\underline{A_{E21}}, A_{E22})$ Oppure: $E_1(\underline{A_{E11}}, A_{E12})$ $E_2(\underline{A_{E21}}, A_{E22})$ $R(\underline{A_{E11}}, \underline{A_{E21}}, A_R)$

Figura 8.26 Traduzioni dal modello E-R al relazionale.

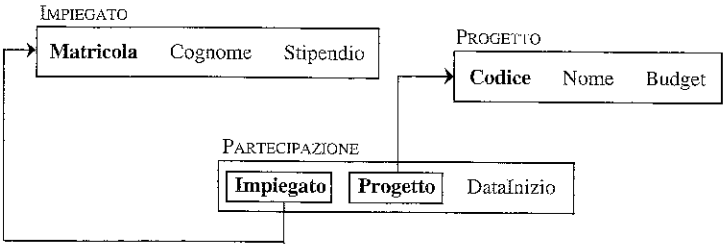

Figura 8.27 Rappresentazione grafica della traduzione nel modello relazionale dello schema in Figura 8.17.

le, le regole aziendali precedentemente definite possono essere usate per documentare anche quest'ultimo. A questa documentazione ne va aggiunta però dell'altra, in grado di descrivere i vincoli di integrità referenziale introdotti dalla traduzione.

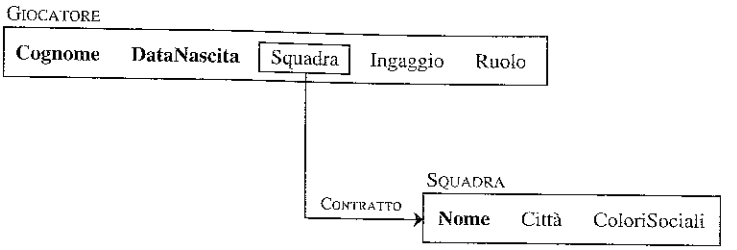

Figura 8.28 Rappresentazione grafica di una traduzione dello schema in Figura 8.20.

A tale riguardo, è possibile adottare un semplice formalismo grafico che permette di rappresentare sia le relazioni con i relativi attributi sia i vincoli di integrità referenziale esistenti tra le varie relazioni. Un esempio di questo tipo di rappresentazione, di facile comprensione, viene dato in Figura 8.27, con riferimento alla traduzione dello schema in Figura 8.17. In questi diagrammi le chiavi delle relazioni sono rappresentate in grassetto, le frecce indicano vincoli di integrità referenziale e la presenza di asterischi sui nomi di attributo indica la possibilità di avere valori nulli.

Si può osservare che, con questo formalismo, si riesce a mantenere traccia delle associazioni dello schema E-R originale. Questo può risultare utile per individuare, in maniera immediata, i *cammini di join*, ovvero le operazioni di join necessarie per ricostruire l'informazione rappresentata dalle associazioni originarie, nel caso dell'esempio, le informazioni sui progetti ai quali gli impiegati partecipano, attraverso il join tra IMPIEGATO, PARTECIPAZIONE e PROGETTO.

Un altro esempio di questi tipi di rappresentazione viene fornito in Figura 8.28, con riferimento alla traduzione dello schema in Figura 8.20.

È interessante osservare come, con questo tipo di rappresentazione, sia possibile rappresentare esplicitamente anche le associazioni dello schema Entità-Relazione di partenza alle quali, nello schema relazionale equivalente, non corrisponde nessuna relazione (l'associazione CONTRATTO nell'esempio in questione).

Come esempio finale, in Figura 8.29 viene riportata la rappresentazione dello schema relazionale ottenuto nel Paragrafo 8.4.5. I legami logici tra le varie relazioni possono essere ora facilmente identificati.

8.5 Un esempio di progettazione logica

Riprendiamo l'esempio presentato nel capitolo precedente relativo alla base di dati della società di formazione, il cui schema concettuale viene riportato, per comodità,

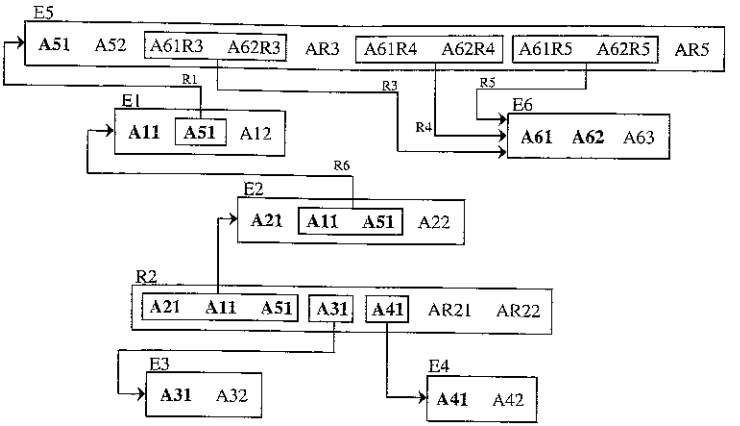

Figura 8.29 Rappresentazione grafica dello schema ottenuto nel Paragrafo 8.4.5.

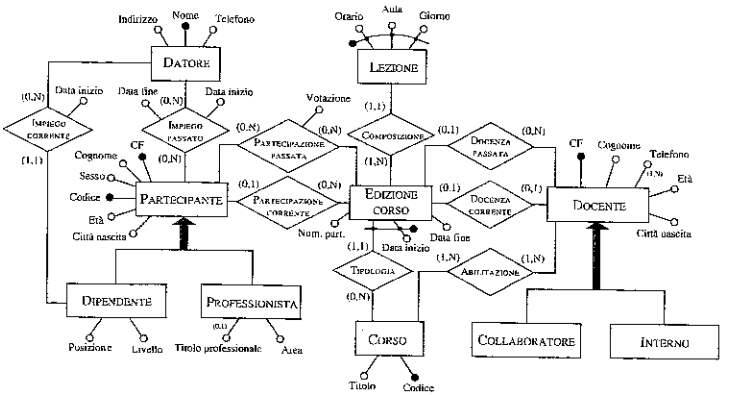

Figura 8.30 Lo schema E-R di una società di formazione.

in Figura 8.30. Le varie ristrutturazioni che discuteremo sono riportate nello schema finale in Figura 8.33.

Sui dati descritti da questo schema erano state previste le seguenti operazioni.

Operazione 1: inserisci un nuovo partecipante indicando tutti i suoi dati.

Operazione 2: assegna un partecipante a una edizione di corso.
Operazione 3: inserisci un nuovo docente indicando tutti i suoi dati e i corsi che può insegnare.
Operazione 4: assegna un docente abilitato a una edizione di un corso.
Operazione 5: stampa tutte le informazioni sulle edizioni passate di un corso con titolo, orari delle lezioni e numero dei partecipanti.
Operazione 6: stampa tutti i corsi offerti, con informazioni sui docenti che possono insegnarli.
Operazione 7: per ogni docente, trova i partecipanti a tutti i corsi da lui insegnati.
Operazione 8: effettua una statistica su tutti i partecipanti a un corso con tutte le informazioni su di essi, sulla edizione alla quale hanno partecipato e la rispettiva votazione.

8.5.1 Fase di ristrutturazione

Supponiamo che i dati di carico siano quelli riportati in Figura 8.31. Eseguiamo, sulla base di questi dati, i vari passi della ristrutturazione.

Analisi delle ridondanze C'è un solo dato ridondante nello schema: l'attributo Numero di partecipanti in EDIZIONE CORSO che può essere derivato dalle associazioni PARTECIPAZIONE CORRENTE e PARTECIPAZIONE PASSATA. Questo dato richiede un quantitativo di memoria pari a $4 \times 1000 = 4000$ byte, avendo assunto che sono necessari 4 byte per ogni occorrenza di EDIZIONECORSO per memorizzare il numero di partecipanti. Le operazioni coinvolte con questo dato sono la 2, la 5 e la 8. L'ultima di queste può essere trascurata perché si tratta di una operazione non frequente ed eseguita in modalità batch. Proviamo a valutare il costo delle Operazioni 2 e 5 in caso di presenza e in assenza di dato ridondante. Possiamo dedurre dalla tavola dei volumi che ogni edizione di corso ha, in media, 8 lezioni e 10 partecipanti. Da questi dati sono facilmente calcolabili le tavole degli accessi riportate in Figura 8.32.

Da queste risulta:

- dato ridondante presente: per l'Operazione 2 abbiamo $2 \times 50 = 100$ accessi in lettura e altrettanti in scrittura al giorno mentre, per l'Operazione 5, abbiamo $19 \times 10 = 190$ accessi in lettura al giorno, per un totale di 490 accessi giornalieri (avendo contato doppie le operazioni di scrittura);
- dato ridondante assente: per l'Operazione 2 abbiamo 50 accessi in lettura e altrettanti in scrittura al giorno, mentre, per l'Operazione 5, abbiamo $29 \times 10 = 290$ accessi in lettura al giorno, per un totale di 440 accessi giornalieri (avendo contato doppie le operazioni di scrittura).

Abbiamo quindi, in presenza di ridondanza, degli svantaggi sia in termini di memoria sia di efficienza. Decidiamo quindi di eliminare l'attributo ridondante Numero di partecipanti dalla relazione EDIZIONE CORSO.

Eliminazione delle gerarchie Nello schema sono presenti due gerarchie: quella relativa ai docenti e quella relativa ai partecipanti. Per i docenti si può notare che le

Tavola dei volumi

Concetto	Tipo	Volume
Lezione	E	8000
Edizione corso	E	1000
Corso	E	200
Docente	E	300
Collaboratore	E	250
Interno	E	50
Partecipante	E	5000
Dipendente	E	4000
Professionista	E	1000
Datore	E	8000
Part. passata	R	10 000
Part. corrente	R	500
Composizione	R	8000
Tipologia	R	1000
Doc. passata	R	900
Doc. corrente	R	100
Abilitazione	R	500
Impiego corrente	R	4000
Impiego passato	R	1000

Tavola delle operazioni

Operazione	Tipo	Frequenza
Op. 1	I	40/giorno
Op. 2	I	50/giorno
Op. 3	I	2/giorno
Op. 4	I	15/giorno
Op. 5	I	10/giorno
Op. 6	I	20/giorno
Op. 7	I	5/sett.
Op. 8	B	10/mese

Figura 8.31 Tavole dei volumi e delle operazioni per lo schema in Figura 8.30.

operazioni che li riguardano, cioè la 3, la 4, la 6 e la 7, non fanno distinzioni tra collaboratori esterni e dipendenti interni della società. Tra l'altro, le entità corrispondenti non hanno attributi specifici che li distinguono. Decidiamo quindi di accorpare le entità figlie della generalizzazione nel genitore aggiungendo un attributo Tipo all'entità DOCENTE che ha un dominio costituito dai simboli *C* (per Collaboratore) e *I* (per Interno).

Per quanto riguarda i partecipanti, osserviamo che anche in questo caso le operazioni che coinvolgono questo dato (la 1, la 2 e la 8) non fanno sostanziali differenze tra i vari tipi di occorrenze. Possiamo però osservare dallo schema che i professionisti e i dipendenti hanno degli attributi che li distinguono gli uni dagli altri. Risulta quindi

Tavole degli accessi in presenza di ridondanza

Operazione 2			
Concetto	**Costr.**	**Acc.**	**Tipo**
Partecipante	E	1	L
Par. corrente	R	1	S
Ediz. corso	E	1	L
Ediz. corso	E	1	S

Operazione 5			
Concetto	**Costr.**	**Acc.**	**Tipo**
Ed. corso	E	1	L
Tipologia	R	1	L
Corso	E	1	L
Composiz.	R	8	L
Lezione	E	8	L

Tavole degli accessi in assenza di ridondanza

Operazione 2			
Concetto	**Costr.**	**Acc.**	**Tipo**
Partecipante	E	1	L
Par. corrente	R	1	S

Operazione 5			
Concetto	**Costr.**	**Acc.**	**Tipo**
Ediz. corso	E	1	L
Tipologia	R	1	L
Corso	E	1	L
Composiz.	R	8	L
Lezione	E	8	L
Par. corrente	R	10	L

Figura 8.32 Tavole degli accessi per lo schema in Figura 8.30.

preferibile lasciare le entità DIPENDENTE e PROFESSIONISTA e aggiungere due associazioni uno a uno tra queste entità e l'entità PARTECIPANTE. In questa maniera, si evita di avere attributi con possibili valori nulli sull'entità genitore della generalizzazione e riduciamo le dimensioni delle relazioni. Il risultato di queste ristrutturazioni e di altre che discuteremo più avanti si può vedere nello schema in Figura 8.33.

Partizionamento/accorpamento di concetti Dall'analisi dei dati e delle operazioni si possono individuare diverse ristrutturazioni di questo tipo. La prima riguarda l'entità EDIZIONE DI CORSO: si può osservare che l'Operazione 5 riguarda solo una frazione delle edizioni, quelle passate, e che le associazioni DOCENZA PASSATA e PARTECIPAZIONE PASSATA fanno riferimento solo a queste edizioni di corso. Si potrebbe quindi pensare, per rendere più efficiente l'operazione suddetta, di decomporre orizzontalmente l'entità in maniera da distinguere le edizioni correnti da quelle passate. L'inconveniente di questa scelta però è che le associazioni COMPOSIZIONE e TIPOLOGIA andrebbero duplicate; inoltre, le Operazioni 7 e 8, che non fanno grosse distinzioni tra le edizioni correnti e quelle passate, risulterebbero più costose perché richiedono la visita di due entità distinte. Decidiamo quindi di non partizionare tale entità.

Due altre possibili ristrutturazioni che si può pensare di effettuare, proprio in conseguenza a quanto detto sulle edizioni dei corsi, sono l'accorpamento delle as-

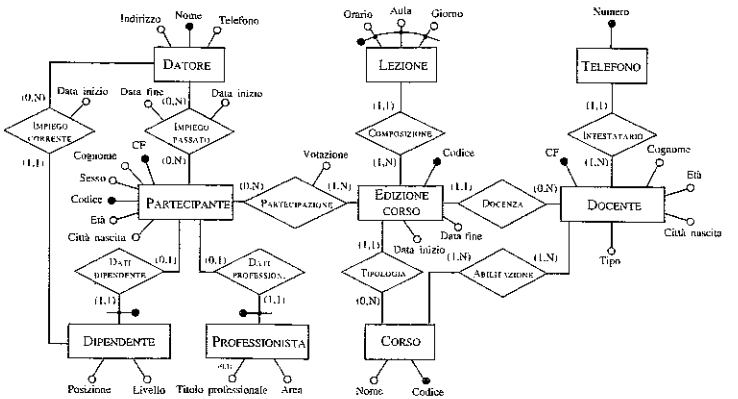

Figura 8.33 Lo schema E-R di Figura 8.30 dopo la fase di ristrutturazione.

sociazioni DOCENZA PASSATA e DOCENZA CORRENTE e delle associazioni analoghe PARTECIPAZIONE PASSATA e PARTECIPAZIONE CORRENTE. Si tratta infatti, in entrambi i casi, di due concetti simili (l'unica differenza è di carattere temporale) tra i quali alcune operazioni non fanno differenza (la 7 e la 8). Il loro accorpamento produrrebbe un altro beneficio: non sarebbe necessario trasferire occorrenze da un'associazione a un'altra quando un'edizione di corso termina. Per le partecipazioni ai corsi, un inconveniente è la presenza dell'attributo Votazione che non si applica alle partecipazioni correnti e quindi provocherebbe la presenza di valori nulli. Del resto, la tavola dei volumi ci dice che il numero medio di occorrenze dell'entità PARTECIPAZIONE CORRENTE è 500 e quindi, supponendo di aver bisogno di 4 byte per memorizzare la votazione, lo spreco di memoria sarebbe di soli 2 kilobyte. Decidiamo quindi di accorpare le due coppie di relazioni come descritto in Figura 8.33. Va aggiunto il vincolo non esprimibile dallo schema che un docente non può insegnare più di una edizione di corso nello stesso periodo e, analogamente, il vincolo che un partecipante non può seguire più di un corso nello stesso periodo.

Infine, bisogna eliminare l'attributo multivalore Telefono associato all'entità DOCENTE. Per far questo, introduciamo una nuova entità TELEFONO legata da una associazione uno a molti con l'entità DOCENTE, che viene privata del relativo attributo.

È interessante osservare che le decisioni prese in questa fase ribaltano, in qualche maniera, decisioni prese in fase di progettazione concettuale. Questo però non deve sorprenderci: l'obiettivo della progettazione concettuale è solo quello di rappresentare nella maniera migliore la realtà d'interesse, mentre nella progettazione logica dobbiamo cercare di ottimizzare le prestazioni ed è quasi inevitabile dover rivedere le decisioni prese.

Scelta degli identificatori principali Solo l'entità PARTECIPANTE presenta due identificatori: il codice fiscale e il codice interno. Tra i due è certamente preferibile scegliere il secondo. Infatti, un codice fiscale richiede 16 byte di memoria mentre un codice interno, che serve a distinguere al più 5000 occorrenze (vedi tavola dei volumi), richiede non più di 2 byte.

C'è in effetti un'altra considerazione di carattere pragmatico da fare sugli identificatori e che riguarda l'entità EDIZIONE CORSO. Questa entità è identificata dall'attributo Data inizio e dall'entità CORSO. Ne risulta un identificatore piuttosto pesante che, in una rappresentazione relazionale, deve essere usato per rappresentare due associazioni (PARTECIPAZIONE e DOCENZA) con molte occorrenze. Si può osservare però che ogni corso ha un codice e che, in media, il numero di edizioni di un corso è pari a cinque. Questo significa che è sufficiente aggiungere un intero di una cifra al codice di un corso per avere un identificatore delle edizioni dei corsi, operazione che può essere fatta durante la creazione di una nuova edizione in maniera piuttosto efficiente e sicura. Da questa discussione risulta che è conveniente definire un nuovo identificatore per le edizioni dei corsi che rimpiazza l'identificatore esterno precedente. Questo è un esempio di analisi e ristrutturazione che non rientra in nessuna delle categorie generali viste ma che, nei casi pratici, capita di incontrare.

Abbiamo con questo terminato la fase di ristrutturazione dello schema E-R originale. Lo schema risultante è quello in Figura 8.33.

8.5.2 Traduzione verso il relazionale

Seguendo la strategia di traduzione descritta in questo capitolo, lo schema E-R in Figura 8.33 può essere tradotto nel seguente schema relazionale.

EDIZIONECORSO(Codice, DataInizio, DataFine, Corso, Docente)

LEZIONE(Ora, Aula, Giorno, EdizioneCorso)

DOCENTE(CF, Cognome, Età, CittàNascita, Tipo)

TELEFONO(Numero, Docente), CORSO(Codice, Nome)

ABILITAZIONE(Corso, Docente)

PARTECIPANTE(Codice, CF, Cognome, Età, CittàNascita, Sesso)

PARTECIPAZIONE(Partecipante, EdizioneCorso, Votazione*)

DATORE(Nome, Telefono, Indirizzo)

IMPIEGOPASSATO(Partecipante, Datore, DataInizio, DataFine)

PROFESSIONISTA(Partecipante, Area, Titolo*)

DIPENDENTE(Partecipante, Livello, Posizione, Datore, DataInizio)

Lo schema logico ottenuto va naturalmente completato con una documentazione di supporto che descriva, tra l'altro, tutti i vincoli di integrità referenziale che sussistono tra le varie relazioni. Questo può essere fatto usando la notazione grafica introdotta nel Paragrafo 8.4.7.

8.6 Progettazione logica con gli strumenti CASE

La fase di progettazione logica viene generalmente supportata da tutti gli strumenti CASE di ausilio allo sviluppo di basi di dati. In particolare, trattandosi di un'operazione basata su criteri precisi, la fase di traduzione verso il modello relazionale viene effettuata da questi sistemi in maniera pressoché automatica. La fase di ristrutturazione dello schema che precede la traduzione vera e propria è invece difficilmente automatizzabile e i vari prodotti non la supportano o lo fanno solo parzialmente, ricorrendo a soluzioni semplificate. Per esempio, alcuni sistemi traducono automaticamente tutte le generalizzazioni secondo uno solo dei metodi descritti nel Paragrafo 8.3.2. Abbiamo visto però che la ristrutturazione di schemi E-R è un momento importante della progettazione perché affronta alcune problematiche (analisi delle ridondanze e trasformazioni orientate all'ottimizzazione) che è possibile risolvere prima di effettuare la traduzione e che non sono di pertinenza della progettazione concettuale. Il progettista dovrebbe quindi curare questo aspetto senza affidarsi completamente allo strumento a disposizione.

Un esempio di prodotto della fase di traduzione automatica fatta con uno strumento CASE viene riportato in Figura 8.34. L'esempio fa riferimento allo schema concettuale riportato nella Figura 7.30 del capitolo precedente. Lo schema risultato viene rappresentato in una forma grafica che rappresenta le tabelle relazionali insie-

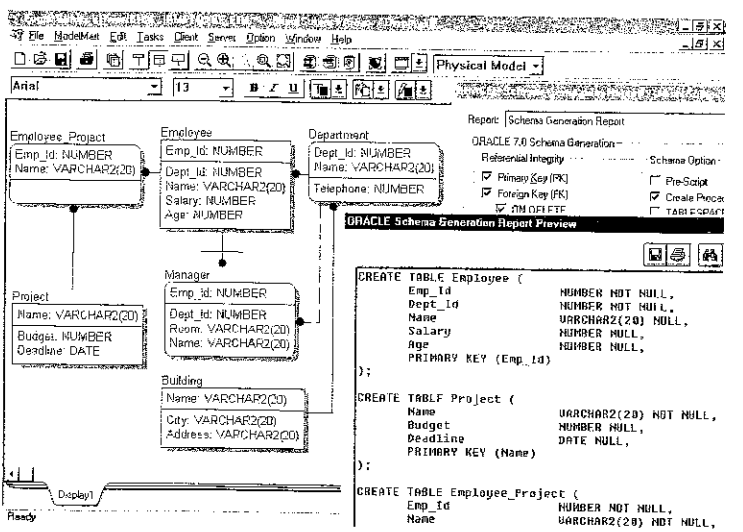

Figura 8.34 Progettazione logica fatta con uno strumento CASE.

me alle associazioni dello schema di partenza. Si osservi come l'associazione molti a molti tra EMPLOYEE e PROJECT sia stata tradotta in una relazione e come siano stati aggiunti nuovi attributi alle relazioni delle entità per rappresentare le associazioni uno a molti e uno a uno. Nella figura viene anche riportato il codice SQL, generato automaticamente dal sistema, che permette di definire la base di dati su uno specifico sistema di gestione di basi di dati. Un altro esempio viene riportato in Figura 8.35 nel quale viene mostrata la rappresentazione in UML di una base di dati relazionale.

Questo schema è stato ottenuto in maniera automatica a partire dallo schema presentato in Figura 7.31. In questa notazione le tabelle vengono rappresentate da classi "speciali" dei diagrammi delle classi, come indicato dal simbolo in alto a destra. I vincoli di integrità referenziale vengono invece rappresentati da associazioni particolari.

Alcuni strumenti CASE sono in grado di comunicare direttamente con un DBMS e costruire autonomamente la corrispondente base di dati relazionale. Altri sistemi forniscono strumenti per effettuare anche l'operazione inversa: ricostruire uno schema concettuale a partire da uno schema relazionale esistente. Questa operazione viene chiamata reingegnerizzazione (o reverse engineering) e risulta particolarmente utile per un'analisi di un sistema informativo precedentemente realizzato, eventualmente orientata a una migrazione verso un nuovo sistema di gestione di dati.

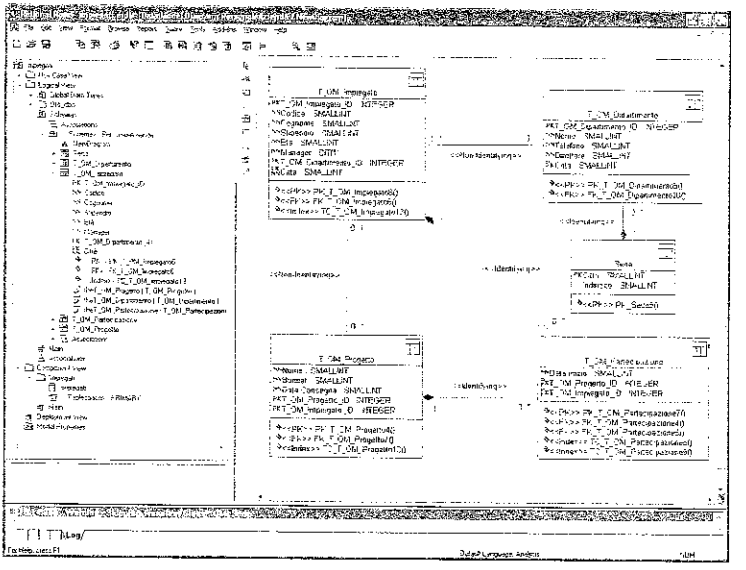

Figura 8.35 Progettazione logica con UML.

Note bibliografiche

Anche la progettazione logica dei dati è affrontata in dettaglio nei libri in inglese di Batini, Ceri e Navathe [8] e Teorey [75]. Il problema della traduzione di uno schema Entità-Relazione nel modello relazionale è discusso nell'articolo originale di Chen [28] e in un articolo di Teorey, Yang e Fry [77].

Esercizi di varia complessità sulla progettazione logica di basi di dati si possono trovare sui testi di Cabibbo, Torlone e Batini [16] e di Francalanci, Schreiber e Tanca [44].

Esercizi

Soluzioni sul sito http://www.ateneonline.it/atzeni

8.1 Si consideri lo schema Entità-Relazione ottenuto come soluzione dell'Esercizio 7.4. Fare delle ipotesi sul volume dei dati e sulle operazioni possibili su questi dati e, sulla base di queste ipotesi, effettuare le necessarie ristrutturazioni dello schema. Effettuare poi la traduzione verso il modello relazionale.

8.2 Tradurre lo schema Entità-Relazione sul personale di un'azienda che abbiamo più volte incontrato (e riportato per comodità in Figura 8.36) in uno schema del modello relazionale.

8.3 Tradurre lo schema Entità-Relazione ottenuto nell'Esercizio 7.6 in uno schema del modello relazionale.

8.4 Definire uno schema logico relazionale corrispondente allo schema E-R ottenuto nell'Esercizio 7.10. Per la fase di ristrutturazione, indicare le possibili alternative e sceglierne poi una, facendo assunzioni sui parametri quantitativi. Come riferimento per i parametri principali, assumere che la base di dati riguardi cen-

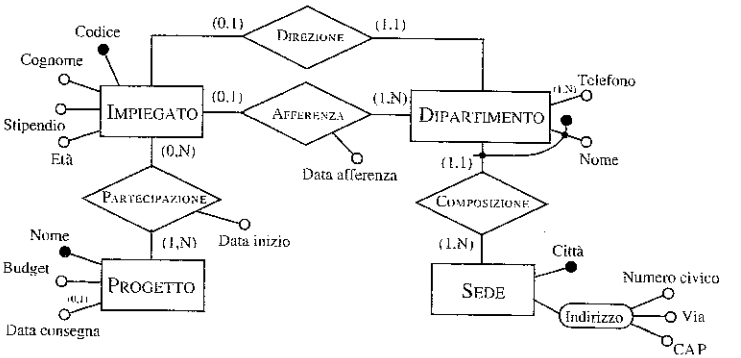

Figura 8.36 Uno schema E-R sul personale di un'azienda.

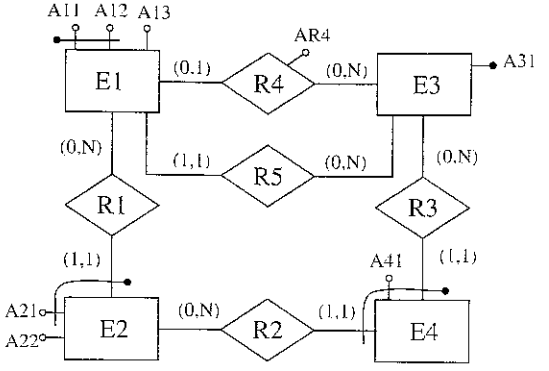

Figura 8.37 Uno schema E-R da tradurre.

to condomini, mediamente con cinque scale ciascuno, e che ogni scala abbia mediamente venti appartamenti e che le operazioni principali siano la registrazione di una spesa (cinquanta all'anno per condominio più dieci per scala e cinque per appartamento) e di un pagamento (dieci all'anno per appartamento); annualmente viene stilato il bilancio di ciascun condominio, con il totale degli accrediti e degli addebiti per ciascun appartamento e quindi il calcolo del nuovo saldo (la stampa di ciascun bilancio deve essere organizzata per scale e ordinata).

8.5 Tradurre lo schema Entità-Relazione di Figura 8.37 in uno schema di basi di dati relazionale. Per ciascuna relazione (dello schema relazionale) si indichi la chiave (che si può supporre unica) e, per ciascun attributo, si specifichi se sono ammessi valori nulli (supponendo che gli attributi dello schema E-R non ammettano valori nulli).

8.6 Sia dato il seguente schema Entità-Relazione in Figura 8.38. Ristrutturare lo schema, eliminando le gerarchie, supponendo che le operazioni più significative siano le seguenti, ciascuna eseguita 10 volte al giorno:

Operazione 1: accesso agli attributi A_{21}, A_{22}, A_{11}, A_{12}, A_{13} dell'entità E_2;
Operazione 2: accesso agli attributi A_{41}, A_{42}, A_{31}, A_{11}, A_{12}, A_{13} dell'entità E_4;
Operazione 3: accesso agli attributi A_{51}, A_{52}, A_{31}, A_{11}, A_{12}, A_{13} dell'entità E_5.

8.7 Si consideri lo schema concettuale di Figura 8.39, che descrive i dati di conti correnti bancari. Si osservi che un cliente può essere titolare di più conti correnti e che uno stesso conto corrente può essere intestato a diversi clienti. Si supponga che su questi dati, siano definite le seguenti operazioni principali.

Operazione 1: apri un conto a un cliente.
Operazione 2: leggi il saldo totale di un cliente.
Operazione 3: leggi il saldo di un conto.
Operazione 4: ritira i soldi da un conto con una transazione allo sportello.
Operazione 5: deposita i soldi in un conto con una transazione allo sportello.

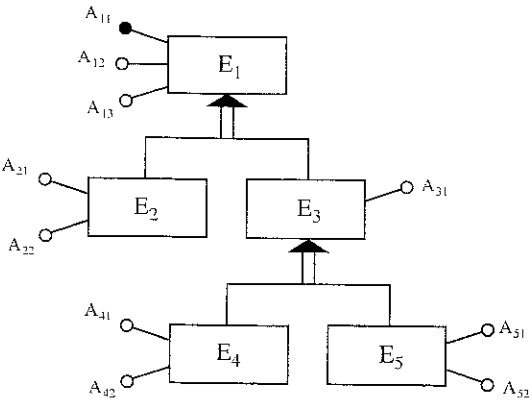

Figura 8.38 Uno schema E-R con generalizzazioni.

Operazione 6: mostra le ultime 10 transazioni di un conto.
Operazione 7: registra transazione esterna per un conto.
Operazione 8: prepara rapporto mensile dei conti.
Operazione 9: trova il numero dei conti posseduti da un cliente.
Operazione 10: mostra le transazione degli ultimi 3 mesi dei conti delle società con saldo negativo.

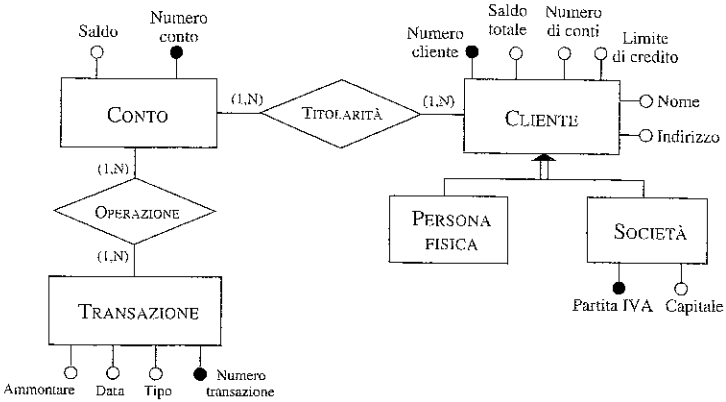

Figura 8.39 Uno schema E-R da tradurre.

Tavola dei volumi

Concetto	Tipo	Volume
Cliente	E	15 000
Conto	E	20 000
Transazione	E	600 000
Persona Fisica	E	14 000
Società	E	1000
Titolarità	R	30 000
Operazione	R	800 000

Tavola delle operazioni

Operazione	Tipo	Frequenza
Op. 1	I	100/giorno
Op. 2	I	2000/giorno
Op. 3	I	1000/giorno
Op. 4	I	2000/giorno
Op. 5	I	1000/giorno
Op. 6	I	200/giorno
Op. 7	B	1500/giorno
Op. 8	B	1/mese
Op. 9	B	75/giorno
Op. 10	I	20/giorno

Figura 8.40 Tavole dei volumi e delle operazioni per lo schema in Figura 8.39.

Si supponga infine che, in fase operativa, i dati di carico per questa applicazione bancaria siano quelli riportati in Figura 8.40.

Effettuare la fase di progettazione logica sullo schema E-R tenendo conto dei dati forniti. Nella fase di ristrutturazione si tenga conto del fatto che sullo schema esistono due ridondanze: gli attributi Saldo totale e Numero di conti dell'entità CLIENTE. Essi possono infatti essere derivati dall'associazione TITOLARITÀ e dall'entità CONTO.

8.8 Si consideri lo schema concettuale della Figura 8.41, nel quale l'attributo Saldo di una occorrenza di CONTOCORRENTE è ottenuto come somma dei valori dell'attributo Importo per le occorrenze di OPERAZIONE a essa correlate tramite l'associazione MOVIMENTO.

Valutare se convenga o meno mantenere la ridondanza, tenendo conto del fatto che le cardinalità delle due entità sono L_{CC} = 2000 e L_{OP} = 20 000 e che le operazioni più importanti sono:

- OP_1 scrittura di un movimento, con frequenza f_1 = 10;
- OP_2 lettura del saldo con frequenza f_2 = 1000.

Figura 8.41 Schema per l'Esercizio 8.8.

8.9 Lo schema concettuale della Figura 8.42 rappresenta un insieme di viaggi e un insieme di partecipanti a questi viaggi. Ogni viaggio ha diversi partecipanti e la stessa persona può partecipare a più viaggi. Nello schema l'attributo Incasso è ridondante perché può essere ottenuto moltiplicando il costo del viaggio per il numero di partecipanti (cioè il prodotto del valore dell'attributo Costo di ogni occorrenza dell'entità Viaggio per il numero di occorrenze dell'entità PARTECIPANTE a cui è correlato tramite l'associazione V-P).
Valutare se convenga o meno mantenere la ridondanza, tenendo conto del fatto che le cardinalità dei concetti in gioco sono NViaggio=20.000, NV-P=300.000 e NPartecipante = 100.000 e che le operazioni più importanti sono:

- Op1 calcolo dell'incasso di un viaggio, con frequenza $f_1 = 10$ al mese;
- Op2 inserimento di un partecipante al viaggio, con frequenza $f_2 = 5$ al giorno.

Assumere che il costo di una lettura e quello di una scrittura siano uguali e che un mese sia di 20 giorni lavorativi.

8.10 Considerare un frammento di schema E-R contenente le entità E_0 (con attributi $A_{0,1}$, identificante, e $A_{0,2}$), E_1 (con attributo $A_{1,1}$), E_2 (con attributo $A_{2,1}$), E_3 (con attributo $A_{3,1}$), E_4 (con attributo $A_{4,1}$) e due generalizzazioni, la prima totale con genitore E_0 e figlie E_1 ed E_2 e la seconda parziale con genitore E_1 e figlie E_3 ed E_4. Supporre paragonabili fra loro le dimensioni degli attributi. Indicare, per ciascuno dei casi seguenti, considerati separatamente, la scelta (o le scelte, qualora ve ne siano diverse paragonabili) che si ritiene preferibile per l'eliminazione delle generalizzazioni nella progettazione logica:

- le operazioni nettamente più frequenti sono due, che accedono rispettivamente a tutte le occorrenze di E_1 (con stampa dei valori di $A_{0,1}$, $A_{0,2}$ e $A_{1,1}$) e a tutte le occorrenze di E_2 (con stampa dei valori di $A_{0,1}$, $A_{0,2}$ e $A_{2,1}$);
- le operazioni nettamente più frequenti sono due, che accedono rispettivamente a tutte le occorrenze di E_1 (con stampa dei valori di $A_{0,1}$, $A_{1,1}$ e, se esiste, $A_{3,1}$) e a tutte le occorrenze di E_2 (con stampa dei valori di $A_{0,1}$ e $A_{2,1}$);
- l'operazione nettamente più frequente prevede l'accesso a tutte le occorrenze di E_0 (con stampa dei valori di $A_{0,1}$, $A_{0,2}$);
- l'operazione nettamente più frequente prevede l'accesso a occorrenze (tutte o alcune) di E_0 (con stampa dei valori di tutti gli attributi, inclusi quelli di tutte le altre entità, ove applicabili).

8.11 Mostrare uno schema E-R che descriva una realtà d'interesse corrispondente a quella rappresentata da uno schema relazionale composto dalle seguenti relazioni:

- CICLISTA(Codice, Cognome, Nome, Squadra);
- COMPETIZIONE(Codice, Nome, Organizzatore, KmTotali);
- TAPPA(Numero, Competizione, Partenza, Arrivo, KM) con vincolo di integrità referenziale fra Competizione e COMPETIZIONE;

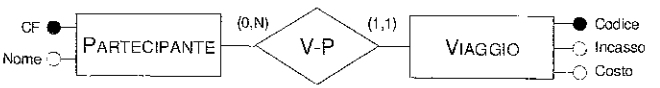

Figura 8.42 Schema per l'Esercizio 8.9.

- CLASSIFICATAPPA(NumTappa, Competizione, Ciclista, Posizione, Distacco) con vincoli di integrità referenziale fra gli attributi NumeroTappa, Competizione e la relazione TAPPA e fra Ciclista e la relazione CICLISTA;
- CLASSIFICAGENERALE(NumTappa, Competizione, Ciclista, Posizione, Distacco) con vincoli di integrità referenziale fra gli attributi NumeroTappa, Competizione e la relazione TAPPA e fra Ciclista e la relazione CICLISTA.

8.12 Per ciascuno dei seguenti schemi logici (in cui A* indica che l'attributo A ammette valori nulli), mostrare uno schema concettuale dal quale possa essere stato ottenuto (indicando anche cardinalità e identificatori).
Schema (a):

- LIBRI(Codice, Titolo, Genere*, Autore) con vincolo di integrità referenziale fra Autore e la relazione SCRITTORI;
- EDIZIONI(Libro, Editore, Collana*, Anno) con vincoli di integrità referenziale fra Libro e la relazione LIBRI e fra Editore e la relazione EDITORI;
- EDITORI(Nome, Città);
- SCRITTORI(Codice, Cognome, Nome).

Schema (b):

- EDITORI e SCRITTORI come nello schema (a);
- LIBRI(Codice, Titolo, Genere*) con vincolo di integrità referenziale fra Genere e la relazione GENERI;
- EDIZIONI(Libro, Editore, Collana*, Anno) con vincoli di integrità referenziale fra Libro e la relazione LIBRI, fra Editore e la relazione EDITORI e tra Collana e la relazione COLLANE;
- AUTORI(Libro, Scrittore) con vincoli di integrià referenziale fra Libro e la relazione LIBRI e fra Scrittore e la relazione SCRITTORI;
- COLLANE(SiglaCollana, Nome);
- GENERI(SiglaGenere, Nome).

8.13 Per ciascuno dei seguenti schemi logici (in cui A* indica che l'attributo A ammette valori nulli), mostrare uno schema concettuale dal quale possa essere stato ottenuto (indicando anche cardinalità e identificatori).
Schema (a):

- CASECOSTRUTTRICI(Codice, Nome, Nazione*);
- MODELLI(Casa, Nome, Categoria*) con vincolo di integrità referenziale fra l'attributo Casa e la relazione CASECOSTRUTTRICI;
- AUTOMOBILI(Targa, Casa, Modello, Anno, Proprietario) con vincoli di integrità referenziale fra gli attributi Casa e Modello e la relazione MODELLI e fra l'attributo Proprietario e la relazione PERSONE;
- PERSONE(CodiceFiscale, Cognome, Nome).

Schema (b):

- MODELLI e PERSONE come nello Schema (a);
- CASECOSTRUTTRICI (Codice, Nome, Nazione*) con vincolo di integrità referenziale fra Nazione e la relazione NAZIONI;
- VERSIONI (Casa, Modello, CodiceVersione, Cilindrata) con vincolo di integrià referenziale fra gli attributi Casa e Modello e la relazione MODELLI;
- AUTOMOBILI(Targa, Casa, Modello, Versione, Anno) con vincolo di integrità referenziale fra gli attributi Casa, Modello, Versione e la relazione VERSIONI;
- ACQUISTO (Auto, Data, Acquirente) con vincoli di integrità referenziale fra Auto e la relazione AUTOMOBILI e fra Acquirente e la relazione PERSONE;
- NAZIONI(SiglaNazione, Nome).

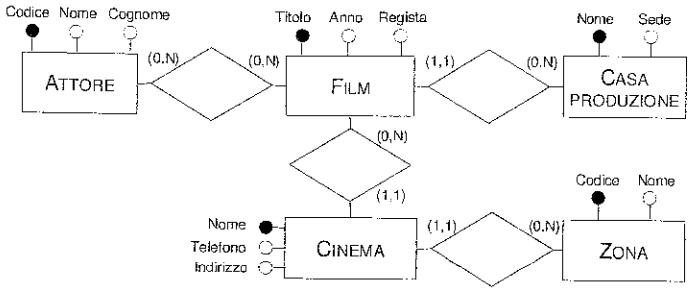

Figura 8.43 Schema per l'Esercizio 8.15..

8.14 Progettare lo schema logico relazionale corrispondente allo schema concettuale definito nell'Esercizio 7.14 mostrando i nomi degli attributi, i vincoli di chiave e di integrità referenziale.

8.15 Mostrare uno schema logico che possa essere ottenuto dallo schema E-R in Figura 8.43.

8.16 Tradurre lo schema E-R della Figura 8.44 nel corrispondente schema relazionale.

8.17 Mostrare uno schema E-R che descriva una realtà d'interesse corrispondente a quella rappresentata da uno schema relazionale composto dalle seguenti relazioni:

- UTENZE(Prefisso,Numero, CodiceCentrale, Titolare, Indirizzo, DataAttivazione) con vincoli di integrità referenziale fra gli attributi Pre-

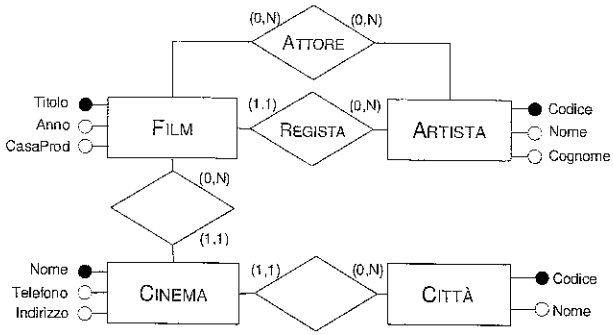

Figura 8.44 Schema per l'Esercizio 8.16..

fisso e CodiceCentrale e la relazione CENTRALI e fra Utente e la relazione UTENTI;

- CENTRALI(Distretto, Codice, Indirizzo, Capacità);
- UTENTI(CodiceFiscale, Cognome, Nome);
- DISTRETTI(Prefisso, Nome, Provincia), con vincolo di integrità referenziale fra l'attributo Provincia e la relazione PROVINCE;
- PROVINCE(Sigla, Nome, Capoluogo);
- BOLLETTE(Prefisso, Numero, DataEmissione, Importo), con vincolo di integrità referenziale fra Prefisso, Numero e la relazione UTENZE;
- PAGAMENTI(Codice,Prefisso,Numero, DataPagamento, Modalità, Importo), con vincolo di integrità referenziale fra Prefisso, Numero e la relazione UTENZE.

8.18 Modificare lo schema prodotto come soluzione all'esercizio precedente supponendo che, oltre alle utenze domestiche, siano descritte anche le utenze radiomobili, ognuna delle quali deve essere associata a un'utenza domestica e quindi avere lo stesso titolare. Le bollette sono emesse con riferimento alle singole utenze. Le utenze radiomobili sono associate a pseudo-distretti, che corrispondono alle aree con lo stesso prefisso.

9 La normalizzazione

In questo capitolo studieremo alcune proprietà, dette *forme normali*, che "certificano" la qualità dello schema di una base di dati relazionale. Vedremo infatti che quando una relazione non soddisfa una forma normale, allora presenta ridondanze e si presta a comportamenti poco desiderabili durante le operazioni di aggiornamento. Questo concetto può di fatto essere utilizzato per effettuare controlli di qualità di basi di dati relazionali e costituisce per questo un utile strumento di analisi nell'ambito dell'attività di progettazione di una base di dati. Per gli schemi che non soddisfano una forma normale, è inoltre possibile applicare un procedimento, detto di *normalizzazione*, che consente di trasformare questi schemi non normalizzati in nuovi schemi per i quali il soddisfacimento di una forma normale è garantito.

Prima di cominciare, bisogna fare due importanti precisazioni. Va detto innanzitutto che le metodologie di progettazione viste nei capitoli precedenti permettono di solito di ottenere schemi che soddisfano una forma normale. In questo contesto, la teoria della normalizzazione costituisce un utile strumento di verifica, in grado di suggerire emendamenti, ma che non può sostituire, soprattutto in applicazioni complesse, le metodologie di analisi e progettazione di più ampio respiro. C'è inoltre da dire che la teoria della normalizzazione è stata studiata nell'ambito del modello relazionale e per questo, all'inizio, presenteremo l'argomento facendo riferimento a questo modello di dati. Vedremo poi però che molte delle considerazioni fatte in precedenza a livello di schema logico possono essere applicate anche su schemi Entità-Relazione e quindi effettuate a monte, per esempio, durante la fase di analisi di qualità della progettazione concettuale.

Affronteremo questo argomento in maniera graduale, discutendo prima i problemi (ridondanze e anomalie) che si possono verificare in una relazione, per poi fornire, partendo sempre da spunti intuitivi, tecniche sistematiche per l'analisi e la normalizzazione. Approfondiremo anche alcuni aspetti da un punto di vista teorico, pur consapevoli che una trattazione completa richiederebbe molto più spazio; tale trattazione avrà quindi scopo prevalentemente esemplificativo. Come già accennato, faremo questo prima con riferimento al modello relazionale, per poi passare a discutere gli stessi concetti nell'ambito del modello Entità-Relazione.

9.1 Ridondanze e anomalie

Introduciamo i primi concetti attraverso un esempio. Consideriamo la relazione di Figura 9.1. Questa relazione ha come chiave l'insieme costituito dagli attributi Impiegato e Progetto. Si può inoltre facilmente verificare che le tuple della relazione soddisfano le seguenti proprietà:

1. lo stipendio di ciascun impiegato è unico ed è funzione del solo impiegato, indipendentemente dai progetti cui partecipa;
2. il bilancio di ciascun progetto è unico e dipende dal solo progetto, indipendentemente dagli impiegati che vi partecipano.

Impiegato	Stipendio	Progetto	Bilancio	Funzione
Rossi	20 000	Marte	2000	tecnico
Verdi	35 000	Giove	15 000	progettista
Verdi	35 000	Venere	15 000	progettista
Neri	55 000	Venere	15 000	direttore
Neri	55 000	Giove	15 000	consulente
Neri	55 000	Marte	2000	consulente
Mori	48 000	Marte	2000	direttore
Mori	48 000	Venere	15 000	progettista
Bianchi	48 000	Venere	15 000	progettista
Bianchi	48 000	Giove	15 000	direttore

Figura 9.1 Esempio di relazione con anomalie.

Questi fatti hanno alcune conseguenze sul contenuto della relazione e sulle operazioni che si possono effettuare su di essa. Limitiamo le nostre considerazioni alla prima proprietà lasciando per esercizio quelle, analoghe, relative alla seconda.

- Il valore dello stipendio di ciascun impiegato è ripetuto in tutte le tuple relative a esso: si ha quindi una *ridondanza*; se per esempio un impiegato partecipasse a 20 progetti, il suo stipendio verrebbe ripetuto 20 volte.
- Se lo stipendio di un impiegato varia, è necessario andarne a modificare il valore in tutte le tuple corrispondenti affinché la dipendenza continui a valere; questo inconveniente, che comporta la necessità di effettuare più modifiche contemporaneamente, va sotto il nome di *anomalia di aggiornamento*.
- Se un impiegato interrompe la partecipazione a tutti i progetti senza lasciare l'azienda, e quindi tutte le corrispondenti tuple vengono eliminate, non è possibile conservare traccia del suo nome e del suo stipendio (a meno di ammettere valori nulli sull'unica chiave, il che è, come abbiamo visto nel Capitolo 2, inammissibile), che potrebbero rimanere di interesse; questo problema viene indicato come *anomalia di cancellazione*.
- Analogamente, se si hanno informazioni su un nuovo impiegato, non è possibile inserirle finchè questi non viene assegnato a un progetto; in questo caso parliamo di *anomalia di inserimento*.

Una motivazione intuitiva della presenza di questi inconvenienti può essere la seguente: abbiamo usato un'unica relazione per rappresentare informazioni eterogenee. In particolare, nella relazione sono rappresentati: gli impiegati con i relativi stipendi, i progetti con i relativi bilanci e le partecipazioni degli impiegati ai progetti con le relative funzioni.

Generalizzando, possiamo arrivare alle seguenti conclusioni, che evidenziano i difetti presentati da relazioni che riuniscono concetti fra loro disomogenei.

- È possibile che alcuni dati debbano essere ripetuti in diverse tuple, senza aggiungere in tal modo informazioni significative.

- Se alcune informazioni sono ripetute in modo ridondante, il relativo aggiornamento (concettualmente atomico) deve essere ripetuto per ciascuna occorrenza dei relativi dati. Il fatto che i linguaggi di manipolazione, per esempio SQL, permettano di specificare aggiornamenti multipli per mezzo di un solo comando risolve il problema solo dal punto di vista del programmatore ma non da quello del sistema, perché comunque le tuple della base di dati vanno aggiornate tutte e quindi si deve fisicamente accedere a ciascuna di esse.
- La cancellazione di una tupla, motivata dal fatto che non è più valido l'intero insieme di concetti da essa espressi, per esempio perché uno di essi non sussiste più, può comportare l'eliminazione di tutti i concetti in questione, cioè anche di quelli che conservano la loro validità.
- L'inserimento di informazioni relative a uno solo dei concetti di pertinenza per una relazione non è possibile se non esiste un intero insieme di concetti in grado di costituire una tupla completa (o almeno la sua chiave primaria).

9.2 Dipendenze funzionali

Per studiare in maniera sistematica i concetti introdotti informalmente nel paragrafo precedente, è necessario far uso di uno specifico strumento di lavoro: la *dipendenza funzionale*. Si tratta di un particolare vincolo di integrità per il modello relazionale che, come ci suggerisce il nome, descrive legami di tipo funzionale tra gli attributi di una relazione.

Consideriamo ancora la relazione in Figura 9.1. Abbiamo osservato che lo stipendio di ciascun impiegato è unico e quindi, ogni volta che in una tupla della relazione compare un certo impiegato, il valore del suo stipendio rimane sempre lo stesso. Possiamo cioè dire che il valore dell'attributo Impiegato *determina* il valore dell'attributo Stipendio o, in maniera più precisa, che esiste una funzione che associa a ogni elemento del dominio dell'attributo Impiegato che compare nella relazione un solo elemento del dominio dell'attributo Stipendio. Un discorso analogo si può fare per il legame che intercorre tra gli attributi Progetto e Bilancio perché il valore del progetto determina il valore del bilancio del progetto stesso e quindi tutte le volte che nella relazione compare il nome di un progetto, il bilancio a esso associato sarà sempre lo stesso.

Questo concetto può essere formalizzato come segue. Data una relazione r su uno schema $R(X)$ e due sottoinsiemi di attributi non vuoti Y e Z di X, diremo che esiste su r una dipendenza funzionale tra Y e Z, se, per ogni coppia di tuple t_1 e t_2 di r aventi gli stessi valori sugli attributi Y, risulta che t_1 e t_2 hanno gli stessi valori anche sugli attributi Z.

Una dipendenza funzionale tra gli attributi Y e Z viene generalmente indicata con la notazione $Y \rightarrow Z$ e, come gli altri vincoli di integrità, viene associata a uno schema: una relazione su quello schema verrà considerata corretta se soddisfa tale dipendenza funzionale. Tornando ai nostri esempi possiamo dunque dire che sulla relazione in Figura 9.1 esistono le dipendenze funzionali:

Impiegato → Stipendio

Progetto → Bilancio

Ci sono alcune osservazioni da fare sulle dipendenze funzionali. La prima è che, se l'insieme Z è composto dagli attributi $A_1, A_2, \ldots, A_k$, allora una relazione soddisfa $Y \rightarrow Z$ se e solo se essa soddisfa tutte le k dipendenze $Y \rightarrow A_1, \ldots, Y \rightarrow A_k$. Di conseguenza, quando opportuno, possiamo, senza perdita di generalità, assumere che le dipendenze abbiano la forma $Y \rightarrow A$, in cui A è un singolo attributo.

Una seconda osservazione è la seguente: in base alle definizione data, possiamo notare che, nella nostra relazione, è verificata anche la dipendenza funzionale:

Impiegato Progetto → Progetto

in quanto due tuple con gli stessi valori sulla coppia di attributi Impiegato e Progetto, hanno ovviamente lo stesso valore sull'attributo Progetto, che è uno dei due. Questa è in effetti una dipendenza funzionale banale perché asserisce una proprietà ovvia di una relazione. Le dipendenze funzionali dovrebbero invece servire a descrivere proprietà significative dell'applicazione che stiamo rappresentando. Diremo quindi che, in generale, una dipendenza funzionale $Y \rightarrow A$ è *non banale* se A non compare tra gli attributi di Y. D'ora in avanti, salvo che nella trattazione teorica del Paragrafo 9.6, faremo riferimento solo a dipendenze funzionali non banali, omettendo spesso per brevità tale aggettivo.

Un'ultima osservazione sulle dipendenze funzionali riguarda il loro legame con il vincolo di chiave. Se prendiamo una chiave K di una relazione r, si può facilmente verificare che esiste una dipendenza funzionale tra K e ogni altro attributo dello schema di r. Questo perché, per definizione stessa di vincolo di chiave, non possono esistere due tuple con gli stessi valori su K e quindi una dipendenza funzionale che ha K al primo membro sarà sempre soddisfatta. Con riferimento al nostro esempio, abbiamo detto che gli attributi Impiegato e Progetto formano una chiave. Possiamo allora affermare che, per esempio, vale la dipendenza funzionale Impiegato Progetto → Funzione. In particolare, esisterà una dipendenza funzionale tra una chiave di una relazione e tutti gli attributi dello schema della relazione (esclusi quelli della chiave stessa per quanto appena detto). Nel nostro caso abbiamo cioè che:

Impiegato Progetto → Stipendio Bilancio Funzione

Possiamo quindi concludere dicendo che il vincolo di dipendenza funzionale *generalizza* il vincolo di chiave. Più precisamente, possiamo dire che una dipendenza funzionale $Y \rightarrow Z$ su uno schema $R(X)$ degenera nel vincolo di chiave se l'unione di Y e Z è pari a X. In tal caso infatti, Y è (super)chiave per lo schema $R(X)$.

9.3 Forma normale di Boyce e Codd

9.3.1 Definizione di forma normale di Boyce e Codd

In questo paragrafo rivisitiamo i concetti illustrati nel Paragrafo 9.1, alla luce di quanto detto sulle dipendenze funzionali: l'idea fondamentale è che si possono introdurre proprietà, dettte *forme normali*, definite con riferimento alle dipendenze funzionali, che sono soddisfatte quando non ci sono anomalie.

Osserviamo che, nel nostro esempio, le due proprietà causa di anomalie corrispondono esattamente ad attributi coinvolti in dipendenze funzionali.

- La proprietà "Lo stipendio di ciascun impiegato è funzione del solo impiegato, indipendentemente dai progetti cui partecipa" implica il soddisfacimento della dipendenza funzionale Impiegato → Stipendio.
- La proprietà "Il bilancio di ciascun progetto dipende dal solo progetto, indipendentemente dagli impiegati che vi partecipano" corrisponde alla dipendenza funzionale Progetto → Bilancio.

Inoltre, è opportuno notare che l'attributo Funzione indica, per ciascuna tupla, il ruolo svolto dall'impiegato nel progetto. Tale ruolo è unico, per ciascuna coppia impiegato-progetto. Anche questa proprietà può essere modellata per mezzo di una dipendenza funzionale.

- La proprietà "In ciascun progetto, ciascuno degli impiegati coinvolti può svolgere una e una sola funzione" implica il soddisfacimento della dipendenza funzionale Impiegato Progetto → Funzione. Come abbiamo accennato nel paragrafo precedente, questo è anche una conseguenza del fatto che gli attributi Impiegato e Progetto formano la chiave della relazione.

Abbiamo visto, nel Paragrafo 9.1, come la prima proprietà (e quindi la corrispondente dipendenza funzionale) generi ridondanze e anomalie indesiderate. Con argomenti analoghi possiamo rilevare come anche la seconda dipendenza funzionale generi ridondanze e anomalie. È diverso il caso per la terza dipendenza che non genera mai ridondanze perché, essendo Impiegato Progetto la chiave, la relazione non può contenere due tuple uguali su questi attributi (e quindi sull'attributo Funzione). Per quanto riguarda le anomalie, da un punto di vista concettuale possiamo dire che essa non ne può generare, in quanto ogni impiegato ha uno stipendio (e uno solo) e ogni progetto ha un bilancio (e uno solo), e quindi per ogni coppia impiegato-progetto è possibile avere valori univoci per tutti gli altri attributi della relazione. In alcuni casi tali valori potrebbero non essere disponibili ma, non facendo essi parte della chiave, potremmo sostituirli senza problemi con valori nulli.

Riassumendo, le dipendenze:

Impiegato → Stipendio

Progetto → Bilancio

sono causa di anomalie, mentre la dipendenza:

Impiegato Progetto → Funzione

non lo è. La differenza, come accennato, risiede nel fatto che Impiegato Progetto è una superchiave (specificamente, è l'unica chiave) della relazione. In effetti, tutti i ragionamenti che abbiamo sviluppato sono legati esclusivamente a questa proprietà e non si riferiscono in alcun modo ad aspetti specifici dell'applicazione d'interesse. Possiamo quindi concludere che le ridondanze e le anomalie sono causate dalle dipendenze funzionali $X \rightarrow A$ che permettono la presenza di più tuple fra loro uguali

sugli attributi in X, cioè, in altre parole, dalle dipendenze funzionali $X \rightarrow A$ tali che X non contiene una chiave.

Precisiamo queste idee per mezzo della più importante delle forme normali, detta di Boyce e Codd, dal nome dei suoi ideatori. Una relazione r è in *forma normale di Boyce e Codd* se per ogni dipendenza funzionale (non banale) $X \rightarrow A$ definita su di essa, X contiene una chiave K di r, cioè X è superchiave per r.

Anomalie e ridondanze, come discusse nel paragrafo precedente, non si presentano per relazioni in forma normale di Boyce e Codd, perché i concetti indipendenti sono separati, uno per relazione.

9.3.2 Decomposizione in forma normale di Boyce e Codd

Data una relazione che non soddisfa la forma normale di Boyce e Codd è possibile, in molti casi, sostituirla con due o più relazioni normalizzate attraverso un processo detto di *normalizzazione*. Questo processo si fonda su un semplice criterio: se una relazione rappresenta più concetti indipendenti, allora va decomposta in relazioni più piccole, una per ogni concetto. Presentiamo in primo luogo l'idea in modo informale per poi precisare, nei paragrafi successivi, alcuni aspetti.

Se alla relazione in Figura 9.1 sostituiamo le tre relazioni in Figura 9.2, ottenute per mezzo di proiezioni sugli insiemi di attributi rispettivamente corrispondenti ai tre concetti prima menzionati, eliminiamo anomalie e ridondanze: le tre relazioni sono infatti in forma normale di Boyce e Codd. Si osservi che abbiamo costruito le relazioni in modo che a ciascuna dipendenza corrisponda una diversa relazione la cui chiave è proprio il primo membro della dipendenza stessa. In tal modo, il soddisfacimento della forma normale di Boyce e Codd è garantito, per la definizione stessa di tale forma normale.

Nell'esempio, la separazione delle dipendenze (e quindi dei concetti da esse rappresentati) è stata facilitata dalla struttura delle dipendenze stesse, "naturalmente" separate e indipendenti l'una dall'altra. In effetti, in molti casi pratici, la decomposizione può essere effettuata producendo tante relazioni quante sono le dipendenze funzionali definite (o meglio, le dipendenze funzionali con diverso primo membro). In generale, purtroppo, le dipendenze possono avere una struttura complessa: può non essere necessario (o possibile) basare la decomposizione su tutte le dipendenze e può essere difficile individuare quelle su cui si deve basare la decomposizione. Per questa ragione è importante studiare formalmente le proprietà delle dipendenze funzionali, cui accenneremo nel Paragrafo 9.6.

9.4 Proprietà delle decomposizioni

In questo paragrafo esaminiamo più in dettaglio il concetto di decomposizione, notando come non tutte le decomposizioni siano desiderabili e individuando alcune proprietà essenziali che devono essere soddisfatte da una "buona" decomposizione.

Impiegato	Stipendio
Rossi	20 000
Verdi	35 000
Neri	55 000
Mori	48 000
Bianchi	48 000

Progetto	Bilancio
Marte	2000
Giove	15 000
Venere	15 000

Impiegato	Progetto	Funzione
Rossi	Marte	tecnico
Verdi	Giove	progettista
Verdi	Venere	progettista
Neri	Venere	direttore
Neri	Giove	consulente
Neri	Marte	consulente
Mori	Marte	direttore
Mori	Venere	progettista
Bianchi	Venere	progettista
Bianchi	Giove	direttore

Figura 9.2 Decomposizione della relazione in Figura 9.1.

9.4.1 Decomposizione senza perdita

Per discutere la prima proprietà, esaminiamo la relazione in Figura 9.3.
Tale relazione soddisfa le dipendenze funzionali:

$$\text{Impiegato} \rightarrow \text{Sede}$$

$$\text{Progetto} \rightarrow \text{Sede}$$

che, sostanzialmente, specificano il fatto che ciascun impiegato opera presso un'unica sede e che ciascun progetto è sviluppato presso un'unica sede. Si osservi che ciascun impiegato può partecipare a più progetti anche se, sulla base delle dipendenze funzionali, devono essere tutti progetti assegnati alla sede cui afferisce.

Impiegato	Progetto	Sede
Rossi	Marte	Roma
Verdi	Giove	Milano
Verdi	Venere	Milano
Neri	Saturno	Milano
Neri	Venere	Milano

Figura 9.3 Relazione per la discussione sulla decomposizione senza perdita.

Operando come nei casi precedenti, separando cioè sulla base delle dipendenze, saremmo portati a decomporre la relazione in due parti:

- una relazione sugli attributi Impiegato e Sede, in corrispondenza alla dipendenza Impiegato → Sede;
- l'altra sugli attributi Progetto e Sede, in corrispondenza alla dipendenza funzionale Progetto → Sede.

L'istanza in Figura 9.3 verrebbe decomposta, per mezzo di proiezioni sugli attributi coinvolti, nelle due relazioni in Figura 9.4.

Esaminiamo in dettaglio le due relazioni. In particolare, consideriamo come sia possibile (provare a) ricostruire informazioni sulla partecipazione degli impiegati ai progetti. L'unica possibilità che abbiamo è di utilizzare l'attributo Sede, che è l'unico attributo comune alle due relazioni: possiamo cioè correlare un impiegato a un progetto se il progetto viene svolto nella sede presso cui l'impiegato opera. Purtroppo, però, in questo caso non riusciamo a ricostruire tutte e sole le informazioni nella relazione originaria: per esempio l'impiegato Verdi lavora a Milano e il progetto Saturno viene svolto presso la sede di Milano, ma in effetti Verdi non lavora a tale progetto.

Possiamo generalizzare l'osservazione notando come la ricostruzione della relazione originaria a partire dalle sue proiezioni (cioè la ricostruzione di tutte le sue tuple a partire dalle tuple nelle proiezioni) debba intuitivamente essere effettuata per mezzo di una operazione di join naturale delle due proiezioni. Purtroppo, il join naturale delle due relazioni in Figura 9.4 produce la relazione in Figura 9.5, che è diversa dalla relazione in Figura 9.3.

In effetti, la relazione in Figura 9.5 contiene tutte le tuple della relazione originaria (Figura 9.3) più altre tuple (nella rappresentazione tabellare, le ultime due). La situazione nell'esempio corrisponde al caso generale: data una relazione r su un insieme di attributi X, se X_1 e X_2 sono due sottoinsiemi di X la cui unione sia pari a X stesso, allora il join delle due relazioni ottenute per proiezione da r su X_1 e X_2, rispettivamente, è una relazione che contiene tutte le tuple di r, più eventualmente altre, che possiamo chiamare "spurie". Diciamo che r *si decompone senza perdita* su X_1 e X_2 se il join delle due proiezioni è uguale a r stessa (cioè non contiene tuple spurie). È chiaramente desiderabile, anzi, è un requisito irrinunciabile, che una decomposizione effettuata a fini di normalizzazione sia senza perdita.

È possibile individuare una condizione che garantisce la decomposizione senza perdita di una relazione, come segue. Sia r una relazione su X e siano X_1 e X_2

Impiegato	Sede
Rossi	Roma
Verdi	Milano
Neri	Milano

Progetto	Sede
Marte	Roma
Giove	Milano
Saturno	Milano
Venere	Milano

Figura 9.4 Relazioni ottenute per proiezione dalla relazione in Figura 9.3.

Impiegato	Progetto	Sede
Rossi	Marte	Roma
Verdi	Giove	Milano
Verdi	Venere	Milano
Neri	Saturno	Milano
Neri	Venere	Milano
Verdi	Saturno	Milano
Neri	Giove	Milano

Figura 9.5 Il risultato del join delle relazioni in Figura 9.4.

sottoinsiemi di X tali che $X_1 \cup X_2 = X$; inoltre, sia $X_0 = X_1 \cap X_2$; allora: r si decompone senza perdita su X_1 e X_2 se soddisfa la dipendenza funzionale $X_0 \rightarrow X_1$ oppure la dipendenza funzionale $X_0 \rightarrow X_2$.

In altre parole, possiamo dire che r si decompone senza perdita su due relazioni se l'insieme degli attributi comuni alle due relazioni è chiave per almeno una delle relazioni decomposte. Nell'esempio, possiamo vedere che l'intersezione degli insiemi di attributi su cui abbiamo effettuato le due proiezioni è costituita dall'attributo Sede, che non è il primo membro di alcuna dipendenza funzionale.

Possiamo giustificare la condizione vista nel modo seguente, con riferimento a una relazione sugli attributi ABC e alle sue proiezioni su AB e AC. Supponiamo che r soddisfi $A \rightarrow C$. Allora, A è chiave per la proiezione di r su AC e quindi non ci sono in tale proiezione due tuple diverse con gli stessi valori su A. Il join costruisce tuple a partire dalle tuple nelle due proiezioni. Consideriamo una generica tupla $t = (a, b, c)$ nel risultato del join e facciamo vedere che appartiene a r, provando così l'uguaglianza delle due relazioni. La tupla t è ottenuta da $t_1 = (a, b)$ nella proiezione di r su AB e $t_2 = (a, c)$ nella proiezione di r su AC. Quindi, per la definizione dell'operatore di proiezione, devono esistere due tuple in r, t_1' con valori a e b su AB e t_2' con valori a e c su AC. Poiché r soddisfa la $A \rightarrow C$, esiste un solo valore su C in r associato al valore a su A: dato che (a, c) compare nella proiezione, tale valore è esattamente c. Quindi, il valore di t_1' su C è proprio c e cosí t_1' (che appartiene a r) ha valori a, b e c, e cioè coincide con t, che quindi appartiene a r, come volevamo dimostrare.

È opportuno notare come la condizione enunciata sia sufficiente ma non necessaria per la decomposizione senza perdita: esistono infatti istanze di relazione che non soddisfano nessuna delle due dipendenze, ma al tempo stesso si decompongono senza perdita. Per esempio, la relazione in Figura 9.5 (ottenuta come join delle proiezioni) si decompone senza perdita sui due insiemi Impiegato, Sede e Progetto, Sede. Peraltro, la condizione in questione garantisce che *tutte* le istanze di relazione che soddisfano un dato insieme di dipendenze si decompongano senza perdita, e questo è in effetti un risultato utilizzabile in pratica: ogniqualvolta decomponiamo una relazione in due parti, se l'insieme degli attributi comuni è chiave per una delle due relazioni, allora possiamo essere certi che tutte le istanze della relazione si decompongono senza perdita.

9.4.2 Conservazione delle dipendenze

Per introdurre la seconda proprietà possiamo esaminare di nuovo la relazione in Figura 9.3. Volendo ancora rimuovere le anomalie, potremmo pensare di sfruttare solo la dipendenza Impiegato → Sede per ottenere una decomposizione senza perdita (potremmo procedere anche utilizzando solo l'altra dipendenza, Progetto → Sede). Otteniamo in questa maniera due relazioni: una sugli attributi Impiegato e Sede e l'altra sugli attributi Impiegato e Progetto. L'istanza in Figura 9.3 verrebbe così decomposta nelle relazioni in Figura 9.6.

Il join delle due relazioni in Figura 9.6 produce effettivamente la relazione in Figura 9.3, per cui possiamo dire che la relazione in Figura 9.3 si decompone senza perdita su Impiegato, Sede e Impiegato, Progetto. In effetti, Impiegato è chiave per la prima relazione, per cui la decomposizione senza perdita è garantita. La decomposizione in Figura 9.6 presenta però un altro inconveniente, che possiamo rilevare nel modo seguente. Supponiamo di voler inserire una nuova tupla che specifica la partecipazione dell'impiegato Neri, che opera a Milano, al progetto Marte. Sulla relazione originaria, cioè quella in Figura 9.3, un tale aggiornamento verrebbe immediatamente individuato come illecito, perché porterebbe a una violazione della dipendenza Progetto → Sede. Sulle relazioni decomposte, invece, non è possibile rilevare alcuna violazione di dipendenze: sulla relazione avente per attributi Impiegato e Progetto non è infatti possibile definire alcuna dipendenza funzionale e quindi non ci possono essere violazioni da rilevare, mentre nella relazione su Impiegato Sede la tupla con valori Neri e Milano soddisfa la dipendenza funzionale Impiegato → Sede. Possiamo quindi notare come non sia possibile effettuare alcuna verifica sulla dipendenza Progetto → Sede, perché i due attributi Progetto e Sede sono stati separati: uno in una relazione e l'altro nell'altra.

Generalizzando, possiamo quindi concludere che, in ogni decomposizione, ciascuna delle dipendenze funzionali dello schema originario dovrebbe coinvolgere attributi che compaiono tutti insieme in uno degli schemi decomposti. In questo modo, è possibile garantire, sullo schema decomposto, il soddisfacimento degli stessi vincoli il cui soddisfacimento è garantito dallo schema originario. Diremo che una decomposizione che soddisfa tale proprietà *conserva le dipendenze* dello schema originario.

Impiegato	Sede
Rossi	Roma
Verdi	Milano
Neri	Milano

Impiegato	Progetto
Rossi	Marte
Verdi	Giove
Verdi	Venere
Neri	Saturno
Neri	Venere

Figura 9.6 Un'altra decomposizione per la relazione in Figura 9.3.

9.4.3 Qualità delle decomposizioni

Per riassumere le considerazioni svolte possiamo affermare che le decomposizioni dovrebbero sempre soddisfare le proprietà di *decomposizione senza perdita* e *conservazione delle dipendenze*.

- La decomposizione senza perdita garantisce che le informazioni nella relazione originaria siano ricostruibili con precisione (cioè senza informazioni spurie) a partire da quelle rappresentate nelle relazioni decomposte. In tal caso, interrogando le relazioni decomposte, otteniamo gli stessi risultati che otterremmo interrogando la relazione originaria.
- La conservazione delle dipendenze garantisce che le relazioni decomposte hanno la stessa capacità della relazione originaria di rappresentare i vincoli di integrità (e cioè le proprietà del frammento di mondo reale di interesse) e quindi di rilevare aggiornamenti illeciti: a ogni aggiornamento lecito (rispettivamente, illecito) sulla relazione originaria corrisponde un aggiornamento lecito (rispettivamente, illecito) sulle relazioni decomposte. Ovviamente, sono possibili sulle relazioni decomposte ulteriori aggiornamenti, legati ai singoli concetti rappresentati in ciascuna di esse, che non hanno un corrispettivo sulla relazione originaria, senza però corrispondere a violazioni dei vincoli: si tratta degli aggiornamenti impossibili sulle relazioni non normalizzate a causa delle anomalie.

Di conseguenza, nel seguito considereremo accettabili, cioè di qualità sufficiente, solo le decomposizioni che soddisfano queste due proprietà. Dato uno schema che violi una forma normale, l'attività di normalizzazione è quindi volta a ottenere una decomposizione che sia senza perdita, che conservi le dipendenze e che contenga relazioni in forma normale. Possiamo notare come la decomposizione discussa nel Paragrafo 9.3.2, con lo scopo di sostituire tre relazioni normalizzate a una non normalizzata, presenti tutte e tre le qualità.

9.5 Terza forma normale

9.5.1 Limitazioni della forma normale di Boyce e Codd

Nella maggior parte dei casi si può raggiungere l'obiettivo di una buona decomposizione in forma normale di Boyce e Codd. Talvolta, però, questo non è possibile, come possiamo vedere discutendo un esempio. Consideriamo la relazione in Figura 9.7. Su di essa, possiamo supporre che siano definite le seguenti dipendenze:

- Dirigente → Sede: ogni dirigente opera presso una sede;
- Progetto Sede → Dirigente: ogni progetto ha più dirigenti che ne sono responsabili, ma in sedi diverse, e ogni dirigente può essere responsabile di più progetti; però, per ogni sede, un progetto ha un solo responsabile.

La relazione non è in forma normale di Boyce e Codd perché il primo membro della dipendenza Dirigente → Sede non è superchiave. Al tempo stesso, possiamo notare come non sia possibile alcuna buona decomposizione di questa relazione: infatti, la dipendenza Progetto Sede → Dirigente coinvolge tutti gli attributi e quindi

Dirigente	Progetto	Sede
Rossi	Marte	Roma
Verdi	Giove	Milano
Verdi	Marte	Milano
Neri	Saturno	Milano
Neri	Venere	Milano

Figura 9.7 Relazione per la discussione di una decomposizione problematica.

nessuna decomposizione è in grado di conservarla. L'esempio ci mostra quindi che esistono schemi che violano la forma normale di Boyce e Codd per i quali non esiste alcuna decomposizione che conservi le dipendenze. Possiamo quindi affermare che, talvolta, "la forma normale di Boyce e Codd non è raggiungibile."

9.5.2 Definizione di terza forma normale

Per trattare casi come quello dell'esempio appena visto, si ricorre a una forma normale meno restrittiva di quella di Boyce e Codd, che sostanzialmente consente situazioni come quella descritta, ma non ammette ulteriori fonti di ridondanza e anomalia. Diciamo che una relazione r è in *terza forma normale* se, per ogni dipendenza funzionale (non banale) $X \rightarrow A$ definita su di essa, almeno una delle seguenti condizioni è verificata:

- X contiene una chiave K di r;
- A appartiene ad almeno una chiave di r.

Ritornando al nostro esempio possiamo facilmente verificare che, sebbene lo schema non soddisfi la forma normale di Boyce e Codd, esso soddisfa la terza forma normale. Infatti, la dipendenza Progetto Sede → Dirigente ha come primo membro una chiave della relazione, mentre Dirigente → Sede, pur non contenendo una chiave al primo membro, ha un unico attributo a secondo membro che fa parte della chiave Progetto Sede. Si osservi che la relazione presenta in effetti una forma di ridondanza: ogni volta che un dirigente compare in una tupla, viene ripetuta per esso la sede in cui opera. Questa ridondanza viene però "tollerata" dalla terza forma normale perché non sarebbe possibile una decomposizione che elimini tale ridondanza e al tempo stesso conservi tutte le dipendenze.

In sostanza, abbiamo che la terza forma normale è meno forte della forma normale di Boyce e Codd e quindi non offre le medesime garanzie di qualità per una relazione; ha però rispetto a essa il vantaggio di essere sempre ottenibile. È possibile infatti dimostrare che una qualunque relazione che non soddisfa la terza forma normale è certamente decomponibile senza perdita e con conservazione delle dipendenze in relazioni in terza forma normale. Svilupperemo questo ragionamento nel Paragrafo 9.6.

Impiegato	Progetto	Stipendio
Rossi	Marte	30 000
Verdi	Giove	30 000
Verdi	Venere	30 000
Neri	Saturno	40 000
Neri	Venere	40 000

Figura 9.8 Relazione per la discussione sulla decomposizione in terza forma normale.

9.5.3 Decomposizione in terza forma normale

Mostreremo nel Paragrafo 9.6 un procedimento algoritmico che permette di ottenere sempre una buona decomposizione in terza forma normale. Osserviamo qui che, intuitivamente, si può procedere come suggerito nel caso della forma normale di Boyce e Codd: una relazione che non soddisfa la terza forma normale si decompone in relazioni ottenute per proiezione sugli attributi corrispondenti alle dipendenze funzionali, con l'unica accortezza di mantenere sempre una relazione che contiene una chiave della relazione originaria. Questo può essere visto con riferimento alla relazione in Figura 9.8, per la quale vale la sola dipendenza funzionale Impiegato → Stipendio.

Una decomposizione in una relazione sugli attributi Impiegato Stipendio e in un'altra sul solo attributo Progetto violerebbe la proprietà di decomposizione senza perdita, proprio perché nessuna delle due relazioni contiene una chiave. Per garantire tale proprietà dobbiamo invece definire la seconda relazione sugli attributi Impiegato Progetto, che formano una chiave della relazione originaria. La conseguente decomposizione è mostrata nella Figura 9.9; notiamo che sulla prima relazione non sono definite dipendenze e che la sua chiave è costituita da entrambi gli attributi. Ribadiamo comunque il fatto che il successo di una decomposizione dipende in buona misura dalle dipendenze che abbiamo individuato.

Per concludere torniamo all'esempio di Figura 9.1 e osserviamo che la relazione non soddisfa neanche la terza forma normale. Procedendo come abbiamo suggerito otteniamo ancora la decomposizione di Figura 9.2 che, incidentalmente, è anche in

Impiegato	Progetto
Rossi	Marte
Verdi	Giove
Verdi	Venere
Neri	Saturno
Neri	Venere

Impiegato	Stipendio
Rossi	30 000
Verdi	30 000
Neri	40 000

Figura 9.9 Una decomposizione in terza forma normale.

forma normale di Boyce e Codd. Questo è in effetti un risultato di validità generale: una decomposizione tesa a ottenere la terza forma normale produce nella maggior parte dei casi schemi in forma normale di Boyce e Codd. In particolare, si può dimostrare che se una relazione ha solo una chiave (come in questo caso) allora le due forme normali coincidono, cioè una relazione in terza forma normale è anche in forma normale di Boyce e Codd.

9.5.4 Altre forme normali

L'aggettivo "terza" nel nome della forma normale suggerisce l'esistenza di altre forme normali che citiamo brevemente. La prima forma normale stabilisce semplicemente una condizione che sta alla base del modello relazionale stesso: gli attributi delle relazioni sono definiti su valori atomici e non su valori complessi quali insiemi o relazioni. Vedremo, nel secondo volume [6], come questo vincolo venga in effetti rilassato in altri modelli per basi di dati.

La seconda forma normale è una variante debole della terza e la introduciamo per mezzo di un esempio, la relazione in Figura 9.10. Essa soddisfa le dipendenze Impiegato → Categoria e Categoria → Stipendio e quindi viola la terza forma normale, perché Categoria non è chiave. La seconda forma normale tollera la dipendenza tra Categoria e Stipendio, perché Stipendio dipende comunque (sia pure attraverso Categoria) dall'intera chiave Impiegato.

Al fine di citare terminologie usate, soprattutto in passato e in particolare nelle proposte originarie, riportiamo le definizioni originarie (anche se un po' informali) di seconda e terza forma normale.

- Una relazione è in seconda forma normale se su di essa non sono definite *dipendenze parziali*, cioè dipendenze fra un sottoinsieme proprio della chiave e altri attributi. Nell'esempio di Figura 9.1 abbiamo due dipendenze parziali Impiegato → Stipendio e Progetto → Budget, perché la chiave è costituita dai due attributi Impiegato e Progetto; tale relazione, quindi, viola anche la seconda forma normale. Invece, la relazione in Figura 9.10 soddisfa la seconda forma normale, perché non vi sono dipendenze parziali, in quanto sia Categoria sia Stipendio dipendono dall'intera chiave Impiegato; in effetti, dalla definizione discende che le relazioni che hanno la chiave composta da un solo attributo sono in seconda forma normale.

Impiegato	Categoria	Stipendio
Neri	3	30 000
Verdi	3	30 000
Rossi	4	50 000
Mori	4	50 000
Bianchi	5	72 000

Figura 9.10 Una relazione con varie dipendenze funzionali.

- Una relazione è in terza forma normale se su di essa non sono definite *dipendenze transitive*, cioè dipendenze della forma $K \rightarrow A$, dove K è la chiave ed esiste un altro insieme di attributi X, non chiave, con le dipendenze $K \rightarrow X$ e $X \rightarrow A$. Nella relazione in Figura 9.10 abbiamo una dipendenza transitiva fra Impiegato e Stipendio per via delle dipendenze Impiegato → Categoria e Categoria → Stipendio.

È importante notare che tanto le dipendenze parziali quanto quelle transitive violano la terza forma normale così come noi l'abbiamo definita perché coinvolgono una dipendenza funzionale il cui primo membro non è superchiave e il cui secondo membro non fa parte della chiave.

Segnaliamo che esistono anche altre forme normali che fanno riferimento peraltro a vincoli di integrità diversi dalle dipendenze funzionali. Tutte queste forme normali vengono poco usate nelle applicazioni odierne in quanto è stato rilevato che la terza forma normale e la forma normale di Boyce e Codd forniscono il giusto compromesso tra semplicità e qualità dei risultati.

9.5.5 Normalizzazione e scelta degli attributi

Con riferimento alla relazione in Figura 9.7, svolgiamo un'ultima considerazione sulle forme normali. Esaminando meglio le specifiche, possiamo arrivare alla conclusione che avremmo potuto descrivere l'applicazione di interesse in maniera più appropriata introducendo un ulteriore attributo Reparto, che partiziona (sulla base dei responsabili) le singole sedi (si veda la relazione in Figura 9.11). Le dipendenze possono, in questo caso, essere così definite:

- Dirigente → Sede Reparto: ogni dirigente opera presso una sede e dirige un reparto;
- Sede Reparto → Dirigente: per ogni sede e reparto c'è un solo dirigente;
- Progetto Sede → Reparto: per ogni sede, un progetto è assegnato a un solo reparto (e, di conseguenza, ha un solo responsabile); la dipendenza funzionale Progetto Sede → Dirigente è quindi ricostruibile (cioè è soddisfatta se lo sono le altre due; formalizzeremo questo concetto con la nozione di implicazione nel prossimo paragrafo).

Dirigente	Progetto	Sede	Reparto
Rossi	Marte	Roma	1
Verdi	Giove	Milano	1
Verdi	Marte	Milano	1
Neri	Saturno	Milano	2
Neri	Venere	Milano	2

Figura 9.11 Modifica della relazione in Figura 9.7.

Dirigente	Sede	Reparto
Rossi	Roma	1
Verdi	Milano	1
Neri	Milano	2

Progetto	Sede	Reparto
Marte	Roma	1
Giove	Milano	1
Marte	Milano	1
Saturno	Milano	2
Venere	Milano	2

Figura 9.12 Una buona decomposizione della relazione in Figura 9.11.

Per questo schema, esiste una buona decomposizione, come mostrato dall'istanza in Figura 9.12. Infatti:

- la decomposizione è senza perdita, perché gli attributi comuni Sede e Reparto formano una chiave per la prima relazione;
- le dipendenze sono conservate, perché per ciascuna dipendenza esiste una relazione decomposta che ne contiene tutti gli attributi;
- entrambe le relazioni sono in forma normale di Boyce e Codd, perché tutte le dipendenze hanno il primo membro costituito da una chiave.

Possiamo quindi concludere affermando che, spesso, come mostrato dall'ultimo esempio, la non raggiungibiltà della forma normale di Boyce e Codd può essere dovuta a un'analisi non sufficientemente accurata dell'applicazione.

9.6 Teoria delle dipendenze e normalizzazione

In questo paragrafo mostriamo, sia pure in modo schematico, come i più importanti concetti discussi nei paragrafi precedenti possano essere formalizzati, arrivando a un processo di normalizzazione realizzabile in modo algoritmico. Ci poniamo cioè il seguente problema: data una relazione e un insieme di dipendenze funzionali definite su di essa, generare una decomposizione della relazione che contenga solo relazioni in forma normale e soddisfi le qualità di decomposizione senza perdita e conservazione delle dipendenze. Poiché, come abbiamo visto, questo obiettivo non è raggiungibile per la forma normale di Boyce e Codd, lo perseguiremo per la terza forma normale. In linea di massima, il procedimento è quello già illustrato informalmente: definire una relazione per ciascun gruppo di dipendenze fra loro strettamente correlate. Il procedimento va formalizzato per definire bene l'insieme di dipendenze di interesse e completato con una verifica finale, che può portare a un passo aggiuntivo.

9.6.1 Implicazione di dipendenze funzionali

Come abbiamo visto per mezzo di alcuni esempi nel Paragrafo 9.3.2, la descrizione delle proprietà di una relazione può essere specificata indifferentemente per mezzo di diversi insiemi di dipendenze funzionali. Precisiamo questa osservazione per mezzo

del concetto di implicazione di vincoli, discusso in questo paragrafo, e con quello di equivalenza di insiemi di vincoli, discusso nel prossimo. Per semplicità di trattazione, consideriamo qui ammissibili, a differenza di quanto fatto nei paragrafi precedenti, anche le dipendenze banali.

Diciamo che un insieme di dipendenze funzionali F *implica* un'altra dipendenza f se ogni relazione che soddisfa tutte le dipendenze in F soddisfa anche f. Con riferimento allo schema della relazione in Figura 9.10, possiamo osservare che le dipendenze Impiegato → Categoria e Categoria → Stipendio implicano la dipendenza Impiegato → Stipendio. Ogni relazione che soddisfa le prime due soddisfa anche la terza, come si può verificare seguendo la definizione: se due tuple hanno lo stesso valore su Impiegato, facciamo vedere che hanno stesso valore su Stipendio; infatti, se hanno lo stesso valore su Impiegato, allora, per la prima dipendenza, esse hanno lo stesso valore su Categoria e quindi, per la seconda dipendenza, anche su Stipendio.

Il primo problema che formalizziamo e studiamo è quello dell'*implicazione* di dipendenze funzionali: dati F e f, come verifichiamo se F implica f?

Per procedere, definiamo un concetto che risulterà molto utile. Siano dati uno schema di relazione $R(U)$ e un insieme di dipendenze funzionali F definite sugli attributi in U. Sia X un insieme di attributi contenuti in U (cioè $X \subseteq U$); la *chiusura* di X rispetto a F, indicata con X_F^+, è l'insieme degli attributi che dipendono funzionalmente da X (esplicitamente o implicitamente):

$$X_F^+ = \{A \mid A \in U \text{ e } F \text{ implica } X \rightarrow A\}$$

L'insieme X_F^+ può risultare molto utile: se vogliamo vedere se $X \rightarrow A$ è implicata da F, basta vedere se A appartiene a X_F^+, a patto di saper calcolare X_F^+. In effetti, questa è una strada valida, perché esiste un algoritmo semplice ed efficiente per il calcolo di X_F^+.

Input: un insieme X di attributi e un insieme F di dipendenze.
Output: un insieme X_P di attributi.
(1) Inizializziamo X_P con l'insieme di input X.
(2) Esaminiamo le dipendenze in F; se esiste una dipendenza $Y \rightarrow A$ con $Y \subseteq X_P$ e $A \notin X_P$ allora aggiungiamo A a X_P.
(3) Ripetiamo il passo (2) fino al momento in cui non vi sono ulteriori attributi che possono essere aggiunti a X_P.

Dimostriamo, sia pur schematicamente, che l'algoritmo termina sempre e che calcola effettivamente la chiusura, cioè che il valore finale di X_P è proprio uguale a X_F^+. Procediamo in tre passi.

- L'algoritmo termina: il passo principale viene ripetuto solo se ci sono attributi da aggiungere a X_P; poiché il numero di attributi di una relazione è finito, prima o poi si raggiunge un punto in cui non vi sono attributi da aggiungere.
- $X_P \subseteq X_F^+$. Siano $X_0, X_1, \ldots, X_h$ i valori assunti da X_P durante l'esecuzione dell'algoritmo, con $X_0 = X$ e X_h pari al valore finale.

La dimostrazione procede per induzione, mostrando che $X_i \subseteq X_F^+$, per ogni i.
Il passo base $X_0 = X \subseteq X_F^+$ è immediato, perché la dipendenza $X \rightarrow A$, per ogni $A \in X$, è banale e quindi sempre soddisfatta e quindi sempre implicata. Il passo induttivo richiede di mostrare che, se $X_i \subseteq X_F^+$, allora $X_{i+1} \subseteq X_F^+$, cioè che se F implica $X \rightarrow A$, per ogni $A \in X_i$ allora F implica $X \rightarrow A$, per ogni $A \in X_{i+1}$. Supponiamo che r soddisfi F e siano t_1 e t_2 due tuple uguali su X; per l'ipotesi induttiva, r soddisfa $X \rightarrow A$, per ogni $A \in X_i$ e quindi t_1 e t_2 sono uguali su X_i; se l'algoritmo viene applicato, allora F contiene una dipendenza $Y \rightarrow A$ con $Y \subseteq X_i$ e $X_{i+1} = X_i A$; ma se r soddisfa F, allora soddisfa $Y \rightarrow A$ e quindi, poiché t_1 e t_2 sono uguali su X_i (e quindi su Y, dato che $Y \subseteq X_i$) allora esse sono uguali su A e quindi anche su $X_{i+1} = X_i A$.

- $X_F^+ \subseteq X_P$. Dobbiamo mostrare che se $A \in X_F^+$ allora $A \in X_P$, cioè (per definizione di X_F^+) che se F implica $X \rightarrow A$, allora $A \in X_P$. Procediamo mostrando che se $A \notin X_P$ allora F non implica $X \rightarrow A$. Allo scopo, mostriamo un controesempio, costituito da una relazione su due tuple, uguali fra loro su tutti gli attributi in X_P e diverse sugli altri, come per esempio la seguente:

r

X_P				$U - X_P$			
0	0	...	0	0	0	...	0
0	0	...	0	1	1	...	1

Questa relazione viola la dipendenza $X \rightarrow A$, perché $X \subseteq X_P$ e $A \notin X_P$ e quindi le due tuple sono uguali su X e diverse su A. Al tempo stesso, la relazione soddisfa tutte le dipendenze in F: consideriamo infatti una generica dipendenza $Z \rightarrow B$ appartenente a F e distinguiamo due casi, $Z \subseteq X_P$ e $Z \nsubseteq X_P$. Nel secondo caso, le due tuple sono diverse su Z e quindi la dipendenza è certamente soddisfatta. Nel primo caso, le due tuple sono uguali su Z; ma, poiché $Z \subseteq X_P$, allora l'algoritmo può considerare la dipendenza $Z \rightarrow B$ e, prima o poi, aggiungere anche B a X_P; di conseguenza, le due tuple sono uguali anche su B.

Prima di mostrare un esempio, illustriamo un'importante applicazione di quanto discusso. Il concetto di chiusura X_F^+ è utile anche per formalizzare il legame fra il concetto di dipendenza funzionale e quello di chiave: un insieme di attributi K è chiave per uno schema di relazione $R(U)$ su cui è definito un insieme di dipendenze funzionali F se F implica $K \rightarrow U$. Di conseguenza, l'algoritmo appena mostrato può essere utilizzato per verificare se un insieme è chiave.

Per vedere un'applicazione dell'algoritmo, consideriamo la relazione in Figura 9.1, indicando i suoi attributi con le rispettive iniziali. Possiamo verificare che gli attributi IP formano una chiave, in quanto $IP^+ = ISPBF$ (e quindi IP è superchiave, perché la relazione è definita su $ISPBF$) e $I^+ = IS$ e $P^+ = PB$ (e quindi nessun sottoinsieme proprio di IP è superchiave). Il calcolo di IP^+ si esegue inizializzando l'insieme di lavoro a IP e aggiungendo (in effetti in qualunque ordine), S utilizzando $I \rightarrow S$, B utilizzando $P \rightarrow B$ e F utilizzando $IP \rightarrow F$. Invece I^+ si calcola partendo da I e potendo aggiungere solo S (grazie a $I \rightarrow S$); le altre due dipendenze non sono utilizzabili, perché P non appartiene all'insieme di lavoro.

9.6.2 Coperture di insiemi di dipendenze funzionali

Come già detto, può essere utile sostituire a un insieme di dipendenze funzionali un altro che specifichi, nella sostanza, le stesse proprietà e che sia più semplice da gestire.

Due insiemi di dipendenze funzionali F_1 e F_2 sono *equivalenti* se F_1 implica ciascuna dipendenza in F_2 e viceversa. Se due insiemi sono equivalenti diciamo anche che ognuno è una *copertura* dell'altro. Si può facilmente dimostrare che, dati due insiemi F_1 e F_2 equivalenti, una relazione soddisfa F_1 se e solo se essa soddisfa F_2. Questa proprietà giustifica quindi l'uso del termine "equivalenza" e la possibilità di utilizzare, dato un insieme di dipendenze, un altro, a esso equivalente, ma più semplice, per esempio con meno dipendenze o meno attributi. In effetti, si possono introdurre diversi criteri di "semplicità" per un insieme di dipendenze funzionali. Nella letteratura sono state introdotte numerose definizioni, fra cui le seguenti, che utilizzeremo: un insieme F è:

- *non ridondante* se non esiste dipendenza $f \in F$ tale che $F - \{f\}$ implica f;
- *ridotto* se è non ridondante e non esiste un insieme F' equivalente a F ottenuto eliminando attributi dai primi membri di una o più dipendenze di F.

Consideriamo alcuni insiemi di dipendenze:

$$F_1 = \{A \rightarrow B, AB \rightarrow C, A \rightarrow C\}$$
$$F_2 = \{A \rightarrow B, AB \rightarrow C\}$$
$$F_3 = \{A \rightarrow B, A \rightarrow C\}$$

Possiamo osservare che:

- F_1 è ridondante, perché $\{A \rightarrow B, AB \rightarrow C\}$ implica $A \rightarrow C$; F_1 è equivalente a F_2;
- F_2 è non ridondante ma non è ridotto, perché B può essere eliminato dal primo membro della seconda dipendenza: F_2 è equivalente a F_3;
- F_3 è ridotto.

Avendo visto nel paragrafo precedente come si verifica l'implicazione, possiamo dire che il calcolo di una copertura non ridondante e di una ridotta, dato un insieme di dipendenze, è abbastanza semplice, in quanto entrambe le definizioni si basano sull'implicazione. Per trovare una copertura non ridondante è sufficiente esaminare ripetutamente le dipendenze dell'insieme dato, eliminando quelle implicate da altre, fermandosi quando non ve ne sono più; l'insieme rimasto è una copertura non ridondante di quello iniziale. Per trovare una copertura ridotta, per un qualunque insieme di dipendenze funzionali, possiamo procedere in tre passi: (a) sostituiamo l'insieme dato con quello equivalente che ha tutti i secondi membri costituiti da singoli attributi; (b) eliminiamo le dipendenze ridondanti; (c) per ogni dipendenza verifichiamo se esistono attributi eliminabili dal primo membro: in pratica, se F è l'insieme corrente, per ogni dipendenza $Y \rightarrow A \in F$, verifichiamo se esiste $Y \subseteq X$ tale che F è equivalente a $F - \{X \rightarrow A\} \cup \{Y \rightarrow A\}$.

Per illustrare questo procedimento, consideriamo un esempio un po' più articolato, con una relazione relativa agli impiegati di un'azienda, sugli attributi Matricola, Cognome, Grado, Retribuzione, Dipartimento, Supervisore, Progetto, Anzianità e con le dipendenze (in cui, di nuovo, abbreviamo gli attributi con le iniziali): $M \rightarrow RSDG$, $MS \rightarrow CD$, $G \rightarrow R$, $D \rightarrow S$, $S \rightarrow D$, $MPD \rightarrow AM$. Dopo avere applicato il passo (a) relativo alla decomposizione dei secondi membri, nel passo (b) possiamo eliminare le dipendenze $M \rightarrow R$, $M \rightarrow S$, $MS \rightarrow D$, $MPD \rightarrow M$. Al passo (c) è poi possibile eliminare S dal primo membro della dipendenza $MS \rightarrow C$, e D dal primo membro di $MPD \rightarrow A$. La copertura ridotta ottenuta alla fine contiene quindi le dipendenze: $M \rightarrow D$, $M \rightarrow G$, $M \rightarrow C$, $G \rightarrow R$, $D \rightarrow S$, $S \rightarrow D$, $MP \rightarrow A$.

9.6.3 Sintesi di schemi in terza forma normale

Sulla base della nozione di copertura ridotta introdotta nel paragrafo precedente, possiamo a questo punto mostrare come si possa ottenere, in modo algoritmico, una decomposizione in terza forma normale (che soddisfi le proprietà di conservazione delle dipendenze e decomposizione senza perdita) per un qualunque schema.

Ricordiamo la definizione di terza forma normale, formalizzandola con i concetti introdotti in questo paragrafo: uno schema di relazione $R(U)$ con l'insieme di dipendenze F è in *terza forma normale* se, per ogni dipendenza funzionale (non banale) $X \rightarrow A \in F$, almeno una delle seguenti condizioni è verificata:

- X contiene una chiave K di r: cioè $X_F^+ = U$;
- A è contenuto in almeno una chiave di r: esiste un insieme di attributi $K \subseteq U$ tale che $K_F^+ = U$ e $(K - A)_F^+ \subset U$.

L'algoritmo per la decomposizione procede come segue, dati $R(U)$ e F:

1. viene calcolata una copertura ridotta G di F;
2. G viene partizionato in sottoinsiemi $G_1, \ldots, G_k$ tali che a ogni insieme appartengono dipendenze che hanno primi membri con la stessa chiusura (cioè, $X \rightarrow A$ e $Y \rightarrow B$ appartengono alla stessa partizione se e solo se $X_G^+ = Y_G^+$);
3. viene costruito un insieme $\mathcal{U}$ di sottoinsiemi di U, uno per ciascuna partizione di dipendenze, con tutti gli attributi coinvolti nella partizione;
4. se un elemento di $\mathcal{U}$ è propriamente contenuto in un altro, allora esso viene eliminato da $\mathcal{U}$;
5. viene costruito uno schema di basi dati con uno schema di relazione $R_i(U_i)$ per ciascun elemento; $U_i \in \mathcal{U}$ con associate le dipendenze in G i cui attributi sono tutti contenuti in U_i;
6. se nessuno degli U_i costituisce una chiave per la relazione originaria $R(U)$, allora viene calcolata una chiave K di $R(U)$ e viene aggiunto allo schema generato al passo precedente uno schema di relazione sugli attributi K, senza dipendenze.

Questo algoritmo va spesso sotto il nome di algoritmo di *sintesi* di schemi in terza forma normale, perché costruisce lo schema finale a partire dalle dipendenze. Non avendo lo spazio per una dimostrazione precisa della correttezza dell'algoritmo, giustifichiamo comunque il raggiungimento dei suoi obiettivi:

- il fatto che ciascuna delle relazioni sia in terza forma normale deriva dal fatto che in ciascuno schema di relazione compaiono dipendenze che hanno primi membri fra loro equivalenti e quindi ciascuno dei primi membri è chiave (questa argomentazione è in effetti sufficiente solo se il passo 4 dell'algoritmo non elimina alcun insieme; altrimenti, la proprietà è pure valida, ma con argomentazioni più laboriose, che omettiamo);
- la conservazione delle dipendenze deriva dal fatto che viene calcolata una copertura ridotta e che ciascuna dipendenza contribuisce a generare una relazione e quindi non può essere trascurata;
- la decomposizione senza perdita è garantita dall'ultimo passo: generalizzando la proprietà illustrata nel Paragrafo 9.4.1, potremmo vedere che, anche in una decomposizione n-aria, se una delle relazioni di una decomposizione contiene una chiave per la relazione originaria, allora la decomposizione risulta essere senza perdita.

Consideriamo due esempi di applicazione dell'algoritmo. Sulla relazione $R(MCGR\ DSPA)$ con le dipendenze mostrate nell'esempio alla fine del Paragrafo 9.6.2, esso produce, al passo 1, la copertura ridotta mostrata in precedenza, al passo 2 partiziona la copertura negli insiemi:

$$G_1 = \{M \rightarrow D,\ M \rightarrow G,\ M \rightarrow C\}$$
$$G_2 = \{G \rightarrow R\}$$
$$G_3 = \{D \rightarrow S,\ S \rightarrow D\}$$
$$G_4 = \{MP \rightarrow A\}$$

Poi, i passi 3, 4 e 5 costruiscono uno schema di relazione per ciascuna partizione (senza bisogno in questo caso di eliminazioni), con le dipendenze corrispondenti. Il passo 6 non ha effetti, perché MP è chiave per la relazione originaria. Quindi, viene generato lo schema con le relazioni:

- $R_1(MDGC)$, con le dipendenze $M \rightarrow D,\ M \rightarrow G,\ M \rightarrow C$
- $R_2GR)$ con $\{G \rightarrow R\}$
- $R_3(DS)$ con $\{D \rightarrow S,\ S \rightarrow D\}$
- $R_4(MPA)$ con $\{MP \rightarrow A\}$

Se consideriamo invece l'esempio mostrato nei paragrafi iniziali di questo capitolo, la relazione $R(ISPBF)$, con le dipendenze $I \rightarrow C, P \rightarrow B, IP \rightarrow F$, otteniamo proprio lo schema mostrato nel Paragrafo 9.3.2, sempre senza utilizzare il passo 6. Se su tale schema fossero invece definite solo le dipendenze $I \rightarrow C, P \rightarrow B$, allora il passo 6 rileverebbe il fatto che nessuno degli schemi contiene una chiave per lo schema originario e allora aggiungerebbe uno schema di relazione $R_3(IPF)$, definito sulla chiave IPF dello schema originario.

9.7 Progettazione di basi di dati e normalizzazione

La teoria della normalizzazione, anche studiata in modo semplificato, può essere utilizzata come base per operazioni di verifica di qualità di schemi, sia nella fa-

se di progettazione concettuale sia in quella di progettazione logica. Vediamo prima brevemente l'utilizzo nella progettazione logica, per poi considerare con maggiore dettaglio l'adattamento dei concetti al modello Entità-Relazione e quindi alla progettazione concettuale.

Le attività di progettazione sono sempre soggette a errori e incompletezze, e quindi una revisione delle relazioni ottenute in sede di progettazione logica può portare all'individuazione di imprecisioni nella formulazione dello schema concettuale: la verifica è spesso relativamente semplice, poiché l'individuazione delle dipendenze funzionali e delle chiavi deve essere svolta nell'ambito di una singola relazione, che corrisponde a un'entità o a un'associazione già analizzata nella progettazione concettuale. In contesti così circoscritti, la struttura delle dipendenze risulta in genere relativamente semplice ed è quindi possibile individuare direttamente la decomposizione necessaria per ottenere la terza forma normale (o la forma normale di Boyce e Codd). Per esempio, la relazione in Figura 9.10 sugli attributi Impiegato, Categoria e Stipendio può essere ottenuta solo se in fase di progettazione concettuale non ci rendiamo conto che il legame tra categoria e stipendio è indipendente dal legame tra impiegato e categoria. Individuando, in sede di verifica di normalizzazione, la sussistenza della dipendenza funzionale, è possibile rimediare all'errore.

9.7.1 Verifiche di normalizzazione su entità

Le idee alla base della normalizzazione possono essere utilizzate anche durante la fase di progettazione concettuale, con riferimento quindi ai costrutti del modello Entità-Relazione, per una verifica della qualità di ciascun elemento dello schema concettuale. In effetti, è possibile considerare ciascuna entità e ciascuna associazione come una relazione. In particolare, la relazione che corrisponde a un'entità ha attributi che corrispondono esattamente agli attributi dell'entità (per le entità con identificatore esterno sono necessari ulteriori attributi in corrispondenza alle entità che partecipano all'identificazione). La verifica di normalizzazione può quindi procedere come visto finora. In pratica, è sufficiente considerare le dipendenze funzionali che sussistono fra gli attributi dell'entità e verificare che ciascuna di esse abbia come primo membro l'identificatore (o lo contenga). Per esempio, consideriamo (Figura 9.13) un'entità Prodotto, con attributi Codice, NomeProdotto, NomeFornitore, Indirizzo, PartitaIVA, dove NomeFornitore è il nome della ditta che fornisce il prodotto, per la quale sono di interesse anche l'indirizzo e il numero della partita IVA.

Nell'individuare le dipendenze relative a tale entità, possiamo notare che possono esistere fornitori diversi con lo stesso nome o stesso indirizzo, mentre tutte le proprietà di ogni fornitore sono identificate dalla partita IVA: sussiste cioè la dipendenza PartitaIVA → NomeFornitore Indirizzo. Inoltre, tutti gli attributi dipendono funzionalmente dall'attributo Codice, che costituisce quindi l'identificatore dell'entità: fissato un codice, sono univocamente determinati il nome del prodotto e il fornitore, con le sue proprietà. Poiché l'unico identificatore dell'entità è l'attributo Codice, possiamo concludere che l'entità viola la terza forma normale, in quanto la dipendenza PartitaIVA → NomeFornitore Indirizzo ha un primo membro che non contiene l'identificatore e un secondo membro composto da attributi che non fanno parte della chiave. In questi casi, la verifica di normalizzazione ci permette di segnalare il fat-

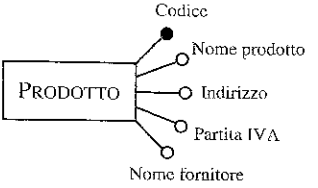

Figura 9.13 Un'entità da sottoporre a verifica di normalizzazione.

to che lo schema concettuale non è accurato e ci suggerisce di decomporre l'entità stessa.

La decomposizione può avvenire, come abbiamo visto in precedenza, con diretto riferimento alle dipendenze, oppure, più semplicemente, ragionando qualitativamente sui concetti rappresentati dall'entità insieme a quelli che derivano dalle dipendenze funzionali. Nel caso in esame, rilevando la dipendenza funzionale, abbiamo capito che il concetto di fornitore è in effetti indipendente da quello di prodotto e ha proprietà associate (partita IVA, nome e indirizzo): quindi, sulla base degli argomenti sviluppati riguardo alla progettazione concettuale, possiamo dire che è opportuno modellare il concetto di fornitore per mezzo di una entità, con identificatore costituto dall'attributo PartitaIVA e ulteriori attributi NomeFornitore e Indirizzo.

Poiché nello schema originario gli attributi di Prodotto e Fornitore compaiono in una stessa entità, è evidente che se viceversa li separiamo in due entità è opportuno che tali entità siano correlate, cioè che esista un'associazione che le collega. Si tratta chiaramente di un'associazione binaria (in questo siamo aiutati dal fatto che consideriamo l'entità in esame separatamente da tutto il resto dello schema), per le cardinalità della quale possiamo ragionare come segue. Poiché esiste una dipendenza funzionale fra Codice del prodotto e PartitaIVA del fornitore, siamo certi che ogni prodotto ha al più un fornitore, e quindi la partecipazione dell'entità PRODOTTO all'associazione deve avere cardinalità massima pari a 1. Poiché viceversa non sussiste alcuna dipendenza fra PartitaIVA e Codice, abbiamo una cardinalità massima non limitata (pari a N) per la partecipazione dell'entità Fornitore all'associazione. Per le cardinalità minime, possiamo ragionare sulla base delle proprietà dell'applicazione. Per esempio, se supponiamo che per ciascun prodotto il fornitore debba essere sempre noto, mentre possiamo avere fornitori che (al momento) non forniscono alcun prodotto, le cardinalità sono quelle in Figura 9.14, in cui è riportato lo schema finale.

Possiamo osservare come la decomposizione ottenuta soddisfi le due proprietà fondamentali: è una decomposizione senza perdita, perché sulla base dei fornitori (la cardinalità massima della partecipazione dell'entità Prodotto all'associazione è pari a 1) è possibile ricostruire i valori degli attributi dell'entità originaria, e conserva le dipendenze, perché ciascuna delle dipendenze è contenuta in una delle entità o è ricostruibile da esse (per esempio, la dipendenza fra i codici dei prodotti e i nomi dei fornitori è ricostruibile tramite l'associazione FORNITURA, che associa a un prodot-

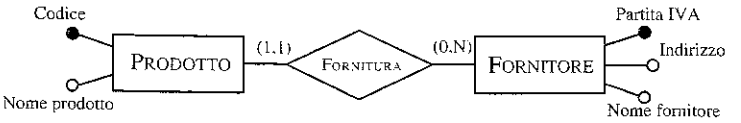

Figura 9.14 Il risultato della decomposizione di una entità.

to uno solo fornitore, e la dipendenza funzionale PartitaIVA → NomeFornitore è definita nell'entità Fornitore). Potremmo rendere questi ragionamenti ancora più precisi, facendo riferimento agli schemi di relazione ottenuti come traduzione, ma lasciamo lo sviluppo per esercizio.

9.7.2 Verifiche di normalizzazione su associazioni

Per quanto riguarda le associazioni, il ragionamento è per certi aspetti più semplice, perché l'insieme delle occorrenze di ciascuna associazione è una relazione, e quindi è possibile applicare direttamente i concetti connessi con le forme normali, ma per altri più complesso, perché i domini su cui tale relazione è definita sono gli insiemi delle occorrenze delle entità coinvolte. Di conseguenza, per verificare il soddisfacimento della terza forma normale, è necessario individuare le dipendenze funzionali che sussistono, nell'ambito dell'associazione in esame, fra le entità coinvolte. Poiché, come è facile vedere, ogni relazione binaria è in terza forma normale (e anche in forma normale di Boyce e Codd), la verifica di normalizzazione va effettuata solo sulle associazioni non binarie, cioè su quelle che coinvolgono almeno tre entità.

Consideriamo per esempio l'associazione TESI in Figura 9.15, che coinvolge le entità STUDENTE, PROFESSORE, CORSODILAUREA e DIPARTIMENTO, il cui scopo

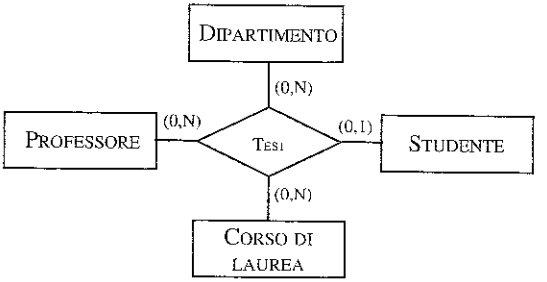

Figura 9.15 Un'associazione da sottoporre a verifica di normalizzazione.

è descrivere il fatto che gli studenti, iscritti ai corsi di laurea, svolgono tesi presso i dipartimenti sotto la guida di professori.

Esaminando in dettaglio l'associazione, possiamo arrivare alle seguenti conclusioni:

- ogni studente è iscritto a un solo corso di laurea;
- ogni studente svolge una tesi sotto la supervisione di un solo professore (che non è necessariamente legato al corso di laurea);
- ogni professore afferisce a un solo dipartimento e gli studenti sotto la sua supervisione svolgono la tesi presso tale dipartimento.

Supponendo che non sia rilevante ai fini della tesi di laurea l'afferenza del professore al corso di laurea cui lo studente è iscritto, possiamo dire che le proprietà dell'applicazione di interesse sono descritte in modo esauriente dalle seguenti tre dipendenze funzionali:

STUDENTE → CORSODILAUREA

STUDENTE → PROFESSORE

PROFESSORE → DIPARTIMENTO

La chiave (unica) della relazione risulta essere costituita da STUDENTE: dato uno studente sono univocamente individuati il corso di laurea, il professore e il dipartimento. Di conseguenza, la terza dipendenza funzionale, ovvero PROFESSORE → DIPARTIMENTO, causa una violazione della terza forma normale. In effetti, l'afferenza di un professore a un dipartimento è un concetto indipendente dall'esistenza di studenti che svolgono la tesi con il professore stesso. Ragionando come nei casi precedenti, possiamo concludere che l'associazione presenta aspetti indesiderabili e che va quindi decomposta, separando le dipendenze funzionali con primi membri diversi. In tal modo, possiamo ottenere lo schema in Figura 9.16, che contiene due associazioni, entrambe in terza forma normale (e anche in forma normale di Boyce e Codd). Anche qui, abbiamo decomposizione senza perdita e conservazione delle dipendenze.

9.7.3 Ulteriori decomposizioni di associazioni

Sullo schema in Figura 9.16 possiamo fare alcune considerazioni aggiuntive, che vanno al di là della teoria della normalizzazione in senso stretto, ma rientrano nell'ambito dell'analisi e verifica di schemi concettuali per mezzo di strumenti formali, nel caso specifico le dipendenze funzionali. L'associazione TESI è in terza forma normale, perché la sua chiave è costituita dall'entità STUDENTE e le uniche dipendenze sono quelle che hanno la chiave STUDENTE come primo membro, e cioè STUDENTE → PROFESSORE e STUDENTE → CORSODILAUREA. D'altra parte, le proprietà descritte dalle due dipendenze sono fra loro indipendenti: non tutti gli studenti stanno svolgendo una tesi e quindi non tutti hanno un relatore. Dal punto di vista della normalizzazione, questa situazione non presenta problemi, perché si assume che le relazioni possano contenere valori nulli, purché non nella chiave, e quindi è

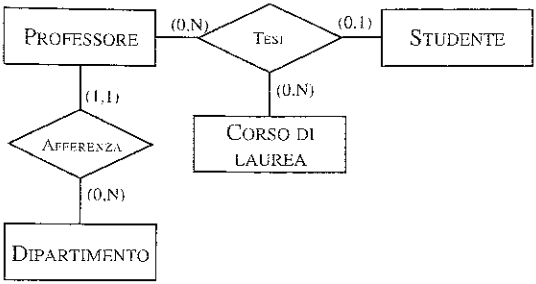

Figura 9.16 Il risultato della decomposizione di un'associazione.

ragionevole accettare dipendenze diverse con lo stesso primo membro. D'altra parte, a livello di modellazione concettuale (cioè di schema entità-relazione) è necessario distinguere i vari concetti (peraltro non esiste, e non avrebbe senso introdurre, un concetto di "valore nullo in un'associazione"). Attraverso le dipendenze possiamo quindi notare che sarebbe opportuno decomporre ulteriormente l'associazione, ottenendo due associazioni, una per ciascuno dei due concetti. La Figura 9.17 mostra lo schema decomposto. La decomposizione è anche in questo caso accettabile, perché conserva le dipendenze ed è senza perdita.

Generalizzando l'argomento appena sviluppato, possiamo arrivare alla conclusione che è opportuno decomporre le associazioni non binarie (anche già normalizzate) sulle quali sia definita qualche dipendenza il cui secondo membro contiene più di una entità. In termini più semplici, poiché è raro incontrare associazioni che

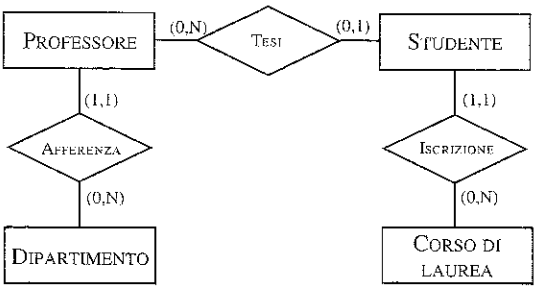

Figura 9.17 Il risultato di un'ulteriore decomposizione di un'associazione.

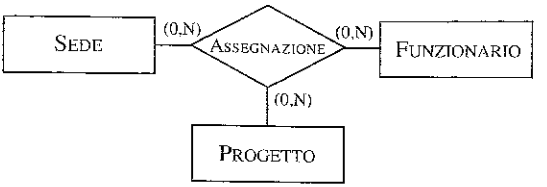

Figura 9.18 Un'associazione difficile da decomporre.

coinvolgano più di tre entità, possiamo affermare che è di solito opportuno decomporre un'associazione ternaria se su di essa è definita una dipendenza funzionale il cui primo membro è costituito da una entità e il secondo membro dalle altre due.

In alcuni casi la decomposizione può risultare non conveniente: per esempio se le due entità nel secondo membro della dipendenza sono fra loro strettamente correlate (nel caso esaminato, se interessano solo studenti che già svolgono una tesi e quindi per ogni studente iscritto a un corso di laurea esiste un professore con cui svolge la tesi), oppure se sull'associazione sono definite altre dipendenze funzionali che non verrebbero conservate nel caso di una decomposizione.

9.7.4 Ulteriori decomposizioni di schemi concettuali

Anche il caso discusso nel Paragrafo 9.5.1 di una relazione per la quale non esiste una buona decomposizione in forma normale di Boyce e Codd, può essere esaminato nell'ambito della progettazione concettuale. Supponiamo che, nel corso dell'analisi, abbiamo definito lo schema in Figura 9.18 per il quale valgono le seguenti dipendenze fra entità: Dirigente → Sede (ogni dirigente lavora presso un'unica sede) e Progetto Sede → Dirigente (per ogni sede, un progetto ha un solo dirigente).

In base alle considerazioni fatte nel Paragrafo 9.5.1, possiamo concludere che l'associazione non è in forma normale di Boyce e Codd e non può essere utilmente decomposta. Proprio in questo contesto, possiamo però individuare la possibilità di introdurre il concetto di reparto, per mezzo di una nuova entità, come mostrato nello schema Figura 9.19, che sostituisce la relazione ternaria.

Questa entità partiziona le singole sedi, come indicato dal suo identificatore esterno. Inoltre, i vincoli di cardinalità ci dicono che ogni reparto di una sede ha associato un direttore e diversi progetti. Dallo schema concettuale ottenuto, attraverso il processo di progettazione logica visto nel capitolo precedente, è possibile ottenere proprio gli schemi delle relazioni in Figura 9.12.

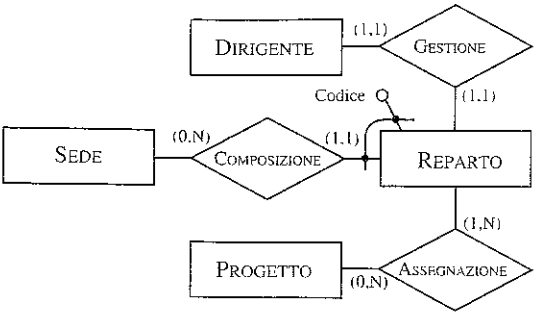

Figura 9.19 Una ristrutturazione dello schema in Figura 9.18.

Note bibliografiche

Le nozioni di base sulla normalizzazione, con la definizione di terza forma normale, sono state proposte da Codd [31]. La teoria della normalizzazione può essere approfondita nel testo in italiano di Atzeni, Batini e De Antonellis [5] oppure nel testo in inglese di Atzeni e De Antonellis [7].
Altri tre testi che studiano in maniera approfondita e formale aspetti legati alle dipendenze funzionali e alla teoria della normalizzazione sono quelli di Maier [56], Ullman [79] e Abiteboul, Hull e Vianu [1].
Sul testo di Cabibbo, Torlone e Batini [16], si trovano diversi esercizi di normalizzazione che fanno riferimento alla forma normale di Boyce e Codd.

Esercizi

Soluzioni sul sito http://www.atenaonline.it/atzeni

9.1 Considerare la relazione in Figura 9.20 e individuare le proprietà della corrispondente applicazione. Individuare inoltre eventuali ridondanze e anomalie nella relazione.

9.2 Individuare la chiave e le dipendenze funzionali della relazione considerata nell'Esercizio 9.1 e individuare poi una decomposizione in forma normale di Boyce e Codd.

9.3 Si consideri la relazione riportata in Figura 9.21 che rappresenta alcune informazioni sui prodotti di una falegnameria e i relativi componenti. Vengono indicati: il tipo del componente di un prodotto (attributo Tipo), la quantità del componente necessaria per un certo prodotto (attributo Q), il prezzo unitario del componente di un certo prodotto (attributo PC), il fornitore del componente (attributo Fornitore) e il prezzo totale del singolo prodotto (attributo PT). Individuare le dipendenze funzionali e la chiave di questa relazione.

Docente	Dipartimento	Facoltà	Preside	Corso
Verdi	Matematica	Ingegneria	Neri	Analisi
Verdi	Matematica	Ingegneria	Neri	Geometria
Rossi	Fisica	Ingegneria	Neri	Analisi
Rossi	Fisica	Scienze	Bruni	Analisi
Bruni	Fisica	Scienze	Bruni	Fisica

Figura 9.20 Relazione per l'Esercizio 9.1.

Prodotto	Componente	Tipo	Q	PC	Fornitore	PT
Libreria	Legno	Noce	50	10 000	Forrest	400 000
Libreria	Bulloni	B212	200	100	Bolt	400 000
Libreria	Vetro	Cristal	3	5000	Clean	400 000
Scaffale	Legno	Mogano	5	15 000	Forrest	300 000
Scaffale	Bulloni	B212	250	100	Bolt	300 000
Scaffale	Bulloni	B412	150	300	Bolt	300 000
Scrivania	Legno	Noce	10	8000	Wood	250 000
Scrivania	Maniglie	H621	10	20 000	Bolt	250 000
Tavolo	Legno	Noce	4	10 000	Forrest	200 000

Figura 9.21 Una relazione contenente dati di una falegnameria.

9.4 Con riferimento alla relazione in Figura 9.21 si considerino le seguenti operazioni di aggiornamento:

- inserimento di un nuovo prodotto;
- cancellazione di un prodotto;
- aggiunta di un componente a un prodotto;
- modifica del prezzo di un prodotto.

Discutere i tipi di anomalia che possono essere causati da tali operazioni.

9.5 Si consideri sempre la relazione in Figura 9.21. Descrivere le ridondanze presenti e individuare una decomposizione della relazione che non presenti tali ridondanze. Fornire infine l'istanza dello schema così ottenuto, corrispondente all'istanza originale. Verificare poi che sia possibile ricostruire l'istanza originale a partire da tale istanza.

9.6 Individuare le dipendenze funzionali definite sulla relazione in Figura 9.21 e decomporre la relazione con l'algoritmo di sintesi di schemi in terza forma normale illustrato nel Paragrafo 9.6.3.

9.7 Considerare uno schema di relazione $R(ENLCSDMPA)$, con le dipendenze $E \rightarrow NS$, $NL \rightarrow EMD$, $EN \rightarrow LCD$, $C \rightarrow S$, $D \rightarrow M$, $M \rightarrow D$, $EPD \rightarrow AE$, $NLCP \rightarrow A$. Calcolare una copertura ridotta per tale insieme e decomporre la relazione in terza forma normale.

9.8 Si consideri lo schema della relazione in Figura 9.22. La chiave di questa relazione è costituita dagli attributi Titolo e Copia, e su di esso è definita la dipen-

denza Titolo → Autore Genere. Verificare se lo schema è o meno in terza forma normale e, in caso negativo, decomporlo opportunamente.

Titolo	Autore	Genere	Copia	Scaffale
Decamerone	Boccaccio	Novelle	1	A75
Divina Commedia	Dante	Poema	1	A90
Divina Commedia	Dante	Poema	2	A90
I Malavoglia	Verga	Romanzo	1	A90
I Malavoglia	Verga	Romanzo	2	A75
I Promessi Sposi	Manzoni	Romanzo	1	B10
Adelchi	Manzoni	Tragedia	1	B20

Figura 9.22 Relazione per l'Esercizio 9.8.

9.9 Si consideri la relazione in Figura 9.23 in cui CM e CD sono, rispettivamente, abbreviazioni di CodiceMateria e CodiceDocente e l'attributo CS assume valori di tipo stringa che indicano in qualche modo il corso di studio o i corsi di studio cui un corso è destinato. Individuare la chiave (o le chiavi) e le dipendenze funzionali definite su di essa (ignorando quelle che si ritiene siano eventualmente "occasionali") e spiegare perché essa non soddisfa la forma normale di Boyce e Codd. Decomporla in forma normale di Boyce e Codd nel modo che si ritiene più opportuno.

CM	Materia	CS	Sem.	CD	NomeDoc	Dipartimento
I01	Analisi I	Inf	I	NR1	Neri	Matematica
I01	Analisi I	El	I	NR2	Neri	Matematica
I02	Analisi II	El-Inf	I	NR1	Neri	Matematica
I04	Fisica I	El	II	BN1	Bianchi	Fisica
I04	Fisica I	Mec	I	BR1	Bruni	Meccanica
I04	Fisica I	Inf	I	BR1	Bruni	Meccanica
I05	Fisica II	El	II	BR1	Bruni	Meccanica
I06	Chimica	Tutti	I	RS1	Rossi	Fisica

Figura 9.23 Relazione per l'Esercizio 9.9.

9.10 Considerare la relazione in Figura 9.24, che contiene informazioni relative ai ristoranti di una città, da riportare in una guida turistica.

Si noti che CT, CC e CZ sono, rispettivamente, abbreviazioni di CodiceTipo, CodiceCarta e CodiceZona. Individuare la chiave (o le chiavi) della relazione e le dipendenze funzionali definite su di essa (ignorando quelle che si ritiene siano eventualmente "occasionali") e spiegare perché essa non soddisa la forma normale di Boyce e Codd. Decomporla in forma normale di Boyce e Codd nel modo che si ritiene più opportuno.

9.11 Si consideri lo schema Entità-Relazione in Figura 9.25.

Sui dati descritti da questo schema valgono le seguenti proprietà:

Cod	Nome	Indirizzo	T	Tipo	CC	Carta	CZ	Zona
342	Da Piero	V. Larga 32	R	Region.	V	VISA	C	Centro
342	Da Piero	V. Larga 32	R	Region.	A	AmEx	C	Centro
421	Buono	Vic. Corto 1	R	Region.	A	AmEx	C	Centro
425	Paris	V. Lunga 4	I	Internaz.	D	Diners	N	Nord
425	Paris	V. Lunga 4	I	Internaz.	A	AmEx	N	Nord
655	Canton	V. Breve 2	C	Cinese	V	VISA	O	Ovest

Figura 9.24 Relazione per l'Esercizio 9.10.

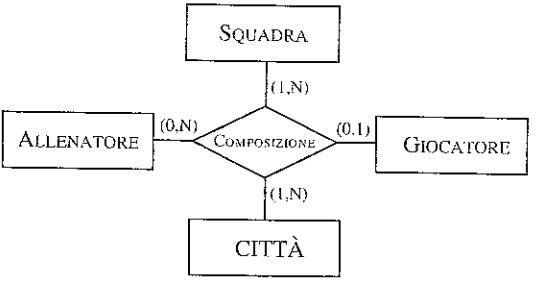

Figura 9.25 Uno schema da sottoporre a verifica di normalizzazione.

- un giocatore può giocare per una sola squadra (o per nessuna);
- un allenatore può allenare una sola squadra (o nessuna);
- una squadra ha un solo allenatore, diversi giocatori e appartiene a un'unica città.

Verificare se lo schema soddisfa la forma normale di Boyce e Codd e, in caso negativo, ristrutturarlo in un nuovo schema in maniera che soddisfi tale forma normale.

9.12 Consideriamo la relazione in Figura 9.26 e le sue seguenti possibile decomposizioni:

- Reparto, Cognome in una relazione e Cognome, Nome, Indirizzo nell'altra;

Reparto	Cognome	Nome	Indirizzo
Vendite	Rossi	Mario	Via Po 20
Acquisti	Rossi	Mario	Via Po 20
Bilancio	Neri	Luca	Via Taro 12
Personale	Rossi	Luigi	Via Taro 12

Figura 9.26 Relazione per l'Esercizio 9.12.

- Reparto, Cognome, Nome in una relazione e Nome, Indirizzo nell'altra;
- Reparto, Cognome, Nome in una relazione e Cognome, Nome, Indirizzo nell'altra.

Individuare, con riferimento sia all'istanza di relazione specifica sia all'insieme delle istanze sullo stesso schema (con le proprietà naturalmente associate), quali di tali decomposizioni sono senza perdita.

9.13 Consideriamo nuovamente la relazione in Figura 9.26. Individuare quali delle seguenti decomposizioni conservano le sue dipendenze.

- Una relazione sugli attributi Reparto, Cognome e Nome e l'altra sugli attributi Cognome e Indirizzo.
- Una relazione su Reparto, Cognome e Nome e l'altra su Cognome, Nome e Indirizzo.
- Una relazione su Reparto e Indirizzo e l'altra su Reparto, Cognome e Nome.

10 Sviluppo di applicazioni per basi di dati

Il dialogo diretto con l'interprete SQL è riservato a pochi utenti esperti; l'accesso di gran lunga più tipico a una base di dati avviene attraverso applicazioni integrate nel sistema informativo. Infatti, nella quasi totalità degli usi delle basi di dati, l'utente accede ai dati usando una applicazione, che può essere interattiva oppure non interattiva (detta anche *batch*); l'applicazione fornisce agli utenti un'interfaccia semplificata che favorisce l'interazione. Lo sviluppo di applicazioni può a sua volta essere facilitato da strumenti di sviluppo, che permettono di costruire applicazioni complete per la gestione dei dati in modo guidato, con minimo ricorso alla programmazione. Buona parte dei sistemi commerciali offrono, oltre al gestore di base di dati vero e proprio, anche un insieme di strumenti specializzati (e proprietari) per lo sviluppo semplificato di applicazioni.

Questo capitolo si concentra sulle tecniche di connessione tra le basi di dati e le applicazioni costruite con i tradizionali linguaggi di programmazione di alto livello, il caso di gran lunga più frequente nel mondo professionale. Il problema da risolvere è l'integrazione tra i comandi SQL, i quali sono responsabili di realizzare l'accesso alla base di dati, e le normali istruzioni del linguaggio di programmazione.

Il capitolo analizzerà diverse tecniche per lo sviluppo di applicazioni nei linguaggi di programmazione di tipo procedurale e a oggetti; descriveremo prima SQL Embedded, in cui SQL viene integrato direttamente all'interno del normale linguaggio di programmazione (Paragrafo 10.1); poi illustreremo gli approcci in cui l'integrazione con SQL avviene tramite l'invocazione di un'opportuna libreria di funzioni (approccio a *Call Level Interface*, Paragrafo 10.2); affronteremo quindi nel Paragrafo 10.3 il modo in cui gestire le transazioni in questi due approcci.

Successivamente tratteremo il metodo più moderno e completo per la connessione tra programmi a oggetti e basi di dati relazionali, la cosiddetta *mappatura relazionale degli oggetti* (in inglese, *Object Relational Mapping - ORM*). Si tratta di una tecnica che automatizza, per una buona parte, il compito di costruire interrogazioni all'interno delle istruzioni di un programma e di trasformare le tuple del risultato in oggetti del linguaggio ospite. Introdurremo il metodo nel Paragrafo 10.4, per poi soffermarci nel Paragrafo 10.5 su Java Persistence API, lo standard più recente emerso dal mondo Java.

10.1 SQL Embedded

Sviluppare un'applicazione che sfrutti i contenuti di una o più basi di dati richiede la programmazione congiunta della logica applicativa, con un linguaggio di programmazione di alto livello (C, C++, Java, ...), e della logica di estrazione e aggiornamento dei dati, espressa in SQL.

L'integrazione del linguaggio SQL con i normali linguaggi di programmazione presenta alcuni ostacoli. Il primo problema deriva dal fatto che SQL è un linguaggio molto ricco, con una propria sintassi. Sono state proposte due soluzioni per consentire l'uso di SQL all'interno di un normale linguaggio di programmazione: l'incastonamento (SQL Embedded) e l'uso di Call Level Interface. Parleremo più avanti della seconda soluzione.

SQL Embedded prevede di introdurre direttamente nel programma sorgente scritto nel linguaggio di alto livello le istruzioni SQL, distinguendole dalle normali istruzioni tramite un opportuno separatore. Lo standard SQL prevede che il codice SQL sia preceduto dalla stringa `exec sql` e termini con il carattere '`;`'.

Dal punto di vista dell'implementazione, è necessario far precedere la compilazione del linguaggio di alto livello dall'esecuzione di un preprocessore che riconosce le istruzioni SQL e sostituisce a esse un insieme di chiamate ai servizi del DBMS, tramite una libreria specifica per ogni sistema. Il preprocessore riconosce il significato dei singoli comandi SQL e predispone l'insieme opportuno di strutture ausiliarie richieste per la loro esecuzione, segnalando al momento della compilazione eventuali errori nell'uso di SQL.

Questo risulta uno dei modi più agevoli per realizzare l'integrazione tra il linguaggio di programmazione e SQL, in quanto buona parte dei problemi vengono gestiti in modo automatico dal preprocessore. Affinché questa soluzione sia applicabile, è necessario un preprocessore per la particolare combinazione di DBMS-piattaforma-linguaggio-compilatore usata per lo sviluppo (per esempio, *Oracle Server, Linux, C, gcc*). I sistemi commerciali di maggiore diffusione mettono a disposizione combinazioni di strumenti di questo tipo per le configurazioni d'uso più frequenti.

La Figura 10.1 mostra un esempio d'uso di SQL Embedded basato su ECPG, un preprocessore di SQL Embedded per il linguaggio C e Postgres.

L'esempio riportato nella Figura 10.1 presenta alcune caratteristiche che vale la pena di mettere in evidenza. In primo luogo, il preprocessore introduce implicitamente la dichiarazione di una particolare struttura, `sqlca` (SQL Communication Area). Si tratta di una struttura dati che permette di gestire la comunicazione tra il programma in esecuzione e il DBMS. Questa struttura è visibile alla linea (10) del programma, dove si fa accesso al campo `sqlcode` della struttura. Il campo `sqlcode` ha proprio lo scopo di mantenere il codice d'errore dell'ultimo comando SQL inviato al DBMS. Un valore pari a zero significa che il comando è stato gestito con successo; un valore diverso da zero segnala invece che si è verificato un errore e il comando non è andato a buon fine.

Si può osservare come le dichiarazioni di variabili del programma C siano racchiuse (linee (4) e (8)) da una coppia di comandi `begin declare section` ed `end declare section`. Questo passo è necessario se si desiderano poi utilizzare le variabili del programma per scambiare dati tra il programma e il DBMS, e in particolare come parametri per i comandi SQL. Una variabile di questo genere prende il nome di *variabile del linguaggio ospite*, o *host variable*, nella terminologia di SQL Embedded. Nell'esempio, le variabili del linguaggio ospite vengono usate nel comando di inserimento che compare alla linea (13) del programma. La sintassi per l'uso delle variabili del linguaggio ospite è molto semplice e richiede solamente di

```
(1)  #include<stdlib.h>
(2)  main()
(3)  {
(4)    exec sql begin declare section;
(5)    char *NomeDip = "Manutenzione";
(6)    char *CittaDip = "Pisa";
(7)    int NumeroDip = 20;
(8)    exec sql end declare section;

(9)    exec sql connect to utente@librobd;
(10)   if (sqlca.sqlcode != 0) {
(11)     printf("Connessione al DB non riuscita\n"); }
(12)   else {
(13)     exec sql insert into Dipartimento
                 values(:NomeDip,:CittaDip,:NumeroDip);
(14)     exec sql disconnect all;
(15)   }
(16) }
```

Figura 10.1 Un programma C con SQL Embedded.

far precedere il nome della variabile dal carattere due punti.[1] Si osserva infine che in questo esempio, così come varrà per i successivi, si assume un modello di interazione con la base di dati che prevede che ogni singolo comando sia gestito da una singola transazione, con un *commit* eseguito automaticamente al termine di ogni comando (cosiddetto *autocommit*). Ciò evita di introdurre negli esempi i comandi espliciti di chiusura della transazione.

Un importante problema che caratterizza l'integrazione tra SQL e i normali linguaggi di programmazione è il cosidetto *conflitto d'impedenza* (*impedance mismatch*[2]). I linguaggi di programmazione accedono agli elementi di una tabella scandendone le righe una a una, utilizzando quello che viene detto un approccio *tuple-oriented*. Al contrario, SQL è un linguaggio di tipo *set-oriented*, che opera su intere tabelle, non su singole righe, e che restituisce come risultato di un'interrogazione un'intera tabella.

Questo problema ammette diverse soluzioni. La prima si basa sull'uso dei *cursori*. I cursori fanno parte dello standard SQL fin dalla versione SQL-2 e sono utilizzabili in combinazione con tutti i principali linguaggi di programmazione.

Una seconda soluzione consiste nell'utilizzare un linguaggio di programmazione che offra costruttori di dati più potenti e in particolare riesca a gestire in modo natura-

[1] Oltre alle variabili dichiarate espressamente nella *declare section*, anche i parametri formali della funzione che contiene il codice SQL Embedded possono essere utilizzati come variabili di scambio con il DBMS.

[2] Il termine deriva dall'ingegneria elettronica, dove si richiede che nell'accoppiamento di circuiti elettronici le impedenze d'ingresso e di uscita dei circuiti collegati siano il più possibile simili.

le una struttura del tipo "insieme di righe". Questa è una soluzione che sta diventando sempre più interessante, grazie alla crescente diffusione dei linguaggi di programmazione a oggetti, caratterizzati da potenti meccanismi di definizione e gestione di tipi. Le soluzioni ADO, ADO.NET e JDBC che descriveremo in seguito (Paragrafo 10.2) seguono questa impostazione per la risoluzione del problema.

Una terza soluzione, la più potente e sofisticata, richiede la presenza di un sistema di mappatura automatica tra gli oggetti del linguaggio ospite e le tabelle dello schema relazionale, nota come *mappatura relazionale degli oggetti* (*Object Relational Mapping – ORM*). La descriveremo approfonditamente nel Paragrafo 10.4.

10.1.1 Cursori

Un cursore è una variabile speciale che permette a un programma di accedere alle righe di una tabella una alla volta; il cursore viene definito su una generica interrogazione. Vediamo dapprima la sintassi per la definizione e l'uso dei cursori:

`declare` *NomeCursore* [`scroll`] `cursor for` *SelectSQL*
[`for` { `read only` | `update` [`of` *Attributo* { , *Attributo* }] }]

Il comando `declare cursor` definisce un cursore, associato a una particolare interrogazione sulla base di dati. L'opzione `scroll` specifica se si vuole permettere al programma di scandire liberamente le righe che formano il risultato dell'interrogazione. L'opzione finale `for update` specifica se il cursore possa essere utilizzato per eseguire un comando di modifica e, in tal caso, gli attributi oggetto del comando di `update`.

`open` *NomeCursore*

Il comando `open` ha come argomento un cursore. Al momento dell'esecuzione del comando `open`, viene eseguita l'interrogazione associata al cursore e il risultato diventa accessibile tramite l'istruzione `fetch`.

`fetch` [*Posizione* `from`] *NomeCursore* `into` *ListaDiFetch*

Il comando `fetch` copia il contenuto di una riga dal cursore nelle variabili del linguaggio ospite enumerate in *ListaDiFetch*. In particolare, *ListaDiFetch* contiene una variabile per ogni elemento della target list dell'interrogazione, con una corrispondenza tra colonne della tabella e variabili del linguaggio ospite dettata dalla posizione della variabile nella lista; ciascuna variabile della lista di fetch deve avere un tipo compatibile con i domini degli elementi della target list dell'interrogazione SQL.

Il cursore è una variabile speciale, dotata di un proprio stato: esiste infatti il concetto di *riga corrente*, che rappresenta l'ultima riga estratta dal cursore. Il parametro *Posizione* permette di specificare quale riga deve essere oggetto dell'operazione di `fetch`; il parametro può assumere i valori:

- `next` (la riga successiva alla corrente);
- `prior` (la riga precedente alla corrente);
- `first` (la prima riga del risultato);
- `last` (l'ultima riga del risultato);
- `absolute` *EspressioneIntera* (la riga che compare in posizione *i*-esima nel cursore, se *i* è il risultato della valutazione dell'espressione);
- `relative` *EspressioneIntera* (la riga che compare alla distanza di *i* posizioni da quella corrente nel cursore, se *i* è il risultato della valutazione dell'espressione).

L'indicazione esplicita della posizione da cui estrarre la riga è utilizzabile a condizione che sia stata specificata al momento della definizione del cursore l'opzione `scroll`, la quale garantisce appunto che sia possibile scandire liberamente il risultato dell'interrogazione. Se l'opzione `scroll` non è specificata, l'unico valore accettabile per il parametro *Posizione* è `next`. L'implementazione di un cursore non scandibile liberamente può essere effettuata in modo più efficiente: le righe del risultato possono essere scartate immediatamente dopo essere state restituite al programma, rilasciando memoria che può essere utilizzata da altri componenti del sistema; inoltre, i tempi di risposta possono essere ridotti, perché non bisogna aspettare che la valutazione della query sia completata prima di poter accedere al risultato. Tutto ciò è utile soprattutto quando l'interrogazione restituisce un gran numero di righe.

I comandi di `update` e `delete` permettono di apportare modifiche alla base di dati tramite l'uso di cursori, nel modo descritto dalle regole sintattiche seguenti.

```
update NomeTabella
    set Attributo = ( Espressione | null | default )
    {, Attributo = ( Espressione | null | default ) }
    where current of NomeCursore

delete from NomeTabella where current of NomeCursore
```

L'unica estensione rispetto ai comandi di `update` e `delete` già visti consiste nella presenza nella clausola `where` del predicato `current of` *NomeCursore*, che identifica la riga corrente (affinché venga aggiornata o rimossa). I comandi di modifica sono utilizzabili solo nel caso in cui il cursore acceda a una precisa riga di una singola tabella e non sono applicabili quando la query associata al cursore esegue un join tra diverse tabelle.

```
close NomeCursore
```

Il comando `close` chiude il cursore, ovvero comunica al sistema che il risultato dell'interrogazione non serve più. A questo punto le risorse impegnate per il cursore vengono liberate, in particolare lo spazio di memoria utilizzato per conservare il risultato.

```
(1)  void VisualizzaStipendiDipart(char NomeDip[])
(2)  {
(3)    exec sql begin declare section;
(4)    char Nome[20], Cognome[20];
(5)    long int Stipendio;
(6)    exec sql end declare section;

(7)    exec sql declare ImpDip cursor for
         select Nome, Cognome, Stipendio
         from Impiegato
         where Dipart = :NomeDip;
(8)    exec sql open ImpDip;
(9)    exec sql fetch ImpDip
                 into :Nome, :Cognome, :Stipendio;
(10)   printf("Dipartimento %s\n",NomeDip);
(11)   while (sqlca.sqlcode == 0)
(12)   {
(13)     printf("Nome e cognome dell'impiegato: %s %s",
                                          Nome,Cognome);
(14)     printf("Attuale stipendio: %d\n",Stipendio);
(15)     exec sql fetch ImpDip
                 into :Nome, :Cognome, :Stipendio;
(16)   }
(17)   exec sql close cursor ImpDip;
(18) }
```

Figura 10.2 Una procedura C che fa uso di cursori.

Un semplice esempio di dichiarazione di cursore è:

```
declare CursoreImpiegati scroll cursor for
    select Cognome, Nome, Stipendio
    from Impiegato
    where Stipendio > 40 and Stipendio < 100
```

Il cursore `CursoreImpiegati` si associa all'interrogazione che estrae i dati relativi ai dipendenti che guadagnano tra 40 e 100.

La Figura 10.2 mostra una semplice funzione C che fa uso dei cursori. Le variabili del programma vengano rappresentate nei comandi SQL con il nome preceduto dal carattere ":" (due punti) e devono avere un tipo compatibile con i valori che dovranno contenere. Per riconoscere quando il cursore ha terminato di estrarre tutte le righe, si fa uso del campo `sqlcode` della struttura predefinita `sqlca` (riga 11).

Le interrogazioni che restituiscono sempre una sola riga (per esempio perché nella condizione compare un predicato di uguaglianza con una chiave) vengono dette query *scalari*; per queste interrogazioni è possibile utilizzare una semplice interfaccia

tra SQL e il linguaggio di programmazione, che non richiede di definire un cursore. Si può infatti in questo caso fare uso della clausola `into`, mediante la quale si stabilisce in modo diretto a quali variabili del programma debba essere assegnato il risultato dell'interrogazione. Un esempio è il seguente:

```
select Nome, Cognome into :nomeDip, :cognomeDip
   from Dipendente
   where Matricola = :matrDip;
```

I valori degli attributi Nome e Cognome del dipendente la cui matricola è contenuta nella variabile `matrDip` verranno rispettivamente copiati nelle variabili `nomeDip` e `cognomeDip`.

10.1.2 SQL dinamico

In molte situazioni sorge la necessità di definire le interrogazioni da effettuare sulla base di dati al momento dell'esecuzione, per esempio perché un parametro di ingresso dell'interrogazione è fornito dall'utente dell'applicazione.

Se le interrogazioni hanno una struttura predefinita e ciò che varia è solamente il valore dei parametri usati in ingresso, allora diventa possibile costruire un'applicazione che gestisce le interrogazioni richieste sfruttando solamente le variabili del linguaggio ospite, come illustrato nell'esempio della Figura 10.2, in cui la funzione `VisualizzaStipendiDipart` riceve il nome del dipartimento come parametro attuale.

In altri casi però l'applicazione richiede di effettuare interrogazioni caratterizzate da estrema variabilità, con la necessità di definire al momento dell'esecuzione del programma non solo i valori dei parametri, ma anche la forma delle interrogazioni, l'insieme di tabelle cui accedere e la struttura del risultato. I meccanismi per l'invocazione di comandi SQL all'interno dei programmi che abbiamo visto finora non vanno bene in questo contesto, dato che richiedono che la struttura dell'interrogazione sia prefissata (si parla infatti di SQL *statico*). Una famiglia alternativa di comandi, che rientra sempre tra le soluzioni basate su SQL Embedded, permette l'uso di SQL *dinamico*. Con questi comandi è possibile costruire un programma che esegue istruzioni SQL create al momento dell'esecuzione. SQL dinamico richiede però un supporto speciale da parte del sistema. Il problema principale è il passaggio di informazioni tra il programma e il comando SQL. Visto che il comando SQL è arbitrario, il programma non ha modo di conoscere al momento della compilazione quali siano i parametri richiesti in ingresso e quali siano le caratteristiche dell'eventuale risultato prodotto dall'esecuzione del comando. Queste informazioni sono però necessarie affinché il programma sia in grado di gestire l'interrogazione al suo interno.

L'uso di SQL dinamico modifica l'interazione tra il programma e il DBMS. Nel caso di SQL statico, i comandi SQL sono noti a tempo di compilazione e vengono gestiti dal preprocessore, che analizza la struttura del comando e la traduce nel linguaggio interno della base di dati. In questo modo il comando non deve essere analizzato e ottimizzato ogni volta che viene richiesta la sua esecuzione. Ciò porta considerevoli vantaggi in termini di prestazioni. SQL dinamico non può avvalersi della fase di

preprocessamento, perché i comandi SQL sono noti solo a tempo di esecuzione, ma cerca di offrire quando possibile gli stessi vantaggi dell'uso del preprocessore, mettendo a disposizione due diverse modalità di interazione tra programma e DBMS: si può eseguire direttamente l'interrogazione, per cui all'analisi segue immediatamente l'esecuzione dell'interrogazione, o la gestione dell'interrogazione può avvenire in due fasi: una prima fase di analisi e una seconda fase in cui l'interrogazione viene propriamente eseguita.

Esecuzione immediata Mediante il comando di `execute immediate` si richiede l'esecuzione di un'istruzione SQL, specificata direttamente o contenuta in un parametro di tipo stringa di caratteri dell'ambiente del programma:

`execute immediate` *IstruzioneSQL*

Il modo immediato può essere utilizzato solo per comandi che non richiedono parametri né in ingresso né in uscita, come per esempio alcuni comandi di inserimento e cancellazione. Un esempio d'uso del comando è il seguente:

```
exec sql execute immediate
  "delete from Impiegato where Nome = 'Mario'"
```

In un programma C si potrebbe invece scrivere:

```
istruzioneSql =
  "delete from Impiegato where Nome = 'Mario'";
...
exec sql execute immediate :istruzioneSql;
```

Quando però un comando deve essere eseguito ripetutamente, o quando il programma ospite deve passare parametri in ingresso o ricevere parametri di uscita dal comando SQL, diventa conveniente distinguere le due fasi di preparazione e di esecuzione.

Fase di preparazione Il comando `prepare` analizza un'istruzione SQL e la traduce nel linguaggio interno del DBMS. Il comando `prepare` associa alla traduzione dell'istruzione un nome, che può essere poi usato dagli altri comandi:

`prepare` *NomeComando* `from` *IstruzioneSQL*

L'istruzione SQL può contenere dei parametri in ingresso, rappresentati dal carattere di punto interrogativo. Per esempio:

```
prepare :comando
  from "select Città from Dipartimento where Nome = ?"
```

In questo modo alla variabile `comando` del programma corrisponde la traduzione dell'istruzione, con un parametro di ingresso che rappresenta il nome del dipartimento che deve essere selezionato dall'interrogazione.

Quando un'istruzione SQL che era stata tradotta non serve più, è possibile rilasciare la memoria occupata dalla traduzione dell'istruzione utilizzando il comando `deallocate prepare`, con la seguente sintassi:

`deallocate prepare` *NomeComando*

Per esempio, per deallocare il comando precedente, si può utilizzare:

```
deallocate prepare :comando
```

Fase di esecuzione Per eseguire un comando che è stato preelaborato da una istruzione `prepare` si usa l'istruzione `execute`, che ha la seguente sintassi:

`execute` *NomeComando* [`into` *ListaTarget*] [`using` *ListaParametri*]

La lista dei target contiene l'elenco delle variabili del linguaggio ospite in cui deve essere copiato il risultato dell'esecuzione del comando (questa parte è opzionale qualora il comando SQL non restituisca alcun valore). La lista dei parametri specifica invece i valori che devono essere assunti dai parametri di input del comando SQL preparato (anche questa parte può essere assente se il comando SQL non contiene parametri di input).

Un esempio può essere il seguente:

```
execute :comando into :citta using :dipartimento
```

Supponendo che la variabile del programma `dipartimento` abbia come valore la stringa `'Produzione'`, l'effetto di questo comando è di eseguire la query:

```
select Città
from Dipartimento
where Nome = 'Produzione'
```

e di ottenere come conseguenza la stringa `'Torino'` nella variabile `citta` del programma ospite.

Cursori con SQL dinamico L'uso dei cursori con SQL dinamico è molto simile all'uso dei cursori che viene fatto da SQL statico. Le uniche due differenze consistono nel fatto che si associa al cursore l'identificativo dell'interrogazione invece che l'interrogazione stessa, e che i comandi d'uso del cursore ammettono le clausole `into` e `using` che permettono la specifica degli eventuali parametri di ingresso e di uscita.

Un esempio d'uso di un cursore dinamico è il seguente, in cui si suppone che l'interrogazione definita nella stringa `istruzioneSQL` ammetta un parametro:

```
prepare :comando from :istruzioneSQL
declare Cursore cursor for :comando

open Cursore using :nome1
```

10.2 Call Level Interface (CLI)

I meccanismi descritti finora ricadono nella famiglia delle soluzioni SQL Embedded, dove si prevede di disporre di un preprocessore, utilizzato a tempo di compilazione o di esecuzione, che espande comandi SQL presenti all'interno di un normale linguaggio di programmazione. Il preprocessore non fa altro che tradurre i comandi SQL in un insieme opportuno di chiamate di sottoprogrammi facenti parte della libreria offerta dal DBMS che realizzano il dialogo con la base di dati.

Una soluzione alternativa consiste nel mettere direttamente a disposizione del programmatore un insieme di primitive che permettano di interagire con il DBMS. Questa famiglia di soluzioni va sotto il nome di *Call Level Interface*, in quanto il programmatore dispone direttamente di una libreria di funzioni per realizzare il dialogo con la base di dati. Rispetto alla soluzione SQL Embedded, con la CLI si dispone normalmente di uno strumento più flessibile, meglio integrato con il linguaggio di programmazione, con l'inconveniente di dover gestire esplicitamente aspetti che in SQL Embedded vengono risolti automaticamente dal preprocessore.

Diversi sistemi offrono delle proprie CLI. Il modo generale d'uso di questi strumenti è il seguente.

1. Si utilizza un servizio della CLI per creare una connessione con il DBMS.
2. Si invia attraverso la connessione un comando SQL che rappresenta la richiesta.
3. Si riceve come risposta del comando una struttura relazionale in un opportuno formato; la CLI dispone di un certo insieme di primitive che permettono di analizzare la struttura del risultato del comando.
4. Al termine della sessione di lavoro, si chiude la connessione e si rilasciano le strutture dati utilizzate per la gestione del dialogo.

Anche in questo campo diverse iniziative hanno portato alla definizione di specifiche e alla loro implementazione, consentendo agli sviluppatori di realizzare applicazioni che accedono a basi di dati in modo relativamente facile. Descriveremo quindi inizialmente le soluzioni che caratterizzano la piattaforma software Microsoft (ODBC, OLE DB, ADO e ADO.NET). Presenteremo poi la soluzione JDBC, che rappresenta invece la soluzione di riferimento per quanto riguarda lo sviluppo di applicazioni nel linguaggio Java. In Figura 10.3 viene mostrato un quadro riassuntivo delle diverse soluzioni.

10.2.1 ODBC e soluzioni proprietarie Microsoft

La piattaforma software Microsoft presenta una certa varietà di soluzioni per l'accesso alle basi di dati. Illustriamo le caratteristiche principali di queste soluzioni.

ODBC *Open DataBase Connectivity* (ODBC) è un'interfaccia applicativa proposta originariamente dalla Microsoft e diventata in seguito uno standard. ODBC costituisce il cuore dell'architettura per l'accesso alle basi di dati dell'attuale piattaforma software Microsoft. ODBC è una soluzione molto ricca, sviluppata con l'obiettivo di consentire l'accesso a basi di dati relazionali in un contesto eterogeneo e distribuito.

ODBC	Interfaccia standard che permette di accedere a basi di dati in qualunque contesto, realizzando interoperabilità con diverse combinazioni di DBMS-Sist.Op.-Reti
OLE DB	Soluzione proprietaria Microsoft, basata sul modello COM, che permette ad applicazioni Windows di accedere a sorgenti dati generiche (non solo DBMS)
ADO	Soluzione proprietaria Microsoft che permette di sfruttare i servizi OLE DB, utilizzando un'interfaccia record-oriented
ADO.NET	Soluzione proprietaria Microsoft che adatta ADO alla piattaforma .NET; offre un'interfaccia set-oriented e introduce i *DataAdapter*
JDBC	Soluzione per l'accesso ai dati in Java sviluppata da Sun Microsystems; offre in quel contesto un servizio simile a ODBC

Figura 10.3 Caratteristiche delle soluzioni CLI.

La descrizione dettagliata di questa soluzione richiederebbe molto spazio. Metteremo invece in evidenza le caratteristiche di fondo della soluzione e descriveremo un modo rapido per sfruttare i suoi servizi nella realizzazione di un'applicazione interfacciata con un DBMS.

L'obiettivo di ODBC è di permettere la costruzione di applicazioni che facciano accesso a fonti di dati di tipo eterogeneo, principalmente relazionali, ma anche non relazionali. ODBC ha incontrato un notevole successo ed è supportata dalla maggior parte dei prodotti relazionali. È di gran lunga la soluzione più significativa per i sistemi Windows ed esistono implementazioni di essa per molti altri sistemi operativi (anche se in questi ambienti le soluzioni sono caratterizzate da livelli di robustezza e diffusione molto minori). Tramite un'interfaccia ODBC, le applicazioni possono accedere a dati memorizzati su sistemi relazionali di diversi venditori, anche installati su server remoti raggiungibili via rete; il linguaggio di interrogazione supportato da ODBC è un SQL "ristretto", formato da un sottoinsieme limitato di istruzioni, standardizzato dal comitato *SQL Access Group* (SAG).

La Figura 10.4 mostra l'architettura di ODBC: il collegamento tra un'applicazione e una o più basi di dati è mediato da uno strato di software di virtualizzazione, che maschera le differenze tecniche di accesso a fonti dati di tipi differenti.

L'architettura ODBC prevede l'uso di un *driver*, una libreria che viene collegata dinamicamente alle applicazioni e viene da esse invocata. Il driver maschera le differenze di interazione legate non solo al DBMS, ma anche al sistema operativo e al protocollo di rete utilizzato. Pertanto, per garantire la compatibilità rispetto allo standard ODBC, ciascun venditore deve fornire dei driver che prevedano l'uso del

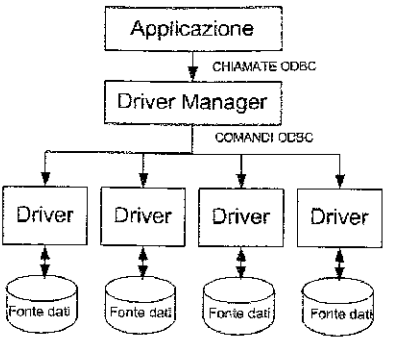

Figura 10.4 L'architettura ODBC.

proprio DBMS nell'ambito di una specifica rete e con uno specifico sistema operativo. Per esempio, la tripla (*Oracle, Windows 8, TCP*) identifica uno specifico driver. Il driver così maschera tutti i problemi d'interoperabilità (non solo quelli posti dal DBMS), e facilita la scrittura delle applicazioni e il loro trasporto da un ambiente di esecuzione a un altro.

Come mostra la Figura 10.4, l'accesso a una base di dati tramite ODBC richiede la cooperazione di quattro componenti di sistema.

- L'*applicazione* richiama le funzioni SQL per eseguire interrogazioni e per acquisirne i risultati. La scelta del protocollo di comunicazione, del server DBMS e del sistema operativo del nodo ove il DBMS è installato sono tutti trasparenti per l'applicazione, essendo mascherati dal driver.
- Il *driver manager* è responsabile di caricare i driver a richiesta dell'applicazione e di rilasciare le risorse da questi richieste al termine dell'utilizzo. Questo componente, fornito direttamente dalla Microsoft per il mondo Windows, garantisce anche alcune funzioni elementari di controllo degli errori e di gestione della corrispondenza fra nomi e funzioni di inizializzazione, che assicurano il corretto funzionamento dei driver.
- I *driver* sono responsabili di eseguire comandi ODBC; pertanto, sono in grado di eseguire interrogazioni in SQL, traducendole in modo da adattarsi alla sintassi e alla semantica delle specifiche fonti dati cui l'applicazione si collega. I driver sono anche responsabili di restituire i risultati alle applicazioni, tramite meccanismi di buffering.
- La fonte dei dati (in inglese: *data source*) è il sistema che esegue le funzioni trasmesse dal *client*. Le fonti d'informazione di interesse principale sono i database server relazionali, ma esistono driver ODBC a funzionalità limitata anche per altri tipi di fonti, quali spreadsheet e dati memorizzati in file di testo strutturati nel formato separato da virgole (in inglese, *comma delimited* o anche *comma-separated values*, CSV).

ODBC standardizza i codici di errore, così da consentire il controllo delle condizioni di errore a tempo di esecuzione con fonti dati differenti. ODBC consente di usare sia interrogazioni SQL statiche sia interrogazioni generate ed eseguite dinamicamente. L'esecuzione può quindi generare degli errori, quando il codice SQL contenuto nella stringa non è corretto.

ODBC risolve un problema significativo: rendere lo sviluppo delle applicazioni meno dipendente dalle caratteristiche tecniche delle CLI dei differenti venditori. Questo beneficio comporta però una certa complessità del codice applicativo, che tipicamente prevede quattro principali fasi: la creazione di una struttura dati contenitore che descrive l'ambiente di esecuzione; la creazione di una struttura dati che rappresenta la connessione alla fonte dati; l'attivazione della connessione alla fonte dati; l'esecuzione di uno o più comandi SQL (statici o dinamici) attraverso la connessione attiva. Queste fasi sono complicate dalla necessità di utilizzare strutture dati di interscambio tra il programma e il DBMS che devono poter rappresentare una vasta gamma di proprietà dei sistemi eterogenei supportati da ODBC e dall'esigenza, come in SQL Embedded, di trasmettere dati in input e in output tra il programma ospite e le funzioni dell'interfaccia ODBC.

Per semplificare il compito del programmatore di applicazioni connesse a DBMS, Microsoft ha sviluppato ulteriori soluzioni proprietarie che migliorano il livello di astrazione mascherando i dettagli tecnici dell'interazione tra programma e base di dati, sfruttano i vantaggi del paradigma di programmazione a oggetti, ed estendono ulteriormente i tipi di sorgenti dati con le quali un'applicazione può interagire. Descriviamo quindi brevemente *OLE DB*, *ADO* e *ADO.NET*.

OLE DB *Object Linking and Embedding for DataBases* (OLE DB) è un'interfaccia generale, che permette di accedere a sorgenti dati di tipo generico: non solo sistemi relazionali, ma anche una vasta tipologia di archivi di dati, come per esempio caselle di posta elettronica o sistemi CAD/CAM. OLE DB si basa sul modello a oggetti COM, che ricopre un ruolo di primo piano nell'architettura software Microsoft. A seconda delle caratteristiche della sorgente dati usata, sarà o meno possibile utilizzare servizi di accesso sofisticati. Per esempio, se la sorgente dati è una base dati relazionale, cui si fa accesso utilizzando ODBC, sarà possibile interagire inviando comandi SQL da eseguire che produrranno come risposta insiemi di tuple, cui sarà possibile accedere con meccanismi di scansione.

ADO *ActiveX Data Object*(ADO) è un'interfaccia di alto livello ai servizi offerti da OLE DB basata su un modello a oggetti, che comprende quattro oggetti fondamentali: la connessione, il comando, la tupla e l'insieme di tuple. La connessione (*Connection*) rappresenta il canale di comunicazione che deve essere stabilito per interagire con una sorgente dati, fornendo la locazione della sorgente dati nel sistema e normalmente lo username e la password da utilizzare per identificarsi sul sistema; un componente importante della connessione è la collezione di errori (*Errors*), che rappresenta l'insieme di anomalie che si possono verificare nell'interazione con la sorgente dati; questa struttura viene definita come una collezione in quanto è possibile che diverse anomalie si manifestino contemporaneamente. Il comando (*Command*)

rappresenta la stringa di caratteri che contiene l'istruzione SQL da eseguire nella sorgente dati. La tupla (*Record*) descrive la singola riga di una tabella e offre strumenti per riconoscere la struttura dei dati che compongono la tupla. L'insieme di tuple (*RecordSet*) definisce la struttura usata per memorizzare i risultati delle interrogazioni inviate alla sorgente dati; il RecordSet offre meccanismi per scandire l'insieme delle tuple e accedere a una tupla alla volta, in modo analogo ai cursori.

ADO è utilizzabile con i diversi linguaggi di programmazione per lo sviluppo di applicazioni disponibili per la piattaforma Microsoft Windows (per esempio, Visual C++, Visual Basic, VBscript, Jscript). La Figura 10.5 presenta un esempio di una procedura Visual Basic che fa uso di ADO. Anche senza conoscere il linguag-

```
(1)  Public Sub InterrogaBD()

(2)    Dim setImpiegati As ADODB.Recordset
(3)    Dim conn As ADODB.Connection
(4)    Dim stringaSQL As String
(5)    Dim comSQL As ADODB.Command
(6)    Dim arrImpiegati As Variant
(7)    Dim messaggio As String
(8)    Dim numRighe As Integer

(9)    Set conn = New ADODB.Connection
(10)   conn.Open "mioServer","giovanni","passwordsegreta"
(11)   Set setImpiegati = New ADODB.Recordset
(12)   stringaSQL = "select Nome, Cognome, Data " _
                    "from Impiegato order by Cognome"
(13)   comSQL.CommandText = stringaSQL
(14)   setImpiegati.Open comSQL, conn, , ,
(15)   Do While Not setImpiegati.EOF
(16)     messaggio = "Impiegato: " & setImpiegati!Nome & _
                 setImpiegati!Cognome & "(record " & _
                 setImpiegati.AbsolutePosition & _
                 " di " & setImpiegati.RecordCount & ")"
(17)     If MsgBox(messaggio, vbOkCancel) = vbCancel
(18)       Then Exit Do
(19)     setImpiegati.MoveNext
(20)   Loop
(21)   setImpiegati.Close
(22)   conn.Close
(23)   Set setImpiegati = Nothing
(24)   Set conn = Nothing

(25)  End Sub
```

Figura 10.5 Una procedura Visual Basic che accede a una sorgente dati ODBC tramite ADO.

gio, dovrebbe essere facile comprendere la struttura della procedura `InterrogaBD` nella Figura 10.5. Nelle istruzioni (2)-(8) si dichiarano le variabili che verranno usate nel programma. Si osserva in particolare l'uso dei tipi `Recordset`, `Connection` e `Command` di ADO, che vengono esplicitamente estratti dal modulo `ADODB`. La procedura inizializza nelle istruzioni (9) e (10) la connessione `conn`. Si inizializzano quindi l'insieme di tuple `setImpiegati` e il comando SQL che verrà eseguito sulla base di dati. L'istruzione (14) invoca il metodo `open` di `RecordSet` e quindi chiede l'esecuzione della query associata al comando `comSQL`. Il programma entra poi nel ciclo descritto dalle istruzioni (15)-(20). In questo ciclo si scandiscono una alla volta (metodo `MoveNext`) le tuple del risultato, fino a che le tuple sono terminate (metodo `EOF`) o l'utente ha selezionato il bottone `Cancel` della finestra che presenta i valori della tupla corrente. Al termine del ciclo, la procedura rilascia gli oggetti utilizzati.

ADO.NET Rappresenta l'evoluzione della soluzione ADO, progettata per operare all'interno della piattaforma .NET di Microsoft. ADO.NET non è una semplice estensione di ADO, bensì è il frutto di un'attività di riprogettazione che trae spunto dalle caratteristiche di .NET. Questa breve descrizione ha lo scopo di mettere in evidenza le caratteristiche di fondo di ADO.NET, segnalando le variazioni rispetto ad ADO. Come al solito, si rimanda alla ricca documentazione disponibile sul sito di Microsoft o a testi specifici sull'argomento per avere il livello di dettaglio che consenta di sviluppare un'applicazione.

La differenza di fondo tra ADO.NET e ADO è il diverso approccio per la gestione del dialogo con la base di dati. In ADO l'oggetto centrale è il *RecordSet*, il quale di norma viene gestito tramite connessioni dirette con la base di dati. L'accesso al *RecordSet* avviene poi con meccanismi che sono analoghi ai cursori.

La Figura 10.6 illustra invece l'architettura di ADO.NET e i principali oggetti che intervengono nella connessione tra un'applicazione e una fonte dati.

L'applicazione, o *data consumer* nella terminologia ADO.NET, è nel caso tipico un programma interattivo realizzato con le tecnologie WinForms o WebForms del

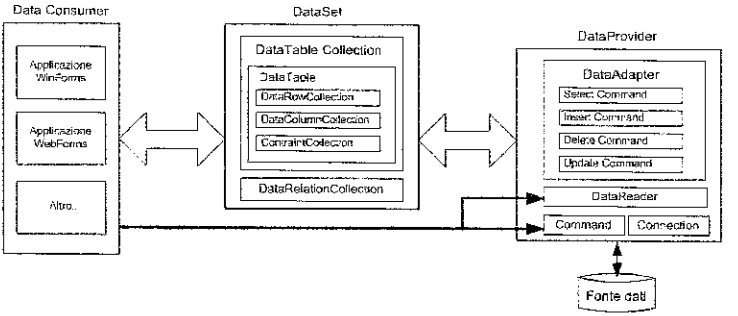

Figura 10.6 L'architettura ADO.NET.

Framework Microsoft .NET. Un data consumer si interfaccia con un oggetto *DataSet*, che rappresenta un magazzino temporaneo in memoria centrale (cache) dei dati estratti dalle fonti di dati e manipolati dall'applicazione.

Un *DataSet* ha una struttura interna articolata, costituita da una collezione di oggetti, tra cui i principali sono gli oggetti di tipo *DataTable*, che rappresentano le tabelle relazionali, gli oggetti di tipo *DataRelation*, che consentono di navigare tra le tabelle seguendo cammini di join, e gli oggetti di tipo *Constraint* per la gestione automatica di alcuni semplici vincoli di integrità sul contenuto di oggetti di tipo *DataTable*. A differenza di ADO, l'accesso al contenuto degli elementi dei *DataSet* e *DataTable* avviene utilizzando i normali meccanismi di accesso alle collezioni e agli attributi degli oggetti.

La flessibilità di ADO.NET nella gestione di relazioni e vincoli di integrità è resa possibile dal fatto che i *DataSet* rappresentano strutture a oggetti del tutto omogenee ai restanti oggetti del programma applicativo, allocate nel medesimo spazio; invece in ADO, così come in SQL Embedded e in ODBC, il risultato di una query viene mantenuto nel contesto della sessione di connessione con la base di dati e deve essere estratto dal programma cliente tupla per tupla, tramite l'invocazione dei metodi di accesso del *RecordSet*.

In ADO.NET il coordinamento tra i *DataSet* e le sorgenti dati avviene tramite componenti specifici, i *DataAdapter*. La classe *DataAdapter* presenta diverse specializzazioni, a seconda delle caratteristiche della sorgente dati (per esempio, *SqlDataAdapter* per accedere a Microsoft SQL Server, *OledbDataAdapter* per accedere a una sorgente OLE DB, *OdbcDataAdapter* per accedere a una sorgente ODBC). I metodi principali del *DataAdapter* sono il metodo *Fill*, che carica dati dalla sorgente e li trasferisce nel *DataSet*, e il metodo *Update*, che aggiorna il contenuto della sorgente dati riportando in essa le variazioni introdotte sul *DataSet*. Vi sono vari vantaggi nell'attribuire ai *DataAdapter* la gestione del dialogo con le sorgenti dati: è possibile minimizzare il numero di connessioni contemporaneamente attive in un sistema multitasking (in quanto ciascun *DataAdapter* occupa di norma la connessione solo per il tempo necessario per eseguire i metodi *Fill* e *Update*); è possibile sfruttare la conoscenza dello schema della base di dati per gestire la comunicazione in modo più efficiente; è facile integrare nello stesso *DataSet* dati che provengono da sorgenti diverse.

Infine, un'ultima caratteristica distintiva interessante di ADO.NET è la coerenza con l'impostazione comune del Framework .NET, che si avvale di XML come formato di rappresentazione dei contenuti e di scambio dati attraverso tutti i livelli di un'architettura applicativa. In particolare, un DataSet può essere riempito con dati provenienti da una fonte XML ed è possibile descrivere il DataSet in XML specificandone la struttura sotto forma di schema XSD (XML Schema Definition Language), indipendentemente dall'origine dati a cui il DataSet è collegato.

10.2.2 Java Database Connectivity (JDBC)

Java è un linguaggio di programmazione proposto dalla Sun Microsystems, che ha riscosso un notevole successo nello sviluppo di applicazioni, soprattutto dove esistono problemi significativi di portabilità tra sistemi operativi diversi e di integrazione

con le reti. Senza descrivere il linguaggio, ciò che andrebbe ben al di là degli obiettivi di questo testo, è possibile fare cenno alle caratteristiche che hanno contribuito al suo successo. Java è un linguaggio di programmazione a oggetti, ma il linguaggio di programmazione è solo il componente centrale di una proposta molto articolata, che comprende librerie, strumenti di sviluppo, esempi di applicazioni e buone pratiche di progettazione per una gamma di situazioni che va dallo sviluppo di soluzioni per sistemi embedded e mobili fino alla creazioni di sistemi informativi distribuiti per le imprese. L'architettura Java prevede che i programmi sorgente vengano tradotti in un formato compatto (bytecode), e vengano quindi eseguiti da una Java Virtual Machine (JVM), normalmente realizzata da un sistema software. Per eseguire un'applicazione Java in un particolare sistema di calcolo è quindi sufficiente disporre di una Java Virtual Machine per quell'ambiente di esecuzione. Visto che esistono diverse implementazioni della Java Virtual Machine per tutti i principali sistemi operativi, un'applicazione Java è in grado di offrire un livello di portabilità molto elevato.

Java, oltre a offrire tutte le caratteristiche di un moderno linguaggio di programmazione (modello a oggetti, gestione delle eccezioni, meccanismi per l'esecuzione concorrente e la sincronizzazione, invocazione remota, sicurezza ecc.), è arricchito da una vasta gamma di moduli di libreria che offrono soluzioni immediate e robuste per costruire applicazioni nei più diversi contesti. Per esempio, il linguaggio offre librerie per costruire interfacce grafiche, per realizzare servizi di crittografia e sicurezza e per gestire contenuti multimediali. Ciascuno di questi moduli esporta i suoi servizi all'ambiente Java tramite un'opportuna Application Programming Interface (API).

Tra le funzioni presenti nell'ambiente Java per la connessione tra applicazioni e basi di dati, un ruolo centrale spetta al modulo *Java Database Connectivity* (*JDBC*), il quale permette ai programmi Java di accedere in modo uniforme a basi di dati relazionali, in modo simile a quanto offerto da ODBC. La Figura 10.7 mostra uno schema dell'architettura JDBC: essa prevede uno strato, costituito dal driver manager, il quale isola l'applicazione dal componente responsabile di implementare il servizio (che è normalmente costruito dagli stessi produttori dei DBMS o da sviluppatori specializzati di software).

Le funzioni di accesso al DBMS possono essere realizzate in vari modi:

1. *Mediante il ponte* (in inglese *bridge*) *JDBC/ODBC*: questa architettura prevede di tradurre le richieste JDBC in richieste al driver manager ODBC. Questa soluzione richiede quindi che la macchina che esegue il codice Java possieda un'installazione di ODBC con il driver specifico per il DBMS cui si vuole fare accesso. Inoltre, esistono differenze nell'insieme di comandi supportati da JDBC e da ODBC, per cui il ponte *JDBC/ODBC* non garantisce tutte le funzionalità previste da JDBC.
2. *Mediante un driver nativo*: questa soluzione prevede la realizzazione di funzioni Java che convertono le richieste JDBC in chiamate alla CLI del DBMS. Il driver è quindi specifico per il DBMS cui si vuole accedere, sfrutta le interfacce di accesso al DBMS create per applicazioni scritte nei linguaggi di alto livello tradizionali, per esempio C e C++, e non è ottimizzato per l'ambiente Java.

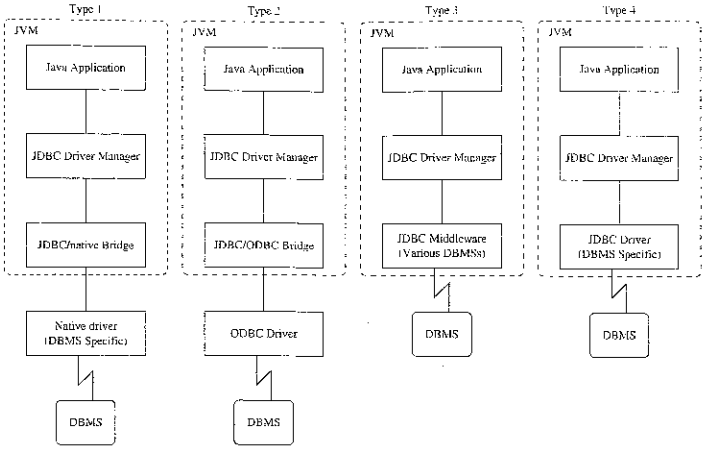

Figura 10.7 Le architetture JDBC.

Queste prime due soluzioni non sono realmente portabili, perché richiedono l'utilizzo di componenti *nativi*, cioè specifici dell'ambiente in cui vengono eseguiti. Esse consentono però di sfruttare librerie di connettività con le basi di dati non ancora disponibili in ambiente Java.

Vi sono poi due soluzioni che si appoggiano su un ambiente Java "puro".

3. *Middleware-server*: questa soluzione prevede l'uso di un server scritto in Java (denominato *application server*) responsabile di tradurre le richieste provenienti dal driver manager nel formato riconosciuto dal particolare DBMS che si intende utilizzare. Sono presenti sul mercato dei prodotti che realizzano queste funzioni permettendo di interagire con i sistemi relazionali più diffusi. Il middleware server può anche essere ospitato in un nodo della rete separato rispetto a quello dove risiedono l'applicazione client e il DBMS, con un miglioramento della scalabilità dell'architettura. L'introduzione di un livello intermedio rende l'ambiente di esecuzione dell'applicazione client indipendente dal DBMS utilizzato.
4. *Driver Java*: questa soluzione prevede l'uso di un driver scritto in Java specifico per il particolare sistema relazionale, in modo analogo ai driver in codice nativo usati in ODBC. I driver vengono normalmente offerti dagli stessi produttori dei sistemi relazionali.

JDBC rappresenta una soluzione molto interessante per la realizzazione di applicazioni portabili che facciano accesso a basi di dati.

Per utilizzare i servizi di JDBC normalmente si seguono i passi seguenti:

1. si carica il driver richiesto;

2. si crea una connessione con la base di dati;
3. si compone il comando SQL e lo si invia alla base di dati;
4. si gestisce il risultato del comando SQL.

La Figura 10.8 mostra un semplice esempio dell'uso di JDBC. La prima istruzione specifica la posizione nell'albero delle classi Java dei servizi JDBC, i quali si trovano nel sottoalbero `sql` dell'albero `java` che contiene i servizi di base del sistema. La connessione viene gestita con un oggetto della classe `Connection` (`conn`), inizializzato a `null`. L'istruzione alla linea (7) specifica il driver che deve essere caricato (trascuriamo i dettagli); in questo caso il driver è il ponte JDBC/ODBC. Nell'istruzione alle linee (9-10) viene stabilita la connessione con la base di dati.

```
(1)  import java.sql.*;
(2)  public class PrimoJdbc {
(3)    public static void main(String[] args) {
(4)      Connection conn = null;
(5)      try {
(6)        // Caricamento del driver
(7)        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
(8)        // Apertura della connessione
(9)        conn = DriverManager.getConnection(
(10)                                "jdbc:odbc:Corsi");
(11)       // Esecuzione dell'interrogazione
(12)       Statement interrogazione =
(13)                    conn.createStatement();
(14)       ResultSet risultato =
(15)         interrogazione.executeQuery(
(16)                            "select * from Corsi");
(17)       while (risultato.next()) {
(18)         String nomeCorso =
(19)                   risultato.getString("NomeCorso");
(20)         System.out.println(nomeCorso);
(21)       }
(22)     }
(23)   catch (SQLException e) {e.printStackTrace();
(24)   }
(25)   catch (Exception e) {e.printStackTrace();
(26)   }
(27)   finally {
(28)   // Chiusura della connessione
(29)   conn.close();
(30) }
(31) }
(32) }
```

Figura 10.8 Una classe Java che accede a una sorgente dati JDBC tramite un bridge JDBC-ODBC.

sfruttando il metodo `getConnection` della classe `DriverManager`, cui viene passato come parametro l'identificativo della base di dati (deve essere stata preventivamente creata la sorgente dati ODBC "Corsi"). In caso di errori, il sistema genererà un'eccezione che verrà inviata sullo stream standard di errore (mediante l'istruzione `e.printStackTrace()` alla linea (25)).

Una volta che la connessione è stata stabilita, è possibile inviare comandi SQL alla base di dati. Nell'istruzione alle linee (12-13) viene creato l'oggetto `interrogazione` appartenente alla classe `Statement`, associato alla connessione `conn`.

Il metodo `executeQuery` offerto dalla classe `Statement` viene invocato sull'oggetto `interrogazione`, con un parametro che contiene il testo della query SQL da eseguire.

L'esecuzione del comando produce un insieme di tuple, che vengono assegnate all'oggetto `risultato` della classe `ResultSet`. Questa classe offre servizi analoghi a quelli dei cursori, che permettono di scandire l'insieme di tuple e di estrarre dati da una tupla alla volta. Come nel caso dei cursori, anche `ResultSet` ha uno stato, costituito dalla tupla corrente (all'inizio sarà la prima); il metodo `next` permette di cambiare la posizione della tupla corrente, spostandola alla tupla successiva. Il metodo `getString` estrae dalla tupla corrente l'attributo il cui nome viene passato come parametro al metodo (in questo caso `NomeCorso`); l'uso di `getString` vale quando l'attributo ha un valore rappresentabile come stringa di caratteri; altri metodi permettono di estrarre dati per ciascun dominio (per esempio, `getInt`, `getFloat`, `getDate` ecc.).

Il ciclo `while` scandisce il contenuto di `risultato` fino a che il metodo `next()` non restituisce il valore booleano falso.

Si noti alla linea (27) la clausola `finally`, che delimita un blocco di codice eseguito alla terminazione del programma e permette di specificare la chiusura della connessione.

Concludiamo questa parte del capitolo che ha mostrato varie architetture per accedere alle basi di dati dall'interno di un programma applicativo con un esempio riepilogativo (nella Figura 10.9) che mostra varie funzionalità offerte da JDBC, tra cui l'accesso ai dati del catalogo relazionale e l'esecuzione di interrogazioni dinamiche costruite esplorando il contenuto del catalogo[3].

Il metodo `mostraDati` della classe *Esempio*, dopo avere aperto una connessione alla base di dati (linee (4-5)) e creato un oggetto di tipo statement (linea (6)), interroga il catalogo (linea (7)): si noti che il metodo `getMetaData()` della classe *Connection* restituisce un oggetto della classe *DatabaseMetaData*, dal quale è possibile estrarre l'elenco desiderato delle tabelle (per esempio, quelle il cui nome contiene la stringa `TABLE`) con il metodo `getTables()`. Il programma itera su tutte le tabelle trovate estraendone il contenuto mediante un'interrogazione dinamica (linea (11)). Anche un oggetto della classe *ResultSet* espone dei metadati che possono essere estratti con il metodo `getMetaData()`. Quest'ultimo restituisce un oggetto di tipo *ResultSetMetaData*, da cui si estraggono il numero delle colonne del risultato (linea (15)), nonché il nome (linea (16)) e il tipo (linea (17)) di ciascuna colonna,

[3]Per brevità omettiamo il codice che tratta le eccezioni e provvede al rilascio della connessione.

```
(1)  import java.sql.*
(2)  public class Esempio {
(3)   public static void mostraDati() throws Exception {
(4)    Class.forName("...nome driver...");
(5)    Connection conn =
       DriverManager.getConnection("..", "..", "..");
(6)    Statement st = conn.createStatement();
(7)    ResultSet cat = conn.getMetaData().
        getTables(null, null, null, new String[]{"TABLE"});
(8)    while (cat.next()) {
(9)     String nomeTab = cat.getString(3);
(10)    System.out.println("\n\nNome Tabella: "+nomeTab);
(11)    ResultSet rs = st.
            executeQuery("select * from " + nomeTab);
(12)   ResultSetMetaData md = rs.getMetaData();
(13)   while (rs.next()) {
(14)    System.out.println(" Riga:");
(15)    for(int i=0; i< md.getColumnCount(); i++) {
(16)     System.out.println("  Nome Colonna:
                                "+ md.getColumnLabel(i+1)+ ", ");
(17)     System.out.println(" Tipo Colonna: "
                              + md.getColumnTypeName(i+1)+ ": ");
(18)     Object val = rs.getObject(i + 1);
(19)     System.out.println("  Valore: "+val+"\n");
(20)     }
(21)    }
(22)   }
(23)  }
(24) }
```

Figura 10.9 Una classe Java che stampa il contenuto della base di dati.

usabili per apporre un'intestazione alla stampa delle righe estratte dalle tabelle (linea (18-19)).

10.3 Il controllo delle transazioni nelle applicazioni

Nel Paragrafo 5.6 abbiamo definito il concetto di transazione e le proprietà associate, cosiddette *acide*. Nel Capitolo 12 analizzeremo approfonditamente da un punto di vista tecnologico gli accorgimenti che il DBMS impiega per realizzare il controllo della concorrenza e della affidabilità, al fine di garantire il comportamento corretto delle transazioni e le proprietà acide.

In questo paragrafo ci occupiamo invece di questi argomenti dal punto di vista delle applicazioni e presentiamo alcune importanti considerazioni che lo sviluppatore deve fare quando intende realizzare applicazioni che operano su dati condivisi in modo concorrente. Come vedremo nel Capitolo 12, lo scopo del controllo di concorrenza è evitare l'insorgenza di vari tipi di anomalie (per esempio la perdita di un aggiornamento a causa di due scritture di transazioni diverse non opportunamente coordinate)

e le tecniche utilizzate bloccano gli accessi o annullano operazioni, per prevenire o rimuovere violazioni delle anomalie. La presenza di questi meccanismi non esime però del tutto lo sviluppatore dalla necessità di progettare in modo adeguato l'uso delle transazioni e di preoccuparsi delle implicazioni dell'accesso concorrente ai dati. Infatti, mirando esclusivamente alla correttezza e alla semplicità del singolo programma, si possono generare situazioni estremamente inefficienti e quindi inaccettabili. Ciò è particolarmente significativo per le applicazioni interattive, che costituiscono una parte molto rilevante dei sistemi informativi e la totalità delle applicazioni con interfaccia Web. Un'applicazione interattiva di aggiornamento dei dati, infatti, procede spesso secondo uno schema ben definito che può essere sintetizzato dicendo che si ha prima una lettura, poi una richiesta all'utente e infine una scrittura. Dovendo essere le tre operazioni correlate, si può pensare all'utilizzo di una transazione, con la struttura seguente:

```
begin transaction;
accedi in lettura ai dati (select);
presenta i dati all'utente nell'interfaccia di modifica;
IF l'utente conferma le modifiche:
    esegui la modifica (update/delete/insert)
    IF si producono violazioni di vincoli
         rollback transaction;
    ELSE commit transaction;
ELSE rollback transaction;
```

L'esempio, anche se semplificato, mostra chiaramente le ragioni per cui non è possibile eseguire un aggiornamento interattivo dei dati mantenendo la proprietà acida dell'isolamento. Per rendere la transazione isolata dalle altre, dovrebbe essere impedito ad altre transazioni di modificare i dati letti.[4] Ma, poichè l'applicazione è interattiva, non si ha alcuna garanzia sui tempi che l'utente impiegherà per decidere se confermare le proprie modifiche o addirittura annullarle e non procedere oltre. Per tutto questo tempo il DBMS dovrebbe impedire l'accesso ai dati coinvolti, con grave danno per le prestazioni complessive del sistema.

Una tecnica di progettazione molto utilizzata in situazioni in cui è lecito aspettarsi un grado di accesso concorrente non troppo spinto ai dati è quella cosiddetta *ottimistica*. Si tratta di un modo di organizzare l'accesso ai dati di un'applicazione che parte dal presupposto, appunto "ottimistico", che l'insorgenza di conflitti sia un evento piuttosto raro, molto meno frequente del completamento regolare delle operazioni di aggiornamento dei dati. Sotto queste ipotesi, risulta conveniente strutturare le applicazioni interattive separando le varie parti in transazioni diverse; una iniziale di *lettura e copia* dei dati di rilievo, che costituisce una transazione a sé, una centrale di modifica da parte dell'utente *della copia locale dei dati*, quindi senza accesso alla base di dati, e una finale di *aggiornamento della base* di dati con i dati modificati dall'applicazione, che costituisce un'ulteriore transazione, separata dalla prima. Questo

[4] Il *locking*, cioè il blocco dei dati, è una delle tecniche utilizzate per il controllo di concorrenza, cioè per garantire l'isolamento. Come vedremo nel Capitolo 12, ce ne sono altre, che presentano comunque problemi paragonabili.

schema evita il blocco dei dati per un periodo di estensione indefinita corrispondente all'interazione dell'utente, ma richiede cautela nella fase di aggiornamento, per evitare che modifiche apportate alla copia locale dei dati vadano in conflitto con modifiche sugli stessi dati effettuate in maniera concorrente da altre transazioni. Il controllo di congruenza tra la copia dei dati locale e i dati persistenti nella base di dati può essere oneroso, ma può essere semplificato aggiungendo alle relazioni un campo che esprime in modo esplicito la versione o l'istante di ultimo aggiornamento di ogni tupla, in modo che il controllo di congruenza possa svolgersi semplicemente su tale campo.

Con l'accorgimento dell'attributo esplicito di versione, l'approccio ottimistico può essere schematizzato nella struttura dell'applicazione interattiva seguente:

```
begin transaction;
   accedi in lettura ai dati (select);
   crea una copia locale dei dati (cache);
commit transaction;
memorizza la versione corrente di ogni tupla letta;
presenta la copia dei dati all'utente per la modifica;
IF l'utente conferma le modifiche:
    begin transaction;
    rileggi l'attributo versione delle tuple modificate;
    IF versione locale = versione della base di dati
      esegui la modifica (update/delete)
      aggiorna il valore dell'attributo versione
      IF si producono violazioni di vincoli
           rollback transaction;
      ELSE commit transaction;
    ELSE rollback transaction; // anomalia rilevata
// se l'utente rinuncia alle modifiche non fare niente
```

La tecnica ottimistica consiste quindi nella creazione di una cache locale dei dati da modificare e nella separazione tra l'aggiornamento interattivo da parte dell'utente e il riallineamento con la base di dati. In fase di riallineamento si verifica preventivamente l'insorgenza di possibili incoerenze controllando che le tuple modificate dall'utente esistano ancora e che nessun'altra transazione le abbia cambiate in modo concorrente (una tale modifica sarebbe riscontrabile dall'incremento del numero di versione).

Questa tecnica non è esente da problemi: si basa sul presupposto che il grado di conflitto tra transazioni concorrenti sia moderato (altrimenti troppe transazioni verrebbero annullate) e che tutte le applicazioni che modificano i dati attualizzino correttamente il valore dell'attributo che rappresenta la versione. Essa risulta poi onerosa, per la necessità di introdurre nel programma il controllo sulla versione. Quest'ultimo problema è in effetti alleviato da sistemi che automatizzano la gestione dell'attributo di versione e la verifica di congruenza tra la copia locale delle tuple e quella nella base di dati. Uno di questi sistemi è Java Persistence API, che vedremo nel Paragrafo 10.5.

Un'altra leva nelle mani del programmatore è costituita dai cosiddetti livelli di isolamento, cioè dalla possibilità di specificare una riduzione controllata della protezione dalle anomalie, che può in alcuni casi risultare utile. L'approfondimento di questo argomento richiede però la comprensione di una serie di concetti relativi al

controllo di concorrenza e lo rimandiamo quindi al capitolo sull'implementazione delle transazioni e in particolare al Paragrafo 12.2.3.

10.3.1 Il controllo della concorrenza e delle transazioni in JDBC

Nello standard JDBC un'applicazione interagisce con il DBMS tramite un oggetto di tipo *Connection*. Gli esempi mostrati nel Paragrafo 10.2.2 non contengono istruzioni esplicite per il controllo delle transazioni. Questo è dovuto al fatto che per default una connessione JDBC opera secondo la modalità di *autocommit*, già vista nel Paragrafo 10.1 per SQL Embedded. La politica di autocommit prevede che ogni comando SQL sia trattato come una transazione a sé stante: al completamento dell'esecuzione viene eseguito il commit automaticamente. Questa politica non permette quindi di raggruppare più istruzioni SQL in un'unica transazione.

JDBC offre primitive per il controllo esplicito delle transazioni, esposte come metodi dell'oggetto di tipo *Connection*. Si noti che l'inizio di una transazione non è definito dall'applicazione; normalmente è il driver JDBC, o la fonte dati stessa, che decide quando avviare una transazione.

Per operare in modo esplicito con le transazioni è necessario disabilitare la modalità di default di autocommit della connessione, tramite il metodo `setAutoCommit()`, che accetta come parametro di ingresso un valore booleano (`false` indica che autocommit è disabilitato):

```
con.setAutoCommit(false);
```

Il metodo duale, `getAutoCommit()` permette di verificare lo stato corrente dell'opzione autocommit di una connessione.

Il controllo esplicito delle transazioni consente di raggruppare più istruzioni da eseguire in modo transazionale, secondo lo schema rappresentato nel codice seguente:

```
try{

// Disabilita autocommit
conn.setAutoCommit(false);
Statement stmt = conn.createStatement();

// Leggi e aggiorna i dati
// ... istruzioni di lettura e aggiornamento
// ...
if ("ci sono le condizioni per il commit")
   conn.commit(); // richiedi il commit
else
   conn.rollback();  // richiedi il rollback
conn.close();
}
catch (SQLException e){
  ... // gestisci l'eccezione
}
```

La richiesta di terminazione della transazione avviene con due metodi appositi dell'oggetto di tipo *Connection*: `commit()` e `rollback()`. Il metodo `close()` rilascia invece la connessione.

JDBC permette anche di specificare e verificare il livello di isolamento delle transazioni associate alla connessione (concetto che, come anticipato sopra, vedremo nel Paragrafo 12.2.3).

In conclusione, lo sviluppo di applicazioni con CLI si avvale normalmente della modalità di gestione della concorrenza e delle transazioni adottate di default dalla connessione con la base di dati, la quale a sua volta dipende dalla modalità di funzionamento di default del gestore delle transazioni del DBMS. Il programmatore deve però strutturare il codice applicativo in modo che le operazioni di lettura e aggiornamento dei dati svolte in maniera interattiva impegnino il meno possibile la connessione con la base di dati.

La trattazione del controllo della concorrenza e delle transazioni dal punto di vista delle applicazioni non si esaurisce con gli argomenti svolti in questo paragrafo. Nei moderni sistemi informativi, specialmente quelli che utilizzano il Web come mezzo per realizzare l'interfaccia utente, l'architettura dell'applicazione è distribuita su più macchine che svolgono ruoli diversi. La gestione delle transazioni in un ambiente distribuito, sia per quanto riguarda il coordinamento di più sorgenti di dati sia per quanto riguarda la distribuzione dell'applicazione interattiva su più macchine, è un argomento avanzato che esula dai contenuti di questo volume.

10.4 Mappatura relazionale degli oggetti e sistemi ORM

Le soluzioni per la connessione delle applicazioni alle basi di dati discusse nei paragrafi precedenti, sia per il mondo Microsoft Windows sia per il mondo Java, sono basate sull'idea di trasferire dati dagli oggetti temporanei dell'applicazione alla fonte dati persistente, e viceversa. I vari approcci variano per la minore o maggiore facilità con cui l'operazione di trasferimento dei dati viene espressa nel programma, per il grado di indipendenza dall'ambiente di esecuzione dell'applicazione e dal DBMS, per la necessità o meno di una fase di preprocessamento del codice. Tutte le soluzioni però considerano gli oggetti applicativi come distinti dalle tuple della base di dati, anche se in realtà molto spesso vi è una totale identità tra l'oggetto del mondo reale (per esempio, un impiegato) rappresentato nella base di dati e nel programma applicativo.

Sorge quindi spontanea la domanda se non vi sia un modo per rendere più semplice, e in qualche modo automatica, la corrispondenza tra gli oggetti gestiti nella memoria centrale dall'applicazione e le tuple relazionali gestite dal DBMS che ne rappresentano la versione persistente.

Questo problema ha delle implicazioni pratiche molto rilevanti. Come si può intuire anche dai pochi esempi di codice applicativo riportati in questo capitolo, l'allineamento tra oggetti e tuple relazionali occupa una gran parte del codice applicativo; si tratta perlopiù di un codice estremamente ripetitivo (operazioni sempre uguali, a meno dello schema dei dati, di copia dei contenuti dal DBMS alle strutture dati del programma, e viceversa) e di difficile manutenzione, poiché ogni cambiamento nello schema della base di dati comporta la necessità di cambiare il codice applicativo in

tutti i punti dove interviene un trasferimento di dati che coinvolge, per esempio, una relazione modificata.

Una soluzione ideale dovrebbe permettere al programmatore applicativo di:

- reperire nella base di dati le informazioni necessarie all'applicazione, in base a un criterio arbitrario di selezione, e automatizzare la creazione di oggetti e collezioni di oggetti del tipo appropriato a partire dalle tuple estratte;
- trasformare automaticamente nuovi oggetti creati dall'applicazione in tuple corrispondenti nella base di dati;
- riflettere in modo automatico le modifiche agli oggetti applicativi in modifiche alle tuple corrispondenti, e viceversa;
- rispecchiare in modo automatico la cancellazione di oggetti applicativi nella cancellazione delle tuple corrispondenti, e viceversa;
- realizzare gli automatismi di cui sopra in modo efficiente, cioè occupando la connessione alla base di dati per il minimo tempo indispensabile, e senza rinunciare a sfruttare tutte le molteplici opzioni che un DBMS offre per controllare le prestazioni e la sicurezza, quali i meccanismi di gestione delle transazioni concorrenti e il controllo degli accessi.

Come già accennato nel Paragrafo 10.1, l'allineamento tra le strutture dati di un programma e le tuple relazionali corrispondenti non è un problema semplice, a causa del *conflitto di impedenza*, che continua a sussistere anche quando il linguaggio di programmazione è a oggetti, e quindi apparentemente più vicino a un modello logico.

A ben guardare, però, il modo in cui gli oggetti vengono gestiti in un linguaggio di programmazione è molto diverso da quello in cui le tuple vengono trattate in un sistema relazionale.

- L'accesso agli oggetti in un linguaggio di programmazione avviene un oggetto per volta, mediante riferimenti realizzati con dei puntatori. L'accesso alle tuple di una relazione avviene invece un insieme alla volta (per esempio, tutte quelle che vengono restituite come risultato di una interrogazione) e non c'è una nozione di indirizzo di basso livello di una tupla che corrisponda al concetto di puntatore.
- La correlazione tra oggetti avviene mediante la navigazione di riferimenti tra oggetti, sfruttando l'indirizzo fisico dell'oggetto. In SQL e nel modello relazionale, il concetto di navigazione tra oggetti correlati è sostituito da quello di join, basato sull'uguaglianza di valori.

Un buon meccanismo di supporto alla mappatura automatica tra oggetti e tuple dovrebbe riconciliare queste differenze e mascherare al programmatore le modalità differenti di gestione degli oggetti e delle tuple.

I sistemi ORM La mappatura degli oggetti di un programma nelle tuple di un sistema relazionale viene affrontata dai cosiddetti sistemi mappatura relazionale degli oggetti (in inglese, *Object Relational Mapping – ORM*).

Il termine ORM abbraccia una vasta gamma di soluzioni, che vanno da semplici tecniche di buona programmazione applicabili dal progettista in qualsiasi linguaggio di programmazione a oggetti, a standard industriali e prodotti software open source

e commerciali. Tutte queste soluzioni hanno in comune l'obiettivo di ridurre la difficoltà della gestione di dati persistenti all'interno dei programmi applicativi a oggetti, risolvendo le questioni legate alle differenze tra il modello relazionale e il modello a oggetti.

Secondo questa interpretazione allargata del termine, anche le tecniche descritte in precedenza in questo capitolo ricadono nel concetto di ORM, in quanto possono essere utilizzate, anche se con un certo sforzo di programmazione, per mappare gli oggetti di un programma nelle tabelle relazionali corrispondenti.

Nel seguito però ci occuperemo di sistemi ORM che forniscono un supporto più completo al programmatore e automatizzano la corrispondenza tra oggetti temporanei e tuple persistenti in un grado maggiore.

L'esempio su cui ci soffermeremo per illustrare le caratteristiche della mappatura relazionale degli oggetti è Java Persistence API, lo standard più recente emerso dal mondo Java. Tuttavia, altre proposte e sistemi affrontano il medesimo problema, tra cui vale la pena citare almeno ADO.NET Entity Framework, un framework open source per la mappatura relazionale degli oggetti basato sull'architettura ADO.NET descritta nel Paragrafo 10.2.1, e Hibernate, una libreria molto utilizzata che estende il linguaggio Java per consentire la mappatura relazionale degli oggetti, fonte di ispirazione per varie altre proposte, tra cui Enterprise Java Beans (EJB) 3 e la stessa libreria JPA, descritta nel resto del capitolo.

10.5 Java Persistence API (JPA)

Java Persistence API (JPA) è la proposta del mondo Java per la mappatura relazionale degli oggetti. La specifica, codificata come uno dei tanti moduli standard dell'architettura Java, tratta vari aspetti:

- un modello logico delle fonti di dati relazionali aderente al modello a oggetti di Java;
- le tecniche linguistiche per definire la mappatura delle classi Java nelle tabelle relazionali corrispondenti;
- un linguaggio di interrogazione compatibile con la sintassi di Java;
- le interfacce di programmazione (API) per la comunicazione tra applicazioni e basi di dati, per l'esecuzione di interrogazioni e la gestione dei dati.

JPA è frutto di un lungo processo di elaborazione che consolida esperienze precedenti, tra cui la progettazione dello standard Enterprise Java Beans (EJB), un'architettura per la costruzione di applicazioni aziendali a oggetti a alta scalabilità, e Hibernate, uno dei principali software open source per la realizzazione di ORM.

10.5.1 Mappatura tra classi e tabelle

JPA offre servizi per la persistenza degli oggetti come un'estensione naturale delle classi del linguaggio Java.

La mappatura tra le classi Java e le tabelle relazionali avviene tramite l'arricchimento del codice mediante *metadati Java*, un meccanismo linguistico previsto a

partire dalla versione 5 del linguaggio che consente di inserire nel sorgente del programma annotazioni che estendono il significato dei normali costrutti a oggetti. In alternativa all'inserimento dei metadati di mappatura nel codice Java, il progettista può esprimere le stesse informazioni in un file XML di configurazione. Nel seguito di questo paragrafo, per semplicità gli esempi mostreranno la mappatura relazionale degli oggetti tramite metadati inseriti nel codice Java.

Il concetto centrale della mappatura è quello di entità. Un'entità JPA rappresenta una tabella della base di dati e un oggetto istanza della classe rappresenta una riga della tabella. La Figura 10.10 mostra un primo frammento di programma Java contenente la definizione di una classe entità: *Dipartimento*.

Come si vede nell'esempio della Figura 10.10, per utilizzare JPA è necessario importare nel sorgente le librerie del modulo `javax.persistence` (linee 2-5), e nel caso che si preveda di trasmettere oggetti a un programma remoto, anche la libreria di sistema `java.io.Serializable` che consente alla classe entità di implementare l'interfaccia `Serializable` (linea 1 e 9).

Un'entità è preceduta dall'annotazione `@Entity`, che mostra un esempio della sintassi dei metadati Java, secondo cui ogni annotazione deve essere introdotta dal carattere speciale `@`. L'annotazione successiva crea la corrispondenza tra la classe `Dipartimento` e la tabella DEPT della base di dati. Lo stato dell'entità è rappresentato da dati membri privati, accessibili con i metodi di accesso (*setter*) e modifica

```
(1)  import java.io.Serializable;
(2)  import java.persistence.Entity;
(3)  import java.persistence.Table;
(4)  import java.persistence.Column;
(5)  import java.persistence.Id;
(6)
(7)  @Entity
(8)  @Table (name="Dept")
(9)  public class Dipartimento implements Serializable {
(10)   @Id
(11)   @Column(name="id", nullable=false)
(12)   private String IdDip;
(13)   @Column(name="name")
(14)   private String NomeDip;
(15)   public void setIdDip(String Iden) {
(16)     this.IdDip = Iden;
(17)   }
(18)   public String getIdDip() {
(19)     return this.IdDip;
(20)   }
(21)   ...
(22) }
```

Figura 10.10 Un programma Java con una entità JPA .

(*getter*) tipici delle classi Java. Opzionalmente, un dato membro può essere preceduto dall'annotazione `@Column` che permette di specificare il nome della colonna della tabella corrispondente al dato membro della classe, nel caso vi sia differenza. Un dato membro può anche essere annotato con `@Id`, il che equivale a dichiarare che tale attributo corrisponde alla chiave primaria degli oggetti istanze dell'entità. L'annotazione `@Id` consente di specificare anche che il valore non può assumere valori nulli, in modo da permettere la verifica che gli oggetti Java, automaticamente allineati con le tuple relazionali corrispondenti, rispettino il vincolo di presenza di valori della chiave. Con una sintassi lievemente più complessa, che omettiamo per brevità, è possibile dichiarare la corrispondenza di più dati membri di una classe con la chiave composta di una tabella.

Una classe Java può corrispondere a una tabella relazionale anche solo parzialmente. In questo caso è possibile apporre l'annotazione `@transient` ai dati membri della classe Java che non trovano corrispondenza con colonne della relazione.

Mappatura delle relazioni Una delle caratteristiche più potenti di JPA è la mappatura delle relazioni tra classi in corrispondenze tra tabelle relazionali basate sull'uso dei vincoli di integrità referenziale. La mappatura riguarda le associazioni tra due classi (cioè binarie) e richiede di specificare tre aspetti: la cardinalità, la navigabilità e le politiche di gestione.

Il primo aspetto è assimilabile a quello della cardinalità delle relazioni nel modello Entità-Relazione (si veda il Capitolo 6): stabilire il numero massimo (ed eventualmente minimo) di oggetti di una classe con cui un certo oggetto dell'altra classe è correlabile.

Il secondo aspetto è invece tipico di ORM: individua una classe come proprietaria (*owner*) dell'associazione e stabilisce se la relazione tra gli oggetti Java è rappresentata solo negli oggetti della classe proprietaria o anche negli oggetti dell'altra classe coinvolta nell'associazione. In altre parole, la navigabilità stabilisce se nel programma Java è possibile navigare solo la relazione *diretta* tra una classe A e una classe B oppure anche la relazione *inversa* tra la classe B e la classe A. La classe proprietaria è anche quella che contiene l'attributo che implementa la relazione, la cui modifica determina anche l'aggiornamento della relazione nella base di dati.

Infine, le politiche di gestione dettano le regole per la propagazione delle operazioni di modifica e di trasferimento da base di dati a memoria centrale delle istanze associate a un oggetto.

La cardinalità delle relazioni si esprime con le annotazioni seguenti:

- `@OneToOne`: si usa sia nella relazione diretta sia in quella inversa per denotare un'associazione uno a uno tra oggetti di due classi;
- `@ManyToOne` e `@OneToMany`: si usano rispettivamente nella relazione diretta e in quella inversa per denotare un'associazione uno a molti tra oggetti di due classi;
- `@ManyToMany`: si usa sia nella relazione diretta sia in quella inversa per denotare un'associazione molti a molti tra oggetti di due classi;

La Figura 10.11 mostra un esempio di mappatura di una relazione uno a molti tra la classe `Corso` (definita come owner) e la classe `Dipartimento`.

```
(1)  @Entity
(2)  public class Corso implements Serializable {
(3)    private Dipartimento dipartimento;
(4)    @ManyToOne
(5)    @JoinColumn(name=IdDip)
(6)    public Dipartimento getDipartimento() {
(7)       return Dipartimento;
(8)    public void setDipartimento (Dipartimento dip){
(9)       this.dipartimento = dip;
(10)   }
(11)  ...
(12) }
(13)
(14) @Entity
(15) @Table (name="Dept")
(16) public class Dipartimento implements Serializable {
(17) ...
(18)   private Collection<Corso> corsi
(19)   @OneToMany(mappedBy="dipartimento",
            fetch=FetchType.EAGER)
(20)   public Collection<Corso> getCorsi() {
(21)     return corsi;
(22)   }
(23)   public void setCorsi(Collection<Corso> corsi) {
(24)     this.corsi = corsi;
(25)   }
(26) }
```

Figura 10.11 Un programma Java con entità e relazioni JPA.

La classe `Corso`, mappata per default sulla tabella omonima, rappresenta la proprietaria della relazione: essa contiene un dato membro (**dipartimento**) che implementa la relazione molti a uno tra i corsi e il dipartimento che li eroga, come specificato dall'annotazione `@ManyToOne` in riga (4). L'attributo **dipartimento** è mappato in riga (5) sulla chiave **IdDip** della tabella DEPT, in corrispondenza del vincolo di integrità referenziale tra la tabella CORSO e la tabella DEPT. La relazione è navigabile nei due sensi: infatti la classe `Dipartimento` contiene un attributo **corsi** di tipo `Java.Collection` (righe (18-19))che implementa la relazione inversa. Si noti come la dichiarazione della relazione inversa faccia uso dell'annotazione duale `@OneToMany` che contiene un attributo `mappedBy` che specifica l'attributo della classe proprietaria che implementa la relazione inversa (in questo caso, **dipartimento**). La specifica della relazione inversa tramite `mappedBy`, nel caso delle relazioni uno a molti, va inserita nella classe i cui oggetti partecipano alla relazione con molteplicità multipla e quindi `mappedBy` non può apparire nell'annotazione `ManyToOne`.

L'esempio mostra in riga (19) anche l'uso di una delle politiche esprimibili sulla relazione: `fetch`. Con tale attributo si specifica come deve comportarsi il sistema

quando carica in memoria centrale un oggetto estratto dalla base di dati: il valore `LAZY` della politica indica che gli oggetti correlati devono essere caricati solo quando servono; il valore `EAGER` invece indica che gli oggetti correlati devono essere caricati subito.

JPA supporta la dichiarazione di altre politiche di gestione delle relazioni, tra cui le principali sono le seguenti.

- *Reazione alla violazione del vincolo di integrità referenziale*: nelle relazioni `@OneToOne` e `@OneToMany` è possibile inserire la clausola `cascadeType =REMOVE` che fa sì che la cancellazione di un oggetto da parte del programma Java comporti la cancellazione a cascata degli oggetti associati nella relazione.
- *Persistenza*: l'inserimento della clausola `cascadeType=PERSIST` nelle relazioni `@OneToOne` e `@OneToMany` fa sì che l'invocazione di un'operazione di memorizzazione nella base di dati di un oggetto da parte del programma Java comporti anche la memorizzazione persistente a cascata degli oggetti associati nella relazione.
- *Gestione del distacco degli oggetti dalla persistenza automatica*: come vedremo nel Paragrafo 10.5.2, JPA permette di interrompere in frangenti particolari l'automatismo con cui gli oggetti Java sono sincronizzati con le tuple relazionali corrispondenti. L'inserimento delle clausole `cascadeType=DETACH` oppure `MERGE` oppure `REFRESH` nelle relazioni `@OneToOne` e `@OneToMany` fa sì che l'applicazione di un'operazione omonima da parte del programma Java a un oggetto comporti l'applicazione della stessa operazione anche agli oggetti associati nella relazione.
- L'uso di una clausola `cascadeType=ALL` equivale all'applicazione di tutte le politiche di propagazione sopra descritte.

Mappatura delle gerarchie La mappatura relazionale degli oggetti si applica anche nel caso in cui le classi del programma Java formino una gerarchia di generalizzazione. È possibile annotare come entità classi Java concrete e astratte, e un'entità può estendere un'altra entità oppure una classe non dichiarata come tale. JPA consente anche di eseguire *interrogazioni polimorfiche*, cioè interrogazioni che si riferiscono a una classe genitore, anche astratta, e possono produrre risultati contenenti istanze di diverse entità figlie.

Come discusso più ampiamente nel capitolo sulla progettazione logica delle basi di dati, dato che i sistemi relazionali tradizionali non consentono di rappresentare direttamente le gerarchie di generalizzazione, risulta necessario mappare questo costrutto in altri costrutti del modello relazionale per i quali esiste invece una implementazione naturale: le relazioni ed eventualmente i cammini di join che collegano le tabelle tramite i vincoli di integrità referenziale.

La mappatura può assumere tre forme diverse, equivalenti alle modalità di eliminazione delle gerarchie discusse nel Paragrafo 8.3.2:

(1) **Accorpamento delle classi figlie della generalizzazione nella classe genitore.** Questa strategia, chiamata *Single Table per Class Hierarchy* nella terminologia JPA, prevede di mappare le sotto-classi C_1 e C_2 e la super-classe genitore C_0 in una sola tabella relazionale che contiene colonne corrispondenti a tutti gli attributi di C_0, C_1 e C_2, più una colonna (chiamata *discriminator column*) che distingue il "tipo" di una istanza di C_0, cioè se tale occorrenza appartiene a C_1, a C_2 o a C_0, nel caso in cui questa sia una classe concreta.

(2) **Accorpamento del genitore della generalizzazione nelle figlie**. Questa strategia, chiamata *Table per Concrete Class* in JPA, prevede una tabella relazionale per ciascuna classe figlia C_1 e C_2 comprendente sia colonne corrispondenti agli attributi locali di C_1 e C_2 sia colonne corrispondenti agli attributi della classe genitore C_0.

(3) **Sostituzione della generalizzazione con associazioni**. Questa strategia, chiamata *Joined Tables* in JPA, mappa la generalizzazione in associazioni uno a uno che legano la tabella corrispondente alla classe genitore C_0 con le tabelle corrispondenti alle classi figlie C_1 e C_2, le quali sono identificate esternamente mediante la chiave primaria della tabella associata a C_0 e possono contenere una colonna che discrimina il "tipo" delle tuple.

La Figura 10.12 mostra un esempio di classi Java in gerarchia, mappate secondo la strategia `JOINED`.

La gerarchia di generalizzazione contiene una classe astratta `Professore`, specializzata in due classi concrete `ProfessoreRuolo` e `ProfessoreContratto`.

```
(1)  @Entity
(2)  @Table(name="PROF")
(3)  @Inheritance(strategy=JOINED)
(4)  @DiscriminatorColumn(name="RUOLO")
(5)  public abstract class Professore {
(6)    @Id protected Integer idProf;
(7)    protected string settore;
(8)    @ManyToOne protected Dipartimento dip;
(9)  }
(10)
(11) @Entity
(12) @Table(name="PROF_RUOLO")
(13) @DiscriminatorValue("PR")
(14) @PrimaryKeyJoinColumn(name="PR_IDPROF")
(15) public class ProfessoreRuolo extends Professore {
(16)  // eredita idProf mappato su PROF_RUOLO.PR_IDPROF
(17)  // eredita settore mappato su PROF.SETTORE
(18)  // eredita dip mappato su PROF.DIP
(19)  // mappato, per default, su PROF_RUOLO.SALARIO
(20)     protected Integer salario;
(21) }
(22) @Entity
(23) @Table(name="PROF_CONTRATTO")
(24) @DiscriminatorValue("PC")
(25) // chiave prim., per default, PROF_CONTRATTO.IDPROF
(26) public class ProfessoreContratto extends Professore {
(27) // mappato, per default, su PROF_CONTRATTO.ORE
(28)     protected Integer ore;
(29) }
```

Figura 10.12 Un programma Java con entità in gerarchia mappate con strategia `JOINED`.

Il tipo delle istanze delle sottoclassi Java è rappresentato nella base di dati relazionale dal valore della colonna discriminatrice ruolo, che può essere `PR` o `PC`. Le tabelle corrispondenti alle sottoclassi includono una colonna che rappresenta la chiave primaria dell'oggetto della superclasse `Professore`: tale colonna viene ridenominata esplicitamente nella tabella corrispondente alla classe `ProfessoreRuolo` e denominata per default nella tabella corrispondente alla classe `ProfessoreContratto`.

In generale, JPA offre molte regole di denominazione di default delle tabelle e delle colonne, che evitano al progettista di specificare esplicitamente i nomi dello schema relazionale nelle annotazioni del codice Java.

10.5.2 Architettura e utilizzo di JPA

L'accesso alle fonti dati da parte di un programma Java che usa JPA avviene attraverso un'interfaccia di programmazione (API), di cui la Figura 10.13 mostra uno schema di alto livello.

Un'applicazione interagisce con una fonte dati attraverso un'istanza di *Entity Manager*, un oggetto che rappresenta una sessione di connessione con la base di dati. Nel caso più generale, l'applicazione si procura tale istanza attraverso un componente di sistema, chiamato *EntityManagerFactory*. Tipicamente l'EntityManagerFactory gestisce le connessioni con una base di dati specifica, occupandosi anche dell'efficienza nel caso di connessioni multiple dalla stessa o da più applicazioni. I dati caratteristici della base di dati (per esempio, la specifica del venditore, il nome della base di dati e le credenziali d'accesso) sono poste all'interno di un file di configurazione XML (chiamato `persistence.xml`).

L'interfaccia dell'*Entity Manager* contiene, tra le altre, le operazioni per rendere persistente un oggetto (`persist()`), cancellare un oggetto (`remove()`), estrarre dalla fonte dati gli oggetti in base vari a criteri di selezione (`find()`), sincronizzare le modifiche fatte in memoria centrale con la fonte dati (`flush()`), imporre un lock

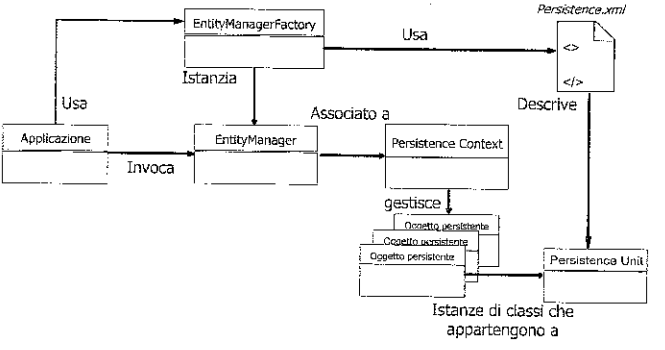

Figura 10.13 L'architettura di accesso alle fonti dati di JPA.

su un'istanza di entità (`lock()`), allineare lo stato di un oggetto in memoria centrale con il contenuto della tupla corrispondente nella base di dati (`refresh()`), e terminare l'interazione (`close()`).

Un *EntityManager* è associato a un contesto di persistenza (*PersistenceContext*), che rappresenta un insieme di entità, cioè oggetti Java associati a tuple relazionali. Ogni EntityManager ha il proprio PersistenceContext, che garantisce che per ogni tupla relazionale esista al più un solo oggetto Java corrispondente con la stessa identità, e svolge il ruolo di memoria cache rispetto al contenuto della base di dati; quando l'applicazione richiede un oggetto già presente nel PersistenceContext, esso viene restituito senza accedere alla base di dati.

Le istanze di entità che appartengono al PersistenceContext hanno un ciclo di vita *gestito* dall'EntityManager (in inglese, si trovano nello stato *managed*). Nel Paragrafo 10.5.2 vedremo cosa si intende per ciclo di vita di un'istanza di entità e in quali stati essa si possa trovare.

Le classi degli oggetti del PersistenceContext costituiscono una *Persistence Unit*: si può fare un parallelo tra questi concetti e quelli relazionali: il PersistenceContext rappresenta l'analogo in Java dell'istanza della base di dati, mentre la Persistence Unit rappresenta lo schema. Le classi che compongono la Persistence Unit sono dichiarate nel file di configurazione `persistence.xml`. La Figura 10.14 mostra un esempio di tale file di configurazione, che specifica: il nome della Persistence Unit (`esempio_pu`), una descrizione, il fornitore della base di dati, il nome di un ulteriore file di configurazione XML che contiene eventuali specifiche di mappatura

```
<persistence>
 <persistence-unit name="esempio_pu">
  <description>Esempio di Persistence Unit</description>
   <provider>
     oracle.toplink.essentials.PersistenceProvider
   </provider>
   <mapping-file>META-INF/mappingFile.xml</mapping-file>
   <jar-file>esempioClassi.jar</jar-file>
   <class>esempio.Dipartimento</class>
   <class>esempio.Corsi</class>
  <properties>
    <property name="toplink.logging.level" value="INFO"/>
    <property name="toplink.jdbc.driver"
              value="oracle.jdbc.OracleDriver"/>
    <property name="toplink.jdbc.url"
              value="jdbc:oracle:thin:@myhost:1521:MYSID"/>
    <property name="toplink.jdbc.password" value="pwd"/>
    <property name="toplink.jdbc.user" value="admin"/>
  </properties>
 </persistence-unit>
</persistence>
```

Figura 10.14 Un esempio di persistence.xml.

relazionale degli oggetti aggiuntive alle annotazioni nelle classi Java, il nome dell'archivio Java che contiene le classi annotate come Entity, la dichiarazione delle entità gestite dalla Persistence Unit e un elenco di proprietà ulteriori che dipendono dal particolare DBMS, nel caso presente i parametri di connessione JDBC per una sorgente dati Oracle.

Un'applicazione che vuole accedere a una fonte dati con JPA deve, come prima cosa, ottenere un oggetto EntityManager e poi utilizzare i metodi della sua interfaccia per operare. Il codice riportato di seguito illustra come un'applicazione Java standard possa creare a partire dall'interfaccia di sistema `Persistence` un EntityManagerFactory (identificata dalla variabile `emf`) e poi ottenere da questa un'istanza di EntityManager (identificata dalla variabile `em`).

```
EntityManagerFactory emf =
   javax.persistence.Persistence.
      createEntityManagerFactory("esempio_pu");
EntityManager em = emf.createEntityManager();
```

L'esempio successivo nella Figura 10.15 mostra invece l'interazione con l'EntityManager nel caso di un oggetto Java di tipo *Enterprise Java Bean* (*EJB*), il quale viene gestito da un ambiente speciale (detto *EJB container*) che fornisce molti servizi automatici, tra cui l'inizializzazione di variabili che si riferiscono a risorse esterne, come nell'esempio l'EntityManager. In questo caso, il reperimento dell'EntityManager avviene semplicemente grazie all'annotazione Java `@PersistenceContext` (linea (3)) che garantisce che l'ambiente di esecuzione inizializzi la variabile `em` con un riferimento all'EntityManager.

Dopo il reperimento dell'EntityManager, l'esempio prosegue mostrando il codice del metodo inserisciCorso, che riceve in ingresso un nuovo corso e l'identificativo di un professore, crea un nuovo corso persistente nella base di dati e aggiorna la relazione tra il corso e un professore. Nella linea (5), il metodo `find()` dell'EntityManager restituisce un oggetto di tipo `Professore` corrispondente alla tupla relazionale avente come chiave primaria il valore memorizzato nel parametro forma-

```
(1)  @Stateless
(2)   public class InserisciCorsoBean implements
      InserisciCorso {
(3)   @PersistenceContext EntityManager em;
(4)   public void inserisciCorso(int idProf, Corso
      nuovoCorso) {
(5)     Professore prof = em.find(Professore.class, idProf);
(6)     prof.getCorsi().add(nuovoCorso);
(7)     nuovoCorso.setProfessore(prof);
(8)     em.persist(nuovoCorso);
(9)   }
(10) }
```

Figura 10.15 Un esempio interazione con l'EntityManager.

le idProf. L'attributo corsi, che rappresenta la relazione tra professore e corso, viene poi aggiornato con l'aggiunta del nuovo corso (linea (6)); la linea (7) aggiorna anche la relazione inversa, rappresentata dall'attributo professore di un oggetto di tipo corso. Infine, il metodo `persist()` dell'EntityManager viene invocato per rendere persistente il nuovo corso e le modifiche alla relazione.

L'esempio sottolinea la necessità di mantenere la consistenza in memoria centrale dei riferimenti tra i dati membri che implementano la direzione diretta e inversa di una relazione bidirezionale. Siccome la scrittura sulla base di dati si basa sui riferimenti contenuti nell'oggetto della classe proprietaria della relazione (la classe `Corso`, nell'esempio), le modifiche fatte all'oggetto non proprietario (`prof`, nell'esempio) devono essere sempre riflesse in quello proprietario (`nuovoCorso`), pena la perdita delle modifiche alla relazione.

Ciclo di vita degli oggetti L'utilità principale di JPA risiede nella sua capacità di riflettere automaticamente le modifiche agli oggetti Java in aggiornamenti alle tuple corrispondenti. Questo automatismo, però, vige solo quando l'istanza di entità è nello stato managed, uno dei cinque possibili.

La Figura 10.16 mostra gli stati possibili per le istanze di entità e il modo in cui le operazioni eseguite dall'EntityManager, dietro richiesta dell'applicazione, modificano lo stato.

Quando si crea un oggetto Java che appartiene a una classe di tipo Entity, questa nuova istanza non possiede ancora un'identità persistente e non è associata al PersistenceContext. L'invocazione del metodo `persist()` dell'*EntityManager* porta l'istanza dallo stato new allo stato managed e la associa a un PersistenceContext. Ciò comporta l'esecuzione di un'operazione di insert nella base di dati, al commit della transazione o quando viene invocato il metodo `flush()` dell'*EntityManager*. L'*EntityManager* produce anche l'attivazione ricorsiva del metodo `persist()` per

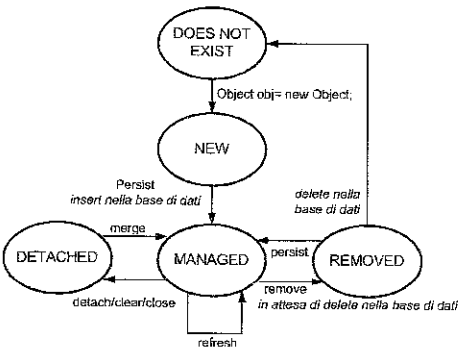

Figura 10.16 Il ciclo di vita di un'istanza di entità in JPA.

tutti gli oggetti correlati mediante relazioni le cui annotazioni includono `cascadeType.PERSIST` oppure `cascadeType.ALL`. Un modo alternativo perché un'istanza entri nello stato managed è che faccia parte del risultato di un'interrogazione. Indipendentemente dal modo con cui sono entrate nello stato managed, per le istanze in questa condizione le modifiche ai dati membri vengono rispecchiate automaticamente nella base di dati.

Per un'entità nello stato managed è anche possibile allineare il valore dei dati membri con lo stato corrente della tupla corrispondente nella base di dati, scartando le eventuali modifiche fatte in memoria centrale; ciò non avviene in modo automatico come per le modifiche sulle entità effettuate in Java, ma su richiesta dell'applicazione, mediante il metodo `refresh()`, che aggiorna sia l'oggetto passato come parametro attuale, sia quelli correlati mediante relazioni con `cascadeType.REFRESH` oppure `cascadeType.ALL`.

Un'istanza nello stato managed può essere eliminata con il metodo `remove()`, che la porta nello stato removed; l'operazione di delete vera e propria nella base di dati viene eseguita al commit della transazione o quando viene invocato il metodo `flush()` dell'*EntityManager*. Il metodo `remove()` viene invocato ricorsivamente per tutti gli oggetti correlati mediante relazioni con `cascadeType.REMOVE` oppure `cascadeType.ALL`.

Un aspetto particolare del ciclo di vita delle istanze di entità è rappresentato dalla possibilità di porle nello stato *detached*. In questo stato un'istanza non è più associata a un PersistenceContext e quindi le modifiche non vengono scritte automaticamente nella base di dati. Tuttavia il suo stato al momento del distacco resta accessibile all'applicazione; più precisamente, restano accessibili:

- i dati membri con politica di fetch diversa da `LAZY`;
- i dati membri acceduti in precedenza dall'applicazione;
- gli oggetti correlati, se sono già stati estratti dalla base di dati in precedenza, a causa di un'interrogazione o di una strategia di fetch `EAGER`.

L'ingresso nello stato detached si produce in vari modi:

- esplicitamente per una specifica istanza, con l'invocazione diretta o a cascata del metodo `detach()`;
- esplicitamente per tutte le istanze del PersistenceContext, con l'invocazione del metodo `clear()` e `close()`;
- implicitamente quando l'oggetto viene serializzato, per esempio per passarlo a un modulo dell'applicazione che risiede su un diverso nodo di rete;
- implicitamente a fronte di eventi transazionali: il rollback della transazione, o anche il commit quando si usa un EntityManager che automatizza la gestione delle transazioni (su questo punto ritorneremo tra poco).

All'operazione di distacco di un'istanza dal PersistenceContext corrisponde un'operazione contraria: (`merge()`), che propaga le modifiche dello stato da un'istanza detached a un'istanza managed. Il metodo funziona *copiando* i valori dei dati membri dell'istanza detached nei dati membri di un'istanza esistente *avente la medesima*

identità, oppure, se una tale istanza non esiste ancora, creandone una nuova per effettuare la copia. Anche l'operazione di `merge()` viene propagata ricorsivamente lungo le relazioni, con annotazione `cascadeType.MERGE`.

Controllo della concorrenza e gestione delle transazioni Uno degli aspetti più complessi della connessione tra applicazioni e basi di dati, dove il supporto di JPA si rivela più utile, è il controllo dell'accesso concorrente ai dati e la gestione delle transazioni.

Per il controllo dell'accesso concorrente ai dati da parte di più applicazioni, JPA offre un meccanismo automatico che facilita la realizzazione della tecnica ottimistica, illustrata nel Paragrafo 10.3. Per abilitare la gestione automatica di questa tecnica, un'entità deve contenere la dichiarazione di un attributo dedicato alla versione, come mostra l'esempio seguente.

```
@Entity
@Table(name="Prof")
  public class Professore {
   @Id protected Integer idProf;
   @Version protected Integer version;
   @ManyToOne protected Dipartimento dip;
   ...
}
```

L'attributo dichiarato come versione viene utilizzato automaticamente dal sistema ad ogni operazione di lettura e scrittura sulle istanze dell'entità; può essere letto, ma non modificato dall'applicazione. Quando un'istanza di entità viene ripristinata nello stato managed tramite un'operazione di `merge()`, il sistema solleva un'eccezione Java se scopre che la copia dell'oggetto che si sta cercando di riallineare con la base di dati non corrisponde più alla tupla associata a causa di qualche modifica eseguita da transazioni concorrenti mentre l'istanza era nello stato detached. In questo modo, il programmatore viene sollevato dall'onere di effettuare manualmente le operazioni di gestione del controllo di versione, che abbiamo visto invece essere necessarie nel caso di accesso alla base di dati con JDBC. Quando sono necessarie politiche di controllo della concorrenza più stringenti, JPA consente comunque la realizzazione di politiche *pessimistiche*, in cui l'applicazione richiede espressamente l'acquisizione delle istanze di entità che devono essere lette o modificate, tramite apposite operazioni di lock che saranno discusse nel capitolo finale.

Per il controllo delle transazioni, JPA offre due soluzioni, a seconda che l'applicazione sia eseguita in ambiente Java standard oppure sfrutti i servizi di automazione delle transazioni offerti dall'ambiente Enterprise Java Beans (EJB).

La prima soluzione (chiamata *transazioni locali alla risorsa*, *resource local transactions*) prevede la demarcazione esplicita della transazione da parte dell'applicazione, attraverso opportuni metodi dell'EntityManager e della classe *EntityTransaction*. Tipicamente, una transazione coincide con la durata della vita utile di un'istanza di EntityManager: la transazione comincia quando si crea l'EntityManager e si conclude con la chiamata del metodo `close()`. Il frammento di codice seguente esemplifica

il modo tipico in cui un'applicazione interagisce con l'EntityManager quando vuole demarcare esplicitamente una transazione:

```
EntityManager em = createEntityManager();
em.getTransaction().begin();
Professore prof = em.find(Professore.class, idProf);
prof.setSalario(prof.getSalario() + 1000);
em.getTransaction().commit();
em.close();
```

Si noti che nel caso in cui la transazione fallisce o viene marcata per il rollback da parte dell'applicazione, solo le modifiche alla base di dati vengono annullate; le istanze di entità lette in precedenza del rollback continuano a esistere in memoria centrale, nello stato detached, mentre il PersistenceContext viene svuotato con il metodo `clear()`. Resta pertanto cura dell'applicazione evitare di riutilizzare oggetti nello stato detached che possono avere un contenuto diverso da quello memorizzato nella base di dati.

La seconda modalità di gestione delle transazioni (chiamata *transazioni JTA*, *JTA transactions*) si appoggia alla specifica Java Transaction API (JTA) ed è usata principalmente nel caso di applicazioni distribuite che fanno uso della tecnologia Enterprise Java Beans. Si tratta di un approccio potente, ma complesso, che richiede nozioni di architetture distribuite e per il Web e perciò esula dai contenuti di questo volume.

10.5.3 Interrogazioni in JPA

Il principale aspetto di JPA approfondito finora è la propagazione automatica delle modifiche, dagli oggetti del programma alle tuple corrispondenti della base di dati. Esiste però un secondo grande vantaggio che distingue JPA dalle tecniche discusse in precedenza, che riguarda la gestione dei risultati delle interrogazioni. Mentre JDBC e le tecniche analoghe basate su ODBC, OLEDB e ADO richiedono la ricopiatura manuale dei risultati di un'interrogazione nelle strutture dati del programma (siano queste semplici variabili, oggetti, o collezioni di oggetti), JPA può sfruttare invece le informazioni della mappatura relazionale degli oggetti per trasformare automaticamente i risultati delle interrogazioni in entità o collezioni di entità. Oltre a questa facilitazione, le informazioni sulla mappatura degli oggetti, e in particolare quelle sulla mappatura delle relazioni, consentono a JPA di supportare uno stile di espressione dei join più vicino alla programmazione a oggetti.

Queste funzionalità sono comprese all'interno di *Java Persistence Query Language* (*JPQL*), il linguaggio di interrogazione e aggiornamento dei dati offerto da JPA. JPQL può essere considerato in due modi: come un'estensione di Java che permette l'interrogazione e modifica orientata agli insiemi delle istanze di entità in memoria centrale, oppure come "una versione potenziata di SQL" che garantisce una migliore integrazione con il linguaggio a oggetti Java, facilmente convertibile in SQL e quindi ottimizzabile da parte del DBMS.

Le caratteristiche principali di JPQL si possono riassumere come segue.

- Le interrogazioni e gli aggiornamenti sono espressi a partire da un modello a oggetti di alto livello (chiamato schema astratto - *abstract schema*), che comprende i concetti visti nel Paragrafo 10.5.1: entità, attributi, relazioni e generalizzazioni.
- Grazie alle gerarchie di generalizzazione, le interrogazioni sono polimorfiche, cioè l'insieme dei risultati può contenere istanze di entità diverse.
- Sono disponibili due tipi di interrogazioni a oggetti, *tipate* e *non tipate*, e anche interrogazioni *native* espresse semplicemente in SQL. La mappatura dei risultati in istanze di entità è fatta dal sistema per le interrogazioni tipate e dal programmatore per quelle native.
- Il linguaggio consente la scrittura di espressioni ed operatori che sfruttano caratteristiche avanzate del modello dei dati, quali la navigazione delle relazioni e l'accesso a collezioni di oggetti indicizzate (mappe).

Dal punto di vista del programmatore, le interrogazioni sono anch'esse oggetti, costruiti attraverso i metodi `EntityManager.createQuery` e `EntityManager.createNamedQuery` dell'EntityManager.

Il metodo `createQuery` serve a costruire interrogazioni dinamiche, come la seguente:

```
public List trovaPerNome(String nome) {
return em.createQuery(
    "select P from Professore P where
         P.Nome like :nomeP")
    .setParameter("nomeP", nome)
    .setMaxResults(10)
    .getResultList();
}
```

I metodi `setParameter`, `setMaxResults` e `getResultList` sono applicati in cascata all'oggetto di tipo `Query` ritornato dal metodo `createQuery`, in perfetto stile a oggetti. Il metodo `setParameter` inizializza il parametro dichiarato nella clausola `WHERE` dell'interrogazione e restituisce di nuovo l'oggetto di tipo Query; su questo, viene invocato il metodo `setMaxResults` che permette una gestione "paginata" dei risultati di un'interrogazione e restituisce di nuovo l'oggetto di tipo Query; infine il metodo `getResultList` produce l'esecuzione dell'interrogazione e restituisce il risultato: siccome si tratta di un'interrogazione non tipata il risultato è semplicemente una collezione di tipo `List` di oggetti della classe Java predefinita *Object*.

Si noti che, con riferimento all'esempio precedente, se al metodo `createQuery` si passa un parametro aggiuntivo che rappresenta la classe del risultato (`Professore.class` nel caso presente), l'EntityManager restituisce un oggetto di tipo `TypedQuery<Professore>`, la cui esecuzione con il metodo `GetResultList` produce un risultato di tipo `List<Professore>` e non semplicemente `List`.

L'esempio mostra anche una differenza rispetto all'uso della clausola SELECT nelle interrogazioni SQL: un'interrogazione di tipo SELECT con JPQL produce come risultato un insieme di oggetti e non un insieme di tuple, come avviene invece in

SQL. Pertanto, a differenza di SQL, l'interrogazione JPQL non contiene una lista di attributi, ma solo la variabile che denota il tipo degli oggetti ritornati (p nel caso dell'esempio sopra riportato).

```
public List<Professore> trovaPerNome(String nome) {
return em.createQuery(
   "select P from Professore P where P.Nome like :nomeP",
   Professore.class)
   .setParameter("nomeP", nome)
   .setMaxResults(10)
   .getResultList();
}
```

La tipizzazione del risultato è possibile anche con le interrogazioni native, e perfino quanto il risultato contiene istanze di più entità, a patto di dichiarare esplicitamente i tipi in cui convertire il risultato. Per far ciò si può usare l'annotazione `@SqlResultSetMapping`, come dimostrato nel seguente esempio.

```
Query q = em.createNativeQuery(
   "select P.IdProf, P.Nome,  D.IdDip, D.Nome,  "+
   "from Professore P, Dipartimento D " +
   "where P.Dipartimento = D.IdDip",
   "RisultatoProfDip");
@SqlResultSetMapping(name="RisultatoProfDip",
   entities={
      @EntityResult(entityClass=Professore.class),
      @EntityResult(entityClass=Dipartimento.class)
})
```

JPQL consente anche di denominare un'interrogazione per poi richiamarla per nome in altri punti del programma:

```
@NamedQuery(
     name="trovaProfPerNome",
     query="select P from Professore P
     where P.Name like :pName"
)
..
risultato = em.createNamedQuery("trovaProfPerNome")
     .setParameter("pName", "Ceri")
     .getResultList();
..
```

Gli esempi sopra riportati usano stringhe costruite dal programma per l'espressione delle interrogazioni dinamiche. JPQL offre una modalità alternativa, più orientata agli oggetti, per lo stesso scopo, tramite un'interfaccia che prende il nome di *Criteria Query API*. Per mostrare la differenza tra le due procedure di costruzione, riformuliamo l'interrogazione `trovaPerNome` con quest'ultima tecnica.

```
public List<Professore> trovaPerNome(String nome) {
  CriteriaBuilder cb = em.getCriteriaBuilder();
  CriteriaQuery<Professore> q =
                    cb.createQuery(Professore.class);
  Root<Professore> prof = q.from(Professore.class);
  q.select(prof);
  ParameterExpression<String> nomeP =
                cb.parameter(String.class, "nomeP");
  q.where(cb.equal(prof.get("nome"), nomeP));
  return q.setParameter("nomeP", nome)
          .setMaxResults(10)
          .getResultList();
}
```

La costruzione di un'interrogazione comincia con il reperimento di un oggetto di tipo `CriteriaBuilder` (`cb`), fornito dal metodo `getCriteriaBuilder` dell'EntityManager. Come suggerisce il nome, CriteriaBuilder serve per costruire tutti gli ingredienti sintattici (termini, espressioni, predicati, clausole di raggruppamento e ordinamento) che possono intervenire in un'interrogazione. Il primo passo prevede la creazione dell'interrogazione, che richiede la specifica della/e entità che formano il risultato (nel nostro caso, `Professore.class`). In seguito, l'interrogazione viene assemblata "pezzo per pezzo" mediante invocazioni successive di metodi opportuni: il metodo `from` costruisce la struttura dati del risultato; nella terminologia JPQL questa viene detta la radice (*root*, in inglese) dell'interrogazione, perchè contiene gli oggetti di più alto livello del risultato da cui è possibile navigare a quelli correlati. Il metodo `select` assegna all'interrogazione l'analogo della target list di un'interrogazione SQL, rappresentata dalla radice precedentemente costruita. Il metodo `where` assegna all'interrogazione l'analogo della condizione `where` di un'interrogazione SQL, sotto forma di un oggetto di tipo *Expression*, costruito con l'ausilio di metodi opportuni del CriteriaBuilder (nell'esempio, il metodo `equal` restituisce un oggetto che rappresenta un'espressione di uguaglianza). Si noti che la condizione `where` è parametrica, per cui bisogna, prima di assegnarla all'interrogazione, costruire un oggetto che rappresenta il parametro, con l'aiuto del metodo `parameter` del CriteriaBuilder.

L'illustrazione della sintassi completa di JPQL, e analogamente di tutte le interfacce Java per la costruzione di Criteria Query, è fuori dalla portata di questo libro; ci limitiamo a osservare che JPQL supporta comandi con una sintassi molto simile alle istruzioni SQL `select`, `insert` e `update` e rimandiamo il lettore alle note bibliografiche per un approfondimento dei riferimenti completi del linguaggio. Nel resto di questo paragrafo, ci limitiamo a sottolineare tramite esempi alcuni aspetti che differenziano JPQL da SQL.

La principale estensione di JPQL rispetto a SQL riguarda le cosiddette *path expression*. Tali espressioni consentono di definire termini a partire dalle entità e dalle relazioni dello schema astratto. JPQL prevede tre tipi di path expression:

- sui campi persistenti: sono analoghe alle espressioni con l'operatore punto viste per SQL (si veda il Paragrafo 4.3.2). Data un'istanza di entità, selezionano un dato membro (campo) persistente. Un esempio è `prof.Salario`;

- sulle relazioni a valore singolo: data un'istanza di entità, selezionano un'istanza correlata da una relazione a valore singolo (uno a uno o molti a uno). Un esempio è `corso.Dipartimento`;
- sulle relazioni a valore multiplo: data un'istanza di entità, selezionano una collezione di istanze correlate da una relazione a valore multiplo (uno a molti o molti a molti). Un esempio è `dip.Corsi`.

L'interrogazione seguente mostra l'uso di una path expression a valore multiplo. Si noti che l'operatore `is empty` (dal significato ovvio di controllo sul fatto che lo stato sia vuoto) viene applicato alla collezione di oggetti denotata dalla path expression `D.Corsi`.

```
select distinct D
from Dipartimento D
where D.Corsi is empty
```

Quando il simbolo terminale di una path expression denota una singola istanza di entità è possibile navigare la path expression con l'operatore punto applicato agli attributi che rappresentano relazioni a singolo valore, per raggiungere altre entità o campi persistenti, come dimostra l'esempio che segue.

```
select distinct P
from Professore P
where P.Dipartimento.Facolta.Nome =
                        "Ingegneria dell'Informazione"
```

Le path expression a valore multiplo sono molto utili per abbreviare la formulazione di interrogazioni che altrimenti richiederebbero l'uso di nidificazione, come dimostra l'esempio seguente, in cui la clausola `where` verifica la presenza dell'oggetto denotato dal parametro `:corso` nella collezione dei corsi di un dipartimento.

```
select distinct D
from Dipartimento D
where :corso member of D.Corsi
```

Note bibliografiche

SQL Embedded fa parte dello standard SQL-2 e quindi si può far riferimento alla versione ufficiale dello standard SQL o ai numerosi libri divulgativi dedicati a SQL, tra i quali il libro di Cannan e Otten [17] (di cui esiste la traduzione italiana) e i libri di Melton e Simon [60, 61].

I documenti di specifica delle diverse soluzioni CLI sono facilmente accessibili in rete. Sul sito Web della Microsoft sono disponibili i documenti di specifica di ODBC, OLE DB, ADO e ADO.NET. Il sito di Oracle dedicato a Java ospita invece le diverse versioni della specifica JDBC. In ciascuno dei siti è inoltre disponibile un insieme di manuali d'uso e spesso di tutorial sugli specifici strumenti. Per quanto riguarda le

estensioni procedurali di SQL conviene fare riferimento ai manuali dei diversi sistemi. Sono poi disponibili su questi argomenti numerosi libri, dedicati al vasto pubblico dei professionisti dell'informatica.

Per una visione approfondita dello standard JPA, si rimanda alla specifica ufficiale *JSR 317: JavaTM Persistence API, Version 2.0*, curata dal Java Persistence 2.0 Expert Group (scaricabile liberamente da internet). Una trattazione molto completa della materia orientata agli sviluppatori si trova anche nel tutorial sull'architettura Java Enterprise Edition (JEE) 6 a cura di Oracle (http://docs.oracle.com/javaee/6/tutorial/doc/index.html).

Esercizi

Soluzioni sul sito http://www.ateneoonline.it/atzeni

10.1 Realizzare una procedura in un linguaggio di programmazione di alto livello che tramite SQL Embedded elimina dalla tabella DIPARTIMENTO l'elemento che ha il nome che viene fornito come parametro alla procedura.

10.2 Realizzare un programma in un linguaggio di programmazione di alto livello che tramite SQL Embedded costruisce una videata in cui si presentano le caratteristiche di ogni dipartimento seguito dall'elenco degli impiegati che lavorano nel dipartimento, ordinati per cognome.

10.3 Realizzare l'esercizio precedente usando ADO.

10.4 Realizzare un programma Java che scandisce gli impiegati ordinati per cognome e inserisce ogni impiegato che si trova in una posizione che è un multiplo di 10 in una tabella IMPIEGATIESTRATTI.

10.5 Realizzare un programma che accede al contenuto di una tabella CAPITOLO(Numero,Titolo,Lunghezza) che descrive i capitoli di un libro, con il titolo e la dimensione in pagine. Il programma quindi popola una tabella k INDICE (Numero, Titolo,NumPagine) in cui si presenta il numero di pagina nel quale inizia ogni capitolo, supponendo che il Capitolo 1 inizi sulla prima pagina e che i capitoli debbano iniziare su pagine dispari (eventualmente introducendo una pagina bianca alla fine del capitolo).

10.6 Si faccia riferimento al seguente schema relazionale:

IMPIEGATI(CodiceFiscale, Cognome, Nome, DataNascita, Dipartimento, Stipendio);
DIPARTIMENTI(Codice, Nome, Sede)
PROGETTI(Sigla, Titolo, Valore)
PARTECIPAZIONE(Impiegato,Progetto, Data)

con vincoli di integrità referenziale tra Dipartimento di Impiegati e la relazione DIPARTIMENTI, Progetto di Partecipazione e la relazione PROGETTI e tra Impiegato di Partecipazione e la relazione IMPIEGATI.

Scrivere un metodo Java con JDBC (o un frammento di programma in SQL immerso in un linguaggio o pseudolinguaggio di programmazione) che inserisca un impiegato con tutti i dati (letti da input o passati come parametri), verificando l'esistenza del dipartimento, con rifiuto dell'operazione in caso negativo. Assumere, per semplicità, che il sistema non supporti i vincoli di riferimento.

10.7 Dato lo schema relazionale seguente:

IMPIEGATI(Codice, Dati, Telefono) con vincolo di integrità referenziale tra Dati e la relazione DATIIMPIEGATO;

DATIIMPIEGATO(CodiceDati, Cognome, Nome),

definire il metodo Java (o un frammento di programma in SQL immerso in un linguaggio o pseudolinguaggio di programmazione) getImpiegatoPerNome(String nome) che, dato il nome di un impiegato, restituisce un Oggetto Impiegato in cui i campi (Codice, Nome, Cognome, Telefono) sono opportunamente valorizzati.

10.8 Si consideri il seguente schema relazionale (con gli evidenti vincoli di integrità referenziale):

VENDITORE(CodArticolo, CodNegozio, CFCliente, Data, Quantitá)
ARTICOLI(CodArticolo, Descrizione, CodMarca, CodCategoria, Prezzo)
CLIENTE(CFCliente, Cognome, Nome, Etá)
NEGOZIO(CodNegozio, Nome, Indirizzo, Cittá, Provincia, Regione)
MARCA(CodMarca, Nome, CodNazione, Nazione)
CATEGORIA(CodCategoria, Descrizione)

Scrivere un metodo Java con JDBC che inserisca un nuovo negozio con codice, nome, indirizzo e città (letti da input o passati come parametri), prelevando provincia e regione da altre della stessa tabella (nell'ipotesi che, fissata la città, provincia e regione siano univocamente determinate) e segnalando come errore (o eccezione) il caso in cui i dati sulla città non siano disponibili.

10.9 Con riferimento allo schema relazionale seguente:

FARMACI(Codice, NomeFarmaco, PrincipioAttivo, Produttore, Prezzo) con vincolo di integrità referenziale fra Produttore e la relazione PRODUTTORI e fra PrincipioAttivo e la relazione SOSTANZE
PRODUTTORI(CodProduttore, Nome, Nazione)
SOSTANZE(ID, NomeSostanza, Categoria)

scrivere un metodo Java con JDBC che (supponendo già disponibile una connessione, passata come parametro) stampi un prospetto con tutti i farmaci, organizzati per produttore:

CodProduttore Nome Nazione
CodiceFarmaco NomeFarmaco Prezzo Sostanza
CodiceFarmaco NomeFarmaco Prezzo Sostanza
...
CodProduttore Nome Nazione
...

10.10 Si consideri una base di dati che contiene informazioni sugli impiegati, i progetti e le sedi di una azienda, con le partecipazioni degli impiegati ai progetti e le sedi di svolgimento dei progetti stessi; essa contiene le seguenti relazioni:

IMPIEGATI(Matricola,Cognome,Nome,Progetto), con vincolo di integrità referenziale fra Progetto e la relazione PROGETTI
PROGETTI(Codice,Titolo)
SEDI(Nome,Cittá,Indirizzo)
SVOLGIMENTO(Progetto,Sede), con vincoli di integrità referenziale fra Progetto e la relazione PROGETTI e fra Sede e la relazione SEDI.

Formulare in Java con JDBC (o con SQL immerso in un linguaggio o pseudolinguaggio di programmazione) una classe (o un frammento di programma) che stampi tutti i progetti (con codice e titolo) e, per ciascuno di essi, gli impiegati coinvolti (mostrando matricola e cognome); in sostanza, va prodotto un prospetto del tipo seguente:

CodProgetto TitoloProgetto
MatricolaImpiegato CognomeImpiegato

MatricolaImpiegato CognomeImpiegato
...
CodProgetto TitoloProgetto
MatricolaImpiegato CognomeImpiegato
...

10.11 Estendere la risposta al quesito precedente mostrando anche, per ciascun progetto, la lista delle sedi di svolgimento, costruendo quindi un prospetto come il seguente:

CodProgetto TitoloProgetto
MatricolaImpiegato CognomeImpiegato
MatricolaImpiegato CognomeImpiegato
...
NomeSede Città
NomeSede Città
...
CodProgetto TitoloProgetto
...

10.12 Si ha uno schema di tabella Impiegato(Nome, Indirizzo, Capo), in cui l'attributo Capo rappresenta il nome del superiore dell'Impiegato, descritto a sua volta nella tabella. Definire il metodo Java in SQL (immerso in un linguaggio o pseudolinguaggio di programmazione) setCapo(Impiegato i1, Impiegato i2) che memorizza la relazione tra capo e sottoposto esistente tra due impiegati rifiutando l'inserimento se la relazione è già rappresentata nella tabella. Supporre che gli oggetti Impiegato abbiano una variabile di istanza per ogni campo della corrispondente colonna nello schema relazionale con opportuna corrispondenza di tipi.

10.13 Con riferimento allo schema relazionale dell'Esercizio 10.10, si scriva un programma Java che realizza quanto richiesto usando JPA. Si mostrino le annotazioni alle classi Java necessarie per creare la mappatura relazionale degli oggetti Java. Tale mappatura deve contenere anche le annotazioni necessarie per la gestione automatica della cancellazione delle istanze persistenti della classe SVOLGIMENTO collegate tramite vincoli di integrità referenziale alle istanze delle classi PROGETTO e SEDE. Si realizzino le funzioni di estrazione dei dati mediante interrogazioni JPQL.

10.14 Si consideri un'applicazione che gestisce l'anagrafica il personale di un'azienda. Ogni membro del personale è caratterizzato da Nome, Cognome e data di nascita. Gli impiegati dell'azienda sono inoltre caratterizzati da una matricola e un salario. I collaboratori esterni sono caratterizzati da un numero di partita IVA e dal costo orario. La classificazione del personale in impiegati e collaboratori è totale ed esclusiva. Si scriva un programma Java che consenta di gestire oggetti delle tre classi descritte e si mostri la mappatura relazionale degli oggetti secondo i tre schemi: *Single table per class hierarchy*, *Table per concrete class* e *Joined tables*. Si includa nel programma Java un metodo che stampa in ordine alfabetico tutti gli oggetti dell'anagrafica e i nomi del personale (per cognome), precisando oltre agli attributi persistenti anche il tipo di ciascun oggetto.

11 Organizzazione fisica e gestione delle interrogazioni

Nel primo capitolo abbiamo visto che i DBMS sono organizzati secondo un'architettura a livelli, in cui, in particolare, si distinguono il livello *logico*, cui fanno riferimento gli utenti (per esempio con istruzioni SQL, in ambiente interattivo o con l'ausilio di un linguaggio di programmazione) e il livello *fisico* o interno, che stabilisce l'effettiva implementazione, in modo adeguatamente efficiente. Le relazioni (o altre strutture logiche, nel caso di modello diverso da quello relazionale) vengono realizzate utilizzando varie possibili strutture fisiche, con molteplici varianti, ognuna delle quali favorisce alcune operazioni e ne penalizza altre. La proprietà di *indipendenza dei dati*, pure discussa nel primo capitolo, permette agli utenti di ignorare completamente le strutture fisiche, al punto che esse possono essere modificate senza che sia necessario modificare i programmi che le utilizzano. In effetti, le operazioni SQL fanno riferimento al livello logico, ed è il DBMS che si preoccupa di tradurre le operazioni al livello più basso, utilizzando in modo efficiente le strutture fisiche. In questo capitolo descriveremo i concetti fondamentali dell'organizzazione fisica dei dati e dell'esecuzione delle interrogazioni: la gestione della memoria centrale e secondaria (Paragrafo 11.1), l'organizzazione delle tuple in pagine (Paragrafo 11.2), le principali strutture per la memorizzazione delle tabelle e degli indici (Paragrafi 11.3, 11.4 e 11.5) e le procedure con cui le interrogazioni sono ottimizzate tenendo conto dei metodi di accesso a disposizione (Paragrafo 11.6). Il capitolo si concluderà con alcune considerazioni su come tenere conto durante la progettazione degli aspetti fisici dell'architettura di un DBMS per migliorare l'efficienza della base di dati (Paragrafo 11.7).

Per cominciare l'illustrazione dei concetti principali dell'organizzazione fisica della base di dati e della gestione delle interrogazioni facciamo riferimento alla Figura 11.1, che illustra schematicamente le componenti di un DBMS coinvolte. Nelle applicazioni per basi di dati, le operazioni vengono specificate in SQL e affidate a un modulo detto *gestore delle interrogazioni*, che traduce le interrogazioni in una forma interna, le trasforma al fine di renderle più efficienti (per questo il modulo viene spesso chiamato *ottimizzatore delle interrogazioni*) e le realizza in termini di operazioni di livello più basso (scansione, ordinamento, accesso diretto), che fanno riferimento alla struttura fisica dei file e sono eseguite da un modulo sottostante, chiamato *gestore dei metodi d'accesso*. Quest'ultimo modulo, conoscendo i dettagli della struttura fisica dei file, trasforma le richieste in operazioni di lettura e scrittura di dati in memoria secondaria, che vengono però mediate da un modulo, detto *gestore del buffer*, che ha la responsabilità di mantenere temporaneamente porzioni della base di dati in memoria centrale, per favorire l'efficienza garantendo al tempo stesso l'affidabilità. Il gestore del buffer invia poi al *gestore della memoria secondaria* le richieste effettive di lettura e scrittura fisica.

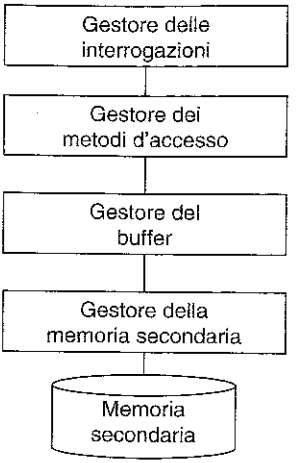

Figura 11.1 Componenti di un DBMS coinvolti nella gestione di interrogazioni e nell'accesso alla memoria secondaria.

Il capitolo presenta questi concetti partendo dal basso, per cui il Paragrafo 11.1 è dedicato alle caratteristiche della memoria secondaria e alla gestione del buffer. Il Paragrafo 11.2 è poi dedicato alla organizzazione interna delle singole pagine che costituiscono i file mentre i due successivi sono dedicati alla organizzazione dei file: il Paragrafo 11.3 illustra le strutture sequenziali e con accesso calcolato, che definiscono due modalità di disposizione delle tuple nei file, mentre il Paragrafo 11.4 illustra le strutture a indice, con le quali è possibile sia disporre le tuple sia soprattutto costruire strutture ausiliarie che favoriscono accessi efficienti. Quindi il Paragrafo 11.5 fornisce una breve descrizione delle strutture fisiche messe a disposizione dai DBMS relazionali. Il Paragrafo 11.6 illustra poi il processo di esecuzione delle interrogazioni, con riferimento sia alla realizzazione delle singole operazioni sia al processo di ottimizzazione. Infine, il Paragrafo 11.7 illustra i principi della progettazione fisica di basi di dati.

11.1 Memoria principale, memoria secondaria e gestione dei buffer

Le basi di dati hanno la necessità di gestire dati in memoria secondaria per due motivi fondamentali. In primo luogo, per quanto la dimensione della memoria principale sia notevolmente cresciuta negli ultimi anni nei sistemi di elaborazione di ogni tipo, essa

non risulta di solito sufficiente per contenere per intero una base di dati. In secondo luogo, una delle caratteristiche fondamentali delle basi di dati è la persistenza: esse hanno un tempo di vita che non è limitato alle singole esecuzioni dei programmi che le utilizzano e debbono essere conservate anche quando i sistemi sono spenti o si guastano. Poiché le memorie centrali sono tuttora sostanzialmente volatili, esse non sono adeguate a supportare la persistenza, mentre lo sono i dispositivi di memoria secondaria (sostanzialmente dischi, al giorno d'oggi, anche se si stanno diffondendo le memorie a stato solido), che sono in grado di mantenere il contenuto inalterato a lungo termine, anche in assenza di alimentazione.

11.1.1 Memoria secondaria: caratteristiche

Ricordiamo che la proprietà che dà il nome alla memoria secondaria è quella in base alla quale essa non è direttamente utilizzabile dai programmi: i dati, per poter essere utilizzati, debbono prima essere trasferiti in memoria principale. Un altro aspetto importante da segnalare è il seguente: sui dischi e sugli altri dispositivi di memoria secondaria, i dati sono organizzati in *blocchi* di dimensione di solito fissa nell'ambito di ciascun sistema; le dimensioni tipiche vanno da alcuni kilobyte ad alcune decine di kilobyte.

Le uniche operazioni possibili su un disco sono la lettura o la scrittura di un intero blocco: questo ha come conseguenza il fatto che l'accesso a un singolo bit ha lo stesso costo dell'accesso a un intero blocco. Inoltre, poiché il tempo necessario per la lettura o la scrittura di un blocco è di vari ordini di grandezza maggiore del tempo necessario per accedere ai dati in memoria centrale ed elaborarli, nelle applicazioni che coinvolgono basi di dati è spesso possibile trascurare i costi di tutte le operazioni esclusi gli accessi a memoria secondaria e quindi utilizzare come approssimazione complessiva il numero di accessi a memoria secondaria (cioè il numero di blocchi letti o scritti).

È anche importante notare, per quanto riprenderemo il punto in seguito solo marginalmente, che il tempo di accesso a un blocco non è costante, ma dipende anche dall'ordine in cui vengono effettuati gli accessi: infatti, il tempo di accesso è costituito di solito da almeno tre componenti: il tempo di *posizionamento della testina* (infatti un disco è organizzato in tracce concentriche ed esiste una sola testina che legge tutte le tracce), il tempo di *latenza* (il tempo che intercorre fra la richiesta e l'istante in cui, durante la rotazione, il blocco di interesse passa sotto la testina) e il tempo di *trasferimento* (necessario per effettuare la lettura o scrittura nel senso stretto del termine). È evidente come le prime due componenti possano variare di molto, a seconda della "distanza" fra i blocchi coinvolti in operazioni successive: se dopo un blocco si legge quello successivo, sulla stessa traccia, il tempo di posizionamento e quello di latenza sono nulli. Di conseguenza, nel caso di letture o scritture massicce il costo complessivo può essere notevolmente ridotto se i blocchi sono adiacenti, cioè se l'allocazione è *contigua*. Notiamo che, con lo sviluppo delle memorie a stato solido, per le quali il tempo di posizionamento e il tempo di latenza non sono rilevanti, queste considerazioni potrebbero presto diventare non significative.

11.1.2 Gestione dei buffer

L'interazione fra memoria centrale e memoria secondaria è realizzata nei DBMS attraverso l'utilizzo di un'apposita, grande zona di memoria centrale detta *buffer*, gestita dal DBMS in modo condiviso per tutte le applicazioni. Con la continua riduzione dei costi delle memorie e quindi la maggiore disponibilità, si è resa possibile l'allocazione di buffer sempre più vasti, che permettono di evitare di ripetere accessi alla memoria secondaria quando uno stesso dato viene utilizzato più volte in tempi ravvicinati. Di conseguenza, la gestione ottimale dei buffer è un aspetto essenziale del funzionamento delle basi di dati, in particolare per quanto concerne le loro prestazioni.

Il buffer è organizzato in *pagine*, che hanno dimensione pari a un numero intero di blocchi. Nel seguito, per semplicità, assumeremo che ciascuna pagina coincide esattamente con un blocco della memoria secondaria, e quindi ogni caricamento o salvataggio di pagina richiede un'operazione su memoria di massa, rispettivamente di lettura o scrittura.

Il gestore del buffer si occupa del caricamento e dello scaricamento (salvataggio) delle pagine dalla memoria centrale alla memoria di massa. Intuitivamente, possiamo pensare al gestore del buffer come a un modulo che riceve, dai programmi, richieste per la lettura e la scrittura di blocchi ed esegue le effettive letture o scritture sulla base di dati, secondo una tempistica che non coincide necessariamente con quella delle richieste ricevute; in particolare:

- in caso di lettura, se la pagina è già presente nel buffer, allora non è necessario effettuare la lettura fisica;
- in caso di scrittura, il gestore del buffer può decidere di differire la scrittura fisica, quando tale attesa è compatibile con le proprietà di affidabilità del sistema; questo aspetto verrà descritto nel prossimo capitolo.

Si noti che entrambi i casi esposti consentono di ridurre il tempo di risposta di una applicazione, evitando operazioni su memoria di massa.

Le politiche di gestione del buffer assomigliano a quelle della gestione della memoria centrale da parte dei sistemi operativi, e obbediscono allo stesso principio, detto di *località dei dati*, in base al quale i dati referenziati di recente hanno maggior probabilità di essere referenziati nuovamente nel futuro. In aggiunta, una nota legge empirica dice che il 20% dei dati è tipicamente acceduto dall'80% delle applicazioni; questa legge ha come conseguenza che generalmente i buffer contengono le pagine cui viene fatta la maggior parte degli accessi.

L'interfaccia offerta dal gestore del buffer ai moduli di livello più alto è in effetti più complessa di quanto sopra accennato; infatti, essa non può limitarsi alle richieste di lettura e scrittura, ma ha bisogno di informazioni sul prevedibile riutilizzo delle pagine (per evitarne la lettura ripetuta) e sulla eventuale necessità di effettuare immediatamente gli aggiornamenti (per garantirne l'effettuazione ed evitare che vadano persi in modo irrecuperabile in caso di guasto). Per la gestione dei buffer è quindi necessario mantenere informazioni sul loro uso, che possiamo schematizzare nel modo seguente.

- Un direttorio descrive il contenuto corrente del buffer indicando per ciascuna pagina quali sono il file fisico e il numero di blocco a essa corrispondente.
- Per ogni pagina del buffer il gestore mantiene alcune "variabili di stato", fra cui le seguenti: un contatore, per indicare quanti programmi utilizzano la pagina;[1] un bit di stato, per indicare se la pagina è stata modificata (e quindi le modifiche vanno prima o poi riportate in memoria secondaria).

Un possibile insieme di operazioni fondamentali allo scopo è il seguente.[2]

- La primitiva *fix* viene usata dalle transazioni per richiedere l'accesso a una pagina; essa restituisce al modulo chiamante il riferimento alla pagina del buffer, in modo che esso possa accedere effettivamente ai dati. L'esecuzione della primitiva è realizzata nel modo seguente.

  1. Si cerca la pagina tra quelle già presenti in memoria. In caso positivo, l'operazione si conclude e l'indirizzo della pagina viene restituito alla transazione richiedente. Per il principio di località dei dati ciò avviene abbastanza spesso.
  2. Altrimenti, viene scelta una pagina nel buffer, cercando fra le pagine libere, cioè con contatore pari a zero. La scelta si può basare su diverse strategie, che cercano di minimizzare le letture fisiche, per esempio selezionando la pagina usata meno di recente (*LRU*, *least recently used*), oppure quella caricata da più tempo (*FIFO*, *first in first out*). Se il bit di stato segnala che la pagina è stata modificata, essa viene aggiornata in memoria di massa (operazione di *flush*). Vengono quindi operate le opportune conversioni di indirizzi in modo da identificare la pagina da caricare nel buffer, e avviene l'operazione di lettura.
  3. Se non esistono pagine libere, il gestore del buffer può comportarsi secondo due politiche alternative. Nel primo caso (politica *steal*) viene sottratta una pagina a un'altra transazione. La pagina selezionata, detta *vittima*, viene scaricata in memoria di massa (operazione di *flush*). Vengono poi operate le opportune conversioni di indirizzi e avviene l'operazione di lettura. Nel secondo caso (politica *no steal*) la transazione viene sospesa, in attesa che si liberino pagine del buffer; entra pertanto in una coda di transazioni, mantenuta dal gestore del buffer. Quando si libera una pagina, il gestore del buffer si comporta come descritto al punto precedente.
  4. In ogni caso, quando si effettua un accesso a una pagina, viene incrementato il contatore relativo all'utilizzo della pagina.

- La primitiva *setDirty* indica al gestore del buffer che una pagina è stata modificata; l'effetto è la modifica del bit di stato relativo.

[1] L'utilizzo di una stessa pagina da parte di più programmi è autorizzato dal gestore della concorrenza, un modulo di livello superiore rispetto al gestore del buffer che filtra le richieste di pagine, come vedremo nel Capitolo 12.

[2] Si potrebbe anche pensare a un'interfaccia del gestore del buffer che permetta di fare riferimento a singoli dati nelle pagine, ma per semplicità ignoriamo la questione.

- La primitiva *unfix* indica al gestore del buffer che il modulo chiamante ha terminato di usare la pagina; come effetto, viene decrementato il contatore di utilizzo della pagina.
- La primitiva *force* trascrive in memoria di massa, in modo sincrono, una pagina del buffer. Come vedremo nel prossimo capitolo, questa operazione viene richiesta dal gestore dell'affidabilità quando è necessario garantire che alcuni dati non vengano persi.

È importante sottolineare che la scrittura in memoria secondaria può avvenire in modo sincrono, all'interno dell'applicazione che la richiede, attraverso la *force*, oppure in modo asincrono (cioè in modo indipendente dall'applicazione) a seguito di decisioni del gestore del buffer (con operazione che viene chiamata in questo caso di *flush*), finalizzate al recupero di spazio, come abbiamo visto sopra, oppure a seguito di scelte di ottimizzazione. In particolare, il gestore del buffer può sfruttare la disponibilità dei dispositivi di ingresso/uscita per anticipare la scrittura delle pagine rese libere da una operazione di unfix e che siano state modificate (cioè con bit di stato posto a "dirty"); a seguito di questa operazione (detta *pre-flushing*) le successive operazioni di fix possono avvenire più rapidamente. Si noti infine che una pagina utilizzata da molte applicazioni può rimanere a lungo nel buffer, anche accumulando varie modifiche, per poi essere trascritta in memoria secondaria con una sola operazione di scrittura.

11.1.3 DBMS e file system

Il file system è un modulo messo a disposizione dal sistema operativo; i DBMS ne utilizzano le funzionalità. Tuttavia, contrariamente alle apparenze, la relazione fra le funzioni affidate al file system e quelle affidate al DBMS non è semplice. Vi è stato un lungo periodo di tempo in cui le funzionalità offerte dai sistemi operativi non potevano garantire efficienza e affidabilità, per cui i DBMS dovevano implementare funzioni proprie di ingresso/uscita. Attualmente, i DBMS usano il file system ma si creano una propria astrazione dei file, per garantire efficienza (tramite i buffer e tramite una gestione di basso livello delle strutture fisiche) e transazionalità (tramite il controllore della affidabilità, di cui parleremo nel prossimo capitolo). Non è escluso che nel futuro queste funzionalità possano migrare verso il sistema operativo, in un'ottica di "esportare" le funzionalità delle basi di dati al di fuori dei DBMS.

Al giorno d'oggi, nella maggior parte dei casi, i DBMS utilizzano solo poche funzionalità di base del sistema operativo per creare ed eliminare file e per leggere e scrivere singoli blocchi o sequenze di blocchi contigui. Tutto ciò che riguarda invece la struttura dei file, sia all'interno dei singoli blocchi sia nell'organizzazione dei dati a partire dai blocchi, è realizzato direttamente dal DBMS.

In molti casi, infatti, il DBMS crea file di grandi dimensioni che utilizza per memorizzare diverse relazioni (al limite, l'intera base di dati); in altri casi, vengono creati file in tempi successivi, seguendo le esigenze, e può succedere che un file contenga i dati di più relazioni e che le varie tuple di una relazione siano in file diversi. In sostanza, il DBMS gestisce i blocchi dei file allocati come se fossero un unico grande spazio di memoria secondaria e costruisce, in tale spazio, le strutture fisiche con

cui implementa le relazioni. Nel caso più frequente, ogni blocco è dedicato a tuple di un'unica relazione, ma esistono tecniche che (come vedremo nel Paragrafo 11.5) prevedono la memorizzazione delle tuple di più tabelle, fra loro correlate, negli stessi blocchi.

11.2 Gestione delle tuple nelle pagine

Sebbene ciascuna organizzazione fisica possa avere una propria specificità nella gestione delle pagine, vi sono molti aspetti comuni che può essere interessante descrivere. In ciascuna pagina sono presenti sia informazione utile sia informazione di controllo; l'informazione utile coincide con i dati veri e propri, l'informazione di controllo consente di accedere all'informazione utile. Avvertendo che, per i particolari, sono possibili molte alternative, presentiamo in dettaglio una possibile organizzazione, mostrata nella Figura 11.2.

- Ogni pagina, in quanto coincidente con un blocco di memoria di massa, ha una parte iniziale (*block header*) e una finale (*block trailer*) contenenti *informazione di controllo utilizzata dal file system*.
- Ogni pagina, in quanto contenente dati, gestiti dal DBMS, ha poi una parte iniziale (*page header*) e una finale (*page trailer*) contenenti *informazione di controllo relativa alla specifica struttura fisica*. Tale informazione può per esempio contenere l'identificatore dell'oggetto (tabella, indice, dizionario dei dati ecc.) contenuto nella pagina, puntatori a pagine successive o precedenti nella struttura dati, numero di dati utili elementari (tuple) contenuti nella pagina, quantità di memoria libera (contigua oppure non contigua) disponibile nella pagina.
- In molti casi, ogni pagina ha poi un suo *dizionario di pagina*, che contiene puntatori a ciascun dato utile elementare contenuto nella pagina, e una *parte utile*, che

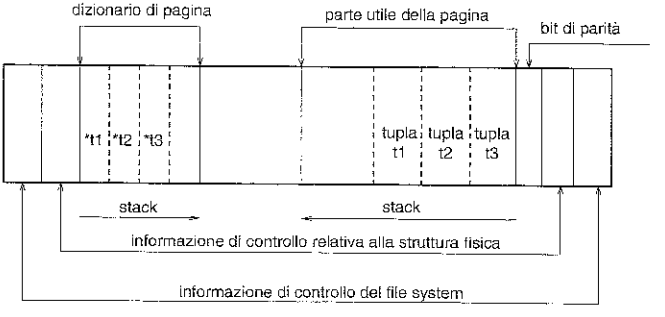

Figura 11.2 Organizzazione di una pagina.

contiene i dati. In genere, dizionario di pagina e dati utili crescono come stack contrapposti, lasciando memoria libera in uno spazio contiguo compreso tra i due stack.
- Infine, ogni pagina contiene *bit di parità*, per verificare che l'informazione in essa contenuta sia valida.

Molti gestori delle pagine non consentono la separazione di una tupla su più pagine, e in tal caso la massima dimensione di una tupla è limitata dal massimo spazio utile disponibile in una pagina; alcuni gestori delle pagine consentono invece di distribuire una tupla su più pagine.

Inoltre, in alcuni casi, tutte le tuple hanno la stessa dimensione; in tal modo, si semplifica la struttura dei dizionari di pagina, ma si corre il rischio di sprecare spazio nella pagina. Come vedremo, alcune organizzazioni fisiche sono caratterizzate da tuple di dimensioni fisse. In altri casi le tuple possono avere lunghezze diverse (soprattutto quando esse contengono attributi di tipo stringa o testo oppure valori nulli); nel caso di tuple a lunghezza variabile, il dizionario di pagina contiene l'indicazione degli offset di ciascuna tupla rispetto all'inizio della parte utile e di ciascun valore dei vari campi presenti nella tupla rispetto all'inizio della tupla stessa. Infine, come già accennato nel paragrafo precedente, in alcuni casi è possibile avere in una stessa pagina tuple appartenenti a relazioni diverse.

Un parametro importante è il *fattore di blocco*, pari al numero di record contenuti in un blocco. In prima approssimazione esso è pari al rapporto fra la dimensione del blocco e la dimensione media del record.

Le primitive offerte dal gestore delle pagine sono le seguenti.

- *Inserimento* e *aggiornamento*, che non comportano riorganizzazioni della pagina se esiste un sufficiente spazio per gestire i byte introdotti in più. Quando lo spazio disponibile non è contiguo, le operazioni devono essere precedute da una riorganizzazione della pagina, che peraltro ha costo limitato in quanto è svolta in memoria centrale. Quando invece lo spazio disponibile è insufficiente, l'operazione comporta una interazione con il file system, che deve allocare nuovi blocchi per il file.
- *Cancellazione*, che è sempre possibile e spesso viene effettuata senza riorganizzare l'informazione nella pagina (cioè senza compattare lo stack relativo alla parte utile), ma semplicemente marcando la tupla come "non valida".
- *Accesso a una particolare tupla*, identificata tramite il valore della chiave (in modo associativo) oppure in base al suo offset, presente nel direttorio.
- *Accesso a un campo di una particolare tupla*, identificato in base all'offset e alla lunghezza del campo stesso dopo aver identificato la tupla tramite la sua chiave o il suo offset (come descritto in precedenza).

Si noti che l'organizzazione delle pagine che abbiamo appena descritto si applica soprattutto ai dati veri e propri e non alle strutture ausiliarie (solitamente ad albero) utilizzate per realizzare gli indici, che verranno illustrate nel Paragrafo 11.4.2.

11.3 Strutture primarie per l'organizzazione di file

Illustriamo in questo paragrafo le principali tecniche utilizzate per la struttura primaria di un file, cioè quella che stabilisce il criterio secondo il quale sono disposte le tuple nell'ambito del file. Le strutture possono essere divise in tre categorie principali, *sequenziali*, *ad accesso calcolato* (o *hash*) e *ad albero*. Le strutture sequenziali sono caratterizzate da una disposizione sostanzialmente consecutiva delle tuple in memoria di massa che si basa su un criterio specifico, che può derivare dall'ordine di inserimento o da un'altra regola opportuna. Le strutture ad accesso calcolato invece collocano le tuple in posizioni determinate sulla base del risultato dell'esecuzione di un algoritmo. Le strutture ad albero sono utilizzate anche e soprattutto come strutture secondarie (cioè aggiuntive rispetto a quelle primarie) e quindi rimandiamo la loro discussione al prossimo paragrafo.

Si noti che si usano spesso in letteratura i termini *struttura* (o *organizzazione*) *fisica* e *metodo d'accesso*, come sinonimi, e anche in questo testo ci adeguiamo per semplicità a questa usanza. Più propriamente, i primi due termini fanno riferimento al modo in cui i dati sono organizzati (disposizione delle tuple e delle pagine e riferimenti fra loro), mentre l'ultimo indica i programmi (disponibili nelle librerie del DBMS e trasparenti per l'utente) che utilizzano tali strutture, traendo profitto dalle loro caratteristiche.

11.3.1 Strutture sequenziali

Nelle strutture sequenziali, un file è costituito da vari blocchi di memoria "logicamente" consecutivi, e le tuple vengono inserite nei blocchi rispettando una sequenza:

- in una organizzazione *disordinata (seriale)*, la sequenza delle tuple è indotta dal loro ordine di immissione;
- in una organizzazione *ad array*, le tuple sono disposte come in un array, e la loro posizione dipende dal valore assunto in ciascuna tupla da un campo di *indice*;
- in una organizzazione *sequenziale ordinata*, la sequenza delle tuple dipende dal valore assunto in ciascuna tupla da un campo del file.

Struttura disordinata La struttura sequenziale più semplice, ma anche più diffusa, è quella detta *seriale* o *disordinata*. I due termini descrivono le due caratteristiche: i record vengono inseriti nel file nell'ordine in cui si presentano e quindi non esiste alcun ordine basato su un qualche valore (per esempio, quello alfabetico dei cognomi, oppure del numero di matricola), a meno che i record non vengano già forniti in ordine (il che non può comunque essere assunto dal DBMS). In inglese questa struttura viene spesso chiamata *heap*, che significa "mucchio", per sottolineare il fatto che i record vengono ammucchiati alla meglio, senza perdere tempo a sistemarli secondo un qualche criterio. È evidente come questa struttura sia molto efficiente per le operazioni di inserimento, in quanto è sufficiente mantenere un riferimento all'ultimo blocco per procedere con un solo accesso. È anche possibile ottenere un buon grado di contiguità se le allocazioni dei blocchi necessari vengono fatte con riferimento a un certo numero di essi, consecutivi. Per quanto riguarda invece le ricerche, la struttura

disordinata, da sola, è poco efficiente, perché, mancando un criterio specifico per la memorizzazione, qualunque ricerca deve considerare tutti i record,[3] cioè richiede una *scansione sequenziale* del file. Pertanto, possiamo dire che il costo di una ricerca in un file sequenziale è *lineare*, nel numero di blocchi del file, in quanto sostanzialmente pari a esso. Per questo motivo, le strutture disordinate sono sì molto usate nei sistemi relazionali, ma insieme a strutture secondarie (che discuteremo fra poco) utilizzate per favorire gli accessi. Va detto peraltro che nelle basi di dati relazionali sono sempre (o quasi) definiti vincoli di chiave, per cui anche le operazioni di inserimento non possono essere effettuate senza verificare tali vincoli. Anche qui, il modo più semplice per effettuare la verifica è la scansione sequenziale dei dati già presenti quando se ne inserisce uno nuovo, ma le strutture secondarie possono rendere la verifica molto più efficiente.

Le strutture disordinate permettono di realizzare in modo abbastanza semplice le operazioni di eliminazione e modifica, una volta individuati i record coinvolti. Per le eliminazioni si procede di solito "marcando" il record come cancellato, ma senza alcuna riorganizzazione locale. Per le modifiche, si procede in loco, se possibile (se c'è spazio nel blocco di interesse, il che si verifica se il record non cresce di dimensione, oppure se è stato lasciato inizialmente spazio libero oppure si è liberato per cancellazioni), altrimenti eliminando la vecchia versione del record e reinserendo la nuova a fine file. Tutto questo può portare a uno spreco di spazio, che fa crescere il numero dei blocchi utilizzati per il file rispetto allo stretto necessario (con un conseguente aumento del numero di accessi necessari per una scansione sequenziale). Per questo motivo, i sistemi prevedono di solito la possibilità di procedere, periodicamente, a riorganizzazioni dei file con eliminazione dello spazio sprecato.

Struttura sequenziale ad array Un'organizzazione sequenziale ad array è possibile solo quando le tuple di una tabella sono di dimensione fissa; in tal caso, al file viene associato un numero n di blocchi contigui e ciascun blocco è dotato di un numero m di "posizioni" disponibili per le tuple, dando luogo a un array di $n \times m$ posizioni complessive; ciascuna tupla è dotata di un valore numerico i che funge da indice e viene posta nella i-esima posizione dell'array. Per il caricamento iniziale del file, gli indici vengono semplicemente ottenuti incrementando un contatore; tuttavia, sono possibili inserimenti e cancellazioni. Le cancellazioni creano posizioni libere, gli inserimenti devono essere effettuati nelle posizioni libere o al termine del file.

Le tipiche primitive garantite da questa organizzazione sono quelle di inserimento, lettura e cancellazione della tupla corrispondente a un determinato valore di indice.

Le strutture ad array non sono quasi mai utilizzate nei DBMS, perché accade assai raramente che i dati rispettino i vincoli di applicabilità delle strutture stesse. Chiaramente, gli array sono strutture assai efficienti per effettuare una ricerca in base al valore dell'indice, perché la pagina e perfino la posizione di una tupla nella pagina possono essere calcolati a partire dal valore dell'indice, dando luogo a un unico

[3] Salvo il caso della ricerca su un campo chiave, che può essere interrotta una volta trovato il record di interesse, nel caso sia presente, ma possiamo trascurare questo aspetto.

accesso alla memoria di massa. Vedremo nella prossima sezione come sia possibile, tramite le strutture ad accesso calcolato, avvicinarsi alle caratteristiche delle strutture sequenziali ad array.

Struttura sequenziale ordinata L'organizzazione sequenziale ordinata (detta anche in alcuni sistemi *clustered*) prevede la memorizzazione dei record secondo un ordinamento fisico coerente con l'ordinamento di un campo (per esempio, la matricola) detto *chiave*.[4] Contrariamente a quanto si potrebbe pensare, il beneficio derivante dalle strutture ordinate non consiste di solito nella possibilità di eseguire ricerche dicotomiche, perché questo richiederebbe informazioni su tutti i blocchi del file (per trovare di volta in volta il blocco mediano), il che non sempre è possibile. In effetti, proprio per questo motivo, nelle basi di dati relazionali, queste strutture sono utilizzate solo in stretta associazione con indici. Al tempo stesso, le strutture ordinate rendono efficienti, come ovvio, le operazioni che hanno bisogno proprio dell'ordinamento utilizzato (per esempio, una struttura ordinata per cognome favorisce la produzione di un elenco ordinato per cognome). Inoltre, esse favoriscono le cosiddette "selezioni su intervallo" (in inglese *range query*), per esempio, la ricerca di tuple con un cognome che inizia con una certa sequenza di lettere oppure con una data compresa in un certo intervallo, in quanto memorizzano in posizioni consecutive i record che soddisfano la condizione: una volta individuato (per esempio tramite un indice) il primo record, l'accesso agli altri sarà molto efficiente. In modo analogo, vengono favorite le operazioni aggregate, se l'ordinamento è sui campi di aggregazione (cioè quelli coinvolti in una clausola `group by`).

È importante osservare che le strutture ordinate presentano inconvenienti in presenza di aggiornamenti, a causa della necessità di mantenere l'ordinamento. Le eliminazioni possono portare a spreco di spazio e gli inserimenti in posizioni intermedie possono richiedere spazio aggiuntivo, spesso con perdita della contiguità. Di conseguenza, le strutture ordinate possono richiedere periodiche riorganizzazioni, ancora più importanti di quelle richieste per le strutture disordinate.

11.3.2 Strutture con accesso calcolato (hash)

Una struttura con accesso calcolato (detta anche *hash*) garantisce un accesso *associativo* ai dati, in cui cioè la locazione fisica dei dati dipende dal valore assunto da un campo chiave.

L'idea alla base delle strutture hash è estendere le caratteristiche di accesso diretto degli array anche al caso in cui l'array non sia immediatamente applicabile. Infatti

[4]Per semplicità ci riferiamo a *un* campo di ordinamento, ma potrebbero essere due o più, con ordinamento sulla base del primo e, in caso di valori uguali, sul secondo, e così via. Inoltre, è importante notare che su questo punto si riscontra nella letteratura una certa confusione terminologica. Infatti, il campo su cui è realizzato l'ordinamento non è necessariamente la chiave della relazione. Più propriamente, qui come nei paragrafi successivi, dovremmo usare il termine *pseudochiave*, per indicare un campo (o un insieme di campi) su cui si basa una tecnica di indirizzamento o di accesso. Poiché i campi che formano la chiave primaria sono quelli su cui è più frequente realizzare ordinamenti, indici e accessi hash, è invalso l'uso improprio del termine.

l'array si presta naturalmente a realizzare una struttura per organizzare un insieme di record che abbiano un campo "chiave" con valori consecutivi. per esempio, se un'azienda ha 10 000 impiegati, e utilizza per ciascuno un numero di matricola diverso compreso fra 1 e 10 000, allora un array di 10 000 elementi con un tipo indice intero permette di accedere in modo efficiente al record di un impiegato dato il suo numero di matricola. Questa tecnica non è viceversa utilizzabile in modo efficace se il numero di possibili valori per la chiave è molto maggiore del numero di elementi: per esempio, se abbiamo 40 studenti con numeri di matricola di sei cifre (cioè un milione di possibili valori) non possiamo certo pensare di utilizzare un array con un milione di posizioni a disposizione.

Volendo continuare a usare qualcosa di simile a un array, ma senza sprecare spazio, possiamo pensare di trasformare i valori della chiave in possibili indici di un array: per esempio, possiamo ricondurre i numeri di matricola a interi compresi in un intervallo dello stesso ordine di grandezza (ma un po' più grande, per ragioni che chiariremo tra poco) della cardinalità dell'insieme di record. La funzione utilizzata per trasformare la chiave in valore dell'indice viene detta *funzione hash* (perché in inglese "to hash" significa "frammentare" o "triturare," e la funzione spesso applica diverse trasformazioni all'operando per favorire la equidistribuzione dei valori del risultato). La funzione hash consente di trasformare la matricola nell'indice di un array, e quindi associare ogni record a una posizione specifica di una struttura sequenziale ad array.

Nell'esempio potremmo utilizzare un array di 50 elementi e una funzione hash (molto semplice, mentre nei sistemi reali le funzioni sono predefinite e complesse, per essere di utilizzabilità generale) che calcola il resto della divisione per 50 (che, supponendo i numeri di matricola scelti a caso, ottiene comunque la loro equidistribuzione). Un esempio tratto da un caso reale[5] è mostrato nella Figura 11.3, ove la prima colonna mostra il valore della matricola e la seconda colonna mostra l'indirizzo ottenuto come resto della divisione.

L'esempio evidenzia il problema fondamentale delle strutture con accesso calcolato: poiché l'insieme delle chiavi è molto più grande dell'insieme dei possibili valori dell'indice, la funzione hash non può essere iniettiva e quindi è sempre possibile che si verifichino collisioni, cioè valori diversi della chiave che portano allo stesso valore dell'indice. L'imprevedibilità dei valori delle chiavi rende inevitabili le collisioni, anche se una buona funzione hash (che distribuisca i risultati in modo casuale) rende bassa la probabilità di collisione e soprattutto quella di collisioni multiple.[6] Si intuisce facilmente, e si può verificare in modo analitico o sperimentale, che la dimensione media delle collisioni diminuisce all'aumentare dello spazio disponibile, in quanto diminuisce la probabilità delle collisioni stesse.

[5] Si tratta dei numeri di matricola di 40 studenti che hanno frequentato alcuni anni fa un corso tenuto da uno degli autori.

[6] Nell'esempio abbiamo una collisione a quattro (sul valore 10 dell'indice), due collisioni a tre (sui valori 2 e 5) e cinque collisioni a due. Assumendo che, in caso di collisione di n elementi, l'accesso al primo costi 1, quello al secondo 2 e così fino a un costo pari ad n per l'n-esimo), il numero medio di accessi alla struttura dati necessario per accedere a un generico elemento risulta in questo caso pari a 1,425.

M	M mod 50
60600	0
66301	1
205751	1
205802	2
200902	2
116202	2
200604	4
66005	5
116455	5
200205	5
201159	9
205610	10
201260	10
102360	10
205460	10
205912	12
205762	12
200464	14
205617	17
205667	17

M	M mod 50
200268	18
205619	19
210522	22
205724	24
205977	27
205478	28
200430	30
210533	33
205887	37
200138	38
102338	38
102690	40
115541	41
206092	42
205693	43
205845	45
200296	46
205796	46
200498	48
206049	49

Figura 11.3 Un insieme di chiavi e le relative collisioni.

Dopo aver introdotto informalmente le caratteristiche delle funzioni hash, vediamo come si applicano all'organizzazione di file. In questo caso, si trae vantaggio dalle proprietà della memoria secondaria, e in particolare dal fatto che l'operazione di costo unitario è l'accesso a un blocco, che può spesso contenere diversi record. Inoltre, i blocchi vengono riempiti solo in parte, e si usa il termine *fattore di riempimento* per indicare la frazione dello spazio fisico disponibile mediamente utilizzata: nell'esempio dei 40 studenti descritto in precedenza, tale fattore è pari a 0,8. Quindi, se T è il numero di tuple previsto per il file, F il fattore di blocco e f il fattore di riempimento, il file può prevedere un numero di blocchi B pari al numero intero immediatamente superiore a $T/(f \times F)$. La funzione hash, applicata alla chiave, restituisce un numero compreso tra 0 e $B - 1$, che viene interpretato come numero del blocco nell'ambito dei blocchi allocati al file. In caso di collisione, i record vengono semplicemente allocati nel blocco fino a esaurimento dello spazio libero (e quindi, vengono mediamente accettate F collisioni). Quando lo spazio relativo a un blocco viene esaurito, viene allocato un ulteriore blocco, collegato al precedente, e il record viene disposto all'interno di esso. Questa tecnica viene replicata anche in caso di completamento dello spazio disponibile nel nuovo blocco, dando luogo a *catene di overflow*.

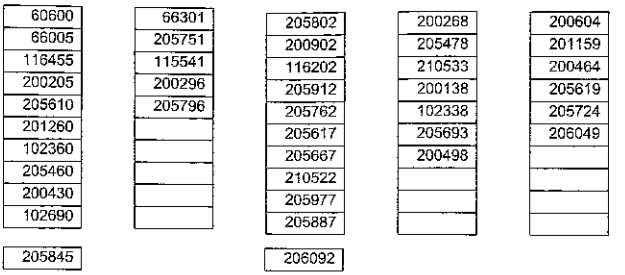

Figura 11.4 L'idea base di una struttura hash.

Riprendendo l'esempio della Figura 11.3 e utilizzando ancora il resto della divisione come funzione hash, vediamo in Figura 11.4 che cosa succede con un fattore di blocco pari a 10 e un utilizzo di 5 blocchi, per un totale quindi sempre di 50 posizioni: in due casi abbiamo 11 record destinati a un blocco, mentre negli altri tre ne abbiamo meno di dieci. Utilizzando in questi due casi una catena di overflow (che chiaramente consta di un singolo blocco), possiamo concludere che, mediamente, sono necessari 1,05 accessi a blocchi per accedere a un record (in 38 casi dobbiamo accedere a un unico blocco e solo nei restanti due casi dobbiamo accedere a due).

Analizzando in dettaglio le collisioni e le loro probabilità, potremmo prevedere con buona precisione il comportamento delle strutture hash. Omettiamo il ragionamento, che porta a calcolare la lunghezza media della catena di overflow come funzione del fattore di blocco e del coefficiente di riempimento del file, riportando solo le conclusioni, per mezzo dei dati della tabella in Figura 11.5 (ripresi dal testo di Gray e Reuter [48]): la lunghezza delle catene di overflow cresce all'aumentare del coefficiente di riempimento, perché aumenta la probabilità di collisione, ma decresce all'aumentare del fattore di blocco, perché le probabilità di collisione elevata si ammortizzano su uno spazio più grande.

		Fattore di blocco			
		1	3	5	10
Coefficiente di riempimento	.7	1.167	0.286	0.136	0.042
	.8	2.000	0.554	0.289	0.110
	.9	4.495	1.377	0.777	0.345

Figura 11.5 Lunghezza media della catena di overflow a seguito di collisioni.

La struttura hash è ideale quando si vuole accedere, in modo associativo, alla tupla che contiene uno specifico valore della chiave (accesso *puntuale*), in quanto l'indirizzo prodotto dall'algoritmo di calcolo viene passato al gestore dei buffer per accedere direttamente al blocco così identificato. In modo analogo, la scrittura di una tupla che contiene un determinato valore di chiave viene effettuata in quel blocco. Quindi, in assenza di collisioni, questo metodo di accesso consente di eseguire letture e scritture di tuple (noto il valore della chiave) con una sola operazione di ingresso/uscita per localizzare il blocco interessato all'operazione, con un rendimento ottimale. Il gestore delle tuple nelle pagine garantisce poi di ritrovare la tupla in base al valore di chiave. La discussione appena svolta conferma come, in molti casi, le collisioni portino peggioramenti non particolarmente significativi. In sostanza, si può affermare che la complessità dell'accesso a un record in questo caso è *costante* (pari a uno o poco più).

Nel caso di accessi basati su intervalli di valori non si ottiene un beneficio analogo a quello degli accessi puntuali, perché le funzioni hash tendono a sparpagliare valori anche vicini su blocchi diversi: per esempio, se dovessimo cercare lo studente con numero di matricola 17012, potremmo trovarlo (o verificare che non esiste) con un solo accesso a memoria secondaria (o poco più, in media), ma se volessimo trovare gli studenti con matricola compresa fra 17010 e 17029 l'unica possibilità, a parte l'accesso sequenziale, sarebbe quella di effettuare 20 accessi diretti diversi, uno per ciascuno dei valori dell'intervallo; come vedremo nel prossimo paragrafo, una struttura ad albero, pur essendo leggermente meno efficiente per accessi puntuali, mantiene la stessa efficienza anche per accessi basati su intervalli. Ovviamente, nessun beneficio viene fornito dalla struttura hash per accessi basati su campi diversi da quello chiave (per esempio, se la matricola è il campo chiave e si vuole cercare uno studente in base al cognome).

Un'altra limitazione della struttura hash è dovuta alla sua scarsa dinamicità: abbiamo visto che le sue prestazioni dipendono in modo significativo dal coefficiente di riempimento; di conseguenza, se il file varia di dimensione e in particolare cresce, allora la struttura perde la sua efficienza e l'unica soluzione possibile consiste nel riorganizzare il file, con un maggior numero di blocchi e una diversa funzione hash. Sono state peraltro proposte in letteratura varianti della struttura hash in grado di evolversi dinamicamente.

11.4 Strutture ad albero

In questo paragrafo studiamo le strutture ad albero, denominate *indici*; queste strutture favoriscono, come nell'organizzazione hash, l'accesso in base al valore di uno o più campi, consentendo sia accessi puntuali sia accessi corrispondenti a intervalli di valori. L'organizzazione ad albero può essere utilizzata per realizzare sia strutture primarie, cioè strutture per contenere i dati, sia strutture secondarie, che favoriscono gli accessi ai dati senza peraltro contenere i dati stessi. Prima di vedere in dettaglio le organizzazioni ad albero, introduciamo brevemente l'uso degli indici distinguendo i casi di strutture primarie e secondarie.

11.4.1 Indici primari e secondari

In prima approssimazione, dato un file f con un campo chiave[7] k, un *indice secondario* è un altro file, in cui ciascun record è logicamente composto di due campi, uno contenente un valore della chiave k del file f e l'altro contenente l'indirizzo o gli indirizzi fisici dei record di f che hanno quel valore di chiave. L'indice secondario è ordinato in base al valore della chiave e consente quindi una rapida ricerca in base a tale valore. Pertanto, l'indice secondario può essere usato da un programma per accedere rapidamente ai dati del file primario, supponendo che l'interrogazione cui dare risposta abbia un predicato basato sul campo k. Intuitivamente, un indice secondario è molto simile all'indice analitico di un libro, che è un elenco di coppie ognuna delle quali contiene un termine e una lista di numeri di pagina ove il termine è citato. L'indice analitico è ordinato alfabeticamente sulla base dei termini.

Quando invece l'indice contiene al suo interno i dati, oppure è realizzato su un file ordinato sullo stesso campo su cui è definito l'indice stesso, esso viene detto *primario*, perché garantisce non solo un accesso in base alla chiave, ma contiene anche i record fisici necessari per memorizzare i dati o comunque ne vincola l'allocazione. Ovviamente, un file può avere un solo indice primario e più indici secondari. Inoltre, un file la cui organizzazione primaria è hash oppure sequenziale, secondo un qualche criterio che non sia l'ordinamento sul campo dell'indice, non può avere un indice primario, perché il criterio di memorizzazione dei dati è in tal caso stabilito dalla funzione di hash o particolare sequenza, e quindi non può essere determinato dall'indice. Continuando il paragone con i libri, possiamo considerare l'indice generale come indice primario (e quindi, l'indice generale determina la posizione dei paragrafi, che possiamo considerare i dati del libro medesimo). Inoltre, il libro può avere vari indici secondari, per esempio molte guide turistiche hanno un indice dei luoghi e un indice dei ristoranti o degli alberghi.

Pur senza entrare troppo nei dettagli, possiamo fare alcune osservazioni sulla struttura degli indici.

- Abbiamo detto sopra che l'indice contiene gli indirizzi fisici dei record del file. In effetti, ciò può essere realizzato attraverso semplici riferimenti ai blocchi oppure con indirizzi che includono anche gli offset all'interno dei blocchi. Le due soluzioni sono diverse nei dettagli (in particolare nel caso di indici su campi non chiave) ma paragonabili nelle prestazioni, perché una ricerca all'interno di un blocco, effettuata in memoria centrale, ha un costo che può essere considerato trascurabile. I puntatori ai blocchi sono più compatti, mentre i puntatori ai record permettono di rendere più efficienti alcune operazioni relative a interrogazioni complesse, effettuandole solo sull'indice, senza accedere al file se non alla fine, per i soli record da recuperare.

[7] Come detto nel paragrafo precedente con riferimento alle strutture ordinate e a quelle hash, anche gli indici possono essere realizzati su più campi (non necessariamente uno solo) e anche su campi che non identificano le tuple di una relazione. Per semplicità continueremo a far riferimento soprattutto a indici su un solo campo.

- Un indice primario, grazie all'ordinamento, può essere realizzato "puntando" a un solo record per ciascun blocco del file, in quanto gli altri record sono posti in modo adiacente a quello puntato; si può scegliere in un indice se puntare agli elementi minimi o massimi dei blocchi (negli esempi si usa il valore minimo). L'indice viene detto in questo caso *sparso* in quanto esistono valori della chiave che non sono presenti nell'indice. Viceversa, un indice secondario deve per forza contenere riferimenti a tutti i valori della chiave, visto che record con valori consecutivi della chiave possono trovarsi in blocchi ben diversi. Gli indici di questo tipo si dicono *densi*. Le Figure 11.6 e 11.7 mostrano due esempi.

Per quanto riguarda le prestazioni, la caratteristica fondamentale degli indici è in ogni caso (anche senza riferirci alle implementazioni migliori che vedremo nel prossimo paragrafo) quella di permettere ricerche efficienti, per esempio binarie. Inoltre, gli indici sono di solito molto più piccoli dei file cui fanno riferimento (perché i loro record contengono solo chiavi e indirizzi) e quindi le ricerche su di essi possono essere più efficienti (per esempio, bastano poche pagine del buffer per caricare l'intero indice, oppure le sue parti più usate). Poiché gli indici sono ordinati, essi rendono efficienti anche le ricerche basate su intervalli e le scansioni sequenziali ordinate, cioè i due

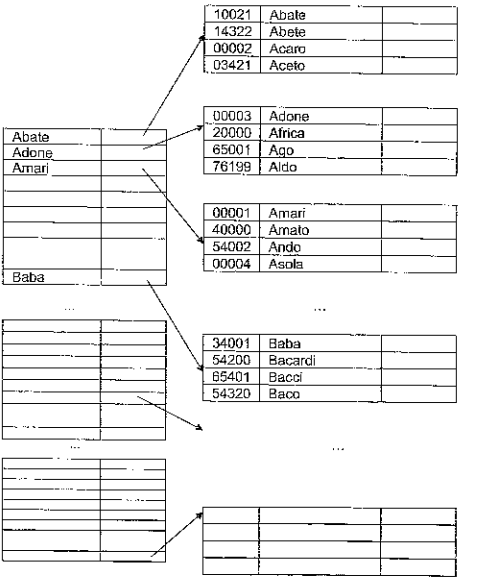

Figura 11.6 Un indice primario sparso.

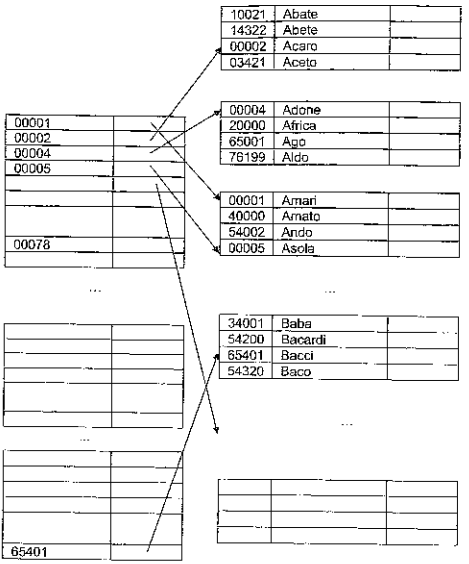

Figura 11.7 Un indice secondario denso.

tipi di operazioni su cui le strutture hash sono inefficienti. D'altra parte, negli accessi puntuali, le strutture hash sono insuperabili: con una forte approssimazione, possiamo dire che una struttura hash ha un tempo di accesso costante (pari mediamente a poco più del tempo necessario per accedere a un blocco), mentre gli indici, che come vedremo sono realizzati tramite alberi, richiedono un tempo logaritmico in funzione del numero di blocchi.[8]

11.4.2 Strutture ad albero dinamiche

Le strutture ad albero dinamiche (cioè efficienti anche in presenza di aggiornamenti), di tipo *B* (*B-tree*) oppure *B+* (*B+-tree*), sono le più frequentemente usate nei DBMS relazionali per la realizzazione di indici.

[8] In effetti, la sempre maggiore disponibilità di memoria centrale permette di avere buffer molto grandi, che possono contenere significative porzioni di indici; di conseguenza, la differenza di costo fra accesso via indice e accesso hash si riduce sempre più.

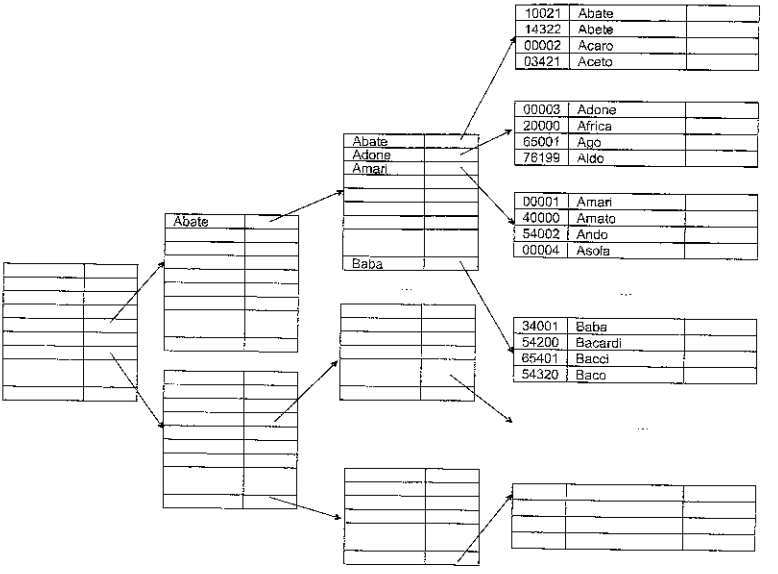

Figura 11.8 Un indice multilivello.

Ogni albero è caratterizzato da un nodo radice, vari nodi intermedi e vari nodi foglia; ogni nodo coincide con una pagina o blocco a livello di file system e di gestore del buffer. I legami tra nodi vengono stabiliti da puntatori che collegano fra loro le pagine; in genere, ogni nodo ha un numero di discendenti abbastanza grande, che dipende dall'ampiezza della pagina (non è raro il caso di alberi in cui ogni nodo ha decine o addirittura centinaia di successori); questo consente di costruire alberi con un numero limitato di *livelli*, nei quali la maggioranza delle pagine è occupata da nodi foglia. Un altro requisito importante per il buon funzionamento di queste strutture dati è che gli alberi siano *bilanciati*, cioè che la lunghezza di un cammino che collega il nodo radice a un qualunque nodo foglia sia costante; in tal caso, il tempo di accesso alle informazioni contenute nell'albero è lo stesso per tutte le foglie ed è pari alla profondità dell'albero. La Figura 11.8 illustra un esempio di indice.

Contenuto dei nodi e tecnica di ricerca La struttura tipica di un nodo non foglia di un albero è illustrata in Figura 11.9. Esso presenta una sequenza di F valori ordinati di chiave. Ogni chiave K_i, $1 \leq i \leq F$, è seguita da un puntatore

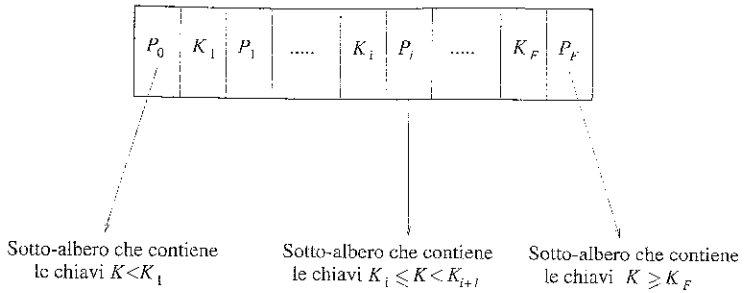

Figura 11.9 Informazione contenuta in un nodo (pagina) di un albero B+.

P_i; K_1 è preceduta da un puntatore P_0. Ciascun puntatore indirizza un sotto-albero, così caratterizzato:

- il puntatore P_0 indirizza al sotto-albero che premette di accedere ai record con chiavi minori di K_1;
- il puntatore P_F indirizza al sotto-albero che premette di accedere ai record con chiavi maggiori o uguali a K_F;
- ciascun puntatore intermedio P_i, $0 < i < F$, indirizza un sotto-albero che contiene chiavi comprese nell'intervallo $[K_i, K_{i+1})$.

In sintesi, ciascun nodo contiene F valori di chiave e $F + 1$ puntatori; il valore $F + 1$ viene detto *fan-out* dell'albero. F dipende dall'ampiezza della pagina e dalla dimensione occupata dai valori di chiave e di puntatori nella "parte utile" di una pagina; si sceglie per F il valore massimo possibile, in modo da ridurre il numero di livelli dell'albero. Peraltro, i vari nodi possono contenere meno di F valori e meno di $F + 1$ puntatori, perché, come vedremo, è necessario prevedere un riempimento parziale e flessibile.

La tipica primitiva di ricerca che viene messa a disposizione dal gestore degli alberi consente un accesso associativo alla tupla o alle tuple che contengono un certo valore di chiave V. Il meccanismo di ricerca consiste nel seguire i puntatori partendo dalla radice. A ogni nodo intermedio:

- se $V < K_1$ si segue il puntatore P_0;
- se $V \geq K_F$ si segue il puntatore P_F;
- altrimenti, si segue il puntatore P_j tale che $K_j \leq V < K_{j+1}$.

La ricerca prosegue in questo modo fino ai nodi foglia dell'albero, che possono essere organizzati in due modi diversi.

- Nel caso di indice primario nel senso stretto del termine, i nodi foglia contengono l'intera tupla. La struttura dati che si ottiene in questo caso è detta *index-sequential*

(o *index-organized table*) e permette di realizzare un file ordinato insieme al suo indice primario; in essa, la posizione di una tupla è vincolata dal valore assunto dal suo campo chiave. Tuttavia, come vedremo, è abbastanza facile inserire o cancellare tuple da questa struttura, in quanto la posizione può variare dinamicamente tramite meccanismi basati sull'uso di puntatori (a prezzo però della perdita di contiguità).

- Nel caso di indice secondario (o primario ma separato dal file), ciascun nodo foglia contiene puntatori ai blocchi della base di dati che contengono tuple con il valore di chiave specificato. La struttura dati che si ottiene in questo caso è detta *indiretta*: il posizionamento delle tuple del file può essere qualsiasi, quindi in particolare questo meccanismo consente di indirizzare tuple allocate tramite un qualunque altro meccanismo "primario" (oltre alle strutture a indice primario, anche strutture sequenziali o ad accesso calcolato). È opportuno anche osservare che, come già detto, un indice primario separato dal file può essere sparso.

Sulla base di quanto detto finora, possiamo sintetizzare alcune considerazioni quantitative sulla profondità degli indici e sul costo dell'accesso puntuale (ricerca di un record sulla base del valore della pseudochiave), per un file con T record di lunghezza L, con pseudochiave di lunghezza k, in un sistema in cui i blocchi hanno dimensione B e i puntatori lunghezza p.

- Indice primario propriamente detto: le foglie contengono i record del file e quindi le foglie sono $T/(B/L)$; l'indice ha complessivamente un numero di livelli pari al logaritmo (in base pari al fan-out e arrotondato all'intero superiore) di $T/(B/L)$. Un accesso puntuale richiede quindi un numero logaritmico di accessi a blocchi.
- Indice primario separato dal file: il file ha $T/(B/L)$ blocchi e l'indice ha un record nelle foglie per ogni blocco del file e così l'albero ha un numero di foglie pari a $T/(B/L)$ diviso per il fan-out e quindi ha un livello in meno del caso precedente. In totale abbiamo lo stesso numero di accessi, perché dopo l'albero si deve accedere ai blocchi del file.
- Indice secondario: l'indice ha un record nelle foglie per ogni record del file[9] e quindi ha $T/(B/(k+p))$ foglie. Ne segue che la profondità dell'albero è pari al logaritmo di $T/(B/(k+p))$ e quindi il numero di accessi a blocchi è in questo caso pari a tale logaritmo più uno (l'accesso al blocco che contiene il record cercato). Nel caso di campo pseudochiave non identificante, se interessa accedere a tutti i record, gli accessi sono in numero maggiore, perché, in linea di principio, ogni record potrebbe trovarsi in un blocco diverso: perciò, se mediamente un valore della pseudochiave si ripete i volte (cioè ci sono i record con tale valore), il numero di accessi sarà pari a i più il logaritmo di $T/(B/(k+p))$. Si noti che questa considerazione non si applica all'indice primario, perché in tal caso i record sono tutti nello stesso blocco (a meno che non siano molti, ma in tal caso il costo è legato alla quantità di dati e non alla ricerca).

[9] Nel caso di pseudochiave non identificante possono essere utilizzate rappresentazioni alternative: per esempio, anziché avere T coppie (valore, puntatore) con ripetizioni sul valore, si possono avere i valori senza ripetizione e liste di puntatori. Per semplicità ignoriamo queste varianti.

Gli inserimenti e le cancellazioni di tuple provocano anche aggiornamenti degli indici, che devono riflettere la situazione generata da una variazione dei valori del campo chiave. Un *inserimento* non provoca problemi quando è possibile inserire il nuovo valore della chiave in una foglia dell'albero, se la pagina ha spazio disponibile; in tal caso, l'indice rimane inalterato e il nuovo valore di chiave viene ritrovato semplicemente applicando l'algoritmo di ricerca. Quando invece la pagina della foglia non ha spazio disponibile, si rende necessaria un'operazione di *split*, che suddivide l'informazione già presente nella foglia e la nuova informazione in due, in modo equilibrato, allocando due foglie al posto di una. Tale operazione richiede una modifica della disposizione dei puntatori, illustrata in Figura 11.10a. Si noti che uno split causa il crescere di una unità dei puntatori al livello superiore dell'albero e in questo modo può provocare una seconda volta il superamento della capacità di una pagina, causando un ulteriore split. In pratica, lo split può propagarsi ricorsivamente verso l'alto fino a raggiungere la radice dell'albero; in casi estremi esso può provocare un aumento di profondità dell'albero. Infatti, se la radice si satura, essa viene divisa in due e viene allocato un nuovo nodo radice, di livello più alto.

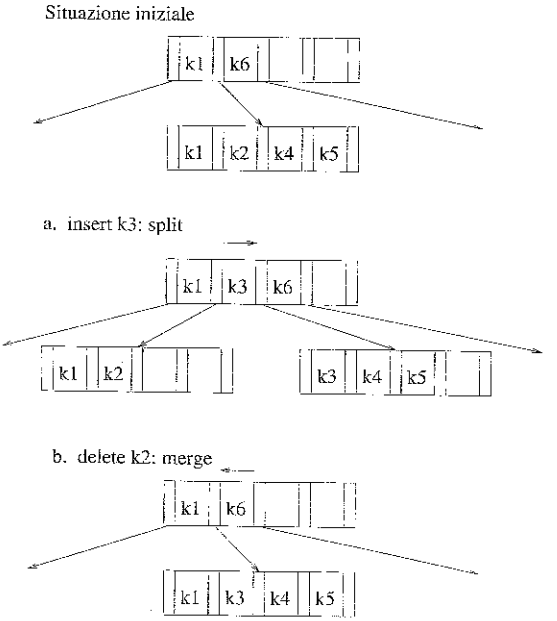

Figura 11.10 Operazioni di split e merge su una struttura ad albero B+.

Una *cancellazione* può essere sempre fatta in loco, marcando lo spazio precedentemente allocato a una tupla come invalido. Vi sono però due problemi.

- Quando, in un indice sparso, la cancellazione coinvolge uno dei valori di chiave presenti nell'albero, è opportuno (anche se non strettamente necessario) recuperare il successivo valore dalla base di dati e introdurlo al posto del valore cancellato. In questo modo, tutti i valori presenti nell'albero appartengono anche alla base di dati.
- Quando la cancellazione lascia due pagine contigue al livello foglia così inutilizzate da consentire che tutta l'informazione in esse presente venga concentrata in un'unica pagina, si può svolgere l'operazione di *merge*, duale all'operazione di *split*, che riunisce tutta l'informazione di due pagine in una sola pagina. Tale operazione richiede una modifica della disposizione dei puntatori, illustrata in Figura 11.10b. Si noti che un merge causa il decrescere di un'unità dei puntatori al livello superiore dell'albero, e in questo modo può provocare un ulteriore merge. In pratica, come nel caso dello split, il merge può propagarsi ricorsivamente verso l'alto fino a raggiungere la radice dell'albero.

La modifica del valore di un campo chiave viene trattata come una cancellazione del suo valore iniziale seguito da un inserimento del suo valore finale, in modo da fare ricadere questo caso in uno dei due casi precedenti.

L'uso attento delle operazioni di split e merge consente di mantenere un riempimento medio di ciascun nodo superiore al 50%; in effetti, valutazioni statistiche indicano il valore medio di riempimento pari a circa il 70%, sotto le opportune ipotesi di implementazione delle operazioni di split e merge.

Alberi B e B+ Esistono due versioni della struttura che abbiamo appena descritto, denominate B e B+. Tale distinzione è semplice: negli alberi B+, i nodi foglia sono collegati da una catena che li connette in base all'ordine imposto dalla chiave, come illustrato nella Figura 11.11. Tale catena consente di svolgere in modo efficiente anche interrogazioni il cui predicato di selezione definisce un *intervallo* di valori ammissibili. In tal caso, infatti, è sufficiente accedere al primo valore dell'intervallo (con una normale ricerca), per poi scandire sequenzialmente i nodi foglia dell'albero fino a trovare valori di chiave maggiori del secondo valore dell'intervallo; la risposta sarà costituita nel caso index sequential da tutte le tuple così ritrovate, mentre nel caso indiretto sarà necessario accedere a tutte le tuple tramite i puntatori così selezionati. In particolare, questa struttura dati consente anche una scansione ordinata in base ai valori di chiave dell'intero file, che risulta abbastanza efficiente. Per questa sua versatilità, nei DBMS è maggiormente usata la struttura B+-tree.

Nelle strutture B-tree non viene previsto di collegare sequenzialmente i nodi foglia. In tal caso, una ottimizzazione possibile prevede di usare nei nodi intermedi due puntatori per ogni valore di chiave K_i; uno dei due puntatori viene utilizzato per puntare direttamente al blocco che contiene la tupla corrispondente a K_i, interrompendo la ricerca; l'altro puntatore serve per proseguire la ricerca nel sotto-albero che comprende i valori di chiave strettamente compresi fra K_i e K_{i+1}, come illustrato in Figura 11.12. Il primo puntatore P_0 individua il sotto-albero corrispondente a va-

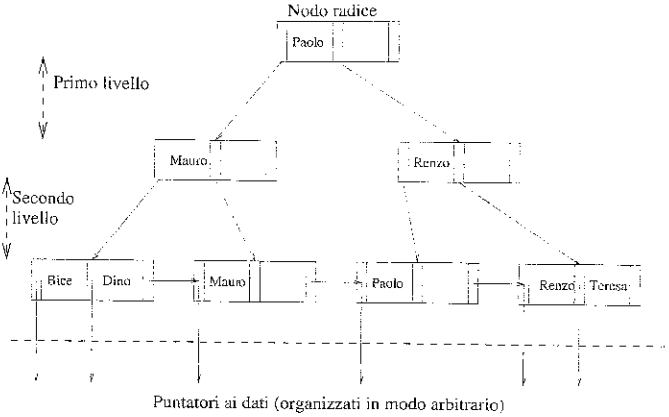

Figura 11.11 Esempio di struttura B+-tree.

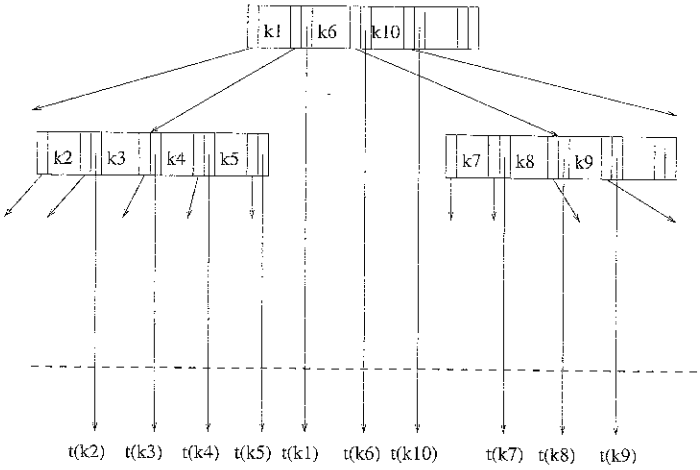

Figura 11.12 Esempio di struttura B-tree.

lori di chiave strettamente inferiori a K_1, mentre l'ultimo puntatore P_T individua il sotto-albero corrispondente a valori di chiave strettamente superiori a K_T.

Questa tecnica fa risparmiare spazio nelle pagine dell'indice, in quanto ciascun valore di chiave è presente al più una volta nell'albero, mentre con gli alberi B+ tutti i valori sono presenti nelle foglie e, quindi, i valori presenti nei livelli superiori vengono ripetuti. Inoltre, con gli alberi B è possibile talvolta terminare la ricerca senza dover percorrere tutti i livelli, tutte le volte che si incontra il valore di chiave cercato in un nodo intermedio.

L'efficienza di un albero B o B+ è normalmente elevata, in quanto spesso le pagine che memorizzano i primi livelli dell'albero risiedono nel buffer perché utilizzate molto frequentemente; questo però può ingenerare un altro problema, e cioè che le pagine contenenti i primi livelli di un indice possano divenire risorse delicate ai fini della concorrenza (problema di cui discuteremo nel prossimo capitolo). Per fortuna, queste pagine sono di solito solo lette e quindi è possibile per vari programmi accedere a esse in modo condiviso (si veda Paragrafo 12.2.4), senza necessità di attese.

Una ottimizzazione dello spazio occupato avviene tramite la compressione dei valori di chiave, per esempio mantenendo solo i loro prefissi nei livelli alti dell'albero, ove si svolge la fase iniziale della ricerca, e solo i loro suffissi, a pari prefisso, nei livelli bassi dell'albero, ove si svolge la parte finale della ricerca.

11.5 Strutture fisiche e indici nei DBMS relazionali

I DBMS attualmente sul mercato differiscono in modo significativo per i dettagli sulle strutture fisiche che mettono a disposizione, soprattutto per quanto riguarda i parametri quantitativi. Al tempo stesso, però, è possibile individuare alcuni principi generali su cui si basano più o meno tutti.

In particolare, tutti i sistemi prevedono una struttura base che è disordinata (e quindi la struttura seriale) su cui è possibile definire indici secondari. Peraltro, quasi tutti i sistemi creano un indice per la chiave primaria, perché in questo modo risulta semplice verificare il rispetto del vincolo (e in generale non esistono altre tecniche efficienti allo scopo). Si noti che alcuni sistemi utilizzano il termine *indice primario* per fare riferimento a un indice definito sulla chiave primaria; si tratta quindi di una terminologia diversa da quella qui adottata secondo cui un indice è primario se realizzato sul campo su cui il file è ordinato.

Molti sistemi prevedono la possibilità di memorizzare in modo contiguo le tuple di una tabella con gli stessi valori su un certo campo. Questa tecnica va di solito sotto il nome di *cluster*, e si presenta in modo diverso nei vari sistemi. In alcuni casi si tratta di vero e proprio ordinamento fisico, mentre in altri casi è associata a una funzione hash oppure a un indice. Alcuni sistemi prevedono la possibilità di definire cluster che coinvolgono più relazioni: in questo modo il join è in un certo senso "preparato," perché le tuple delle due relazioni con valori uguali sul campo di join sono memorizzate insieme ed è quindi sufficiente una semplice scansione per realizzarlo.

Tutti i sistemi permettono di realizzare indici e praticamente tutti prevedono realizzazioni basate su B-tree (o meglio, su B+-tree, anche se la documentazione parla spesso di B-tree), mentre alcuni sistemi prevedono anche altri tipi di indici, per esempio quelli di tipo bitmap, di cui si parla nel secondo volume, perché particolarmente interessanti nell'ambito dei data warehouse. Un'altra variante interessante è il cosiddetto *indice hash*, che realizza una struttura secondaria basata sull'accesso calcolato: i valori della funzione hash vengono usati per determinare l'indirizzo di un blocco in cui non compare il record con il valore della chiave cercato, bensì un riferimento ad esso.

Per concludere questa breve panoramica sulle strutture fisiche nei DBMS, vediamo brevemente quali comandi sono messi a disposizione nei sistemi relazionali per la creazione e cancellazione degli indici. Questi comandi non fanno parte dello standard SQL-3 del linguaggio (né delle edizioni precedenti), per due motivi; in primo luogo non si è raggiunto un accordo all'interno del comitato di standardizzazione, in secondo luogo gli indici sono ritenuti un aspetto strettamente legato all'implementazione del sistema e difficilmente uniformabile. La sintassi che illustreremo è comunque utilizzata dai sistemi commerciali di maggiore diffusione.

La sintassi del comando per la creazione di un indice è:

`create` [`unique`] `index` *NomeIndice* `on` *NomeTabella* (*ListaAttributi*)

Con questo comando si crea un indice di nome *NomeIndice* sulla tabella *NomeTabella*, operante sugli attributi elencati in *ListaAttributi*. L'ordine in cui compaiono gli attributi nella lista è significativo: le chiavi dell'indice vengono infatti ordinate prima in base ai valori del primo attributo della lista, poi a pari valore del primo attributo si usano i valori del secondo attributo, e così in sequenza fino all'ultimo attributo. L'uso della parola `unique` specifica che nella tabella non sono ammesse tuple che concordino su tutti gli attributi dell'indice (in altri termini, essa specifica che gli attributi in questione formano una superchiave per la tabella, come discusso nel Capitolo 2[10]).

Per eliminare un indice si usa invece il comando di `drop index`, caratterizzato da una sintassi estremamente semplice:

`drop index` *NomeIndice*

Per esemplificare l'uso dei comandi appena visti, possiamo specificare un indice sulla tabella IMPIEGATO, che permette di accedere in modo efficiente ai dati dell'impiegato avendo a disposizione il cognome e la città:

```
create index NomeCittaIdx on Impiegato(Cognome,Citta)
```

Per eliminare l'indice, si darà il comando:

```
drop index NomeCittaIdx
```

[10] L'uso della parola chiave `unique` e la conseguente possibilità di definire superchiavi sono un difetto del linguaggio, risultato di una scelta infelice che porta a confondere un aspetto logico, quello di chiave, con uno fisico, quello di indice.

11.6 Gestore delle interrogazioni: esecuzione e ottimizzazione

Il gestore delle interrogazioni è un modulo cruciale dell'architettura di un DBMS, in quanto responsabile dell'esecuzione efficiente di operazioni che sono specificate a livello molto alto (e sono quindi potenzialmente molto complesse). Esso riceve in ingresso un'interrogazione scritta in SQL, passata tramite un qualunque meccanismo dell'interfaccia applicativa (per esempio, tramite l'ambiente interattivo oppure programmi con SQL embedded). L'interrogazione viene inizialmente analizzata per determinare eventuali suoi errori lessicali, sintattici o semantici, che vengono opportunamente segnalati. Durante questa fase, il sistema accede al dizionario dei dati (si veda il Paragrafo 4.2.9) per leggere l'informazione ivi contenuta e consentire così i controlli; dal dizionario dei dati vengono lette anche informazioni statistiche relative alle dimensioni delle tabelle. Una volta accettata, l'interrogazione viene tradotta in una forma interna di tipo algebrico. A questo punto, l'ottimizzazione vera e propria ha inizio. Essa si compone delle seguenti fasi:

- Innanzitutto viene svolta un'ottimizzazione di tipo algebrico, che consiste nell'effettuare tutte le trasformazioni algebriche che sono sempre convenienti, quali per esempio il "push" delle selezioni e delle proiezioni, come descritto nel Paragrafo 3.1.7. Questa ottimizzazione, di tipo logico, avviene indipendentemente dal modello dei costi assunto nel sistema.
- Successivamente, viene svolta un'ottimizzazione che dipende sia dalla tipologia dei metodi di accesso ai dati supportati dal sottostante livello sia dal modello dei costi assunto. Per questa fase, pur essendo l'ottimizzazione delle interrogazioni assai ben definita e frutto di tecniche consolidate (specie nel contesto dei sistemi relazionali), ciascun sistema presenta di fatto caratteristiche peculiari.
- Infine, avviene la generazione del codice, che utilizza i metodi di accesso ai dati. Si ottengono cioè dei *programmi di accesso* in formato "oggetto" o "interno" che richiedono l'uso delle strutture dati fornite dal sistema (tra cui, per esempio, gli indici).

Il processo di ottimizzazione di un'interrogazione è illustrato in Figura 11.13. Si noti che, a differenza di molti altri moduli di sistema, in particolare quelli descritti nel prossimo capitolo (e finalizzati alla gestione delle transazioni), l'ottimizzatore agisce a tempo di compilazione. Nel caso comune in cui l'interrogazione venga compilata una volta ed eseguita molteplici volte (approccio *compile-and-store* o *prepare-execute*, si veda il Capitolo 10), il codice viene prodotto e memorizzato nella base di dati, assieme a un'indicazione delle *dipendenze* del codice dalle particolari versioni di tabelle e indici della base di dati, descritte nel dizionario dei dati. In tal modo, se la base di dati cambia in modo significativo per l'interrogazione (per esempio perché viene aggiunto un indice), la compilazione dell'interrogazione viene invalidata e ripetuta. Talvolta, invece, un'interrogazione viene compilata e immediatamente eseguita (approccio *compile-and-go* o *execute-immediate*), senza essere memorizzata.

Nel seguito ci concentriamo sulla fase centrale di questo processo, relativa all'ottimizzazione dipendente dai costi; dato che questa parte dell'ottimizzazione dipende

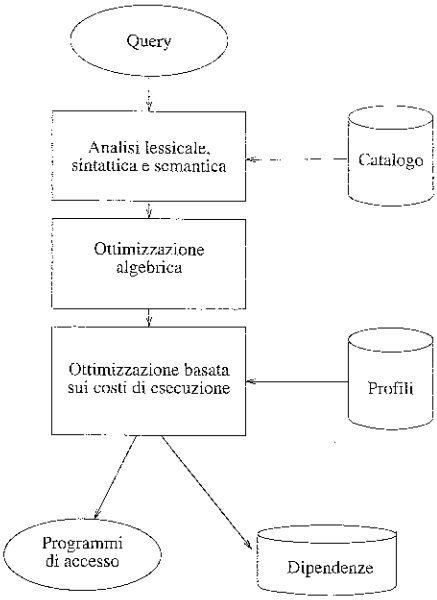

Figura 11.13 Compilazione di una interrogazione.

in modo specifico dalle strutture di memorizzazione e dal modello di costi prescelto per un DBMS, potremo darne solo una descrizione qualitativa e approssimata. Premessa a questa fase è che l'ottimizzazione algebrica abbia prodotto una descrizione ottimizzata dell'interrogazione, in cui tutte le naturali trasformazioni algebriche siano state realizzate. Il risultato di tale lavoro rappresenta ogni interrogazione SQL sotto forma di un albero, i cui nodi foglia rappresentano tabelle e i cui nodi intermedi rappresentano operazioni dell'algebra relazionale.

11.6.1 Profili delle relazioni

Ciascun DBMS commerciale possiede informazioni quantitative relative alle caratteristiche delle tabelle, organizzate in strutture dati, dette *profili delle relazioni*, che vengono memorizzate nel dizionario dei dati. I profili contengono alcune delle seguenti informazioni:

- la *cardinalità* CARD(T) (numero di tuple) di ciascuna tabella T;
- la *dimensione* in byte, SIZE(T), di ciascuna tupla di T;

- la *dimensione* in byte, SIZE(A_j, T), di ciascun attributo A_j di T;
- il *numero di valori distinti*, VAL(A_j, T), di ciascun attributo A_j di T;
- il valore *minimo* MIN(A_j, T) e quello *massimo* MAX(A_j, T) di ciascun attributo A_j di T.

I profili vengono calcolati in base ai dati effettivamente memorizzati nelle tabelle, utilizzando opportune primitive di sistema (per esempio, la primitiva `update statistics`); è compito dell'amministratore della base di dati richiamare periodicamente queste primitive. Normalmente viene esclusa, perché troppo onerosa, la possibilità di tenere aggiornati i profili durante la normale esecuzione delle transazioni. In genere, è sufficiente che i profili contengano valori approssimati, in quanto comunque i modelli statistici che si applicano loro sono anch'essi approssimati.

L'ottimizzazione dipendente dai costi richiede di formulare ipotesi sulle dimensioni dei risultati intermedi prodotti dalla valutazione di operazioni algebriche con un approccio di tipo statistico. Per esempio, vediamo i profili relativi alle operazioni algebriche più classiche: selezione, proiezione e join.

Profili delle selezioni Il profilo di una tabella T' prodotta da una *selezione* $T' = \sigma_{A_i=v}(T)$ è ottenuto tramite le seguenti formule, la cui giustificazione è lasciata come esercizio:

1. CARD(T') = (1/VAL(A_i, T))×CARD(T);
2. SIZE(T') = SIZE(T);
3. VAL(A_i, T') = 1;
4. VAL(A_j, T') = col(CARD(T),VAL(A_j, T),CARD(T')), $j \neq i$;[11]
5. MAX(A_i, T') = MIN(A_i, T') = v;
6. MAX(A_j, T') e MIN(A_j, T') (per $j \neq i$) hanno rispettivamente gli stessi valori di MAX(A_j, T) e MIN(A_j, T).

Profili delle proiezioni Il profilo di una tabella T' prodotta da una *proiezione* $T' = \pi_L(T)$, ove L è l'insieme di attributi $A_1, A_2, \ldots A_n$, è ottenuto tramite le seguenti formule:

1. CARD(T') = MIN(CARD(T), $\prod_{i=1}^{n}$VAL(A_i, T));
2. SIZE(T') = $\sum_{i=1}^{n}$SIZE(A_i, T);
3. VAL(A_i, T'), MAX(A_i, T'), MIN(A_i, T') hanno rispettivamente gli stessi valori di VAL(A_i, T), MAX(A_i, T), MIN(A_i, T).

---

[11] La formula $col(n, m, k)$ relativa a VAL(A_j, T'), calcola il numero di colori distinti presenti in k oggetti estratti a partire da n oggetti di m colori distinti, distribuiti omogeneamente: ciascun colore rappresenta uno dei diversi valori presenti nell'attributo A_j. Tale formula ammette la seguente approssimazione:

(a) $col(n, m, k) = k$ se $k \leq m/2$;
(b) $col(n, m, k) = (k + m)/3$ se $m/2 \leq k \leq 2m$;
(c) $col(n, m, k) = m$ se $k \geq 2m$.

Profili dei join Il profilo di una tabella T^J prodotta da un *equi-join* $T^J = T' \bowtie_{A=B} T''$, assumendo che A e B abbiano domini identici e in particolare $\text{VAL}(A, T') = \text{VAL}(B, T'')$, è ottenuto tramite le seguenti formule:

1. $\text{CARD}(T^J) = (1/\text{VAL}(A, T')) \times \text{CARD}(T') \times \text{CARD}(T'')$;
2. $\text{SIZE}(T^J) = \text{SIZE}(T') + \text{SIZE}(T'')$;
3. $\text{VAL}(A_i, T^J)$, $\text{MAX}(A_i, T^J)$, $\text{MIN}(A_i, T^J)$ hanno i valori che assumevano nelle loro rispettive tabelle prima di effettuare il join.

Le formule precedenti mostrano i limiti di questo tipo di analisi statistica. Per esempio, tutte le formule assumono una distribuzione uniforme dei dati nelle tabelle e una assenza di correlazione tra le varie condizioni presenti in un'interrogazione. Si noti che spesso le formule assegnano al risultato di un'operazione parametri identici a quelli dei loro operandi (per esempio, per quanto riguarda i valori minimi e massimi di un certo attributo) perché non è possibile prevedere meglio gli effetti dell'operazione stessa. Comunque, questa analisi è in grado di definire, anche approssimativamente, l'ordine di grandezza delle dimensioni dei risultati intermedi (per esempio, il numero di pagine occupate); questi dati quantitativi sono sufficienti a svolgere l'ottimizzazione.

11.6.2 Rappresentazione interna delle interrogazioni

La rappresentazione che viene data dall'ottimizzatore a un'interrogazione tiene conto della struttura fisica utilizzata per implementare le tabelle, nonché degli indici disponibili su di esse. Pertanto, la prima trasformazione consiste nel cambiare i nodi foglia con nodi che tengono conto delle strutture fisiche. Successivamente, i nodi intermedi vengono trasformati in operazioni di accesso ai dati che sono supportate sulle strutture fisiche. Tipicamente, le operazioni supportate dai DBMS relazionali comprendono le scansioni sequenziali, gli ordinamenti, gli accessi diretti e varie modalità di join.

Operazioni di scansione Un'operazione di *scansione* (*scan*) opera contestualmente varie operazioni di tipo algebrico ed extra-algebrico:

- proiezione su di una lista di attributi (senza eliminazione dei duplicati, che richiede un ordinamento);
- selezione su di un predicato;
- inserimenti, cancellazioni e modifiche delle tuple quando vi si fa accesso durante la scansione.

Ordinamenti La necessità di operazioni di ordinamento emerge sia ai fini delle applicazioni, perché si desiderano risultati ordinati, sia per una corretta realizzazione delle proiezioni, con eliminazione dei duplicati. Inoltre, essa può essere utile ai fini di successive operazioni, per esempio di join, come vedremo fra poco, oppure di raggruppamento. Peraltro, talvolta l'ordinamento non è necessario, perché alcune operazioni, per esempio alcune implementazioni del join, producono risultati ordinati.

Il problema di ordinare strutture dati è un problema classico della teoria degli algoritmi; vari metodi, descritti in letteratura, consentono di ottenere prestazioni ottimali per ordinare dati contenuti in memoria centrale, tipicamente rappresentati tramite array di record. Le tecniche di ordinamento dei dati utilizzate dai DBMS utilizzano varianti di questi algoritmi che tengono conto delle caratteristiche della memoria secondaria (con i costi calcolati rispetto al numero di blocchi acceduti) e alla disponibilità di buffer. Infatti, in presenza di buffer molto grandi, è possibile utilizzare algoritmi molto efficienti che effettuano un ordinamento accedendo a ciascun dato una o al massimo due volte. In particolare, si può utilizzare una variante del merge-sort che ordina, inizialmente, porzioni di tabella pari alla dimensione del buffer e poi procede alla fusione di tante porzioni quante sono le pagine disponibili del buffer (anziché due sole porzioni come nel merge-sort tradizionale). In questo modo, si può ordinare, con due sole passate, una tabella con un numero di blocchi pari al quadrato del numero di pagine di buffer disponibili (e in N passate una tabella con numero di blocchi pari alla N-esima potenza della dimensione del buffer).

In presenza di un indice secondario sul campo di ordinamento, l'ordinamento può essere realizzato anche attraverso una scansione delle foglie dell'indice stesso.

Accesso diretto Si usa il termine *accesso diretto* quando è possibile leggere o scrivere un record senza dover necessariamente esaminare il file in modo sequenziale, ma è possibile ottenere l'indirizzo del blocco in cui il record si trova in altro modo, a partire dal valore di un campo. Per eseguire l'accesso diretto, è necessaria la presenza di una struttura, hash o indice, che lo permetta. Gli indici, come abbiamo visto, favoriscono le interrogazioni che richiedono accessi puntuali (del tipo $A_i = V$) oppure a intervallo (del tipo $V_1 \leq A_i \leq V_2$). Si dice in tal caso che un predicato dell'interrogazione è *valutabile* tramite l'indice. Le strutture hash sono ancora più efficienti per gli accessi puntuali, ma non supportano le ricerche per intervallo.

In genere, se l'interrogazione presenta un solo predicato valutabile (anche in presenza di altri predicati in congiunzione) c'è convenienza a usare l'indice o la struttura hash. Quando un'interrogazione presenta una *congiunzione* di predicati valutabili tramite indice o funzione hash, il DBMS sceglie il più selettivo dei due per l'accesso diretto; il secondo predicato viene valutato una volta caricate nel buffer le pagine che soddisfano il primo predicato. Quando invece l'interrogazione presenta una *disgiunzione* di predicati, basta che uno di loro sia non valutabile per imporre l'uso di una scansione completa. Se infine si ha una disgiunzione di predicati tutti valutabili, è possibile utilizzare gli indici oppure una scansione; nel caso di uso di indici, però, è necessario eliminare i duplicati di quelle tuple che vengono ritrovate tramite più indici.

Infine, occorre tenere presente che l'uso dell'indice richiede molteplici accessi per ogni tupla individuata; quando l'interrogazione è poco selettiva, l'uso dell'indice può essere dominato da una semplice scansione.

Metodo di join Il join è l'operazione più gravosa per un DBMS, in quanto è presente il rischio di un'esplosione del numero di tuple del risultato: un join su campi che non siano chiave per nessuno dei due operandi può avere un numero di tuple

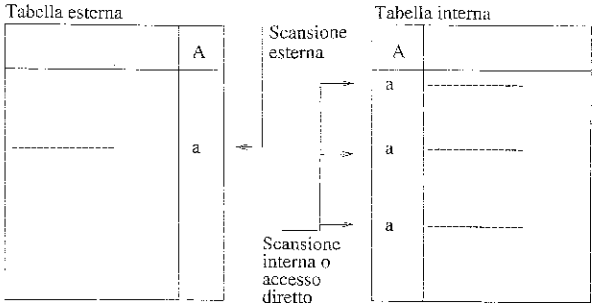

Figura 11.14 Tecnica di join nested loop.

paragonabile al prodotto delle cardinalità degli operandi. Pertanto, non stupisce che la tecnologia dei DBMS abbia prodotto vari algoritmi per la realizzazione dei join, e che definire l'ordine e la modalità di realizzazione delle operazioni di join abbia un ruolo centrale nell'ottimizzazione globale delle interrogazioni. Solo in tempi recenti, con il crescere dell'interesse per le operazioni aggregate, analoghi algoritmi e approcci quantitativi sono stati dedicati alle operazioni aggregate e al raggruppamento. Nel seguito, vediamo tre tecniche per la realizzazione dei join, denominate *nested loop*, *merge scan*, e *hash-based*.

- *Nested loop*. Nel nested loop una tabella viene definita come *esterna* e l'altra come *interna*. Si esegue una scansione sulla tabella esterna; per ogni tupla ritrovata dalla scansione, si preleva il valore dell'attributo di join e si cercano le tuple della tabella interna che hanno lo stesso valore. Per questa seconda parte dell'algoritmo è utile una struttura hash o un indice sull'attributo di join della tabella interna, che può essere creato *ad hoc*; altrimenti, è necessario eseguire una scansione sulla tabella interna per ogni valore di join della tabella esterna. A questa tecnica viene dato il nome "nested loop" perché propone una scansione "nidificata" nell'altra. Si noti che questa tecnica ha costi diversi a seconda della attribuzione alle tabelle operandi dei ruoli di interna e esterna e della presenza di strutture per l'accesso diretto nonché della disponibilità di buffer.
- *Merge scan*. Questa tecnica richiede di esaminare le tabelle secondo l'ordine degli attributi di join ed è quindi particolarmente efficiente quando le tabelle sono già ordinate oppure quando sono definiti su di esse indici adeguati. Essa viene eseguita per mezzo di scansioni parallele sulle due tabelle, basate sull'ordinamento, come nei classici algoritmi di fusione (*merge*). Le scansioni possono così ritrovare nelle tuple valori ordinati degli attributi di join; quando coincidono, vengono generate ordinatamente tuple del risultato.
- *Hash-based*. Questo metodo viene eseguito in due passi: in primo luogo, una stessa funzione h di hash sugli attributi di join viene utilizzata per memorizzare una

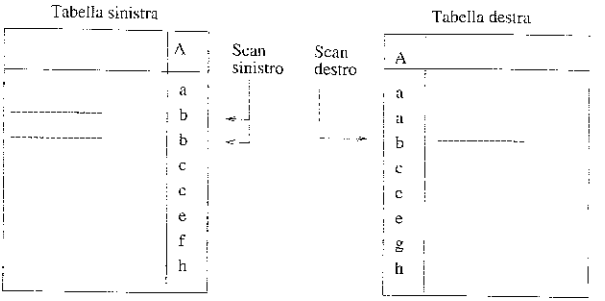

Figura 11.15 Tecnica di join merge scan.

copia di ciascuna delle due tabelle (o almeno di parte delle informazioni di ciascuna tupla). Supponendo che la funzione h faccia corrispondere i valori del dominio di tali attributi a B partizioni su ciascuna tabella, per costruzione le tuple con gli stessi valori nell'attributo di join verranno poste in partizioni di identico indice. Perciò sarà successivamente possibile trovare tutte le tuple risultanti dal join effettuando B semplici join tra le partizioni a pari indice, come illustrato in Figura 11.16. Varie versioni di questo metodo consentono di ottimizzare le prestazioni tramite un'attenta costruzione delle funzioni di hash e un'attenta gestione dei buffer di memoria centrale. Inoltre, segnaliamo la variante che elabora preli-

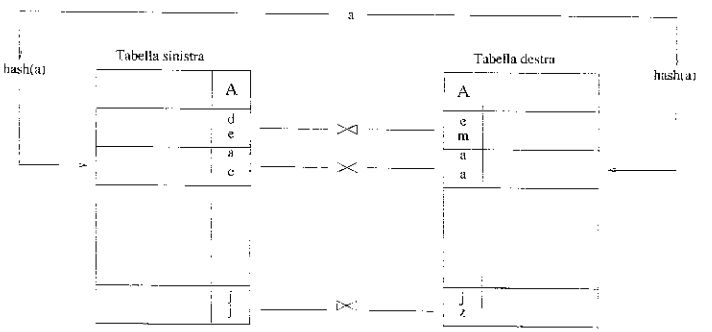

Figura 11.16 Tecnica di join hash-based.

minarmente una sola delle due relazioni, effettuando il join insieme alla scansione della seconda.

È chiaro che ciascuna delle tre tecniche ha un costo che dipende dalla "situazione iniziale" cui si applica (per esempio, la presenza di indici o la possibilità di caricare un operando completamente nei buffer di memoria centrale date le sue ridotte dimensioni), e produce una "condizione finale" (per esempio, il fatto che il risultato si presenti in modo ordinato). Pertanto, il costo di realizzare una qualunque di queste tecniche non può essere valutato in astratto, ma deve essere valutato in funzione delle scelte che precedono o seguono.

Concludiamo questa discussione sull'operatore di join ribadendo che esso viene eseguito in modo efficiente dai DBMS, che cercano di scegliere la migliore fra le varie alternative possibili. Pertanto, è fondamentale, in un'applicazione, delegare al DBMS l'esecuzione del join evitando, come invece talvolta i programmatori inesperti fanno, di simularlo, attraverso l'utilizzo di due scansioni (di cursori o result set) nidificate: in questo caso, infatti, si eseguirebbe un nested loop (per giunta con molte chiamate al DBMS) rinunciando alle altre possibilità disponibili nel sistema.

11.6.3 Ottimizzazione basata sui costi

Vediamo infine come viene svolto l'ultimo passo dell'ottimizzazione, che porta a definire il piano di accesso. Il problema si presenta assai difficile su un piano computazionale, in quanto sono presenti varie dimensioni di ottimizzazione.

- Occorre scegliere, in presenza di alternative, quali operazioni di accesso ai dati svolgere. In particolare, per quanto concerne il primo accesso ai dati, occorre talvolta scegliere fra una scansione e un accesso tramite indici.
- Occorre scegliere l'ordine delle operazioni da compiere (per esempio, l'ordine fra i vari join presenti in un'interrogazione).
- Quando un sistema offre varie alternative per la realizzazione di un'operazione, occorre scegliere quale alternativa associare a ciascuna operazione (per esempio, scegliere il metodo di join).
- Quando l'interrogazione oppure il metodo di realizzazione di un'operazione richiedono un ordinamento, occorre definire a quale livello della strategia svolgere l'operazione di ordinamento.

Di fronte a un problema così complesso, gli ottimizzatori dispongono in genere di formule di costo approssimate. Essi costruiscono un *albero delle alternative*, in cui ogni nodo corrisponde a fissare una particolare opzione fra quelle citate in precedenza. Ovviamente, le dimensioni di tale albero crescono notevolmente in funzione del numero di alternative presenti a ciascun livello. Ogni nodo foglia dell'albero corrisponde a una specifica *strategia di esecuzione* dell'interrogazione, descritta dalle scelte che si trovano percorrendo il cammino che va dalla radice al nodo foglia. Quindi, il problema di ottimizzazione è riformulato nella ricerca del nodo-foglia cui corrisponde il costo minore.

In Figura 11.17 mostriamo le alternative di esecuzione di un'interrogazione congiuntiva (cioè con sole selezioni, proiezioni e join) con tre tabelle e due join, in cui

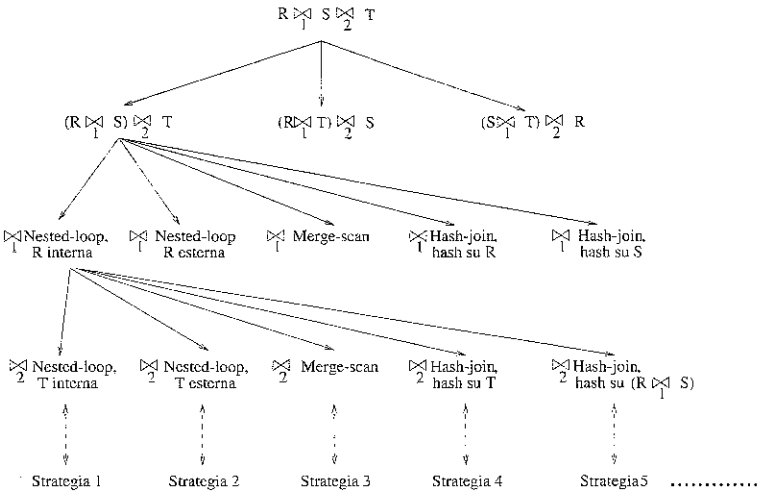

Figura 11.17 Alternative di esecuzione in una interrogazione congiuntiva.

l'ottimizzatore deve decidere solo l'ordinamento e il metodo di join da utilizzare. Sono presenti 3 possibili ordinamenti dei join e 5 possibili modi per svolgere operazioni di join (assumendo un'implementazione asimmetrica per l'hash-join), dando luogo a 75 alternative. Questo semplice esempio è indicativo della complessità del problema nei suoi termini più generali.

Il problema viene tipicamente risolto tramite formule di costo che associano a ciascuna operazione intermedia un costo in termini di operazioni di ingresso/uscita e di istruzioni necessarie per valutare il risultato della interrogazione. In questo modo, è possibile associare a un nodo foglia un costo:

$$C_{total} = C_{I/O} \times n_{I/O} + C_{cpu} \times n_{cpu}$$

ove $C_{I/O}$, C_{cpu} sono parametri noti (costo unitario di un accesso a memoria secondaria e costo unitario di esecuzione di un'istruzione in memoria centrale) e $n_{I/O}$, n_{cpu} indicano il numero di operazioni di ingresso/uscita e il numero di istruzioni necessarie per valutare il risultato dell'interrogazione. Tale costo è dato dalla somma di tutti i costi accumulati ai nodi intermedi per svolgere le operazioni che caratterizzano la strategia. Si noti che molto spesso il costo è dominato dal primo termine e quindi l'unico aspetto da valutare è il numero di accessi a memoria secondaria.

Talvolta, la strategia ottimale richiede la realizzazione *ad hoc* di strutture temporanee, per esempio indici od ordinamenti, e quindi il costo della strategia include il costo di costruzione di tali strutture. I risultati intermedi prodotti vengono spesso allocati nei buffer e consumati subito dopo la loro produzione, sfruttando il *pipelining* delle operazioni, cioè la possibilità di percorrere l'intero albero delle operazioni per una parte delle tuple estratte, invece che procedere a realizzare interamente ciascuna operazione. Talvolta, invece, è necessario riscrivere i risultati delle operazioni intermedie in memoria di massa; in tal caso, fa parte dei costi di una strategia anche il costo di tale riscrittura.

Gli ottimizzatori si accontentano tipicamente di ottenere soluzioni "buone", cioè che rientrino nello stesso "ordine di grandezza prestazionale" rispetto alla soluzione ottima del problema. Essi possono escludere interi sotto-alberi dello spazio delle soluzioni quando il loro costo parziale è peggiore del costo di una strategia globale già individuata; vengono quindi adottate tecniche di ricerca operativa, denominate *branch and bound*, per l'eliminazione esatta o approssimata di sotto-alberi.

Con un approccio "compile-and-go", gli ottimizzatori sono spesso vincolati a operare in tempi ristretti; infatti, i tempi di risposta non possono essere penalizzati dal dedicare troppo tempo alla ottimizzazione. Non ha senso trovare la soluzione ottima in un tempo molto elevato, quando è possibile trovare una soluzione "buona" in un tempo limitato e realizzare questa strategia in un tempo globale (che cioè include il tempo dedicato alla ottimizzazione) inferiore a quello della soluzione ottima. In conclusione, gli ottimizzatori sono moduli assai complessi, da cui dipende gran parte dell'efficienza del DBMS nel valutare interrogazioni.

11.7 Progettazione fisica di una base di dati

Nell'ambito del processo di progettazione di una base di dati, così come lo abbiamo esaminato nei Capitoli 6–8, la fase finale è quella della *progettazione fisica*, che, ricevendo in ingresso lo schema logico della base di dati, le caratteristiche del sistema scelto e le previsioni sul carico applicativo, produce in uscita lo schema fisico della base di dati, costituito dalle effettive definizioni delle relazioni e soprattutto delle strutture fisiche utilizzate, con i relativi parametri.

L'attività di progettazione fisica di una base di dati relazionale può essere molto complessa perché, oltre alle scelte relative alle strutture fisiche (che possono essere compiute in una rosa più o meno ampia, a seconda dei sistemi, come abbiamo visto nel Paragrafo 11.5), può essere necessario definire molti parametri, che vanno dalle dimensioni iniziali dei file alle possibilità di espansione, dalla contiguità di allocazione alla quantità e alle dimensioni dei buffer. Alcuni sistemi mettono a disposizione varie decine di parametri, i cui valori possono risultare significativi ai fini delle prestazioni delle applicazioni. Peraltro, esistono sempre valori standard per questi parametri, che vengono assunti dal sistema se non altrimenti specificati in modo esplicito.

La maggior parte delle scelte da effettuare nel corso della progettazione fisica dipende in effetti dallo specifico sistema di gestione utilizzato, quindi risulta difficile fornire una panoramica completa e di validità generale. Indicheremo solo le linee

principali, che possono però essere considerate sufficienti in presenza di basi di dati di dimensioni non enormi o con carichi di lavoro non particolarmente complessi.

Le scelte fondamentali nella progettazione sono da ricondurre a due:

- scelta della struttura primaria per ciascuna relazione, fra quelle rese disponibili dal DBMS;
- definizione di eventuali indici secondari.

Per orientarci nelle scelte, è opportuno ricordare che, come detto nel Paragrafo 11.6, le operazioni più delicate in una base di dati relazionale sono quelle di selezione (che corrisponde all'accesso a uno o più record sulla base dei valori di uno o più attributi) e di join (che richiede di combinare tuple di relazioni diverse sulla base dei valori di uno o più attributi di ognuna di tali relazioni).

Ciascuna delle due operazioni può essere eseguita in modo molto più efficiente se sui campi interessati è definito un indice (primario o secondario) o una struttura hash, rendendo così possibile un accesso diretto.

Consideriamo per esempio una base di dati su due relazioni: la relazione IMPIEGATO (Matricola, Cognome, Nome, Dipartimento) (nella quale l'attributo Dipartimento indica il codice del dipartimento di afferenza) e la relazione DIPARTIMENTO (Codice, Nome, Direttore).

Supponiamo ora di voler effettuare la ricerca di un impiegato dato il suo numero di matricola, ovvero una selezione sull'attributo Matricola della relazione IMPIEGATO. Se sulla relazione è presente un indice su tale attributo oppure se la relazione ha una struttura primaria hash basata su di esso, allora si può procedere con un accesso diretto, molto efficiente, altrimenti si deve effettuare un accesso sequenziale, con un costo proporzionale alla dimensione del file. Lo stesso vale per una ricerca basata sul cognome dell'impiegato; vale la pena notare che se è definito un indice su un attributo, solo le ricerche basate su tale attributo possono trarne beneficio: se la relazione ha un indice oppure una struttura hash su Matricola e non ha un indice su Cognome, le selezioni su Matricola potranno essere eseguite in modo efficiente, mentre quelle su Cognome rimarranno inefficienti.

Un equi-join fra le due relazioni volto a collegare ciascun impiegato con il corrispondente dipartimento, in presenza di un indice o struttura hash sulla chiave Codice della relazione DIPARTIMENTO, può essere effettuato in modo efficiente tramite il metodo di *nested loop*: si scandisce sequenzialmente la relazione IMPIEGATO (e questo non è un problema, perché tutte le sue tuple contribuiscono al risultato) e, per ciascuna di esse, si effettua un accesso diretto alla relazione DIPARTIMENTO sulla base dell'indice. Se l'indice non è stato definito, l'accesso alla relazione DIPARTIMENTO risulta inefficiente, e tutto il join risulta molto più costoso (sono possibili in alcuni casi miglioramenti con altri algoritmi, ma spesso senza l'indice si ha grande inefficienza).

È importante ricordare come molti dei join che si presentano nelle nostre applicazioni siano equi-join e per almeno una delle due relazioni i campi coinvolti formino una chiave, come nell'esempio appena mostrato. Al tempo stesso, possiamo notare come quasi sempre la chiave di una relazione sia coinvolta in operazioni di selezione o di join (o entrambe).

Possiamo pertanto riassumere le attività di progettazione fisica come segue. Innanzitutto, è ragionevole definire, su ciascuna relazione, un indice in corrispondenza

della relativa chiave primaria; la maggior parte dei DBMS costruisce questo indice automaticamente. Un'alternativa lasciata al progettista è di solito la possibilità di rendere primario tale indice (cioè di avere un'organizzazione ordinata per il file) oppure di sostituirlo con una struttura hash. Infine, per i sistemi che la prevedono, può essere valutata la possibilità di definire cluster multirelazionali. Inoltre, possono essere definiti ulteriori indici su altri campi su cui vengono effettuate operazioni di selezione oppure su cui è richiesto un ordinamento in uscita (perché un indice ordina logicamente i record di un file, rendendo nullo il costo di un ordinamento). Queste osservazioni costituiscono la base per una semplice strategia di progettazione fisica.

La scelta eventuale di strutture primarie particolari, hash o indici primari, può essere motivata da operazioni particolarmente importanti, ma, al tempo stesso, dall'assenza di operazioni che siano da esse penalizzate: abbiamo visto nei paragrafi precedenti che una struttura hash è particolarmente efficiente per accessi puntuali, ma peggiore di una struttura a indice per ricerche su intervalli; le strutture hash e quelle ordinate (associate agli indici primari) possono portare a comportamenti non desiderabili in presenza di aggiornamenti frequenti.

Un approccio sistematico alla definizione delle strutture fisiche può essere basato sul carico applicativo, nel modo seguente. Si supponga di avere un certo insieme di operazioni $o_1, o_2, \ldots, o_n$, ciascuna con la relativa frequenza $f_1, f_2, \ldots, f_n$. Per ogni operazione o_i, è possibile definire il costo c_i (tempo di esecuzione), che può variare, a seconda delle possibili scelte di strutture fisiche. Obiettivo della progettazione fisica è minimizzare il costo complessivo, cioè il costo delle varie operazioni pesato con la relativa frequenza: $\Sigma_{i=1}^{n}(c_i \times f_i)$. Il costo di ciascuna operazione può essere schematizzato con il numero di accessi a memoria secondaria.

In realtà un approccio così sistematico alla progettazione fisica non può essere messo in pratica. Innanzitutto, il costo del singolo accesso può variare, per via della contiguità dei blocchi. Inoltre, non sempre è possibile prevedere con precisione il numero di accessi fisici, a causa dei benefici derivanti dalla gestione dei buffer. Supponendo possibile valutare il costo di ciascuna operazione, l'attività di progettazione risulta comunque difficile da gestire in modo completo, perché le alternative possibili sono molte e non possono certo essere valutate tutte. Infine, resta l'incognita di capire come si comporterà il sistema in pratica; infatti, la scelta delle strategie di esecuzione delle query è fatta da un ottimizzatore, che potrebbe attuare scelte diverse da quelle utilizzate nel modello per valutare i costi di esecuzione.

Pertanto, è opportuno individuare, sulla base delle considerazioni generali sopra citate, le alternative principali, per poi valutare analiticamente i casi specifici che restano incerti e che siano significativi da un punto di vista quantitativo. Infatti, è opportuno sottolineare che è quasi sempre possibile ignorare tanto le operazioni poco frequenti quanto le relazioni più piccole, in quanto in entrambi i casi l'impatto sul costo complessivo sarebbe limitato.

Discutiamo brevemente un semplice esempio per illustrare una possibile procedura. Consideriamo una relazione IMPIEGATO(Matricola, Cognome, Nome, DataNascita) con un numero di tuple pari a $N = 10\,000\,000$ abbastanza stabile nel tempo (pur con inserimenti ed eliminazioni) e una dimensione di ciascuna tupla (a lunghezza fissa) pari a $L = 100$ byte, di cui $K = 2$ byte per la chiave Matricola e $C = 15$ byte per il campo Cognome. Supponiamo di avere a disposizione un

DBMS che permetta strutture fisiche disordinate (heap), ordinate (con indice primario sparso) e hash e che preveda la possibilità di definire indici secondari e un sistema operativo che utilizzi blocchi di dimensione $B = 2000$ byte e con puntatori ai blocchi di $P = 4$ caratteri. Supponiamo che le operazioni principali siano le seguenti:

o_1 ricerca sul numero di matricola con frequenza $f_1 = 2000$ volte al minuto;
o_2 ricerca sul cognome (o una sua sottostringa iniziale, abbastanza selettiva, in media una sottostringa identifica $S = 10$ tuple) con frequenza $f_2 = 100$ volte al minuto.

È evidente nell'esempio che è comunque necessaria una struttura ad accesso diretto tanto per la matricola quanto per il cognome, in quanto una scansione sequenziale sarebbe troppo costosa. Per quanto riguarda il cognome, la struttura hash non è possibile, perché le ricerche possono essere sulla sottostringa; un indice primario potrebbe essere molto interessante proprio per quest'ultimo motivo. Per la ricerca sulla matricola, la struttura hash è la più efficiente (ed è possibile, perché la dimensione è stabile nel tempo), ma anche un indice (primario o secondario) può essere utile in alternativa. Di conseguenza, una riflessione qualitativa può portare a individuare come comunque importante la scelta della struttura primaria, che può essere o quella hash sulla matricola oppure l'ordinamento con indice primario su cognome. La struttura disordinata in questo caso sembra comunque essere dominata dalle altre due, visti i benefici che ne potrebbero derivare. Per rendere efficiente l'operazione che non viene favorita dalla struttura primaria, è opportuno definire un indice secondario (essendo la struttura primaria definita in altro modo). Di conseguenza, risultano possibili due alternative:

A. struttura hash su matricola e indice secondario su cognome;
B. indice primario su cognome e secondario su matricola.

A questo punto, possiamo valutare i costi delle due operazioni in ciascuno dei due casi:

$c_{1,A}$ (costo dell'operazione o_1 nel caso A) costante, pari circa a 1;
$c_{2,A}$ richiede la visita dell'albero (profondità p_C, che, con i dati disponibili può essere stimata pari a quattro, calcolando il logaritmo della dimensione del file, come discusso in precendenza) più (mediamente) S accessi ai vari record, che si trovano in blocchi diversi;
$c_{1,B}$ richiede la visita dell'albero dell'indice secondario (profondità p_M, pari a tre) più un accesso per il blocco in cui si trova la tupla interessata (che è una sola, perché la matricola è la chiave della relazione);
$c_{2,B}$ richiede la visita dell'albero dell'indice primario (profondità p'_C, pari a quattro nel caso di indice primario vero e proprio); non vi è aggravio per le ricerche su sottostringa, perché le tuple interessate sono tutte nello stesso blocco.

Possiamo quindi riassumere le valutazioni quantitative come segue:

$$c_A = c_{1,A} \times f_1 + c_{2,A} \times f_2$$
$$1 \times f_1 + (4 + 10) \times f_2 = 2800$$
$$c_B = c_{1,B} \times f_1 + c_{2,B} \times f_2 =$$
$$(3 + 1) \times f_1 + 4 \times f_2 = 10\,400$$

In questo caso, quindi, l'alternativa A risulta più conveniente. Vale la pena notare che se le frequenze fossero state diverse, per esempio invertite, $f_1 = 100$ volte al minuto e $f_2 = 2000$ volte al minuto, allora la scelta B sarebbe stata più conveniente.

Definite strutture primarie e indici sulla base di queste considerazioni, si può sperimentare sul campo il comportamento della nostra applicazione: se le prestazioni risultano insoddisfacenti, si possono aggiungere altri indici, procedendo però con grande attenzione, in quanto l'aggiunta di un indice comporta un aggravio del carico per far fronte alle operazioni di modifica. Talvolta, inoltre, il comportamento del sistema è imprevedibile, e l'aggiunta di indici non altera la strategia di ottimizzazione delle interrogazioni principali, risultando del tutto inefficace.

È buona norma, dopo l'aggiunta di un indice, verificare che le interrogazioni ne facciano uso (in genere, esiste un comando `show plan` che descrive la strategia di accesso scelta dal DBMS). Per questo motivo, l'attività di scelta degli indici nell'ambito del progetto fisico delle basi di dati relazionali è svolta spesso in modo empirico, con un approccio per tentativi; più in generale, l'attività di *regolazione* (*tuning*) del progetto fisico consente spesso di migliorare le prestazioni della base di dati.

Note bibliografiche

Gli argomenti presentati in questo capitolo e nel successivo vengono trattati sia nei libri di testo generali sulle basi di dati (in particolare quelli di Garcia-Molina, Ullman e Widom [45], Elmasri e Navathe [41], Korth, Silberschatz e Sudarshan [71] e Ramakhrishnan e Gehrke [66], che coprono adeguatamente gli aspetti tecnologici), sia in libri più specifici. Fra questi ultimi, si segnalano il testo di Teorey e Fry [76] che tratta in modo approfondito molte strutture fisiche e quello di Albano [3] che descrive approfonditamente sia l'organizzazione fisica dei dati sia i metodi di accesso. La gestione dei buffer è trattata in modo molto approfondito nel testo *Transaction Processing Systems*, di Gray e Reuter [48], essenziale riferimento anche per il prossimo capitolo. I fondamenti delle tecniche di ordinamento e delle strutture hash in memoria secondaria (oltre che in memoria centrale) sono discussi in modo esteso in un altro testo molto famoso, il terzo volume di una nota serie di Knuth [52]. Alcuni dei testi sopra citati discutono le estensioni dell'organizzazione hash con caratteristiche dinamiche, cioè che non degenerano al crescere delle dimensioni. Un buon testo di introduzione al progetto delle strutture fisiche e al loro dimensionamento è *Database Tuning: Principles, Experiments, and Troubleshooting Techniques* di Bonnet e Shasha [13], versione rinnovata di un testo dello stesso Shasha [69].

Esercizi

Soluzioni sul sito http://www.ateneonline.it/atzeni

11.1 Definire le strutture dati necessarie per la gestione dei buffer; implementare in un qualunque linguaggio le funzioni `fix`, `use` e `unfix`. Si suppongano disponibili le funzioni di file system descritte nel Paragrafo 11.1.3.

11.2 Si consideri una base di dati gestita tramite hashing, il cui campo chiave contenga i seguenti nominativi:

Green, Lovano, Osby, Peterson, Pullen, Scofield, Allen, Haden, Shepp, Harris, McCann, Mann, Brown, Hutcherson, Newmann, Ponty, Cobham, Coleman, Mingus, Lloyd, Tyner, Fortune, Coltrane.

1. Proporre un algoritmo di hashing con $B = 8$ e $F = 4$.
2. Supponendo $B = 40$ e $F = 1$, qual è la probabilità di conflitto? E con $B = 20$ e $F = 2$?
3. Con $F = 5$ e $B = 7$, quanto vale approssimativamente la lunghezza media della catena di overflow?

11.3 Si consideri una base di dati gestita tramite alberi B+, il cui campo chiave contenga i dati elencati nel precedente esercizio.

1. Descrivere una struttura ad albero B+ bilanciato, con $F = 2$, che contenga i dati citati.
2. Introdurre un dato che provochi lo split di un nodo al livello foglia, e mostrare cosa accade al livello foglia e al livello superiore.
3. Introdurre un dato che provochi il merge di un nodo al livello foglia, e mostrare cosa accade al livello foglia e al livello superiore.
4. Indicare una sequenza di inserimenti che provochi, ricorsivamente, lo split della radice e l'allungamento dell'albero.
5. Descrivere una struttura ad albero B, con $F = 3$, che contenga i dati citati.

11.4 Si consideri la base di dati costituita dalle seguenti relazioni:

PRODUZIONE(NumeroSerie,TipoParte,Modello,Quan,Macchina)
PRELIEVO(NumeroSerie,Lotto)
ORDINE(Lotto,Cliente,Ammontare)
COMMISSIONE(Lotto,Venditore,Ammontare)

Si assumano inoltre i seguenti profili:

CARD(PRODUZIONE)=200 000	SIZE(PRODUZIONE)=41
CARD(PRELIEVO)=50 000	SIZE(PRELIEVO)=15
CARD(ORDINE)=10 000	SIZE(ORDINE)=45
CARD(COMMISSIONE)=5000	SIZE(COMMISSIONE)=35

SIZE(NumeroSerie)=10	VAL(NumeroSerie)=200 000
SIZE(TipoParte)=1	VAL(TipoParte)=4
SIZE(Modello)=10	VAL(Modello)=400
SIZE(Quan)=10	VAL(Quan)=100
SIZE(Macchina)=10	VAL(Macchina)=50
SIZE(Lotto)=5	VAL(Lotto)=10 000
SIZE(Cliente)=30	VAL(Cliente)=400
SIZE(Ammontare)=10	VAL(Ammontare)=5000
SIZE(Venditore)=20	VAL(Venditore)=25

Descrivere l'ottimizzazione algebrica e il calcolo dei profili dei risultati intermedi relativi alle seguenti interrogazioni, che vanno inizialmente espresse in SQL e poi tradotte in algebra relazionale:

1. Determinare la quantità disponibile del prodotto 77Y6878.
2. Determinare le macchine utilizzate per la produzione dei pezzi venduti al cliente Rossi.

3. Determinare i clienti che hanno comprato dal rivenditore Bianchi un box modello 3478.

Per le ultime interrogazioni, che richiedono join fra tre o quattro tabelle, indicare gli ordinamenti tra join che sembrano più convenienti sulla base delle dimensioni degli operandi. Descrivere poi, prevedendo solo due alternative a scelta per l'esecuzione dei join, l'albero delle alternative relativo alla seconda interrogazione.

11.5 Elencare le condizioni (dimensioni delle tabelle, presenza di indici o di organizzazioni sequenziali o a hash) che rendono più o meno conveniente la realizzazione di join con i metodi nested loop, merge scan e hash-based; per alcune di queste condizioni, proporre delle formule di costo che tengano conto essenzialmente del numero di operazioni di ingresso/uscita come funzione dei costi medi delle operazioni di accesso coinvolte (scansioni, ordinamenti, accessi via indici).

11.6 Considerare la seguente interrogazione in SQL:

```
select distinct C, L
from R1, R2, R3
where R1.C = R2.D and R2.F=R3.G and R1.B=R3.L and R3.H>10
```

Mostrare un possibile piano di esecuzione (in termini di operatori dell'algebra relazionale e loro realizzazioni; prestare attenzione anche alla `DISTINCT`, in quanto le realizzazioni degli operatori non producono necessariamente insiemi, ma liste di tuple), giustificando brevemente le scelte più significative, con riferimento alle seguenti informazioni sulla base di dati:

- la relazione $R_1(\underline{A}BC)$ ha 100 000 tuple, una struttura heap e un indice secondario su C;
- la relazione $R_2(\underline{D}EF)$ ha 30 000 tuple, una struttura heap e un indice secondario sulla chiave D;
- la relazione $R_3(\underline{G}HL)$ ha 10 000 tuple, una struttura heap e un indice secondario sulla chiave G.

11.7 Considerare la seguente interrogazione in SQL:

```
select distinct A, L
from T1, T2, T3
where T1.C = T2.D and T2.E = T3.F and T1.B = 3
```

Mostrare un possibile piano di esecuzione (in termini di operatori dell'algebra relazionale e loro realizzazioni), giustificando brevemente le scelte più significative, con riferimento alle seguenti informazioni sulla base di dati:

- la relazione $T_1(\underline{A}BC)$ ha 800 000 tuple e 100 000 valori diversi per l'attributo B, distribuiti uniformemente; ha una struttura heap e un indice secondario sulla chiave A;
- la relazione $T_2(\underline{D}E)$ ha 500 000 tuple; è definito un vincolo di riferimento fra l'attributo E e la chiave F della relazione T_3; ha una struttura heap e un indice secondario sulla chiave D;
- la relazione $T_3(\underline{F}L)$ ha 1000 tuple e ha una struttura hash sulla chiave F.

11.8 Calcolare il fattore di blocco e il numero di blocchi occupati da una relazione con $T = 1\,000\,000$ di tuple di lunghezza fissa pari a $L = 200$ byte in un sistema con blocchi di dimensione pari a $B = 2$ kilobyte.

11.9 Si considerino:

- un sistema con blocchi di $B = 1000$ byte e indirizzi ai blocchi di $p = 2$ byte; il sistema (in effetti un po' obsoleto) prevede indici, primari o secondari, a un solo livello e solo sul campo chiave;
- una relazione con $N = 1\,000\,000$ tuple, ciascuna di $l = 100$ byte, di cui $k = 8$ byte per la chiave.

Calcolare:

1. il numero dei blocchi necessari per un indice primario sul campo chiave;
2. il numero dei blocchi necessari per un indice secondario sul campo chiave;
3. il numero di accessi a memoria secondaria necessari per una ricerca sequenziale sulla chiave;
4. il numero di accessi a memoria secondaria necessari per una ricerca che utilizzi un indice primario;
5. il numero di accessi a memoria secondaria necessari per una ricerca che utilizzi un indice secondario.

Calcolare il numero di accessi sia nel caso peggiore sia con riferimento a un caso medio, nell'ipotesi che le ricerche abbiano successo in una frazione $s = 0.8$ dei casi (cioè otto volte su dieci il record cercato esiste).

11.10 Si considerino un sistema con blocchi di dimensione $B = 1\,000$ byte e puntatori ai blocchi di $P = 2$ byte e una relazione *R(A,B,C,D,E)* di cardinalità pari circa a $N = 1\,000\,000$, con tuple di $L = 50$ byte e campo chiave A di $K = 5$ byte. Valutare i pro e i contro (in termini di numero di accessi a memoria secondaria e trascurando le problematiche relative alla concorrenza) relativamente alla presenza di un indice secondario sulla chiave A e di un altro, pure secondario, su B, in presenza del seguente carico applicativo:

1. inserimento di una nuova tupla (con verifica del soddisfacimento del vincolo di chiave), con frequenza $f_1 = 500$ volte al minuto;
2. ricerca di una tupla sulla base del valore della chiave A con frequenza $f_2 = 500$ volte al minuto;
3. ricerca di tuple sulla base del valore di B con frequenza $f_3 = 100$.

11.11 Come accennato nel testo, alcuni DBMS permettono una tecnica di memorizzazione chiamata "clustering multirelazionale," in cui un file contiene record di due o più relazioni e tali record sono raggruppati (eventualmente, ma non necessariamente, ordinati) secondo i valori di opportuni campi dell'una e dell'altra relazione. Per esempio, date due relazioni

- ORDINI(CodiceOrdine,Cliente,Data,ImportoTotale);
- LINEEORDINE(CodiceOrdine,Linea,Prodotto,Quantità,Importo);

questa tecnica (con riferimento agli attributi CodiceOrdine delle due relazioni) permetterebbe una memorizzazione contigua di ciascun ordine con le rispettive "linee d'ordine," cioè dei prodotti ordinati (ciascun ordine fa riferimento a più prodotti, ognuno su una "linea").
Con riferimento all'esempio, indicare quali delle seguenti operazioni possono trarre vantaggio dall'uso di questa opportunità e quali ne possono essere penalizzate (spiegare la risposta anche in termini quantitativi, individuando valori opportuni per i principali parametri d'interesse; supporre che siano utilizzati indici su CodiceOrdine, in tutti i casi, due per la memorizzazione tradizionale e uno nel caso di utilizzo del cluster eterogeneo):

1. stampa dei dettagli (cioè delle linee d'ordine) di tutti gli ordini (ordinati per codice);
2. stampa dei dettagli di un ordine;

3. stampa delle informazioni sintetiche (codice, cliente, data, totale) di tutti gli ordini.

11.12 Si consideri una relazione IMPIEGATO (Matricola, Cognome, Nome, Data-Nascita) con un numero di tuple pari a N abbastanza stabile nel tempo (pur con molti inserimenti ed eliminazioni) e una dimensione di ciascuna tupla (a lunghezza fissa) pari a L byte, di cui K per la chiave Matricola e C per il campo Cognome.
Supporre di avere a disposizione un DBMS che permetta strutture fisiche disordinate (heap), ordinate (con indice primario sparso) e hash e che preveda la possibilità di definire indici secondari e operi su un sistema operativo che utilizza blocchi di dimensione B e con puntatori ai blocchi di P caratteri.
Indicare quale possa essere l'organizzazione fisica preferita nel caso in cui le operazioni principali siano le seguenti:

1. ricerca sul cognome (o una sua sottostringa iniziale, abbastanza selettiva, si supponga che mediamente una sottostringa identifichi $S = 10$ tuple) con frequenza f_1;
2. ricerca sul numero di matricola, con frequenza f_2;
3. ricerca sulla base di un intervallo della data di nascita (poco selettivo, restituisce circa il 5% delle tuple), con frequenza f_3 molto minore di f_1 e f_2;

assumendo $N = 10\,000\,000$ tuple, $L = 100$ byte, $K = 5$ byte, $C = 20$ byte, $B = 1000$ byte, $P = 4$ byte, $f_1 = 100$ volte al minuto, $f_2 = 2000$ volte al minuto, $f_3 = 1$ volte al minuto.
Individuare le alternative più sensate sulla base di ragionamenti qualitativi e poi valutarle quantitativamente, ignorando i benefici derivanti dai buffer e dalla contiguità di memorizzazione.

Gestione delle transazioni

In questo capitolo continuiamo la discussione degli aspetti tecnologici che caratterizzano il funzionamento dei DBMS, illustrando le modalità secondo cui essi garantiscono la possibilità di conservare a lungo termine e in modo corretto il contenuto delle basi di dati, anche in presenza di guasti e favorendo anche gli indispensabili accessi concorrenti. Queste funzionalità sono particolarmente delicate in presenza di aggiornamenti delle basi di dati, cioè quando le operazioni non sono solo *interrogazioni*, ma anche *transazioni*.

La nozione di transazione è stata introdotta nel Paragrafo 5.6, con la definizione delle proprietà acide (atomicità, consistenza, isolamento e durabilità o persistenza), e utilizzata poi nel Capitolo 10 e in particolare nei Paragrafi 10.3 e 10.5.2. La Figura 12.1 mostra, in modo schematico, come l'architettura di un DBMS già discussa nel capitolo precedente (Figura 11.1) con riferimento alle sole interrogazioni, possa essere estesa per tenere conto della necessità di garantire le proprietà acide delle transazioni. Il modulo indicato nella figura come *gestore delle transazioni* coordina tutte le attività connesse appunto con le transazioni, attraverso l'esecuzione delle istruzioni `start transaction`, `commit` e `rollback`. Il modulo di gestione della affidabilità ha l'obiettivo di garantire atomicità e persistenza e, allo scopo, interagisce sia con il gestore dei metodi d'accesso (per tenere traccia delle operazioni richieste) sia con il gestore del buffer (per richiedere, quando necessario, scritture fisiche sui dischi, al fine di evitare che informazioni delicate rimangano solo nei buffer, che sono volatili). Il gestore della concorrenza si occupa invece di garantire l'isolamento, e lo fa "filtrando," ed eventualmente ripianificando, le operazioni di accesso richieste dal gestore degli accessi. Notiamo che invece la consistenza è gestita dai compilatori del DDL, che introducono opportuni controlli sui dati e opportune procedure per la verifica, eseguite poi dalle transazioni.

I due paragrafi di questo capitolo sono quindi rispettivamente dedicati alla presentazione delle tecniche per il controllo dell'*affidabilità* e di quelle per il controllo della *concorrenza*.

12.1 Controllo di affidabilità

Il controllo dell'affidabilità garantisce due proprietà fondamentali delle transazioni: l'atomicità e la persistenza. In pratica, esso garantisce che le transazioni non vengano lasciate incomplete, con alcune operazioni eseguite e le altre no, e che gli effetti di ciascuna transazione conclusa con un commit siano mantenuti in modo permanente. Il controllore dell'affidabilità svolge il proprio compito attraverso il *log*, un archivio persistente su cui registra le varie azioni svolte dal DBMS. Come vedremo, ogni azione di scrittura sulla base di dati viene protetta tramite una azione sul log, in modo che sia possibile "disfare" (*undo*) le azioni a seguito di malfunzionamenti o guasti precedenti il commit, oppure "rifare" (*redo*) queste azioni qualora la loro buona riuscita sia incerta e le transazioni abbiano effettuato un commit.

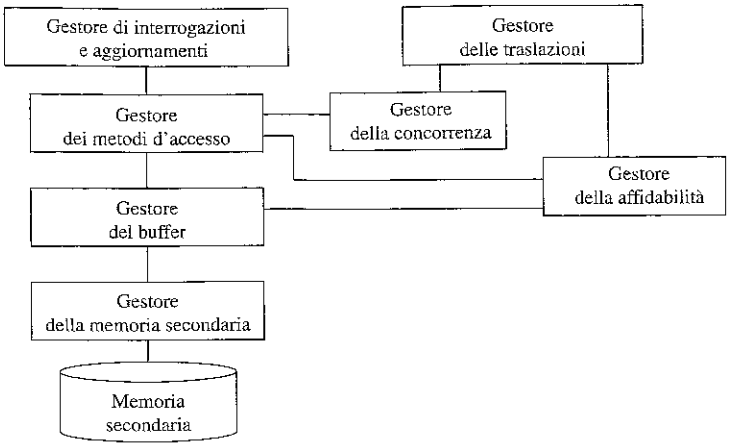

Figura 12.1 Componenti di un DBMS coinvolti nella gestione di interrogazioni e transazioni.

Per dare l'intuizione sul ruolo del log è possibile ricorrere a due metafore, una mitologica e l'altra basata su una fiaba popolare. Il log può essere paragonato al "filo di Arianna", usato da Teseo per ritrovare l'uscita del palazzo del Minotauro; in tal caso, riavvolgendo il filo Teseo riesce a "disfare" il cammino percorso. Un ruolo analogo era affidato da Hansel e Gretel ai granelli di pane lasciati lungo il loro cammino nella foresta, ma nel caso della favola dei fratelli Grimm i granelli venivano mangiati dagli uccellini, e Hansel e Gretel si perdevano nel bosco. Questa analogia mostra che, per poter svolgere con efficacia il suo compito, il log deve essere sufficientemente robusto.

12.1.1 Architettura del controllore dell'affidabilità

Il controllore dell'affidabilità è responsabile di realizzare i comandi transazionali e di realizzare le operazioni di ripristino dopo i malfunzionamenti, dette rispettivamente *ripresa a caldo* e *ripresa a freddo*. Inoltre, il controllore di affidabilità riceve richieste di accessi a pagine in lettura e scrittura, che passa al buffer manager, e genera altre richieste di letture e scritture di pagine necessarie a garantire la robustezza e la resistenza ai guasti. Infine, il controllore dell'affidabilità predispone i dati necessari per eseguire i meccanismi di ripristino dai guasti, in particolare realizzando azioni di *checkpoint* e di *dump*.

Per semplicità, in questo paragrafo (e, con qualche differenza, nel prossimo), facciamo riferimento alle azioni sulla base di dati come se fossero sempre operazioni

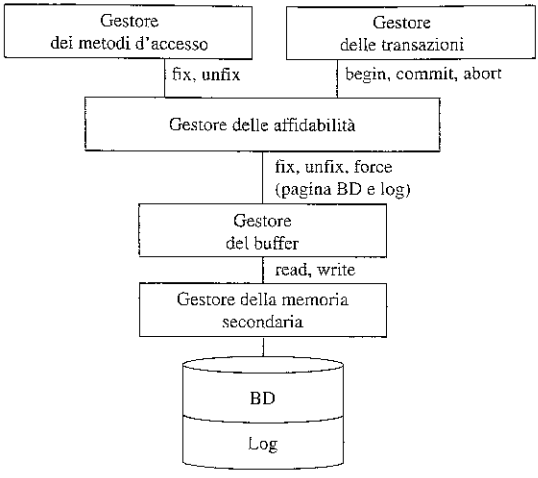

Figura 12.2 Architettura del controllore di affidabilità.

di lettura o scrittura di un'intera pagina. Ovviamente, la realtà è molto più complessa, ma la schematizzazione è sufficiente per capire i princìpi fondamentali.

Memoria stabile Per poter operare, il controllore dell'affidabilità deve disporre di *memoria stabile*, cioè di memoria che risulti resistente ai guasti. La memoria stabile è una astrazione, in quanto nessuna memoria può avere una probabilità nulla di fallimento; tuttavia, meccanismi di replicazione e protocolli di scrittura robusti possono rendere tale probabilità realmente prossima allo zero. I meccanismi di controllo di affidabilità vengono definiti come se la memoria stabile fosse effettivamente esente da guasti; un guasto della memoria stabile viene considerato *catastrofico* e non previsto, perlomeno in questo paragrafo.

A seconda dei casi, la memoria stabile viene realizzata in modi diversi. In alcune applicazioni, si assume che una unità a nastro sia stabile. In altri casi, si assume come stabile una coppia di dispositivi, per esempio una unità a nastro e un disco su cui vengono scritte le stesse informazioni. Una tipica realizzazione di memoria stabile utilizza, al posto di una sola unità a disco, due unità a disco che si dicono "speculari" (*mirrored*), destinate a contenere esattamente la stessa informazione e a essere scritte con un'operazione di "scrittura attenta" che si ritiene riuscita solo se l'informazione viene registrata su entrambi i dischi. In questo modo, l'informazione stabile è anche "in linea", cioè disponibile su un dispositivo ad accesso diretto.

12.1.2 Organizzazione del log

Il *log* è un file sequenziale di cui è responsabile il controllore dell'affidabilità, scritto in memoria stabile e contenente informazione ridondante che permette di ricostruire il contenuto della base di dati a seguito di guasti. Sul log vengono registrate le azioni svolte dalle varie transazioni, nell'ordine temporale di esecuzione delle azioni stesse. Pertanto, il log ha un blocco corrente (detto *top*), l'ultimo a essere stato allocato al log stesso; i record nel log vengono scritti sequenzialmente nel blocco corrente, e quando esso termina vengono scritti in un successivo blocco, allocato al log, che diventa il blocco corrente.

I record del log sono di due tipi: di transazione e di sistema. I record di transazione registrano le attività svolte da ciascuna transazione, nell'ordine in cui esse vengono effettuate. Pertanto, ogni transazione inserisce nel log un record di `begin` (corrispondente al comando `start transaction`), vari record relativi alle azioni effettuate (corrispondenti ai comandi `insert`, `delete`, `update`) e un record di `commit` (corrispondente al comando `commit`) oppure di `abort` (corrispondente al comando `rollback`). La Figura 12.3 mostra la sequenza di record presenti in un log. Vengono evidenziati i record scritti relativamente alla transazione t_1, in un log che viene scritto anche da altre transazioni. La transazione t_1 esegue due modifiche (U, per `update`) prima di andare a buon fine terminando con un `commit`. I record di sistema indicano l'effettuazione di operazioni specifiche del controllore dell'affidabilità chiamate *dump* (abbastanza rare) e di *checkpoint* (più frequenti), che illustreremo in dettaglio più avanti. La Figura 12.3 evidenzia la presenza nel log di un record di dump e di vari record di checkpoint.

Struttura dei record nel log I record di log che vengono scritti per descrivere le attività di una transazione t_i sono elencati di seguito.

- I record di `begin`, `commit` e `abort`, che contengono, oltre all'indicazione del tipo di record, anche l'identificativo T della transazione.
- I record di `update`, che contengono l'identificativo T della transazione, l'identificativo O dell'oggetto su cui avviene l'update, e poi due valori BS e AS che descrivono rispettivamente il valore dell'oggetto O precedentemente alla modifica, detto *before state*, e successivamente alla modifica, detto *after state*. Possiamo

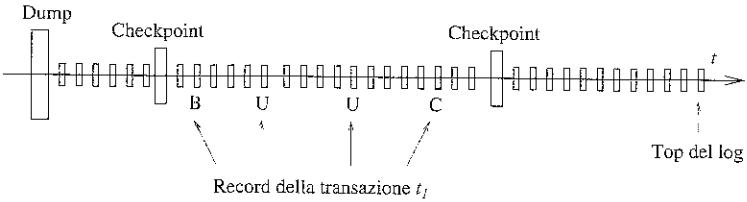

Figura 12.3 Descrizione di un log.

assumere per semplicità che AS e BS contengano copie complete delle pagine modificate, ma nella realtà queste informazioni sono molto più sintetiche.
- I record di `insert` e di `delete` sono analoghi a quelli di `update`, da cui si differenziano per l'assenza nei primi del before state e nei secondi dell'after state.

Nel seguito, useremo i simboli $B(T)$, $A(T)$, $C(T)$ per denotare i record di `begin`, `abort` e `commit` e $U(T, O, BS, AS)$, $I(T, O, AS)$ e $D(T, O, BS)$ per denotare i record di `update`, `insert` e `delete`. Questi record consentono di disfare e rifare le rispettive azioni sulla base di dati, attraverso operazioni specifiche, di competenza del gestore dell'affidabilità, chiamate *Undo* e *Redo* e realizzate nel modo seguente.

- *Undo*: per disfare una azione su un oggetto O è sufficiente ricopiare nell'oggetto O il valore BS; l'insert viene disfatto cancellando l'oggetto O.
- *Redo*: per rifare una azione su un oggetto O è sufficiente ricopiare nell'oggetto O il valore AS; il delete viene rifatto cancellando l'oggetto O.

Dato che le primitive di *Undo* e *Redo* sono definite tramite una azione di copiatura, vale per esse una proprietà essenziale, detta *idempotenza*, per la quale l'effettuazione di un numero arbitrario di undo o redo della stessa azione equivale allo svolgimento di tale azione una sola volta:

$$Undo(Undo(A)) = Undo(A) \qquad Redo(Redo(A)) = Redo(A)$$

Questa proprietà è utile perché, come vedremo, si potrebbero avere errori durante le operazioni di ripristino, che portano alla ripetizione delle operazioni di *Undo* e *Redo*.

Checkpoint e dump I record di log che abbiamo appena visto potrebbero essere sufficienti per svolgere un'operazione di ripristino a seguito di un guasto. Questa sarebbe però molto lunga, perché dovrebbe utilizzare l'intero log; per semplificarla sono stati inventati opportuni accorgimenti, che ora illustriamo. Un *checkpoint* è una operazione che viene svolta periodicamente dal gestore dell'affidabilità (in stretto coordinamento con il buffer manager), con l'obiettivo di registrare quali transazioni sono attive. Esistono in effetti diverse versioni dell'operazione, ma qui, per semplicità didattica, ci limitiamo a vedere la più semplice e intuitiva, costituita dai passi sotto elencati.

1. Si sospende l'accettazione di operazioni di scrittura, commit o abort, da parte di ogni transazione.
2. Si trasferiscono in memoria di massa (tramite opportune operazioni di *force*) tutte le pagine del buffer su cui sono state eseguite modifiche da parte di transazioni che hanno già effettuato il commit.
3. Si scrive in modo sincrono (*force*) nel log un record di checkpoint che contiene gli identificatori delle transazioni attive.
4. Si riprende l'accettazione delle operazioni sopra sospese.

Si noti che questa tecnica garantisce che gli effetti delle transazioni che hanno eseguito il commit siano registrati nella base di dati in modo permanente, mentre nel checkpoint sono elencate le transazioni non hanno ancora espresso la loro scelta relativamente all'esito finale (commit o abort).

Un *dump* è una copia completa e consistente della base di dati, che viene normalmente effettuata in mutua esclusione con tutte le altre transazioni quando il sistema non è operativo (per esempio, di notte oppure durante il fine settimana). La copia viene memorizzata su memoria stabile, tipicamente su nastro, ed è anche chiamata *backup*. Dopo la conclusione dell'operazione di dump viene scritto nel log un *record di dump*, che segnala appunto la presenza di una copia fatta in un determinato istante; dopodiché il sistema può tornare al suo funzionamento normale. I sistemi commerciali oggigiorno offrono spesso la possibilità di eseguire le operazioni di creazione del backup mentre il sistema transazionale rimane attivo. La funzionalità è chiamata *hot backup*. Non tratteremo la gestione interna di questa modalità ed assumeremo che il *dump* venga eseguito durante la sospensione del sistema transazionale.

Nel seguito, useremo i simboli *DUMP* per denotare il record di dump e $CK(T_1, T_2, \ldots, T_n)$ per denotare un record di checkpoint, ove $T_1, T_2, \ldots, T_n$ denotano gli identificatori delle transazioni attive all'istante del checkpoint.

12.1.3 Esecuzione delle transazioni e scrittura del log

Durante il funzionamento normale delle transazioni, il controllore dell'affidabilità deve garantire che siano seguite due regole, che definiscono i requisiti minimi che consentono di ripristinare la correttezza della base di dati a fronte di guasti.

- La *regola WAL* (dall'inglese *Write Ahead Log*, cioè letteralmente: scrivi il log per primo) impone che la parte before state dei record di un log venga scritta nel log (con una operazione di scrittura su memoria stabile) *prima* di effettuare la corrispondente operazione sulla base di dati. Questa regola consente di disfare le scritture già effettuate in memoria di massa da parte di una transazione che non ha ancora effettuato un commit, poiché per ogni aggiornamento viene reso disponibile in modo affidabile il valore precedente la scrittura.
- La *regola di Commit-Precedenza* impone che la parte after state dei record di un log venga scritta nel log (con una operazione di scrittura su memoria stabile) *prima* di effettuare il commit. Questa regola consente di rifare le scritture già decise da una transazione che ha effettuato il commit ma le cui pagine modificate non sono ancora state trascritte dal buffer manager in memoria di massa.

In realtà, anche se le regole fanno riferimento a before state e after state dei record di log, nella pratica entrambe le componenti del record di log vengono scritte assieme; pertanto, una versione semplificata di WAL impone che *i record di log siano scritti prima dei corrispondenti record della base di dati*, mentre una versione semplificata della regola del commit impone che *i record di log siano scritti prima della effettuazione dell'operazione di commit*.

La transazione sceglie, in modo atomico e indivisibile, l'esito di commit nel momento in cui scrive sul log, in modo sincrono (tramite la primitiva *force*), il *record di commit*. Prima di questa azione, un eventuale guasto comporta l'*Undo* delle azioni effettuate, ricostruendo così lo stato iniziale della base di dati. Dopo l'azione di commit, un eventuale guasto comporta il *Redo* delle azioni effettuate, in modo da ricostruire con certezza lo stato finale della transazione. Il record di abort definisce in modo

atomico la scelta di abortire, prodotta dal "suicidio" della transazione o imposta dal sistema; dato però che essa, come vedremo, non modifica le decisioni del controllore dell'affidabilità, il record di abort può essere semplicemente scritto in modo asincrono nel buffer che contiene il blocco corrente del log, che può essere successivamente riscritto sul log con una operazione di flush generata dal gestore del buffer.

Scrittura congiunta di log e base di dati Le due regole WAL e di Commit-Precedenza impongono alcune restrizioni ai protocolli per la scrittura del log e della base di dati, ma lasciano aperte varie possibilità, illustrate in Figura 12.4. Supponiamo per semplicità che le azioni svolte dalle transazioni siano `update`, ma non cambierebbe nulla nel caso di `insert` o `delete`. Distinguiamo tre schemi.

- Nel primo schema, illustrato in Figura 12.4a, la transazione scrive inizialmente il record $B(T)$, quindi svolge le sue azioni di update scrivendo prima il record di log $U(T, O, BS, AS)$ e successivamente la pagina della base di dati, che così passa dal valore BS al valore AS. Tutte queste pagine sono effettivamente scritte (autonomamente dal gestore del buffer oppure con esplicite richieste di *force*) dal buffer manager prima del commit, il quale comporta una scrittura sincrona (*force*). In questo modo, al commit tutte le pagine della base di dati modificate dalla transazione sono certamente scritte in memoria di massa; questo schema non richiede operazioni di *Redo*.
- Nel secondo schema, illustrato in Figura 12.4b, la scrittura dei record di log precede quella delle azioni sulla base di dati, che però avvengono dopo la decisione

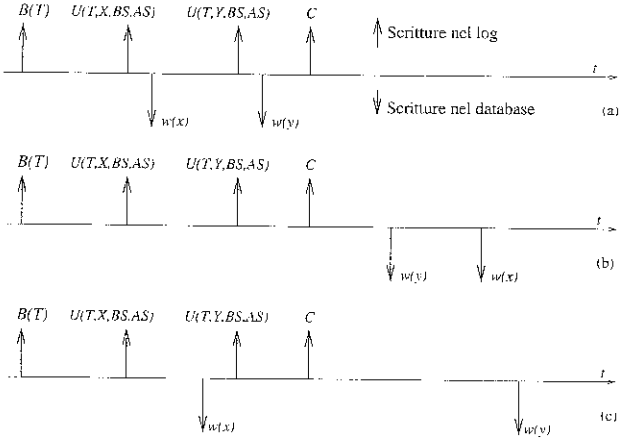

Figura 12.4 Descrizione del protocollo per la scrittura congiunta di log e base di dati.

di commit e la conseguente scrittura sincrona del record di commit sul log; questo schema non richiede operazioni di *Undo*.

- Il terzo schema, più generale e comunemente usato, viene illustrato in Figura 12.4c. Secondo questo schema, le scritture nella base di dati, una volta protette dalle opportune scritture sul log, possono avvenire in un qualunque momento rispetto alla scrittura del record di commit sul log. Questo schema consente al buffer manager di ottimizzare le operazioni di flush relative ai suoi buffer, indipendentemente dal controllore dell'affidabilità; esso però richiede sia *Undo* sia *Redo*.

Si noti che tutti e tre i protocolli rispettano entrambe le regole (WAL e Commit-Precedenza) e scrivono il record di commit in modo sincrono; essi si differenziano solo per il momento in cui scrivono le pagine della base di dati.

Abbiamo così visto quali azioni vengono svolte dal controllore dell'affidabilità per predisporre nel log informazione utile ad azioni di ripristino da malfunzionamenti. Queste azioni hanno un costo, paragonabile al costo di aggiornare la base di dati; in effetti, l'uso di protocolli transazionali robusti rappresenta un sensibile sovraccarico per il sistema, ma è irrinunciabile per garantire le proprietà "acide" delle transazioni. Le operazioni sul log possono essere ottimizzate, per esempio scrivendo i record di log relativi a una transazione nella stessa pagina in cui si scrive il record di commit della transazione (così da ottenere il loro flush contestualmente alla scrittura sincrona del record di commit). Altre tecniche di ottimizzazione consentono il commit di gruppi di transazioni (*group-commit*): vari record di commit vengono posti nella stessa pagina del log e scritti con una unica scrittura sincrona, attesa da tutte le transazioni richiedenti.

12.1.4 Gestione dei guasti

Prima di intraprendere lo studio dei meccanismi di gestione dei guasti, è opportuno darne una classificazione. Dal punto di vista di un DBMS, i guasti si suddividono in due classi:

- *Guasti di sistema*: sono guasti indotti da "bachi software", per esempio del sistema operativo, oppure da interruzioni del funzionamento dei dispositivi, per esempio dovuti a cali di tensione, i quali portano il sistema in uno stato inconsistente. Si traducono in una perdita del contenuto della memoria centrale (e quindi di tutti i buffer), mantenendo invece valido il contenuto della memoria di massa (e quindi della base di dati e del log).
- *Guasti di dispositivo*: sono guasti relativi ai dispositivi di gestione della memoria di massa (per esempio, lo strisciamento delle testine di un disco, che causa la perdita del suo contenuto). Data la nostra assunzione che il log venga scritto sulla memoria stabile, i guasti di dispositivo si traducono in una perdita del contenuto della base di dati, ma non del log. La perdita del contenuto del log è perciò classificata come evento catastrofico, per il quale non viene suggerito alcun rimedio.

Il modello ideale in cui ci poniamo è detto di *fail-stop*: quando il sistema individua un guasto, sia di sistema sia di dispositivo, esso forza immediatamente un arresto

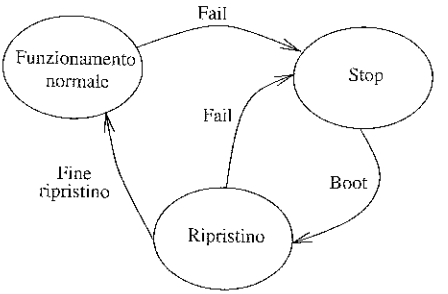

Figura 12.5 Modello fail-stop di funzionamento di un DBMS.

completo delle transazioni e il successivo ripristino del corretto funzionamento del sistema operativo (*boot*). Quindi, viene attivata una procedura di ripresa, denominata *ripresa a caldo* (*warm restart*) nel caso di guasto di sistema e *ripresa a freddo* (*cold restart*) nel caso di guasto di dispositivo. Al termine delle procedure di ripresa, il sistema diventa nuovamente utilizzabile dalle transazioni; il buffer è completamente vuoto e può riprendere a caricare pagine della base di dati o del log. Questo modello di comportamento è illustrato in Figura 12.5.

Con questo modello, il guasto è un evento istantaneo che accade a un certo istante dell'evoluzione della base di dati. Vediamo quali obiettivi si pone la procedura di ripresa: esisteranno transazioni potenzialmente attive all'atto del guasto, cioè delle quali non si conosce se abbiano ultimato le loro azioni sulla base di dati (perché il buffer manager ha perso ogni informazione utile), e queste si classificano in due categorie in base all'informazione presente nel log. Alcune di esse hanno effettuato il commit, e per loro è necessario rifare le azioni al fine di garantire la persistenza (poiché la transazione che ha effettuato il commit si è impegnata a eseguire tutte le proprie azioni). Altre non hanno effettuato il commit, e per loro è necessario disfare le azioni, in quanto non è noto quali azioni siano state realmente effettuate, ma vi è un impegno a lasciare la base di dati nel suo stato iniziale precedente alla esecuzione della transazione. Si noti che alcuni sistemi, per semplificare i protocolli di ripristino, aggiungono al log un altro record, detto *record di end*, quando le operazioni di trascrizione (*flush*) delle pagine a opera del buffer manager sono terminate. Ciò consente di individuare una terza classe di transazioni, per le quali non è necessario né disfare né rifare le azioni. Però in genere il record di end non è previsto, in modo da non complicare la gestione delle transazioni. Nel seguito assumeremo un modello fail-stop dei guasti e l'assenza di record di end.

Ripresa a caldo La ripresa a caldo è articolata in quattro fasi successive.

1. Si accede all'ultimo blocco del log, che era corrente all'istante del guasto, e si ripercorre all'indietro il log fino all'ultimo (cioè più recente) record di checkpoint.

2. Si decidono le transazioni da rifare o disfare. Si costruiscono due insiemi, detti di *UNDO* e di *REDO*, contenenti identificativi di transazioni. Si inizializza l'insieme di *UNDO* con le transazioni attive al checkpoint e l'insieme di *REDO* con l'insieme vuoto. Si percorre poi il log in avanti, aggiungendo all'insieme di *UNDO* tutte le transazioni di cui è presente un record di begin, e spostando dall'insieme di *UNDO* all'insieme di *REDO* tutti gli identificativi delle transazioni di cui è presente il record di commit. Al termine di questa fase, gli insiemi di *UNDO* e *REDO* contengono rispettivamente tutti gli identificativi delle transazioni da disfare e di quelle da rifare.
3. Si ripercorre all'indietro il log disfacendo le transazioni nel set di *UNDO*, risalendo fino alla prima azione della transazione più "vecchia" nei due insiemi di *UNDO* e *REDO*; si noti che questa azione potrebbe precedere il record di checkpoint nel log.
4. Infine, nella quarta fase si applicano le azioni di *Redo* nell'ordine in cui sono registrate nel log. In questo modo, viene replicato esattamente il comportamento delle transazioni originali.

Questo meccanismo garantisce l'atomicità e la persistenza delle transazioni. Per quanto concerne l'atomicità, viene garantito che le transazioni in corso all'istante del guasto lascino la base di dati nello stato iniziale oppure in quello finale; per quanto concerne la persistenza, viene garantito che le pagine nel buffer relative a transazioni completate ma non ancora trascritte in memoria di massa vengano effettivamente completate con una scrittura in memoria di massa. Si noti che ciascuna transazione "incerta", cioè presente nell'ultimo record di checkpoint o iniziata dopo l'ultimo checkpoint, viene disfatta se l'ultimo suo record scritto nel log è un record che descrive un'azione oppure un record di `abort`, oppure rifatta se l'ultimo suo record nel log è il record di commit.

Vediamo un esempio di applicazione del protocollo. Si supponga che nel log vengano registrate le azioni:

$$B(T1),\ B(T2),\ U(T2, O1, B1, A1),\ I(T1, O2, A2),\ B(T3),\ C(T1),$$
$$B(T4),\ U(T3, O2, B3, A3),\ U(T4, O3, B4, A4),\ CK(T2, T3, T4),$$
$$C(T4),\ B(T5),\ U(T3, O3, B5, A5),\ U(T5, O4, B6, A6),\ D(T3, O5, B7),$$
$$A(T3),\ C(T5),\ I(T2, O6, A8).$$

Successivamente, si verifica un guasto. Il protocollo opera come segue.

1. Si accede al record di checkpoint; $UNDO = \{T2, T3, T4\}$, $REDO = \{\}$.
2. Successivamente si percorre in avanti il record di log, e si aggiornano gli insiemi di *UNDO* e *REDO*:
   (a) $C(T4)$: $UNDO = \{T2, T3\}$, $REDO = \{T4\}$
   (b) $B(T5)$: $UNDO = \{T2, T3, T5\}$, $REDO = \{T4\}$
   (c) $C(T5)$: $UNDO = \{T2, T3\}$, $REDO = \{T4, T5\}$
3. Successivamente si ripercorre indietro il log fino all'azione $U(T2, O1, B1, A1)$, eseguendo le seguenti azioni di *Undo*:

(a) Delete($O6$)
(b) Re-Insert($O5 = B7$)
(c) $O3 = B5$
(d) $O2 = B3$
(e) $O1 = B1$

4. Infine, vengono svolte le azioni di *Redo*:
   (a) $O3 = A4$ (nota: $A4 = B5$!)
   (b) $O4 = A6$

Ripresa a freddo La ripresa a freddo risponde a un guasto che provoca il deterioramento di una parte della base di dati; è articolata in tre fasi successive.

1. Durante la prima fase, si accede al *dump* e si ricopia selettivamente la parte deteriorata della base di dati. Si accede anche al più recente record di *dump* nel log.
2. Si ripercorre in avanti il log, applicando relativamente alla parte deteriorata della base di dati sia le azioni sulla base di dati sia le azioni di commit o abort e riportandosi così nella situazione precedente al guasto.
3. Infine, si svolge una ripresa a caldo.

Questo schema ricostruisce tutto il lavoro svolto sulla parte della base di dati soggetta al guasto e quindi, con una ripresa a caldo, garantisce la persistenza e atomicità relativamente all'istante del guasto. La seconda fase dell'algoritmo può essere ottimizzata, per esempio effettuando solo le azioni di transazioni che abbiano eseguito precedentemente il commit.

12.2 Controllo di concorrenza

Un DBMS deve spesso servire diverse applicazioni, e rispondere alle richieste provenienti da diversi utenti. Un'unità di misura che viene solitamente utilizzata per caratterizzare il carico applicativo di un DBMS è il *numero di transazioni al secondo*, abbreviato in *tps* (dall'inglese: *transactions per second*) che devono essere gestite dal DBMS per soddisfare le applicazioni. Sistemi tipici, quali per esempio sistemi informativi bancari o finanziari, devono rispondere a carichi dell'ordine di decine o centinaia di tps; sistemi di prenotazione delle grandi compagnie aeree o di gestione delle carte di credito possono arrivare a migliaia di tps. Per questo motivo, è indispensabile che le transazioni di un DBMS vengano eseguite concorrentemente; è impensabile infatti una loro esecuzione seriale, in cui cioè le transazioni vengono eseguite una alla volta. Solo la concorrenza delle transazioni consente un uso efficiente del DBMS, massimizzando il numero di transazioni servite per secondo e minimizzando i tempi di risposta.

12.2.1 Architettura

Le funzionalità del controllo di concorrenza interagiscono, come abbiamo già visto nella Figura 12.1, con quelle relative alla gestione dei metodi di accesso. In modo un

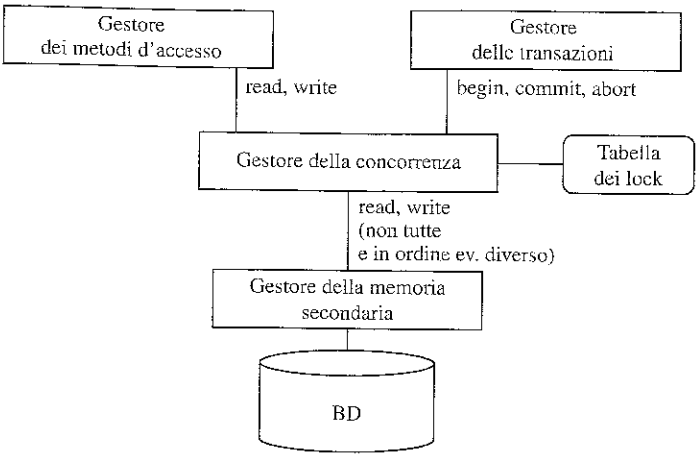

Figura 12.6 Architettura per il controllo di concorrenza.

po' semplificato, ma efficace, possiamo pensare che il controllore della concorrenza riceva le richieste di accesso ai dati (che, come abbiamo visto nel capitolo precedente, sono in effetti richieste al buffer) e decide se autorizzarle o meno, eventualmente riordinandole (come mostrato in modo schematico nella Figura 12.6, in cui sono ignorate le problematiche relative alla gestione del buffer e dell'affidabilità). Poiché in pratica esso stabilisce l'ordine degli accessi, il controllore della concorrenza viene anche chiamato *scheduler*.

In questo paragrafo daremo una descrizione astratta della base di dati nei termini di oggetti x, y, z ed esemplificheremo operazioni in memoria centrale su di esse come se i loro valori fossero numerici. In realtà la loro lettura e scrittura richiede la lettura e scrittura dell'intera pagina in cui risiedono tali dati. Infatti, le azioni $r(x)$ e $w(x)$ denotano rispettivamente la lettura e la scrittura della pagina in cui il dato x è memorizzato nel DBMS.

12.2.2 Anomalie delle transazioni concorrenti

L'esecuzione concorrente di varie transazioni può causare alcuni problemi di correttezza, o *anomalie*; la presenza di queste anomalie motiva la necessità di controllare la concorrenza. Vediamo cinque casi tipici.

Perdita di aggiornamento Supponiamo di avere due transazioni identiche che operano sullo stesso oggetto della base di dati:

$$t_1: r_1(x), x = x + 1, w_1(x)$$
$$t_2: r_2(x), x = x + 1, w_2(x)$$

dove $r_i(x)$ denota la lettura del generico oggetto x da parte della transazione t_i e $w_i(x)$ la scrittura dello stesso oggetto. L'incremento dell'oggetto x è effettuato per opera di un programma applicativo, per esempio un `update` in SQL. Supponiamo che il valore iniziale di x sia 2. Se eseguiamo in sequenza le due transazioni t_1 e t_2, alla fine x assumerà il valore 4. Analizziamo ora una possibile esecuzione concorrente delle due transazioni, che evidenzia la progressione temporale delle azioni:

Transazione t_1	Transazione t_2
$r_1(x)$	
$x = x + 1$	
	$r_2(x)$
	$x = x + 1$
	$w_2(x)$
	`commit`
$w_1(x)$	
`commit`	

In questo caso il valore finale di x è 3, perché entrambe le transazioni leggono 2 come valore iniziale di x. Questa anomalia è nota come *perdita di aggiornamento* (o *lost update*), in quanto gli effetti della transazione t_2 (la prima che scrive il nuovo valore per x) sono persi.

Lettura sporca Supponiamo di avere una transazione come le precedenti, che incrementa x di 1, ma va in abort dopo avere scritto, e un'altra transazione che legge tale dato, eseguite in modo concorrente come segue:

Transazione t_1	Transazione t_2
$r_1(x)$	
$x = x + 1$	
$w_1(x)$	
	$r_2(x)$
	`commit`
`abort`	

Il valore finale di x al termine dell'esecuzione è 2, ma la seconda transazione ha letto (e potenzialmente comunicato all'esterno) il valore 3, che, essendo la transazione andata in abort, è come se non fosse mai esistito. L'aspetto critico di questa esecuzione è la lettura della transazione t_2, che vede uno stato intermedio generato dalla transazione t_1; ma t_2 non avrebbe dovuto vedere tale stato, perché prodotto dalla transazione t_1, che successivamente esegue un abort. Questa anomalia prende il nome di *lettura sporca* (o *dirty read*), in quanto viene letto un dato che rappresenta uno stato intermedio nell'evoluzione di una transazione. Si noti che l'unico modo per ripristinare la correttezza a seguito dell'abort di t_1 sarebbe imporre l'abort di t_2 e, ricorsivamente,

di tutte le transazioni che avessero letto dati modificati da t_2; questa situazione, denominata "effetto domino", è però assai onerosa da gestire e talvolta non praticabile, perché (come nel caso in esame) il risultato può essere stato comunicato all'esterno.

Letture inconsistenti Supponiamo invece che la transazione t_1 svolga solamente operazioni di lettura, ma che ripeta la lettura del dato x in istanti successivi, come descritto nella seguente esecuzione:

Transazione t_1	Transazione t_2
$r_1(x)$	
	$r_2(x)$
	$x = x + 1$
	$w_2(x)$
	`commit`
$r_1(x)$	
`commit`	

In questo caso, x assume il valore 2 dopo la prima operazione di lettura e il valore 3 dopo la seconda operazione di lettura. Invece, è opportuno che una transazione che accede due volte alla base di dati trovi esattamente lo stesso valore per ciascun dato letto, e non risenta dell'effetto di altre transazioni.

Aggiornamento fantasma Si consideri una base di dati con 3 oggetti x, y e z che soddisfano un vincolo di integrità, tali cioè che $x + y + z = 1000$, ed eseguiamo le seguenti transazioni:

Transazione t_1	Transazione t_2
$r_1(x)$	
	$r_2(y)$
$r_1(y)$	
	$y = y - 100$
	$r_2(z)$
	$z = z + 100$
	$w_2(y)$
	$w_2(z)$
	`commit`
$r_1(z)$	
$s = x + y + z$	
`commit`	

La transazione t_2 non altera la somma dei valori e quindi non viola il vincolo di integrità; però la variabile s della transazione t_1, che dovrebbe contenere la somma di x, y e z, contiene in effetti al termine dell'esecuzione il valore 1100. In altri termini, la transazione t_1 osserva solo in parte gli effetti della transazione t_2, e quindi osserva uno stato che non soddisfa i vincoli di integrità. Questa anomalia prende il nome di *aggiornamento fantasma* (o *ghost update*).

Inserimento fantasma Accenniamo infine a una situazione la cui importanza risulterà chiara quando spiegheremo come viene gestito il controllo della concorrenza. Consideriamo una transazione che valuta un valore aggregato relativo all'insieme di tutti gli elementi che soddisfano un predicato di selezione; per esempio, il voto medio degli studenti del primo anno. Consideriamo il caso in cui tale valore aggregato viene valutato due volte, e tra la prima e la seconda valutazione viene inserito un nuovo studente del primo anno. In tal caso, i due valori medi letti dalla transazione potranno essere differenti; notiamo che questa anomalia non può essere evitata facendo riferimento solo ai dati già presenti nella base di dati: è necessario tenere presente che vi è una nuova tupla che compare improvvisamente, come uno spettro (da cui il termine usato in inglese per questa anomalia: *phantom*).

12.2.3 Gestione della concorrenza in SQL e in JDBC

Come abbiamo già osservato nel Paragrafo 10.3, la garanzia dell'isolamento può essere molto onerosa e per questo i DBMS (e gli standard) prevedono la possibilità di specificare un grado di protezione ridotto, definito specificando che alcune delle anomalie che abbiamo visto nel paragrafo precedente vengono evitate, ma altre possono manifestarsi.[1] In questo modo, si permette al programmatore (e all'utente evoluto) di decidere di rinunciare a un controllo di concorrenza stringente, proprio al fine di migliorare le prestazioni.

Specificamente, è possibile, in SQL e, in modo analogo anche in JDBC, indicare per ciascuna transazione[2] il livello di isolamento, scegliendo tra quattro possibilità che si differenziano rispetto alle varie anomalie di lettura:

- *read uncommitted*: permette le anomalie di lettura sporca, lettura inconsistente, aggiornamento fantasma, inserimento fantasma;
- *read committed*: evita la lettura sporca, ma non la lettura inconsistente, l'aggiornamento fantasma, e l'inserimento fantasma;
- *repeatable read*: evita tutte le anomalie, escluso l'inserimento fantasma (phantom);
- *serializable*: evita tutte le anomalie.

I livelli di isolamento diversi da serializable (l'unico che garantisce i massimi requisiti di isolamento) vanno usati esclusivamente con transazioni di sola lettura; quando una transazione effettua letture e scritture è opportuno usare comunque il livello di isolamento massimo, anche perché alcuni sistemi si comportano erroneamente se in una stessa transazione letture ad un livello di isolamento inferiore precedono le scritture, e ciò può causare la perdita di aggiornamento, cioè la anomalia più grave.

[1] È interessante notare che non vi è una definizione rigorosa di questi concetti che, anche nella specifica degli standard, sono indicati per mezzo di esempi.

[2] Si noti quindi che in generale possono esistere diverse transazioni attive in un certo istante con livelli di isolamento diversi.

I sistemi più evoluti mettono a disposizione del programmatore tutti e quattro i livelli di isolamento. Sta al programmatore delle applicazioni scegliere quale livello utilizzare in funzione del livello di isolamento e delle prestazioni richieste. Per le applicazioni nelle quali la correttezza dei dati letti è vitale (per esempio, applicazioni finanziarie) verrà scelto il livello più elevato, mentre per altre applicazioni in cui la correttezza non è importante (per esempio, valutazioni statistiche in cui valori approssimati sono accettabili) verranno scelti livelli inferiori.

12.2.4 Teoria del controllo di concorrenza

Diamo ora una trattazione precisa dei problemi posti dall'esecuzione concorrente di transazioni. Per questo scopo, è necessario definire un modello formale di transazione. Definiamo una transazione come una sequenza di azioni di lettura o scrittura. Quindi, rispetto agli esempi di anomalie illustrati precedentemente, viene omesso da questo modello ogni riferimento alle operazioni di manipolazione dei dati da parte delle transazioni; per quanto concerne la teoria del controllo di concorrenza, ogni transazione è un oggetto sintattico, di cui si conoscono soltanto le azioni di ingresso/uscita. Per esempio, una transazione t_1 è rappresentata dalla sequenza:

$$t_1 : \quad r_1(x)\, r_1(y)\, w_1(x)\, w_1(y)$$

Dato che le transazioni avvengono in modo concorrente, le operazioni di ingresso/uscita vengono richieste da varie transazioni in istanti successivi. Uno *schedule* rappresenta la sequenza di operazioni di ingresso/uscita presentate da transazioni concorrenti. Uno schedule S_1 è quindi una sequenza del tipo:

$$S_1 : \quad r_1(x)\, r_2(z)\, w_1(x)\, w_2(z)\, \ldots$$

dove $r_1(x)$ rappresenta la lettura dell'oggetto x effettuata dalla transazione t_1, e $w_2(z)$ la scrittura dell'oggetto z effettuata dalla transazione t_2. Le operazioni compaiono nello schedule seguendo l'ordine temporale con cui sono eseguite sulla base di dati.

Il controllo di concorrenza ha la funzione di accettare alcuni schedule e rifiutarne altri, in modo per esempio di evitare che si verifichino le anomalie illustrate nel paragrafo precedente. Per questa ragione, il modulo che gestisce il controllo di concorrenza è chiamato anche *scheduler*: esso ha il compito di intercettare le operazioni compiute sulla base di dati dalle transazioni, decidendo per ciascuna se rifiutarla o accettarla, eventualmente dopo una fase di attesa, con la possibilità quindi di sequenze di esecuzione diverse rispetto a quelle di richiesta.

Inizialmente ci occuperemo di caratterizzare la correttezza degli schedule assumendo che le transazioni che compaiono in esse abbiano un esito (commit o abort) noto; in questo modo, è possibile ignorare le transazioni che producono un abort, togliendo dallo schedule tutte le loro azioni, e concentrarsi solo sulle transazioni che producono un commit. Lo schedule si dice in tal caso una *commit-proiezione* della esecuzione delle operazioni di ingresso/uscita, contenente le sole azioni di transazioni che producono un commit. Questa assunzione è funzionale a uno studio teorico ma è inaccettabile in pratica, perché lo scheduler deve decidere se accettare o rifiutare le azioni di una transazione senza conoscere il suo esito finale, a priori incerto.

Per esempio, questa assunzione non consente di trattare l'anomalia di "lettura sporca" vista in precedenza, che si genera quando una transazione decide un abort. Perciò, dovremo rinunciare a questa assunzione quando affronteremo i metodi per il controllo di concorrenza che vengono effettivamente utilizzati nei DBMS.

Definiamo *seriale* uno schedule in cui le azioni di ciascuna transazione compaiono in sequenza, senza essere inframmezzate da istruzioni di altre transazioni. Lo schedule S_2 è per esempio uno schedule seriale in cui vengono eseguite in sequenza le transazioni t_0, t_1 e t_2:

$$S_2: \quad r_0(x)\, r_0(y)\, w_0(x)\, r_1(y)\, r_1(x)\, w_1(y)\, r_2(x)\, r_2(y)\, r_2(z)\, w_2(z)$$

L'esecuzione di uno schedule S_i è corretta quando produce lo stesso risultato prodotto da un qualunque schedule seriale S_j delle stesse transazioni. In tal caso, diremo che S_i è *serializzabile*. Si noti che questo corrisponde ad assumere che ciascuna transazione sia corretta e che l'esecuzione di una sequenza di transazioni seriali, cioè isolate l'una dall'altra, sia corretta. Per formalizzare che cosa si intende con la frase "due schedule producono lo stesso risultato" è necessario disporre di una definizione di equivalenza fra schedule. La prossima sezione è dedicata all'introduzione della nozione di *view-equivalenza*, concettualmente molto interessante ma purtroppo inutilizzabile in pratica per ragioni computazionali. Pertanto, introdurremo poi una nozione più restrittiva di equivalenza (detta *conflict*-equivalenza) e successivamente due metodi utilizzabili (e utilizzati) in pratica, il *locking a due fasi* e il controllo di concorrenza basato su *timestamp*, che garantiscono la serializzabilità, nel senso che accettano solo schedule serializzabili (anche se, oltre agli schedule non serializzabili, rifiutano anche alcuni schedule che sono serializzabili).

View-equivalenza Definiamo dapprima una relazione che lega coppie di operazioni di lettura e scrittura: una operazione di lettura $r_i(x)$ *legge da* una scrittura $w_j(x)$ quando $w_j(x)$ precede $r_i(x)$ e non vi è alcun $w_k(x)$ compreso tra le due operazioni. Una operazione di scrittura $w_i(x)$ viene detta una *scrittura finale* se è l'ultima scrittura dell'oggetto x che appare nello schedule.

Due schedule vengono detti *view-equivalenti* ($S_i \approx_V S_j$) se possiedono la stessa relazione "legge-da" e le stesse scritture finali. Uno schedule viene detto *view-serializzabile* se esiste uno schedule seriale view-equivalente ad esso. Indichiamo con *VSR* l'insieme degli schedule view-serializzabili.

Si considerino gli schedule seguenti:

$$S_3: \quad w_0(x)\, r_2(x)\, r_1(x)\, w_2(x)\, w_2(z)$$

$$S_4: \quad w_0(x)\, r_1(x)\, r_2(x)\, w_2(x)\, w_2(z)$$

$$S_5: \quad w_0(x)\, r_1(x)\, w_1(x)\, r_2(x)\, w_1(z)$$

$$S_6: \quad w_0(x)\, r_1(x)\, w_1(x)\, w_1(z)\, r_2(x)$$

S_3 è view-equivalente allo schedule seriale S_4 (quindi è view-serializzabile); S_5 non è invece view-equivalente a S_4, ma è view-equivalente allo schedule seriale S_6, e quindi risulta anch'esso view-serializzabile.

Notiamo che i seguenti schedule, corrispondenti alle anomalie di perdita di aggiornamento, di letture inconsistenti e di aggiornamento fantasma, non sono view-serializzabili:

$$S_7: \quad r_1(x)\, r_2(x)\, w_2(x)\, w_1(x)$$

$$S_8: \quad r_1(x)\, r_2(x)\, w_2(x)\, r_1(x)$$

$$S_9: \quad r_1(x)\, r_1(y)\, r_2(z)\, r_2(y)\, w_2(y)\, w_2(z)\, r_1(z)$$

Determinare la view-equivalenza di due schedule è un problema con complessità lineare. Determinare se uno schedule è view-equivalente a un qualsiasi schedule seriale è però un problema NP-difficile, perchè può esistere un numero esponenziale di schedule seriali (tutte le permutazioni delle transazioni) con cui confrontare lo schedule dato, senza alcun indizio per scegliere quelli con cui confrontarlo. Questo risultato sulla complessità illustra che il concetto di view-equivalenza non può essere usato al fine di caratterizzare la serializzabilità. Si preferisce quindi definire una condizione di equivalenza più ristretta, la quale non copra tutti i casi di equivalenza tra schedule, ma sia utilizzabile nella pratica, presentando una complessità inferiore.

Conflict-equivalenza Una nozione di equivalenza più facilmente utilizzabile si basa sulla definizione di conflitto. Date due azioni a_i e a_j, con $i \neq j$, si dice che a_i è in *conflitto* con a_j se esse operano sullo stesso oggetto e almeno una di esse è una scrittura. Possono quindi esistere conflitti lettura-scrittura (rw o wr) e conflitti scrittura-scrittura (ww).

Si dice che lo schedule S_i è *conflict-equivalente* allo schedule S_j ($S_i \approx_C S_j$) se i due schedule presentano le stesse operazioni e ogni coppia di operazioni in conflitto è nello stesso ordine nei due schedule. Uno schedule risulta quindi *conflict-serializzabile* se esiste uno schedule seriale a esso conflict-equivalente. Chiamiamo *CSR* l'insieme degli schedule conflict-serializzabili.

È possibile dimostrare che la classe degli schedule CSR è strettamente inclusa in quella degli schedule VSR: esistono cioè schedule che appartengono a VSR ma non a CSR, mentre tutti gli schedule CSR appartengono a VSR. Quindi la conflict-serializzabilità è condizione sufficiente, ma non necessaria, per la view-serializzabilità.

La Figura 12.7 illustra uno schedule S_{10} conflict-serializzabile, di cui sono posti in evidenza i conflitti, e il corrispondente schedule equivalente seriale S_{11}.

È possibile determinare se uno schedule è conflict-serializzabile tramite il *grafo dei conflitti*. Il grafo è costruito facendo corrispondere un nodo a ogni transazione. Si traccia quindi un arco orientato da t_i a t_j se esiste almeno un conflitto tra un'azione a_i e un'azione a_j e si ha che a_i precede a_j; si veda l'esempio in Figura 12.7. Si può dimostrare che uno schedule è in CSR se e solo se il suo grafo dei conflitti è aciclico:

- Se uno schedule S è in CSR allora, per definizione, esso è conflict-equivalente ad uno schedule seriale S_0. Sia $t_1, t_2, \ldots, t_n$, la sequenza delle transazioni in S_0. Poiché lo schedule seriale S_0 ha tutti i conflitti nello stesso ordine dello schedule S, nel grafo di S ci possono essere solo archi (i, j) con $i < j$ e quindi il grafo non può avere cicli, perché un ciclo richiede almeno un arco (i, j) con $i > j$.

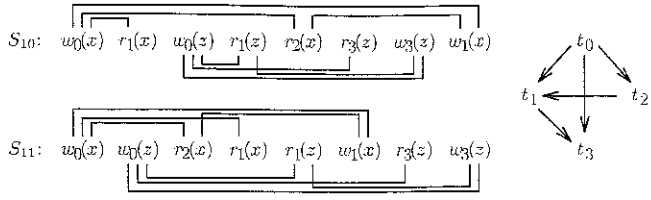

Figura 12.7 Uno schedule S_{10} conflict-equivalente a uno schedule seriale S_{11}.

- Se il grafo di S è aciclico, allora esiste fra i nodi un "ordinamento topologico" (cioè una numerazione dei nodi tale che il grafo contiene solo archi (i, j) con $i < j$). Lo schedule seriale le cui transazioni sono ordinate secondo l'ordinamento topologico è equivalente a S, perché per ogni conflitto (i, j) si ha sempre $i < j$.

L'analisi di ciclicità di un grafo ha una complessità lineare rispetto alle dimensioni del grafo stesso, però la conflict-serializzabilità risulta ancora eccessivamente onerosa in pratica. Per esempio, in un sistema con 100 tps e transazioni che accedono a 10 pagine e durano mediamente 5 secondi, sarà necessario in ogni istante gestire un grafo con 500 nodi e ricordare i 5000 accessi delle 500 transazioni attive; questo grafo continua peraltro a modificarsi dinamicamente, rendendo molto ardue le decisioni dello scheduler. La tecnica risulta assolutamente inaccettabile in un contesto di base di dati distribuita perché, come vedremo nel secondo volume, il grafo deve essere ricostruito a partire da archi che vengono riconosciuti sui diversi server del sistema distribuito.

Locking a due fasi I meccanismi di controllo di concorrenza utilizzati in pratica superano le limitazioni discusse in precedenza. Tra di essi, il più noto, e il primo usato dai DBMS commerciali, si basa sul *locking*, una tecnica che, come quasi tutte le idee che incontrano un notevole successo applicativo, utilizza un principio molto semplice: tutte le operazioni di lettura e scrittura devono essere protette tramite la esecuzione di opportune primitive, *r_lock*, *w_lock* e *unlock*; lo scheduler (che viene in questo caso detto anche *lock manager*, perché la sua funzione fondamentale è di gestire i lock) riceve una sequenza di richieste di esecuzione di queste primitive da parte delle transazioni, e ne determina la correttezza con una semplice *ispezione* di una struttura dati.

Nell'esecuzione di operazioni di lettura e scrittura si devono rispettare i seguenti vincoli.

1. Ogni operazione di lettura deve essere preceduta da un *r_lock* e seguita da un *unlock*; il lock si dice in questo caso *condiviso*, perché su un dato possono essere contemporaneamente attivi più lock di questo tipo.

2. Ogni operazione di scrittura deve essere preceduta da un *w_lock* e seguita da un *unlock*; il lock si dice in tal caso *esclusivo*, perché non può coesistere con altri lock (esclusivi o condivisi) sullo stesso dato.

Quando una transazione segue queste regole si dice *ben formata rispetto al locking*; si noti che le regole di precedenza precedentemente illustrate non sono strette, e quindi l'operazione di lock di una risorsa può avvenire molto prima di una azione di lettura o scrittura di quella risorsa. Se una transazione deve contemporaneamente leggere e scrivere, la transazione può richiedere solo un lock di tipo esclusivo, oppure passare al momento opportuno da un lock condiviso a un lock esclusivo, "incrementando" il livello di lock (*lock upgrade*). In alcuni sistemi, è disponibile una sola primitiva di lock, che non distingue tra lettura e scrittura, e di fatto si comporta come un lock esclusivo.

In genere, le transazioni sono automaticamente ben formate rispetto al locking, poiché emettono le opportune richieste di lock e unlock in modo trasparente al programma applicativo. Il gestore della concorrenza riceve le richieste di lock provenienti dalle transazioni, e concede i lock in base ai lock precedentemente concessi alle altre transazioni. Quando una richiesta di lock è concessa, si dice che la corrispondente risorsa viene *acquisita* dalla transazione richiedente; all'atto dell'unlock, la risorsa viene *rilasciata*. Quando una richiesta di lock non viene concessa, la transazione richiedente viene messa in *stato di attesa*; l'attesa termina quando la risorsa viene sbloccata e diviene disponibile. I lock già concessi vengono memorizzati in *tabelle di lock*, gestite dal lock manager.

Ogni richiesta di lock che perviene al lock manager è caratterizzata solo dall'identificativo della transazione che fa la richiesta, e dalla risorsa per la quale la richiesta viene effettuata. La politica che viene seguita dal lock manager per concedere i lock è rappresentata nella *tabella dei conflitti* illustrata in Figura 12.8, in cui le righe identificano le richieste, le colonne lo stato della risorsa richiesta, il primo valore nella cella l'esito della richiesta e il secondo valore nella cella lo stato che verrà assunto dalla risorsa dopo l'esecuzione della primitiva.

I tre *No* presenti nella tabella rappresentano i conflitti che si possono presentare, quando si richiede una lettura o una scrittura su un oggetto già bloccato in scrittura, o una scrittura su un oggetto già bloccato in lettura. In pratica, solo quando un oggetto è bloccato in lettura è possibile dare risposta positiva a un'altra richiesta di lock in

Richiesta	Stato risorsa		
	libero	*r_locked*	*w_locked*
r_lock	OK / *r_locked*	OK / *r_locked*	No / *w_locked*
w_lock	OK / *w_locked*	No / *r_locked*	No / *w_locked*
unlock	*error*	OK / *dipende*	OK / *libero*

Figura 12.8 Tabella dei conflitti per il metodo di locking.

lettura; da questa caratteristica discende il nome di *lock condiviso* attribuito al lock in lettura. Nel caso di unlock di una risorsa bloccata in modo condiviso, la risorsa ritorna *libera* quando non ci sono altre transazioni in lettura che operano su di essa; altrimenti, essa rimane bloccata in lettura. Per questo motivo, la corrispondente casella della matrice dei conflitti assume il valore *dipende*. Per tener conto del numero di lettori è necessario introdurre un contatore, che viene incrementato a ogni richiesta di *r_lock* concessa, e decrementato a ogni unlock.

I meccanismi di locking possono essere usati per garantire che le transazioni che lavorano sulla base di dati accedano ai dati in mutua esclusione; è questo il tradizionale contesto d'uso dei meccanismi di controllo su lettura e scrittura presenti nell'ambito dei sistemi operativi. Per avere però la garanzia che le transazioni seguano uno schedule serializzabile è necessario porre una restrizione sull'ordinamento delle richieste di lock, che prende il nome di principio del *locking a due fasi* (in inglese: *Two Phase Locking*, abbreviato in 2PL).

Locking a due fasi (2PL): Una transazione, dopo aver rilasciato un lock, non può acquisirne altri.

Come conseguenza di questo principio si possono distinguere nell'esecuzione della transazione due diverse fasi: una prima fase in cui si acquisiscono i lock per le risorse cui si deve accedere (*fase crescente*), e una seconda fase in cui i lock acquisiti vengono rilasciati (*fase calante*). Si tenga conto che il passaggio da un *r_lock* a un *w_lock* costituisce un incremento del livello di lock sulla risorsa, che può quindi comparire nella fase crescente della transazione. La Figura 12.9 mostra una rappresentazione grafica del comportamento richiesto dal protocollo di lock a due fasi. L'ascissa rappresenta il tempo, l'ordinata rappresenta il numero di risorse ottenute da una transazione durante la sua esecuzione.

Un sistema in cui le transazioni sono ben formate rispetto al locking (ovvero richiedono sempre un opportuno lock prima di accedere alle risorse e lo rilasciano

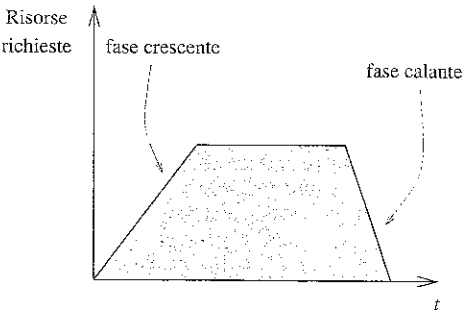

Figura 12.9 Rappresentazione delle risorse allocate a una transazione con un protocollo di lock a due fasi.

prima del termine della transazione), con un lock manager che rispetta la politica descritta nella tabella, e in cui le transazioni seguono il principio del lock a due fasi, è un sistema transazionale caratterizzato dalla serializzabilità delle proprie transazioni. La classe 2PL contiene gli schedule che soddisfano queste condizioni.

Dimostriamo, sia pure informalmente, che ogni schedule che rispetti i requisiti del protocollo di lock a due fasi risulta anche uno schedule serializzabile rispetto alla conflict-equivalenza, ovvero che *la classe* 2PL *è contenuta nella classe* CSR. Ipotizziamo per assurdo che esista uno schedule S tale che $S \in$ 2PL e $S \notin$ CSR. Se lo schedule non appartiene a CSR vuol dire che costruendo il grafo delle dipendenze tra le transazioni si ottiene un ciclo $t_1, t_2, \ldots, t_n, t_1$. Se esiste un conflitto tra t_1 e t_2, vuol dire che esiste una risorsa x su cui operano entrambe le transazioni in modo conflittuale. Affinché la transazione t_2 possa procedere, è necessario che la transazione t_1 rilasci il lock su x. D'altra parte, se osserviamo il conflitto tra t_n e t_1, vuol dire che esiste una risorsa y su cui operano entrambe le transazioni in modo conflittuale. Affinché la transazione t_1 possa procedere, è necessario che la transazione t_1 acquisisca il lock sulla risorsa y, rilasciato da t_n. Quindi, la transazione t_1 non può essere a due fasi: essa rilascia la risorsa x prima di acquisire la y.

Si può poi dimostrare che le classi 2PL e CSR non sono equivalenti, e quindi che 2PL è inclusa strettamente in CSR. Per questo basta mostrare un esempio di schedule non in 2PL ma in CSR, come il seguente:

$$S_{12}: \quad r_1(x)\, w_1(x)\, r_2(x)\, w_2(x)\, r_3(y)\, w_1(y)$$

In questo caso, la transazione t_1 deve cedere un lock esclusivo sulla risorsa x e successivamente richiedere un lock esclusivo sulla risorsa y, risultando pertanto non a due fasi. Viceversa, lo schedule è conflict-serializzabile rispetto alla sequenza: t_3, t_1, t_2.

In conclusione, quindi possiamo dire che *la classe* 2PL *è strettamente contenuta nella classe* CSR.

Riprendendo gli esempi discussi nel Paragrafo 12.2.2, possiamo vedere come il 2PL permette di evitare l'insorgere delle anomalie (anche se potrebbe introdurre situazioni di blocco critico o stallo, che discuteremo nel Paragrafo 12.2.6).

Esaminiamo in particolare una delle anomalie, l'aggiornamento fantasma. Consideriamo l'esempio introdotto nel Paragrafo 12.2.2; rappresentiamo la stessa sequenza di accessi e mostriamo che il 2PL risolve il problema. La Figura 12.10 descrive per ogni risorsa il suo stato come libero (*free*), bloccato in lettura dalla i-esima transazione ($i : read$) oppure bloccato in scrittura dalla i-esima transazione ($i : write$); illustriamo anche l'esito negativo di una richiesta di lock della i-esima transazione, posta in stato di attesa ($i : wait$). Si noti che, per effetto del 2PL, le richieste di lock di t_1 relative alle risorse z e x vengono messe in attesa, e la transazione t_1 può procedere solo quando tali risorse vengono sbloccate da t_2. Al termine della esecuzione, la variabile s contiene il valore corretto della somma $x + y + z$.

È possibile vedere che le anomalie di lettura inconsistente e di perdita di aggiornamento vengono ugualmente risolte dal 2PL. Qualche osservazione in più è necessaria invece per quanto riguarda le anomalie di lettura sporca e di inserimento fantasma.

Per la lettura sporca, è evidente che finora abbiamo ignorato il problema, in quanto abbiamo ragionato nell'ipotesi di commit-proiezione, trascurando le transazioni

t_1	t_2	x	y	z
		free	free	free
$r_lock_1(x)$		1:read		
$r_1(x)$				
	$w_lock_2(y)$		2:write	
	$r_2(y)$			
$r_lock_1(y)$			1:wait	
	$y = y - 100$			
	$w_lock_2(z)$			2:write
	$r_2(z)$			
	$z = z + 100$			
	$w_2(y)$			
	$w_2(z)$			
	commit			
	$unlock_2(y)$		1:read	
$r_1(y)$				
$r_lock_1(z)$				1:wait
	$unlock_2(z)$			1:read
$r_1(z)$				
$s = x + y + z$				
commit				
$unlock_1(x)$		free		
$unlock_1(y)$			free	
$unlock_1(z)$				free

Figura 12.10 Prevenzione di un aggiornamento fantasma tramite l'uso del locking a due fasi.

che si concludono con un abort. Per rimuovere tale ipotesi ed evitare l'anomalia, si può procedere attraverso una restrizione del protocollo 2PL, che porta al cosiddetto *2PL stretto* (*strict 2PL*):

Locking a due fasi stretto (strict 2PL): I lock di una transazione possono essere rilasciati solo dopo aver correttamente effettuato le operazioni di commit/abort.

In pratica, con questo vincolo i lock vengono rilasciati solo al termine della transazione, dopo che ciascun dato è stato portato nel suo stato finale. Questa versione del 2PL viene usata da molti sistemi commerciali. L'esempio in Figura 12.10 utilizza il 2PL stretto, poiché le azioni di rilascio dei lock seguono l'azione di commit, esplicitamente indicata nello schedule. Tramite il 2PL stretto viene reso impossibile il verificarsi di letture sporche, perché viene impedito l'accesso (da parte di altre transazioni) a dati scritti da transazioni che ancora non hanno effettuato il commit.

L'anomalia di inserimento fantasma (phantom) richiede invece una più precisa definizione del concetto di lock: finora abbiamo assunto che i lock siano definiti con riferimento agli oggetti *presenti* nella base di dati. L'anomalia è legata al fatto che la lettura ripetuta fa riferimento alle tuple che soddisfano una certa condizione,

indipendentemente dal fatto che siano presenti nella base di dati oppure no. Il problema sarebbe evitato se potessimo impedire l'inserimento di un nuovo studente del primo anno (visto che stiamo lavorando sugli studenti del primo anno). Allo scopo, è necessario che i lock possano essere definiti anche con riferimento a condizioni (o *predicati*) di selezione, impedendo non solo l'accesso ai dati coinvolti ma anche la scrittura di nuovi dati che soddisfano il predicato. I *lock di predicato* sono realizzati nei sistemi relazionali con l'ausilio degli indici o, nel caso in cui essi non esistano, bloccando intere relazioni. È quindi evidente come essi possano penalizzare molto l'efficienza dei sistemi.

Concludiamo questo paragrafo osservando come nel contesto del 2PL possono essere realizzati i diversi livelli di isolamento (per le transazioni di sola lettura) illustrati nel Paragrafo 12.2.3:

- read uncommitted: la transazione non chiede lock e non osserva nemmeno i lock esclusivi posti da altre transazioni;
- read committed: richiede lock per le letture, rilasciandoli subito dopo, quindi senza 2PL; in questo modo si evitano le letture sporche, ma non le altre anomalie tipiche delle letture;
- repeatable read: applica in 2PL stretto, ma applicando i lock a singole tuple; sono evitate tutte le anomalie ma non l'inserimento fantasma (phantom), perché non è possibile impedire l'inserimento di nuove tuple;
- serializable: applica il 2PL stretto e i lock di predicato e quindi evita tutte le anomalie.

Controllo di concorrenza basato sui timestamp Illustriamo infine un altro metodo per il controllo di concorrenza assai semplice da realizzare nella sua versione base e diffusosi attraverso una interessante variante. Questo metodo utilizza i *timestamp*, cioè identificatori associati ad ogni evento temporale che definiscono un ordinamento totale sugli eventi. Nei sistemi centralizzati, il timestamp viene generato leggendo il valore dell'orologio di sistema al momento in cui è avvenuto l'evento. Il controllo di concorrenza mediante timestamp (*metodo TS*) avviene nel seguente modo:

- a ogni transazione si associa un timestamp che rappresenta il momento di inizio della transazione;
- si accetta uno schedule solo se esso riflette l'ordinamento seriale delle transazioni in base al valore del timestamp di ciascuna transazione.

Questo metodo di controllo di concorrenza, forse il più semplice di tutti dal punto di vista della realizzazione, impone che le transazioni risultino serializzate in base all'ordine in cui esse acquisiscono il loro timestamp. A ogni oggetto x vengono associati due indicatori, $\mathrm{WTM}(x)$ e $\mathrm{RTM}(x)$, che sono rispettivamente i timestamp della transazione che ha eseguito l'ultima scrittura e della transazione con t più grande che ha letto x. Allo scheduler arrivano le richieste di accesso agli oggetti del tipo $r_t(x)$ o $w_t(x)$, dove t rappresenta il timestamp della transazione che esegue la lettura o la scrittura. Lo scheduler non fa altro che permettere o no l'operazione, secondo la seguente politica:

- $r_t(x)$: se $t < \text{WTM}(x)$ la transazione viene uccisa, altrimenti la richiesta viene accettata; in tal caso $\text{RTM}(x)$ viene aggiornato e posto pari al massimo tra $\text{RTM}(x)$ e t;
- $w_t(x)$: se $t < \text{WTM}(x)$ o $t < \text{RTM}(x)$ la transazione viene uccisa, altrimenti la richiesta viene accettata; in tal caso $\text{WTM}(x)$ viene aggiornato e posto pari a t.

In pratica, ogni transazione non può leggere o scrivere un dato scritto da una transazione con timestamp superiore, e non può scrivere su un dato che è già stato letto da una transazione con timestamp superiore.

Vediamo un esempio. Si supponga che sia $\text{RTM}(x) = 7$ e $\text{WTM}(x) = 5$ (ovvero l'oggetto x è stato letto da transazioni con timestamp 7 o minore e scritto l'ultima volta dalla transazione con timestamp 5). Nel seguito, descriviamo la risposta dello scheduler alle richieste di lettura e scrittura ricevute:

Richieste	Risposte	Nuovi valori
$r_6(x)$	ok	
$r_7(x)$	ok	
$r_9(x)$	ok	$\text{RTM}(x) = 9$
$w_8(x)$	no	t_8 uccisa
$w_{11}(x)$	ok	$\text{WTM}(x) = 11$
$r_{10}(x)$	no	t_{10} uccisa

Il metodo TS comporta l'uccisione di un gran numero di transazioni; inoltre, questa versione del metodo è corretta sotto l'ipotesi di uso di commit-proiezioni. Per rimuovere questa ipotesi è necessario "bufferizzare" le scritture, cioè effettuarle in memoria e trascriverle in memoria di massa solo dopo il commit; ciò comporta che le letture da parte di altre transazioni dei dati memorizzati nel buffer e in attesa di commit vengano a loro volta messe in attesa del commit della transazione scrivente, in pratica introducendo meccanismi di attesa analoghi a quelli di locking.

Una modifica del metodo, interessante sia sul piano teorico sia su quello pratico, è l'uso delle *multiversioni*, che consiste nel mantenere diverse copie degli oggetti della base di dati, per ogni transazione che modifica la base di dati. Ogni volta che una transazione scrive un oggetto, la vecchia copia non viene persa, ma una nuova N-esima copia viene creata, con un corrispondente $\text{WTM}_N(x)$. Si ha invece un solo $\text{RTM}(x)$ globale. Quindi, in un generico istante sono attive $N \geq 1$ copie di ciascun oggetto x; con questo metodo, le richieste di lettura non vengono mai rifiutate, ma vengono dirette alla versione dei dati corretta rispetto al timestamp della transazione richiedente. Le copie vengono rilasciate quando sono divenute inutili, in quanto non esistono più transazioni in lettura interessate al loro valore. Le regole di comportamento diventano:

- $r_t(x)$: una lettura è sempre accettata. Si legge un x_k siffatto: se $t > \text{WTM}_N(x)$, allora $k = N$, altrimenti si prende i in modo che sia $\text{WTM}_i(x) < t < \text{WTM}_{i+1}(x)$;
- $w_t(x)$: se $t < \text{RTM}(x)$ si rifiuta la richiesta, altrimenti si aggiunge una nuova versione del dato (N cresce di uno) con $\text{WTM}_N(x) = t$.

L'idea di adottare più versioni, introdotta teoricamente nel contesto dei metodi basati su timestamp, è poi stata estesa anche agli altri metodi, e in particolare viene usata nel contesto del locking a due fasi. Un uso interessante delle versioni si ottiene limitando le copie a due, tenendo cioè una copia precedente e una successiva a ogni aggiornamento durante le operazioni di scrittura; le transazioni in lettura che sono sincronizzate prima della transazione in scrittura possono in tal caso accedere alla copia più vecchia.

Come abbiamo fatto per il 2PL, vediamo come i vari livelli di isolamento possono essere implementati in un sistema che utilizzi il controllo di concorrenza multiversione:

- read uncommitted: in effetti in molti sistemi questo livello è implementato come il successivo, read committed;
- read committed: ogni operazione legge i dati validi (quindi non quelli di transazioni in corso) al momento dell'inizio dell'operazione stessa; è evidente come questo possa causare anomalie di vario tipo, esclusa la lettura sporca;
- repeatable read e serializable sono implementati nello stesso modo, evitando quindi anche l'inserimento fantasma: ogni lettura di ciascuna transazione vede la versione dei dati valida all'inizio della transazione stessa; inoltre, la transazione viene abortita se tenta di modificare dati che, nel frattempo (cioè dopo l'inizio della transazione) sono stati modificati da altre transazioni.

Confronto fra VSR, CSR, 2PL e TS La Figura 12.11 illustra la tassonomia dei metodi VSR, CSR, 2PL e TS fin qui introdotti. Si osserva che la classe VSR è la classe più generale: essa include strettamente al suo interno CSR, la quale a sua volta include sia la classe 2PL sia la classe TS. 2PL e TS a loro volta presentano una intersezione non nulla, ma nessuna presenta una relazione di inclusione con l'altra. Quest'ultima caratteristica può essere verificata facilmente, costruendo schedule che sono in TS ma non in 2PL, oppure in 2PL ma non in TS, oppure infine in 2PL e in TS.

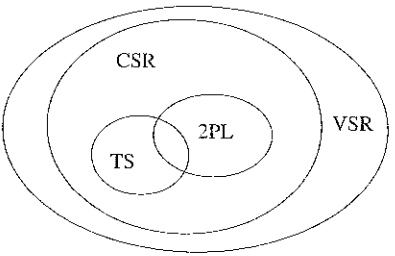

Figura 12.11 Tassonomia delle classi di schedule accettate dai metodi VSR, CSR, 2PL e TS.

Dapprima mostriamo che esistono degli schedule che sono in TS ma non in 2PL. Si consideri lo schedule S_{13}:

$$S_{13}: \quad r_1(x)\, w_2(x)\, r_3(x)\, r_1(y)\, w_2(y)\, r_1(v)\, w_3(v)\, r_4(v)\, w_4(y)\, w_5(y)$$

Il corrispondente grafo dei conflitti, illustrato in Figura 12.12, mostra l'assenza di ciclicità e quindi l'appartenenza a CSR dello schedule. L'ordinamento seriale delle transazioni conflict-equivalenti allo schedule di partenza è $t_1 t_2 t_3 t_4 t_5$. Lo schedule non risulta essere 2PL in quanto t_2 prima rilascia x affinché venga letto da t_3 e poi acquisisce y, rilasciato da t_1, ma risulta essere in TS, poiché presso ogni oggetto le transazioni operano nell'ordine indotto dai timestamp.

Uno schedule che è sia in TS sia in 2PL è per esempio il semplice schedule (seriale) $r_1(x)\, w_1(x)\, r_2(x)\, w_2(x)$. Invece lo schedule $r_2(x)\, w_2(x)\, r_1(x)\, w_1(x)$, in cui cioè la transazione t_2 acquisisce il timestamp dopo la transazione t_1 ma si presenta per prima all'oggetto x, non appartiene a TS ma appartiene a 2PL.

Confrontiamo tra di loro le tecniche 2PL e TS. Emergono alcune differenze significative.

- Nel 2PL le transazioni sono poste in attesa. Nel TS esse sono uccise e poi riavviate.
- L'ordine di serializzazione nel 2PL è imposto dai conflitti, mentre nel TS è imposto dai timestamp stessi.
- La necessità di attendere l'esito della transazione comporta l'allungarsi del tempo di blocco in 2PL (il passaggio da 2PL a 2PL stretto) e la creazione di condizioni di attesa in TS.
- Il metodo 2PL può presentare il problema del blocco critico, che vedremo nel prossimo paragrafo.
- Il restart usato dal TS costa in genere più dell'attesa imposta da 2PL.

I DBMS commerciali utilizzano in effetti varianti di queste tecniche che cercano di limitare gli inconvenienti, soprattutto in termini di prestazioni.

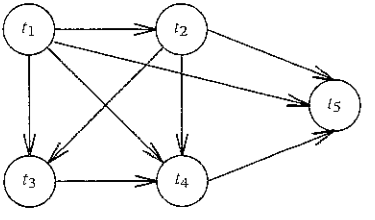

Figura 12.12 Grafo dei conflitti per lo schedule S_{13}.

12.2.5 Meccanismi per la gestione dei lock

Un lock manager è un processo in grado di essere invocato da tutti i processi che intendono accedere alla base di dati. I processi per accedere alle risorse dovranno eseguire delle procedure di *r_lock*, *w_lock* e *unlock*, in genere caratterizzate dai seguenti parametri:

r_lock(T, x, errcode, timeout)
w_lock(T, x, errcode, timeout)
unlock(T,x)

T rappresenta l'identificativo della transazione, *x* l'elemento per il quale si richiede o si rilascia il lock. Rispetto alla definizione della procedura che era stata data nel precedente paragrafo, compare qualche ulteriore parametro: *errcode* rappresenta un valore restituito dal lock manager, e vale zero qualora la richiesta sia stata soddisfatta, mentre assume un valore diverso da zero qualora la richiesta non sia stata soddisfatta; *timeout* rappresenta l'intervallo massimo di tempo che la procedura chiamante è disposta ad aspettare per ottenere il lock sulla risorsa.

Quando un processo richiede una risorsa e la richiesta può essere soddisfatta, il lock manager tiene traccia del cambiamento dello stato della risorsa nelle sue tabelle interne e restituisce immediatamente il controllo al processo; in questo caso, il ritardo introdotto dal lock manager sul tempo di esecuzione della transazione è molto modesto, in quanto la richiesta non comporta operazioni di ingresso-uscita.

Quando invece la richiesta non può essere immediatamente soddisfatta, il sistema inserisce il processo richiedente in una coda associata alla risorsa; ciò comporta un'attesa arbitrariamente lunga, e quindi il processo associato alla transazione viene sospeso. Appena una risorsa viene rilasciata, il lock manager controlla se esistono dei processi in attesa della risorsa e nel caso prende il primo processo della coda e concede a esso la risorsa. L'efficienza del lock manager dipende perciò dalla probabilità che le richieste di una transazione vadano in conflitto; tale probabilità è pari circa a $k \times m/n$, dove k è il numero di transazioni operanti sul sistema, m il numero medio di risorse cui accede una transazione, e n il numero di diversi oggetti presenti nella base di dati.

Quando infine scatta un timeout e la richiesta è insoddisfatta, la transazione richiedente può eseguire un `rollback`, cui generalmente seguirà una ripartenza della stessa transazione, oppure decidere di proseguire, richiedendo nuovamente il lock, in quanto un fallimento nella richiesta di lock non comporta un rilascio delle altre risorse acquisite dalla transazione in precedenza.

Alle tabelle di lock si accede molto di frequente; per questo, il lock manager mantiene queste informazioni in memoria centrale, in modo da minimizzare i tempi di accesso. Le tabelle hanno la seguente struttura: a ciascun oggetto si associano due bit di stato (per rappresentare i tre possibili stati servono almeno due bit) e un contatore, che rappresenta il numero di processi in attesa di quell'oggetto.

Lock gerarchico Per ora si è parlato dei problemi di lock citando generiche risorse e oggetti della base di dati, perché i principi teorici su cui si basa il 2PL sono indipendenti dalla tipologia degli oggetti cui il metodo viene applicato. In molti sistemi

reali, è però possibile specificare i lock a livelli diversi: si parla allora di *granularità dei lock*. Per esempio, è possibile bloccare intere tabelle, o insiemi di tuple, o campi di singole tuple.

Per introdurre livelli diversi di granularità del locking, si opera una estensione del protocollo di lock tradizionale, detta *lock gerarchico* (*hierarchical locking*). Si consideri la Figura 12.13, che illustra la gerarchia delle risorse che fanno parte di una base di dati. La tecnica del lock gerarchico permette alle transazioni di definire in modo molto efficiente i lock di cui hanno bisogno, operando al livello prescelto della gerarchia. Così è possibile per una transazione ottenere un lock per l'intera base di dati (come può essere richiesto quando si vuole effettuare un salvataggio dello stato della base di dati), o per una specifica tupla.

La tecnica fornisce un insieme più ricco di primitive di richiesta di lock, ciascuna con gli opportuni parametri, come visto nel paragrafo precedente:

- XL: lock esclusivo (*exclusive lock*). Corrisponde al write-lock del protocollo normale;
- SL: lock condiviso (*shared lock*). Corrisponde al read-lock del protocollo normale.

I tre lock successivi sono specifici di questa tecnica:

- ISL: intenzione di lock condiviso (*intentional shared lock*). Esprime l'intenzione di bloccare in modo condiviso uno dei nodi che discendono dal nodo corrente;
- IXL: intenzione di lock esclusivo (*intentional exclusive lock*). Esprime l'intenzione di bloccare in modo esclusivo uno dei nodi che discendono dal nodo corrente;
- SIXL: lock condiviso, intenzione di lock esclusivo (*shared intentional-exclusive lock*). Blocca il nodo corrente in modo condiviso ed esprime l'intenzione di bloccare in modo esclusivo uno dei nodi che discendono dal nodo corrente.

Se per esempio si vuole bloccare in scrittura una tupla della tabella, e la gerarchia è quella rappresentata in Figura 12.13, allora bisognerà prima richiedere un IXL al livello della base di dati. Quando la richiesta verrà soddisfatta si potranno chiedere

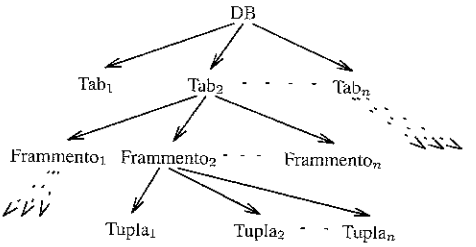

Figura 12.13 La gerarchia delle risorse.

in sequenza un IXL per la relazione e uno per la partizione in cui risiede la tupla desiderata. Quando il lock IXL al livello della partizione sarà stato concesso, si potrà richiedere un lock esclusivo per la particolare tupla. Quando poi la transazione avrà terminato, dovrà rilasciare i lock nell'ordine inverso in cui questi sono stati ottenuti, risalendo un passo alla volta la gerarchia.

Descriviamo in modo più formale le regole che devono essere rispettate dal protocollo.

1. Si richiedono i lock partendo dalla radice e scendendo lungo l'albero.
2. Si rilasciano i lock partendo dal nodo bloccato di granularità più piccola e risalendo lungo l'albero.
3. Per poter richiedere un lock SL o ISL su un nodo, si deve già possedere un lock ISL o IXL sul suo nodo padre.
4. Per poter richiedere un lock IXL, XL o SIXL su un nodo, si deve già possedere un lock SIXL o IXL sul suo nodo padre.
5. Le regole di compatibilità utilizzate dal lock manager per decidere se accettare la richiesta di lock in base allo stato del nodo e al tipo di richiesta sono riportate nella tabella in Figura 12.14.

La scelta del livello di lock è lasciata al progettista delle applicazioni o dell'amministratore della base di dati, sulla base delle caratteristiche delle transazioni: transazioni che effettuano modifiche "localizzate", accedendo a un insieme limitato di oggetti, utilizzeranno una granularità fine; transazioni che effettuano accessi a grandi moli di dati utilizzeranno granularità più grossolana. La scelta deve essere svolta con accortezza, in quanto l'uso di un livello troppo grossolano può porre limitazioni al parallelismo (perché rende elevata la probabilità di conflitti), mentre l'uso di un livello troppo fine costringe a richiedere un gran numero di lock uno per uno, costringendo il lock manager a una quantità considerevole di lavoro ed esponendolo al rischio di un fallimento dopo aver acquisito molte risorse.

Richiesta	Stato risorsa				
	ISL	IXL	SL	SIXL	XL
ISL	OK	OK	OK	OK	No
IXL	OK	OK	No	No	No
SL	OK	No	OK	No	No
SIXL	OK	No	No	No	No
XL	No	No	No	No	No

Figura 12.14 Compatibilità tra le modalità di lock in presenza di gerarchie.

12.2.6 Blocco critico

Il blocco critico (detto anche stallo, oppure abbraccio mortale, dal termine inglese *deadlock*) costituisce un problema rilevante, tipico dei sistemi concorrenti in cui si introducono condizioni di attesa. Supponiamo di avere una transazione t_1 che deve eseguire le operazioni $r(x)$, $w(y)$, e una seconda transazione t_2 che deve eseguire $r(y)$, $w(x)$. Se viene usato il protocollo di lock a due fasi, si può presentare il seguente schedule:

$$r_lock_1(x),\ r_lock_2(y),\ read_1(x),\ read_2(y),\ w_lock_1(y),\ w_lock_2(x)$$

A questo punto nessuna delle due transazioni riesce a procedere e il sistema è bloccato. Il problema è dato dal fatto che t_1 è in attesa che si liberi l'oggetto y, che è bloccato da t_2, e a sua volta t_2 è in attesa dell'oggetto x, bloccato da t_1. Questa situazione è caratteristica di tutti i sistemi in cui si utilizzano dei meccanismi di blocco delle risorse.

Valutiamo la probabilità che si verifichi un evento del genere: consideriamo una tabella che consiste di n diverse tuple, con identica probabilità di accesso. La probabilità che due transazioni che operano un solo accesso vadano in conflitto è $1/n$; la probabilità che si verifichi un blocco critico di lunghezza 2 è pari alla probabilità di un secondo conflitto, e quindi vale $1/n^2$. Non consideriamo il caso dei blocchi critici generati da catene più lunghe, poiché in questo caso la probabilità decresce in modo esponenziale con la crescita della catena, e il contributo al numero totale di blocchi critici che si possono verificare in un sistema reale è trascurabile. Limitatamente al caso di blocchi critici costituiti da coppie di transazioni, la probabilità di conflitto cresce linearmente col numero globale k di transazioni presenti nel sistema e quadraticamente col numero medio m di risorse cui ciascuna transazione fa accesso. L'effettiva probabilità di occorrenza del blocco critico è leggermente superiore a quello che la semplice analisi statistica precedente faccia pensare, a causa delle dipendenze che esistono tra i dati oppure tra le diverse transazioni (per cui, quando una transazione accede a un dato, è più probabile che acceda a un altro dato legato a questo da una qualche relazione). In conclusione, possiamo assumere che la probabilità che si verifichi un blocco critico nei sistemi transazionali sia bassa, ma non nulla; questa considerazione è confermata da valutazioni sperimentali su sistemi standard nel mondo delle applicazioni finanziarie.

Tre sono le tecniche che vengono comunemente usate per risolvere il problema del blocco critico:

1. timeout;
2. prevenzione (*deadlock prevention*);
3. rilevamento (*deadlock detection*).

Uso del timeout La tecnica del timeout è molto semplice. Le transazioni rimangono in attesa di una risorsa per un tempo prefissato. Se passa questo tempo e la risorsa non è stata ancora concessa, allora alla richiesta di lock viene data risposta negativa; in questo modo, una transazione che fosse in deadlock verrebbe comunque tolta dalla condizione di attesa, e presumibilmente abortita. Per la sua semplicità, questa tecnica si lascia preferire nella stragrande maggioranza dei DBMS commerciali.

Per quanto riguarda la scelta del valore di timeout, bisogna saper valutare pro e contro tra due diversi aspetti: da una parte un valore elevato del timeout tende a risolvere tardi i blocchi critici, dopo che le transazioni coinvolte nel blocco hanno passato del tempo in attesa, d'altra parte un timeout troppo basso corre il rischio di rilevare come blocchi critici anche situazioni in cui una transazione sta aspettando una risorsa senza che vi sia un vero deadlock, uccidendo inutilmente una transazione e sprecando il lavoro già svolto dalla transazione.

Prevenzione dei blocchi critici Vi sono diverse tecniche che possono essere utilizzate per prevenire l'insorgenza di un blocco critico. Una tecnica prevede di richiedere il lock di tutte le risorse necessarie alla transazione in una sola volta. Essa presenta però il problema che le transazioni spesso non conoscono a priori le risorse cui vogliono accedere.

Un altro metodo si basa sul fatto che le transazioni acquisiscano un timestamp, e consiste nel consentire l'attesa di una transazione t_i su una risorsa acquisita da t_j solamente se vale una determinata relazione di precedenza fra i timestamp di t_i e t_j (per esempio, $i < j$). In questo modo, circa il 50% delle richieste che generano un conflitto possono attendere in coda, mentre nel restante 50% dei casi una transazione deve essere uccisa. Per quanto riguarda la politica di scelta della transazione da uccidere vi sono diverse alternative. Distinguiamo prima di tutto le politiche in politiche interrompenti (*preemptive*) e non interrompenti. Una politica è interrompente se può risolvere il conflitto uccidendo la transazione che possiede la risorsa (in modo tale che questa rilasci la risorsa, che può così essere concessa all'altra transazione). In caso contrario, la politica è non interrompente, e una transazione può essere uccisa solo all'atto di fare una nuova richiesta. Una politica può essere quella di uccidere le transazioni che hanno fatto meno lavoro. Un problema di questa politica è che può capitare che una transazione faccia accesso, all'inizio della propria elaborazione, a un oggetto cui accedono molte altre transazioni. Può così capitare che la transazione trovi sempre un conflitto, ed essendo la transazione che ha fatto meno lavoro, venga uccisa ripetutamente. La situazione che si presenta è quella di un sistema senza blocchi critici, ma in cui vi sono delle transazioni in *blocco individuale* (*starvation*). Per risolvere questo problema è necessario garantire che ogni transazione non possa essere uccisa un numero illimitato di volte. Una soluzione che si adotta è quella di mantenere lo stesso timestamp quando una transazione viene fatta abortire e ripartire, dando nel contempo priorità crescente alle transazioni più "anziane". Questa tecnica non viene mai usata nei DBMS commerciali, in quanto mediamente si uccide una transazione ogni due conflitti, mentre la probabilità di insorgenza del blocco critico è di gran lunga inferiore alla probabilità di un conflitto.

Un'altra tecnica finalizzata alla prevenzione, recentemente proposta, si basa sull'osservazione che molti deadlock si verificano all'atto di "incrementare" un lock (lock upgrade), passando cioè da un lock in lettura ad un lock di scrittura. Questa situazione si verifica spesso durante l'esecuzione delle transazioni, quando sono previste inizialmente istruzioni di lettura, seguite dalla valutazione di condizioni (talvolta basate su input esterni), e quindi da istruzioni di modifica. Un deadlock si genera quando due transazioni acquisiscono lock condivisi sullo stesso dato e poi cercano

entrambe di incrementare il lock, bloccandosi a vicenda. Per risolvere questo problema, è stato recentemente introdotto un nuovo tipo di lock, detto di *update*, che viene chiesto da una transazione all'atto di leggere un dato su cui intende successivamente scrivere. Questo lock è incompatibile con altri lock di update (quindi, in un certo istante, una sola transazione può avere un lock di update su un dato), ma è compatibile con un lock di lettura (quindi, vari lettori possono essere concorrenti ad un unico lettore che detiene il lock di update). All'atto di passare alla scrittura, la transazione deve comunque richiedere un lock di scrittura, e quindi resta in attesa che eventuali lettori terminino la loro azione. Però con questa modalità di locking si escludono due letture di un medesimo dato su cui due transazioni potrebbero voler poi scrivere, e quindi si esclude la condizione che portava a generare il deadlock descritto in precedenza.

Rilevamento dei blocchi critici Questa tecnica prevede di non porre vincoli al comportamento del sistema, ma di controllare il contenuto delle tabelle di lock, tutte le volte che si ritiene necessario, per rilevare eventuali situazioni di blocco. Il controllo può essere effettuato a intervalli prefissati, o quando scade un timeout di attesa di una transazione. Il rilevamento di un blocco critico richiede di analizzare le relazioni di attesa tra le varie transazioni e di determinare se esiste un ciclo. La ricerca di cicli in un grafo, specie se effettuata periodicamente, risulta abbastanza efficiente; per questo motivo, alcuni DBMS commerciali utilizzano questa tecnica, che verrà descritta in modo diffuso nel secondo volume, nel contesto dei sistemi distribuiti.

Note bibliografiche

Molti riferimenti di interesse per questo capitolo coincidono con quelli già citati nel capitolo precedente. In particolare, il riferimento principale per l'organizzazione del capitolo è il monumentale libro *Transaction Processing Systems*, di Gray e Reuter [48]. Un testo più recente su tematiche simili e ugualmente di ampio respiro è quello di Weikum e Vossen [83]. Per una visione pragmatica, è interessante anche il testo di Bernstein e Newcomer [12]. Segnaliamo inoltre per il controllo di concorrenza la trattazione di Vossen [82] e, per una visione più formale di controllo di concorrenza e di affidabilità, quello di Bernstein, Hadzilacos e Goodman [11]. L'impostazione del controllo di affidabilità è tratta dal libro di Ceri e Pelagatti [23], con opportuni aggiornamenti. Il concetto di transazione introdotto in questo capitolo è stato recentemente esteso, introducendo modelli transazionali più complessi, tra cui le transazioni nidificate o di "lunga vita"; un buon riferimento è il libro edito da Elmagarmid [40].

Esercizi

Soluzioni sul sito http://www.ateneonline.it/atzeni

12.1 Descrivere la ripresa a caldo, indicando la costituzione progressiva degli insiemi di *UNDO* e *REDO* e le azioni di recovery, a fronte del seguente input:

$DUMP$, $B(T1)$, $B(T2)$, $B(T3)$, $I(T1, O1, A1)$, $D(T2, O2, B2)$, $B(T4)$, $U(T4, O3, B3, A3)$, $U(T1, O4, B4, A4)$, $C(T2)$, $CK(T1, T3, T4)$, $B(T5)$, $B(T6)$, $U(T5, O5, B5, A5)$, $A(T3)$, $CK(T1, T4, T5, T6)$, $B(T7)$, $A(T4)$, $U(T7, O6, B6, A6)$, $U(T6, O3, B7, A7)$, $B(T8)$, $A(T7)$, guasto

12.2 Si supponga che nella situazione precedente si verifichi un guasto di dispositivo che coinvolge gli oggetti $O1$, $O2$ e $O3$; descrivere la ripresa a freddo.

12.3 Il check-point, nei vari DBMS, viene realizzato in due modi diversi:

1. in alcuni sistemi si prende nota delle transazioni attive e si rifiutano (momentaneamente) nuovi commit
2. in altri si inibisce l'avvio di nuove transazioni e si attende invece la conclusione (commit o abort) delle transazioni attive

Spiegare, intuitivamente, le differenze che ne conseguono sulla gestione delle riprese a caldo.

12.4 Indicare se i seguenti schedule possono produrre anomalie; i simboli c_i e a_i indicano l'esito (commit o abort) della transazione.

1. $r_1(x), w_1(x), r_2(x), w_2(y), a_1, c_2$
2. $r_1(x), w_1(x), r_2(y), w_2(y), a_1, c_2$
3. $r_1(x), r_2(x), r_2(y), w_2(y), r_1(z), a_1, c_2$
4. $r_1(x), r_2(x), w_2(x), w_1(x), c_1, c_2$
5. $r_1(x), r_2(x), w_2(x), r_1(y), c_1, c_2$
6. $r_1(x), w_1(x), r_2(x), w_2(x), c_1, c_2$

12.5 Indicare se i seguenti schedule sono VSR.

1. $r_1(x), r_2(y), w_1(y), r_2(x), w_2(x)$
2. $r_1(x), r_2(y), w_1(x), w_1(y), r_2(x), w_2(x)$
3. $r_1(x), r_1(y), r_2(y), w_2(z), w_1(z), w_3(z), w_3(x)$
4. $r_1(y), r_1(y), w_2(z), w_1(z), w_3(z), w_3(x), w_1(x)$

12.6 Classificare i seguenti schedule (come: NonSR, VSR, CSR); nel caso uno schedule sia VSR oppure CSR, indicare tutti gli schedule seriali a esso equivalenti.

1. $r_1(x), w_1(x), r_2(z), r_1(y), w_1(y), r_2(x), w_2(x), w_2(z)$
2. $r_1(x), w_1(x), w_3(x), r_2(y), r_3(y), w_3(y), w_1(y), r_2(x)$
3. $r_1(x), r_2(x), w_2(x), r_3(x), r_4(z), w_1(x), w_3(y), w_3(x), w_1(y), w_5(x), w_1(z), w_5(y), r_5(z)$
4. $r_1(x), r_3(y), w_1(y), w_4(x), w_1(t), w_5(x), r_2(z), r_3(z), w_2(z), w_5(z), r_4(t), r_5(t)$
5. $r_1(x), r_2(x), w_2(x), r_3(x), r_4(z), w_1(x), r_3(y), r_3(x), w_1(y), w_5(x), w_1(z), r_5(y), r_5(z)$
6. $r_1(x), r_1(t), r_3(z), r_4(z), w_2(z), r_4(x), r_3(x), w_4(x), w_4(y), w_3(y), w_1(y), w_2(t)$
7. $r_1(x), r_4(x), w_4(x), r_1(y), r_4(z), w_4(z), w_3(y), w_3(z), w_1(t), w_2(z), w_2(t)$

12.7 Se gli schedule dell'esercizio precedente si presentassero a uno scheduler che usa il locking a due fasi, quali transazioni verrebbero messe in attesa? (si noti che, una volta posta in attesa una transazione, le sue successive azioni non vanno più considerate).

12.8 Definire le strutture dati necessarie per la gestione del locking supponendo un modello non gerarchico e con read ripetibili; implementare in un linguaggio di

programmazione a scelta le funzioni `lock_r`, `lock_w` e `unlock`. Si supponga disponibile un tipo di dato astratto "coda" con le opportune funzioni per inserire un elemento in coda ed estrarre il primo elemento dalla coda.

12.9 Facendo riferimento all'esercizio precedente, aggiungere un meccanismo di timeout; si suppongano disponibili le funzioni di sistema per impostare il timeout, per verificare (a controllo di programma) se il tempo fissato è trascorso, e per estrarre uno specifico elemento da una coda.

12.10 Se gli schedule descritti nell'esercizio 12.6 si presentassero a uno scheduler basato sui timestamp, quali transazioni verrebbero abortite?

12.11 Si consideri un oggetto X sul quale opera un controllo di concorrenza basato su timestamp, con $\mathrm{WTM}(X) = 5$, $\mathrm{RTM}(X) = 7$. Indicare le azioni dello scheduler a fronte del seguente input nel caso mono-versione e in quello multi-versione:

$$r_8(x),\ r_{17}(x),\ w_{16}(x),\ w_{18}(x),\ w_{23}(x),\ w_{29}(x),\ r_{20}(x),\ r_{30}(x),\ r_{25}(x)$$

12.12 Spiegare perché il livello più alto di isolamento previsto nello standard SQL ("`serializable`") può avere effetti molto più pesanti anche rispetto al livello immediatamente inferiore ("`repeatable read`").

13

Basi di dati a oggetti, multimediali e NoSQL

Il modello relazionale ha avuto un notevole successo, particolarmente nell'ambito delle tradizionali applicazioni gestionali. In alcuni domini applicativi le caratteristiche del modello relazionale hanno però mostrato alcuni limiti, legati per esempio alla necessità di gestire in modo diretto oggetti complessi, di manipolare contenuti multimediali e di adattarsi ai requisiti di efficienza su grande scala del mondo Web. Ciò ha portato a una serie di iniziative che hanno perseguito l'obiettivo di costruire DBMS in grado di gestire modelli dei dati diversi da quello relazionale.

Le basi di dati a oggetti, sviluppate a partire dalla seconda metà degli anni Ottanta, integrano la tecnologia delle basi di dati con il paradigma a oggetti, sviluppato nell'ambito dei linguaggi di programmazione e utilizzato, sul piano metodologico, nell'ambito dell'ingegneria del software. Nelle basi di dati a oggetti ogni entità del mondo reale è rappresentata da un *oggetto*. Esempi di oggetti possono essere:

- componenti elettronici, progettati tramite un sistema di *Computer-Aided Design* (CAD);
- componenti meccanici, progettati tramite un sistema di *Computer-Aided Manufacturing* (CAM);
- specifiche e programmi, gestiti da un ambiente di *Computer-Aided Software Engineering* (CASE).

Vi sono poi altre tipologie di oggetti che richiedono un trattamento particolare, più ricco di quello che i normali sistemi relazionali riescono normalmente a offrire, che rientrano nell'area delle basi di dati multimediali:

- dati multimediali, che comprendono testi liberi, immagini e suoni, collezionati tramite *sistemi per la gestione di documenti multimediali*;
- dati spaziali o geografici, che descrivono per esempio figure geometriche o mappe geografiche, gestiti dai *sistemi informativi geografici* (GIS).

Nel mondo del Web e delle applicazioni sociali vi sono particolari tipologie di dati che devono essere trattati in modo particolarmente efficiente, rinunciando ai requisiti di transazionalità e limitando il supporto per le operazioni di manipolazione. Questi vengono gestiti da sistemi chiamati NoSQL; in origine il nome rappresentava soluzioni che rinunciavano del tutto all'uso di SQL; oggi si interpreta il nome anche come Not-only SQL. Questi rientrano in diverse famiglie, a seconda della varietà di dati gestiti:

- raccolte estese di coppie chiave-valore, gestite da *key-value data store*;
- raccolte di documenti in formati nativi per il Web, gestite da *document store*;
- collezioni di record estendibili con soluzioni di archiviazione *a colonne*, gestite da *extensible record store*.

Queste applicazioni presentano caratteristiche abbastanza differenti fra loro. Tuttavia è comune a tutti questi campi applicativi il bisogno di organizzare i dati e gestirli in un modo diverso dai normali sistemi relazionali. Per quanto riguarda i sistemi a oggetti, nel modello relazionale ogni oggetto si trova distribuito su di un alto numero di tabelle; una visione unitaria dell'og-

getto richiede query complesse che lo ricostruiscano estraendone i componenti dalle varie tabelle, tramite join. Sul fronte dei sistemi multimediali, risulta necessario disporre di strutture di memorizzazione e accesso specifiche. I sistemi NoSQL da parte loro sono in grado di offrire prestazioni e immediatezza maggiore, a spese di limitazioni sul fronte del potere espressivo e di una riduzione delle garanzie di consistenza dei dati.

In questo capitolo focalizzeremo l'attenzione sui sistemi a oggetti, presentando i tipici componenti di un modello a oggetti e mostrando poi le caratteristiche degli OODBMS e degli ORDBMS. Discuteremo poi alcuni aspetti tecnologici relativi all'implementazione di questi sistemi. Descriveremo quindi i sistemi multimediali, che possono anche essere considerati un caso particolare di sistema a oggetti. Tratteremo infine la famiglia delle soluzioni NoSQL. È bene osservare che tra le soluzioni di gestione dei dati che vanno oltre il modello relazionale rientrano anche i sistemi per la gestione di XML descritti nel Capitolo 14 e le basi di dati semantiche che verranno trattate nel Capitolo 15.

13.1 Basi di dati a oggetti

In questo paragrafo introduciamo progressivamente i concetti necessari per descrivere una base di dati a oggetti; essi estendono notevolmente il potere espressivo del modello dei dati rispetto alla spartana semplicità del modello relazionale, e ritroviamo nel modello dei dati molte delle caratteristiche del modello Entità-Relazione, visto nel Capitolo 6.

13.1.1 Tipi

I tipi, in un linguaggio di programmazione a oggetti, consentono di definire le proprietà degli oggetti; in particolare, distinguiamo proprietà *statiche* (che descrivono la struttura degli oggetti) e *dinamiche* (che descrivono le operazioni, o "metodi", applicabili agli oggetti).

Partiamo, nella descrizione dei tipi, dalla parte statica; la natura dinamica dei tipi verrà illustrata nel Paragrafo 13.1.3. La parte statica di un tipo è costruita usando i cosiddetti *costruttori di tipo* e un insieme abbastanza esteso di *tipi di dati atomici*, che comprendono i classici tipi di dati presenti nei linguaggi di programmazione: per esempio, interi, reali, booleani, stringhe di caratteri. I tipi atomici includono gli identificatori di oggetto (*Object Identifier*, OID), che verranno introdotti più avanti in questo capitolo. La maggior parte dei sistemi include il valore nullo (`nil`) in tutti i tipi atomici; si dice che `nil` è un valore *polimorfo*, che appartiene cioè a molti tipi.

Ogni definizione di tipo associa un nome (etichetta) a un tipo; per esempio, `Indirizzo: string` è una definizione di tipo che associa il nome "Indirizzo" al tipo atomico `string`.

Tipi di dati complessi I costruttori di tipo consentono di costruire tipi, detti *tipi di dati complessi*, che descrivono la struttura delle istanze (anche dette *oggetti complessi* di una base di dati a oggetti). Una definizione ricorsiva dei tipi di dati complessi, basata sui costruttori di tipo, è la seguente. Assumiamo come dato un insieme di tipi di dati atomici.

- Il costruttore *record* consente di costruire tipi le cui istanze sono tuple di valori (complessi) di tipi differenti. Se $T_1, \ldots, T_n$ sono nomi di tipo e $A_1, \ldots, A_n$

sono etichette distinte, che chiameremo *attributi* del record, $T = record\text{-}of(A_1 : T_1, \ldots, A_n : T_n)$ è un *tipo record*.

- I costruttori di *insieme*, *multi-insieme* e *lista* consentono di costruire collezioni di valori (eventualmente complessi) dello stesso tipo. Gli insiemi (*set*) sono collezioni non ordinate e prive di duplicati, i multi-insiemi (*bag*) sono collezioni non ordinate che possono presentare elementi duplicati, le liste (*list*) sono collezioni ordinate che possono presentare elementi duplicati. Se T_1 è un tipo, $T = set\text{-}of(T_1)$ è un *tipo insieme*, $T = bag\text{-}of(T_1)$ è un *tipo multi-insieme* e $T = list\text{-}of(T_1)$ è un *tipo lista*.

Dato un tipo complesso T, un oggetto che ha per tipo T si dice *istanza* di T. Come è consuetudine in molte basi di dati a oggetti, assumiamo che una definizione di tipo di dato inizi sempre con un costruttore di tipo record. Dato un oggetto x di tipo $T = record\text{-}of(A_1 : T_1, \ldots, A_n : T_n)$, possiamo in tal modo parlare degli attributi $A_1, \ldots, A_n$ come *proprietà* di x.

L'uso dei costruttori di tipo garantisce la cosiddetta *complessità strutturale* degli oggetti; in particolare, se un oggetto del mondo reale è complesso, i costruttori di tipo ci consentono di modellarlo in modo accurato. Alcuni sistemi non supportano però tutti i costruttori, e d'altra parte non è in genere conveniente costruire tipi eccessivamente complessi, perché poi diventa difficile accedere, tramite linguaggi di programmazione e interrogazione, ai componenti del tipo.

Vediamo un esempio di definizione di tipo per l'oggetto complesso AUTOMOBILE, caratterizzato da varie proprietà: Targa, Modello, Produttore, Colore, Prezzo, PartiMeccaniche. Alcune di queste proprietà hanno a loro volta una struttura complessa:

```
Automobile: record-of(
              Targa: string,
              Modello: string,
              Produttore: record-of(
                          Nome: string,
                          Presidente: string,
                          Stabilimenti: set-of(
                            record-of(
                               Nome: string,
                               Città: string,
                               Addetti: integer))),
              Colore: string,
              Prezzo: integer,
              PartiMeccaniche: record-of(
                          Motore: string,
                          Ammortizzatore: string))
```

Data questa definizione di tipo, è possibile esemplificare valori che sono compatibili con essa. Per esempio, assumendo una rappresentazione dei valori in cui i record sono

contenuti all'interno di parentesi quadre e gli insiemi all'interno di parentesi graffe, un valore compatibile con la definizione precedente è:

```
V1: ["DB123MS", "Punto",
       ["Fiat", "Agnelli",
          {["Mirafiori","Torino", 10000],
           ["trattori", "Modena", 1000]}],
     "blu", 9500, ["1100cc", "A olio"]}
```

Dato un valore di tipo record, è possibile accedere ai suoi componenti tramite la classica *notazione punto* (*dot notation*), che può essere applicata in modo ricorsivo. Per esempio:

```
V1.Colore="blu"
V1.Produttore.Presidente="Agnelli"
V1.PartiMeccaniche.Ammortizzatore="A olio"
```

Oggetti e valori La possibilità di introdurre un'arbitraria complessità strutturale, illustrata da questo esempio, soddisfa l'esigenza di associare a un unico oggetto una struttura arbitrariamente complessa; quindi, un automezzo (oppure un circuito integrato) viene descritto in modo più articolato e unitario che non, per esempio, usando il modello relazionale e separandone la descrizione in tante tabelle. Questo esempio illustra, però, anche il principale limite di una descrizione strutturale basata su "valori": per ogni automobile, per esempio della Fiat, viene ripetuta la descrizione del costruttore, che a sua volta si compone di vari dati, tra cui il nome del presidente e l'insieme degli stabilimenti. Tale descrizione introduce ovviamente della ridondanza, e contraddice i principi di normalizzazione dei dati, presentati nel Capitolo 9.

Per ovviare a questo problema è necessario decomporre la struttura precedente; per rappresentare i legami tra i diversi frammenti si usano i riferimenti fra oggetti (OID). La parte strutturale di un oggetto è in realtà costituita da una coppia (OID, *Valore*); il valore è un'istanza del tipo dell'oggetto; lo chiamiamo lo "stato" dell'oggetto. L'OID garantisce l'individuazione in modo univoco dell'oggetto nella base di dati e consente di costruire riferimenti fra oggetti; è assegnato dal DBMS e in molti sistemi non è visibile all'utente. Un oggetto può includere riferimenti espliciti ad altri oggetti; denotiamo ciò, a livello di schema, usando la notazione $*T$, che denota gli OID di oggetti di tipo T. Se una proprietà di un oggetto ha tipo $*T$, si dice che essa ha *per valore un oggetto* (è *object-valued*). La seguente definizione introduce riferimenti a oggetti:

```
Automobile: record-of(Targa: string,
                      Modello: string,
                      Produttore: *Costruttore,
                      Colore: string,
                      Prezzo: integer,
                      PartiMeccaniche: record-of
                          (Motore: string,
```

```
                                    Ammortizzatore: string))
Costruttore: record-of(Nome: string,
                       Presidente: string,
                       Stabilimenti:
                           set-of(*Stabilimento))
Stabilimento: record-of(Nome: string,
                        Città: string,
                        Addetti: integer)
```

Un insieme di oggetti compatibili con le definizioni di tipo appena introdotte è il seguente:

```
O1:  <OID1, ["DB123MS", "Punto", OID2, "blu", 9500,
            ["1100cc", "A olio"]]>
O2:  <OID2, ["Fiat","Agnelli", {OID3,OID4}]>
O3:  <OID3, ["Mirafiori", "Torino", 10000]>
O4:  <OID4, ["trattori", "Modena", 1000]>
```

L'esempio mostra che le proprietà object-valued consentono riferimenti fra oggetti (da una automobile al suo costruttore, dal costruttore ai suoi stabilimenti) e la condivisione di oggetti da parte di altri oggetti (lo stesso costruttore viene referenziato da varie automobili). Per esempio:

- il valore `O1.Produttore` è l'OID dell'oggetto `O2`;
- il valore `O1.Produttore.Presidente` è la stringa `Agnelli`.

Identità e uguaglianza L'uso di OID consente anche di garantire la possibilità che due oggetti distinti abbiano lo stesso stato e differiscano solo per l'OID (per esempio, due automobili con le stesse proprietà); questa possibilità non è consentita dal modello relazionale. Diremo quindi che due oggetti `O1` e `O2` sono identici (`O1=O2`) quando hanno lo stesso identificatore (e ovviamente anche lo stesso stato), e utilizzeremo la nozione di uguaglianza per confrontare lo stato di due oggetti.

Nel modello a oggetti, esistono due nozioni di uguaglianza:

- l'*uguaglianza superficiale* (==) richiede che due oggetti abbiano lo stesso stato.
- l'*uguaglianza profonda* (===) richiede che due oggetti abbiano identici valori ottenuti sostituendo ricorsivamente gli oggetti raggiungibili tramite OID agli OID stessi, ma non necessariamente lo stesso stato.

Si noti che lo stato include le proprietà object-valued (OID), perciò l'uguaglianza superficiale implica l'uguaglianza profonda; si noti inoltre che la procedura indicata per costruire i valori raggiungibili nella definizione di uguaglianza profonda deve essere costruita con attenzione, altrimenti potrebbe non terminare in presenza di riferimenti ciclici. In genere, i sistemi offrono solo un operatore per verificare l'uguaglianza superficiale di due oggetti, mentre l'uguaglianza profonda deve essere programmata tramite opportuni predicati sui valori.

Per esempio, si considerino le seguenti definizioni di tipo e i seguenti oggetti:

```
T1: record-of(A: integer, B: *T2)
T2: record-of(C: character, D: *T3)
T3: record-of(E: integer)

O1: <OID1, [120, OID4]> di tipo T1
O2: <OID2, [120, OID4]> di tipo T1
O3: <OID3, [120, OID5]> di tipo T1
O4: <OID4, [a, OID6]> di tipo T2
O5: <OID5, [a, OID7]> di tipo T2
O6: <OID6, [15]> di tipo T3
O7: <OID7, [15]> di tipo T3
```

In questo caso:

- le uguaglianze superficiali sono: `O1==O2`, `O6==O7`;
- le uguaglianze profonde sono: `O1===O2`, `O1===O3`, `O2===O3`, `O4===O5`, `O6===O7`;
- la condizione per definire l'uguaglianza profonda di oggetti `X` e `Y` di tipo `T1`, che può essere programmata, è:

```
X.A=Y.A and X.B.C=Y.B.C and X.B.D.E==Y.B.D.E.
```

13.1.2 Classi

Una classe raccoglie oggetti dello stesso tipo; essa funge cioè da contenitore di oggetti, che possono essere dinamicamente aggiunti e tolti alla classe. Gli oggetti che appartengono alla stessa classe sono omogenei, sono cioè dotati dello stesso tipo. Nel DDL, le definizioni di tipo possono essere talvolta date nell'ambito della definizione della classe. In genere, la definizione di una classe è separata in due parti.

- L'*interfaccia* descrive il tipo statico e dinamico degli oggetti appartenenti alla classe; il tipo dinamico include la segnatura di tutti i metodi della classe. Ogni segnatura contiene l'elenco del nome e del tipo dei parametri del metodo, che possono essere di input e/o di output; la conoscenza della segnatura consente di invocare il metodo da un programma.
- L'*implementazione* descrive il codice dei metodi e, talvolta, le strutture dati per memorizzare gli oggetti.

Affronteremo la descrizione del tipo dinamico e dell'implementazione nel Paragrafo 13.1.3, dedicato ai metodi.

Il principio di *incapsulamento*, un'importante astrazione dei linguaggi a oggetti, deriva dal più generale concetto di *tipo di dato astratto*, che attribuisce a ciascun oggetto un'interfaccia e un'implementazione. L'interfaccia descrive solo le operazioni applicabili sull'oggetto, mentre l'implementazione nasconde la struttura dati e l'effettiva costruzione delle operazioni. Nelle basi di dati a oggetti, tuttavia, lo stato è

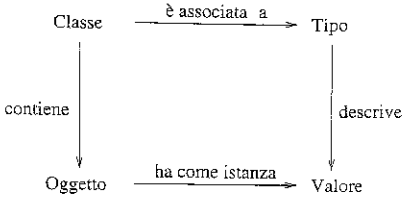

Figura 13.1 Relazione fra valori, oggetti, tipi e classi.

spesso visibile tramite alcune interfacce utente (per esempio, dal linguaggio di interrogazione); pertanto, rispetto a un'interpretazione rigorosa di tipo di dato astratto, le basi di dati a oggetti "espongono" ai loro utenti la struttura dello stato degli oggetti.

La distinzione fra tipi e classi è uno degli argomenti più controversi nell'ambito dei linguaggi di programmazione e delle basi di dati a oggetti; nel nostro modello dei dati, i tipi sono astrazioni che consentono di descrivere sia lo stato sia il comportamento, mentre le classi descrivono sia la rappresentazione estensionale degli oggetti, sia l'implementazione dei metodi relativi a un tipo; il tipo, cioè, descrive proprietà astratte, mentre la classe descrive la realizzazione di tali proprietà astratte tramite strutture dati e programmi. Abbiamo quindi presentato un modello dei dati in cui:

- tipi e classi sono concetti distinti;
- ogni classe è associata a un solo tipo;
- il concetto di classe descrive sia l'implementazione sia l'estensione di un tipo.

Il rapporto che intercorre tra valori, oggetti, tipi e classi è sintetizzato in Figura 13.1. Ogni oggetto ha un valore, che appartiene a un tipo; ogni oggetto appartiene a una classe, che è associata a un tipo; il tipo della classe è il tipo cui appartiene il valore degli oggetti contenuti nella classe.

Modelli dei dati più complessi riservano al concetto di classe il solo ruolo di definire l'implementazione dei metodi, introducendo poi un terzo concetto, quello di *estensione*, che consente di inserire oggetti dello stesso tipo in collezioni differenti e dare nomi distinti a queste collezioni (per esempio, il tipo Cittadino potrebbe corrispondere a una medesima classe ma a collezioni differenti, denominate MILANESE e FIORENTINO). In tal caso sarebbero presenti, ben distinti, i tre concetti di tipo, classe ed estensione. D'altra parte, alcuni modelli dei dati più semplici non distinguono neppure tra tipi e classi, in quanto fanno coincidere i due concetti e attribuiscono al tipo anche il ruolo di definire estensioni e implementazioni dei metodi.

Per esempio, vediamo come la definizione di classe includa sintatticamente anche la definizione di tipo; si noti l'uso di *nomi di classi* nella definizione di tipo, che vengono implicitamente interpretati come riferimenti:[1]

[1] La sintassi che verrà utilizzata per presentare le caratteristiche strutturali delle classi deriva dal sistema O2, una base di dati a oggetti che ha avuto una particolare influenza nello sviluppo di questa

```
add class Automobile
  type tuple(Targa: string,
             Modello: string,
             Produttore: Costruttore,
             Colore: string,
             Prezzo: integer,
             PartiMeccaniche:
                  tuple(Motore: string,
                        Ammortizzatore: string))
add class Costruttore
  type tuple(Nome: string,
             Presidente: Persona,
             Stabilimenti:
                  set(Stabilimento))

              add class Stabilimento
                type tuple(Nome: string,
                           Città: string,
                           Addetti: integer)

              add class Persona
                type tuple(Nome: string,
                           Residenza: string,
                           CodFisc: string)
```

La struttura delle classi può essere rappresentata in modo grafico, mettendo in evidenza i legami fra le classi dovuti a proprietà object-valued. La Figura 13.2 mostra le quattro classi fin qui introdotte, inserite in uno schema che comprende anche altre classi e una gerarchia di generalizzazione, illustrata nel seguito. In aggiunta, la figura illustra una classe PERSONA, cui fanno riferimento sia l'attributo **Presidente** nella classe COSTRUTTORE, sia l'attributo **Pilota** introdotto nella classe AUTOSPORTIVA.

13.1.3 Metodi

I metodi vengono usati per manipolare gli oggetti di un OODBMS; la loro presenza è il principale elemento di innovazione di un OODBMS rispetto a una base di dati relazionale. Un metodo ha una *segnatura*, che ne descrive l'interfaccia e comprende tutte le informazioni che consentono di invocarlo, e un'*implementazione*, che contiene il codice del metodo; spesso le implementazioni dei metodi sono scritte in linguaggi di programmazione a oggetti, quali C++, Java e C#. Più propriamente, come già discusso, la segnatura del metodo è una delle componenti della definizione di tipo, mentre l'implementazione del metodo è una delle componenti della definizione di classe.

Ciascun metodo è associato con una specifica classe di oggetti. Assumeremo che ogni metodo abbia un numero arbitrario di *parametri di ingresso* e un unico *parametro di uscita*.

---

tecnologia, ma che non è più disponibile in commercio.

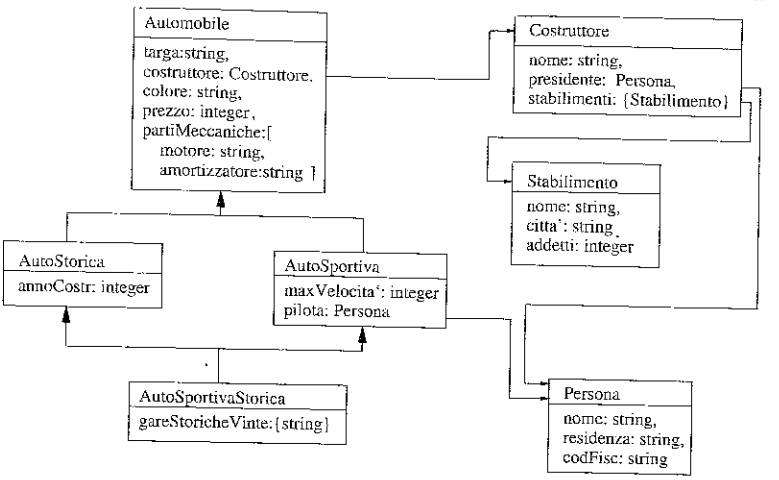

Figura 13.2 Schema di una base di dati a oggetti per la descrizione di automobili.

I metodi presenti in un OODBMS possono essere classificati in quattro categorie:

- i *costruttori* sono utilizzati per costruire gli oggetti a partire da parametri di ingresso;
- i *distruttori* servono per cancellare gli oggetti ed eventuali altri oggetti a essi collegati;
- gli *accessori* restituiscono informazioni sul contenuto degli oggetti;
- i *trasformatori* cambiano il contenuto dello stato degli oggetti.

Altri metodi non possono essere classificati in base a questo schema, e rispondono a specifiche esigenze applicative.

In molti sistemi si distingue tra metodi *pubblici* e *privati*; i metodi pubblici sono richiamabili in qualunque punto del programma applicativo, mentre i metodi privati sono richiamabili solo all'interno degli altri metodi della stessa classe. In base a questa distinzione, gli oggetti risultano massimamente incapsulati quando è possibile accedere loro soltanto tramite i metodi pubblici.

La segnatura può far parte sintatticamente della definizione della classe, oppure ciascun metodo può essere introdotto da una definizione specifica, in cui si nomina la classe cui appartiene. Questa seconda opzione consente una definizione incrementale dello schema, e verrà utilizzata nel seguito. Il metodo init è un trasformatore e inizializza alcune delle proprietà delle AUTOMOBILI; esso riceve come parametri di ingresso alcuni valori con cui inizializzare lo stato di un oggetto, e restituisce come parametro di uscita l'oggetto stesso. Ogni metodo si applica a un oggetto della classe

AUTOMOBILE, che si può perciò considerare un parametro di ingresso implicitamente definito. Vediamo la definizione della segnatura (istruzione `add method`):

```
add method init(Targa_par: string,
                Modello_par: string,
                Colore_par: string,
                Prezzo_par: integer): Automobile
            in class Automobile is public
```

Vediamo ora l'implementazione nella sintassi del linguaggio C++:

```
Ref<Automobile> Automobile::init
          (string Targa_par, string Modello_par,
           string Colore_par,  integer Prezzo_par)
{    self -> Targa = Targa_par;
     self -> Modello = Modello_par;
     self -> Colore = Colore_par;
     self -> Prezzo = Prezzo_par;
     return(self); }
```

Le implementazioni dei metodi sono scritte in C++; per l'integrazione tra le basi di dati a oggetti e il linguaggio C++, facciamo riferimento alla proposta del consorzio ODMG. L'espressione `Ref<Automobile>` dichiara che il tipo del risultato dell'invocazione del metodo è un riferimento a un oggetto della classe AUTOMOBILE; il costrutto `Ref`, che sfrutta un apposito meccanismo del linguaggio C++ (i *template*), consente di definire all'interno del programma C++ il tipo degli OID degli oggetti conservati nella base di dati. I riferimenti possono essere utilizzati come i puntatori nell'ambito del programma. L'espressione `Automobile::init` specifica invece che si sta definendo l'implementazione del metodo init della classe AUTOMOBILE. La variabile `self`, introdotta implicitamente nell' implementazione, denota l'oggetto cui viene applicato il metodo; con altra terminologia, `self` denota l'oggetto *ricevente*, cioè quello al quale viene inviato un *messaggio*. La notazione punto, introdotta nel Paragrafo 13.1.1, è resa con i meccanismi di accesso propri del C++, per cui la composizione dell'operatore che segue un riferimento (`*`) e dell'operatore che accede a un attributo (`.`) viene rappresentata con una freccia (`->`).

L'invocazione del metodo init in un programma avviene nel modo seguente:

```
Ref<Automobile> X;
X = new(Automobile);
X -> init("DB313MS", "Panda", "blu", 7000);
```

La prima istruzione del programma definisce una variabile di nome `X` e di tipo `Ref<Automobile>`. La seconda istruzione crea un oggetto della classe AUTOMOBILE, tramite l'invocazione del metodo new, e lo rende accessibile tramite la variabile `X`; il metodo polimorfo new è offerto da tutte le classi per creare nuovi oggetti e inserirli nella classe. Infine, la terza istruzione applica il metodo init all'oggetto, inizializzandone alcune proprietà. Si noti che nella chiamata di un metodo si

deve indicare l'oggetto target (o ricevente) e il nome del metodo, seguito dall'elenco dei valori attuali dei parametri di ingresso. Al termine dell'esecuzione del metodo, viene restituito come parametro di uscita l'identificatore dell'oggetto su cui il metodo stesso è invocato.

Il metodo Aumento nella classe AUTOMOBILE è un *trasformatore*; esso incrementa il Prezzo di un certo `Ammontare`; `Ammontare` è l'unico parametro del metodo.

```
add method Aumento(Ammontare:integer)
    in class Automobile is public
...
void Automobile::Aumento(Ammontare:integer)
{ self -> Prezzo += Ammontare;}
```

Il prossimo esempio mostra l'invocazione innestata dei metodi init e Aumento, possibile in quanto il metodo init restituisce come parametro di uscita l'oggetto target:[2]

```
Ref<Automobile> X;
X = new(Automobile);
(X -> init("DB313MS", "Panda", "blu", 7000))
   -> Aumento(1200);
```

Possiamo infine riassumere le proprietà degli oggetti. Ogni oggetto ha un identificatore, uno stato e un comportamento. L'*identificatore* garantisce l'individuazione in modo univoco dell'oggetto nella base di dati, e consente di costruire riferimenti fra oggetti. Lo *stato* di un oggetto è l'insieme dei valori assunti dalle sue proprietà in un determinato istante. Infine, il *comportamento* di un oggetto è definito dai metodi che possono essere applicati all'oggetto stesso, e ne predefiniscono l'evoluzione nel tempo.

Conflitto di impedenza Gli esempi di metodi appena visti consentono di introdurre una discussione relativa a una caratteristica importante delle basi di dati a oggetti, nelle quali il programmatore può manipolare oggetti persistenti tramite le istruzioni del linguaggio di programmazione. Si dice che i DBMS a oggetti risolvono il cosiddetto *conflitto d'impedenza* (*impedance mismatch*) che caratterizza invece i sistemi relazionali; esso consiste nella necessità di far dialogare un linguaggio di programmazione, contenente variabili scalari, con il linguaggio SQL, che estrae insiemi di tuple. In effetti, i meccanismi di interazione fra l'ambiente di programmazione e i server SQL, basati sull'uso di cursori per scandire il risultato delle interrogazioni (descritti nel Capitolo 10), sono assai rigidi e poco "amichevoli"; viceversa, nella programmazione degli OODBMS gli oggetti persistenti e temporanei possono essere gestiti con semplicità (per esempio, tramite meccanismi di assegnamento). Si dice in tal caso che la persistenza è una caratteristica *ortogonale*, di cui il programmatore può essere inconsapevole.

[2] Si noti che l'esempio non è molto sensato sul piano applicativo, perché prima inizializza e poi modifica la proprietà Prezzo, dandone un immediato aumento.

13.1.4 Gerarchie di generalizzazione

La possibilità di stabilire gerarchie di generalizzazione fra classi è probabilmente l'astrazione più importante nei linguaggi e nelle basi di dati a oggetti. Le generalizzazioni consentono la definizione di una sotto-classe a partire da una super-classe; la sotto-classe *eredita* lo stato e il comportamento della super-classe, e può in aggiunta rendere il proprio stato e comportamento più specifici tramite l'aggiunta di attributi e metodi. Le gerarchie di generalizzazione fra classi sono del tutto analoghe alle gerarchie di generalizzazione fra entità offerte nel modello Entità-Relazione; esse garantiscono la cosiddetta *complessità semantica* degli oggetti. In una gerarchia di generalizzazione:

- tutti gli oggetti delle sotto-classi appartengono automaticamente alle super-classi;
- tutti gli attributi e i metodi delle super-classi vengono *ereditati* dalle sotto-classi;
- è possibile introdurre nella descrizione delle sotto-classi dei nuovi attributi e dei nuovi metodi.

È possibile ridefinire l'implementazione di un metodo senza modificare la sua interfaccia, in modo da ottenere varie implementazioni dello stesso metodo, richiamabili in modo uniforme su oggetti di tipo diverso appartenenti alla gerarchia: il sistema utilizza l'implementazione più specifica in base al tipo dell'oggetto; affronteremo questo aspetto nel Paragrafo 13.1.6. È poi possibile, anche se ciò pone dei problemi, *raffinare* (cioè rendere più specifici) gli attributi e i metodi presenti nelle super-classi; affronteremo questo aspetto nel Paragrafo 13.1.7.

Le generalizzazioni godono della proprietà transitiva; perciò, se C_1 è una sotto-classe di C_2 e C_2 è una sotto-classe di C_3, anche C_1 è una sotto-classe di C_3. La relazione di sotto-classe deve essere aciclica.

In virtù della proprietà di ereditarietà, gli attributi e i metodi definiti nell'ambito di tutte le super-classi sono automaticamente ereditati dalle sotto-classi, le cui definizioni possono pertanto limitarsi a introdurre nuovi attributi e metodi. Per esempio, definiamo le sotto-classi AUTOSPORTIVA e AUTOSTORICA della classe AUTOMOBILE:

```
add class AutoSportiva
  inherits Automobile
  type tuple(MaxVelocita:integer,
             Pilota:Persona)

add class AutoStorica
  inherits Automobile
  type tuple(AnnoCostr:integer)
```

In virtù della ereditarietà, la classe AUTOSTORICA eredita gli attributi e i metodi definiti per AUTOMOBILE (per esempio, gli attributi Modello e Colore e i metodi init e Aumento). È possibile quindi invocare il metodo init su un oggetto della classe AUTOSTORICA:

```
Ref<AutoStorica> X;
X = new(AutoStorica);
X -> init("Mi56543", "Ferrari", "rossa", 170000);
X -> AnnoCostr = 1957;
```

L'implementazione di un metodo m richiamabile in una classe C_1 può non essere definita nella classe C_1; in tal caso, è necessario che un'implementazione di m sia definita per una super-classe C_2 di C_1; nel caso vi siano due super-classi C_2 e C_3 della classe C_1 che possiedono un'implementazione di m, si sceglie l'implementazione nella classe inferiore in gerarchia, cioè l'implementazione più specifica rispetto alla classe C_1.

Migrazioni fra classi In presenza di gerarchie di generalizzazione, in alcuni OODBMS gli oggetti possono migrare da un livello della gerarchia a un altro; in altri OODBMS, invece, gli oggetti rimangono nella classe in cui sono creati durante tutta la loro esistenza. Si chiama *specializzazione* l'operazione tramite la quale un oggetto migra da una super-classe a una sotto-classe; in questo "cammino", lo stato dell'oggetto in genere si modifica, aggiungendo nuove proprietà. La migrazione inversa è detta *generalizzazione* e consente a un oggetto di migrare da una sotto-classe a una super-classe; lo stato, in genere, perde alcune delle sue proprietà.

Per esempio, un oggetto della classe AUTOMOBILE può essere specializzato, entrando a far parte della classe AUTOSTORICA, a un certo punto della sua esistenza; viceversa, un oggetto della classe AUTOSPORTIVA cui è applicata l'operazione di generalizzazione cessa di appartenere a tale classe, rimanendo però nella classe AUTOMOBILE.

Distinguiamo tra *istanze* e *membri* di una classe; un oggetto è istanza di una classe solo se essa è la *classe più specializzata* per l'oggetto nell'ambito di una gerarchia di generalizzazione; le istanze di una classe sono automaticamente membri delle sue super-classi. In alcuni sistemi, ogni oggetto può essere un'istanza di una sola classe; in altri, invece, alcuni oggetti possono essere istanze di più classi, cioè appartenere a due o più distinte classi più specializzate, tra loro incomparabili dal punto di vista della gerarchia. Nel nostro esempio, un oggetto della classe AUTOMOBILE può essere istanza di due classi se esso viene specializzato sia nella classe AUTOSPORTIVA sia nella classe AUTOSTORICA.

Ereditarietà multipla In alcuni sistemi è consentito a una classe di ereditare da più super-classi; questa situazione si dice *ereditarietà multipla*. Per esempio, è possibile definire la classe AUTOSPORTIVASTORICA, nel modo seguente:

```
add class AutoSportivaStorica
  inherits AutoSportiva, AutoStorica
  type tuple(GareStoricheVinte: set(string))
```

Si noti che questa gerarchia di classi definisce una situazione di appartenenza alle classi, illustrata in Figura 13.3, in cui:

- le istanze della classe AUTOSPORTIVASTORICA sono automaticamente membri delle classi AUTOMOBILE, AUTOSPORTIVA e AUTOSTORICA;
- le istanze della classe AUTOSPORTIVA o AUTOSTORICA che non fanno parte della classe AUTOSPORTIVASTORICA sono automaticamente membri della classe AUTOMOBILE;

- infine, esistono istanze della classe AUTOMOBILE che non fanno parte delle classi AUTOSPORTIVA o AUTOSTORICA.

Si noti che, qualora il sistema ammetta oggetti che sono contemporaneamente istanze delle due classi AUTOSPORTIVA e AUTOSTORICA (come classi più specializzate), essi possono esistere nella base di dati *senza* far parte della classe AUTOSPORTIVASTORICA; l'appartenenza di un oggetto a una classe richiede infatti un'esplicita operazione di inserimento di quell'oggetto nella classe. Questo fatto è illustrato in Figura 13.3 tramite una rappresentazione della classe AUTOSPORTIVASTORICA come sotto-insieme dell'intersezione delle classi AUTOSPORTIVA e AUTOSTORICA.

Conflitti L'ereditarietà multipla oppure la presenza di oggetti che sono istanze di molteplici classi possono essere la fonte di *conflitti di nome*, qualora due o più superclassi abbiano attributi o metodi con lo stesso nome. In tal caso, occorre definire delle politiche per la risoluzione del conflitto, in modo da rendere non ambiguo il meccanismo di ereditarietà. Alcune delle soluzioni possibili sono elencate di seguito.

- Rilevare il conflitto all'atto della definizione delle classi e non accettare come corrette le definizioni. Se si esclude la possibilità di molteplici istanze per un oggetto, è sufficiente fare questa verifica all'atto della definizione di classi con ereditarietà multipla. Questa soluzione ha lo svantaggio di imporre un ripensamento dello schema relativamente a parti già consolidate, in quanto il problema si elimina solo cambiando la struttura delle classi che sono causa di conflitto o rinunciando alla definizione della sotto-classe.
- Definire dei meccanismi per togliere ogni ambiguità dalla scelta; per esempio, utilizzando un ordinamento fra le classi definito a priori, oppure applicando il metodo nel contesto di una variabile target che sia stata esplicitamente definita membro di una sola delle classi sulle quali sono definiti gli attributi e i metodi che danno luogo a problemi di ambiguità.
- Ridefinire gli attributi e i metodi localmente, come descritto nel Paragrafo 13.1.6; la ridefinizione locale, infatti, elimina la visibilità del conflitto.

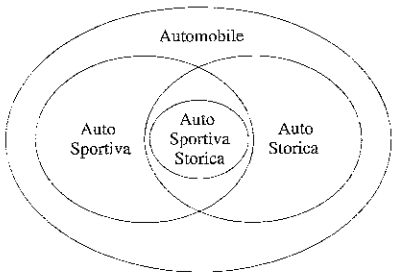

Figura 13.3 Rappresentazione degli oggetti appartenenti alle classi AUTOMOBILE, AUTOSPORTIVA, AUTOSTORICA e AUTOSPORTIVASTORICA.

13.1.5 Persistenza

Gli oggetti definiti in un programma possono essere *persistenti* oppure *temporanei*; gli oggetti temporanei cessano di esistere al termine dell'esecuzione del programma, mentre gli oggetti persistenti vengono inseriti nell'OODB. In genere, un oggetto diviene persistente tramite i seguenti meccanismi:

- tramite l'inserimento in una classe che è definita come persistente. Si adopera in tal caso la primitiva new, come descritto in alcuni esempi di questo paragrafo;
- tramite la *raggiungibilità* a partire da un altro oggetto persistente; per esempio, se vengono creati oggetti di tipo Automobile e di tipo Costruttore in cui il primo fa riferimento al secondo e poi il primo viene inserito in una classe persistente, anche il secondo diviene persistente. In tal modo, lo stato di un oggetto ricostruibile ricorsivamente tramite riferimenti è persistente;
- tramite la *denominazione*, cioè dando a un oggetto un nome (detto *handle*, cioè letteralmente "maniglia" di un oggetto) che possa essere utilizzato per ritrovarlo a una successiva invocazione del programma. Nell'esempio, usando l'interfaccia di integrazione per C++ proposta dall'ODMG, è possibile dichiarare una variabile e poi darle un nome, rendendo il corrispondente oggetto persistente:

```
Ref<AutoSportiva> X;
MioDB->set_object_name(X,"Ferrari_360_Modena");
```

Non tutti questi meccanismi per dare la persistenza agli oggetti sono supportati in ogni OODB. In alcuni sistemi, si definisce la persistenza a livello di classe, introducendo cioè classi persistenti e classi temporanee; la natura persistente o temporanea degli oggetti viene definita quando l'oggetto è inserito nella classe.

La persistenza tramite raggiungibilità garantisce la cosiddetta "integrità referenziale" dell'OODB, cioè il mantenimento automatico di vincoli di integrità referenziale fra le classi che sono simili ai vincoli di riferimento fra tabelle, discussi nel Capitolo 2. Tuttavia questo tipo di persistenza comporta una qualche difficoltà nella cancellazione degli oggetti dall'OODB. In pratica, un oggetto può essere cancellato dal sistema solo quando esso non è più raggiungibile tramite denominazione o tramite riferimenti da altri oggetti persistenti; questo processo si chiama *garbage collection*.

13.1.6 Ridefinizioni dei metodi

Una volta introdotta una gerarchia, è possibile ridefinire i metodi delle sotto-classi; questa tecnica, detta *overriding* dei metodi, è estremamente utile per garantire una interfaccia uniforme dei metodi, applicabile indipendentemente dall'appartenenza di un oggetto a uno specifico elemento della gerarchia delle classi, e un'implementazione che tiene viceversa conto della classe di appartenenza degli oggetti.

L'esempio classico che viene utilizzato per introdurre questo approccio consiste nel metodo display, che fornisce una rappresentazione dell'oggetto; supponiamo l'esistenza di una generica classe OGGETTO, da cui tutte le altre classi ereditano, in cui è definita l'interfaccia del metodo display, che non abbia parametri espliciti di input (l'oggetto target è un parametro implicito) e sia associata a un'implementazione fittizia.

Il metodo viene poi ridefinito nell'ambito di tutte le altri classi, che ereditano da OGGETTO; avremo per esempio le classi PROPRIETARIO, ABITAZIONE, MAPPA, CONDIZIONIVENDITA relative alle attività di un'agenzia immobiliare. Nell'ambito di ciascuna classe, il metodo display viene implementato in modo differente; per esempio, display applicato all'abitazione mostra una sua fotografia, alla mappa mostra una planimetria dell'abitazione, al proprietario e alle condizioni di vendita mostra una schermata con i dati relativi. È possibile cioè scrivere del codice che, in modo compatto, dato un insieme anche eterogeneo di oggetti S, richiama la funzione display su di essi; per esempio:

```
for X in S do X.display()
```

Ovviamente, la scelta di quale metodo invocare dipende dal tipo dell'oggetto cui il metodo si applica; in particolare, se il tipo dell'oggetto non può essere noto a tempo di compilazione (per esempio perché gli oggetti possono migrare fra le classi), tale scelta deve avvenire a tempo di esecuzione. Questa caratteristica è detta *late binding* (la traduzione letterale è: "collegamento tardivo") e comporta, da parte del sistema, la scelta del codice specifico per l'esecuzione del metodo e il suo collegamento con il resto del codice applicativo a tempo di esecuzione.

Per effetto della ridefinizione, è possibile avere quindi varie versioni dello stesso metodo con identica interfaccia (e, in particolare, identico nome); questo fenomeno si chiama *overloading* dei nomi di metodi.

Vediamo un esempio di uso di metodo con overloading e overriding. Si consideri la gestione di dati per assistere il progetto di sistemi software. In tal caso, introduciamo una gerarchia di generalizzazione con la classe generica FILE e le sotto-classi SORGENTE e DOCUMENTAZIONE. Queste classi sono caratterizzate da attributi introdotti localmente; il metodo init è in grado di inizializzare gli oggetti in base al loro tipo.

```
add class File
type tuple(Nome: string,
           Creatore: Utente,
           Data: date)
method init(Nome_par: string) is public

add class Sorgente
    inherits File
type tuple(Responsabile: Utente)

add class Documentazione
    inherits File
type tuple(Validità: date)
...

Ref<File> File::init(string Nome_par)
{      self -> Nome = Nome_par;
       self -> Creatore = DatabaseUser();
       self -> Data = Today();
       return(self);}
```

```
Ref<Sorgente> Sorgente::init(string Nome_par)
{       (Ref<File>) self -> init(Nome_par);
        self -> Responsabile = DatabaseUser(); }

Ref<Documentazione> Documentazione::init
                                (string Nome_par)
{       (Ref<File>) self -> init(Nome_par);
        self -> Validità = EndOfPeriod; }
```

Si noti che il metodo init, definito nell'ambito della classe FILE, viene riutilizzato nell'ambito delle due sotto-classi; il nome del tipo `Ref<File>`, che compare racchiuso tra parentesi prima dell'invocazione di init all'interno dei metodi per SORGENTE e IMPLEMENTAZIONE, rappresenta l'applicazione dell'operatore di *cast*, con il quale si specifica di considerare la variabile come istanza di un tipo diverso rispetto a quello dichiarato; in entrambi i casi il metodo init che viene richiamato è quello definito nell'ambito della classe FILE. Questo esempio ci mostra pertanto un'insieme di metodi che soddisfano l'ultimo criterio di buona progettazione introdotto nel Paragrafo 13.1.3, cioè il riutilizzo dei metodi definiti nelle classi generiche nell'implementazione dei metodi nelle classi più specifiche.

Si noti inoltre l'uso delle funzioni `DatabaseUser()` e `Today()`, e della variabile globale `EndOfPeriod`, che pur essendo oggetti accessibili a livello globale sono accettabili nel contesto delle tre implementazioni dei metodi. Tramite queste definizioni, è possibile inizializzare i file indipendentemente dalla loro appartenenza alla gerarchia; nel codice seguente, è necessario semplicemente porre al posto di CLASSE uno qualunque dei tre nomi di classi introdotti nell'esempio:

```
Ref<CLASSE> X;
X = new(CLASSE);
X -> init("MioProgramma");
```

13.1.7 Ridefinizione con raffinamenti di tipo

I meccanismi di ridefinizione visti nel paragrafo precedente non modificano l'interfaccia dei metodi. È però possibile raffinare attributi e metodi modificandone l'interfaccia, introducendo i meccanismi di sotto-tipazione.

La sotto-tipazione è una relazione fra tipi. Intuitivamente, T_1 è un sotto-tipo di T_2 se ammette un insieme di valori possibili più specifico. Per esempio, se T_2 è un tipo enumerativo, T_1 può essere definito come un sotto-insieme dei valori di T_2. Ogni tipo è sotto-tipo di se stesso, e nei sistemi in cui le classi sono interpretate anche come tipi, due classi in relazione di sotto-classe sono anche in relazione di sotto-tipo.

Un caso importante è quello dei record: dato un tipo record $T_X = [A_1 : T_1, \ldots, A_m : T_m]$, un altro tipo record T_Y è un sotto-tipo di T_X se ha la struttura $T_Y = [A_1 : T'_1, \ldots, A_m : T'_m, A_{m+1} : T'_{m+1}, \ldots, A_n : T'_n]$, con T'_i sotto-tipo di T_i per $1 \leq i \leq m$ e nessuna restrizione su T'_i per $(m+1) \leq i \leq n$; il sotto-tipo può cioè avere attributi T'_i che possono essere sotto-tipi degli attributi T_i, e avere attributi aggiuntivi che lo rendono più specifico. In realtà, abbiamo già visto che l'aggiunta di attributi caratterizza normalmente la definizione delle sotto-classi.

Data una definizione precisa di sotto-tipo, una ridefinizione con raffinamento può far riferimento sia agli attributi sia ai metodi.

- Si consideri la definizione di una classe C_2 che eredita da C_1 e un generico attributo $A : T$ di C_1. La *co-varianza degli attributi* consente di associare all'attributo A, ridefinito in C_2, un sotto-tipo T' del tipo T.
- Si consideri la definizione di una classe C_2 che eredita da C_1 e un generico metodo m caratterizzato da un certo numero di parametri di ingresso, di tipo T_i, e da un parametro di uscita, di tipo T. La *co-varianza del parametro di uscita* si ottiene dando al metodo m' ridefinito in C_2 un sotto-tipo T' di T. Per quanto riguarda i parametri di ingresso:
  - se, per ogni parametro di ingresso di m, T_i' è un sotto-tipo di T_i, si ha la *co-varianza dei parametri di ingresso*;
  - se, per ogni argomento di ingresso di m, T_i è un sotto-tipo di T_i', si ha la *contro-varianza dei parametri di ingresso*.

La co-varianza degli attributi e dei parametri di ingresso dei metodi, adottata nella maggior parte dei sistemi, è la nozione più intuitiva e utile nello specializzare attributi e metodi, ma non assicura la correttezza della sostituzione dai parametri formali a quelli attuali a tempo di compilazione. Ciò è illustrato dal seguente esempio.

Riprendiamo la situazione del Paragrafo 13.1.6 e introduciamo due nuove classi UTENTE e PROGRAMMATORE, la cui struttura però non interessa:

```
add class Utente ....
add class Programmatore inherits Utente ....

add class File
type tuple(Nome: string,
           Creatore: Utente,
           Data: date)
method init(Nome_par: string,
            Utente_par: Utente):
         File is public

add class Sorgente inherits File
type tuple(Creatore: Programmatore)
method init(Nome_par: string,
            Utente_par: Programmatore): Sorgente
    is public
...
Ref<File> File::init
            (string Nome_par, Utente Utente_par)
{      self -> Nome = Nome_par;
       self -> Creatore = Utente_par;
       self -> Data = Today();
       return(self); }
```

```
Ref<Sorgente> Sorgente::init(string Nome_par,
                                Programmatore Utente_par)
{       (Ref<File>)self -> init(Nome_par, Utente_par);
        return(self); }
```

Osserviamo che l'attributo Creatore viene ridefinito nel contesto della classe SORGENTE: il creatore non è un generico utente, bensì un programmatore. Questo è un esempio di *ridefinizione co-variante di attributi*, che può aver senso da un punto di vista applicativo in quanto si vuole imporre che solo i programmatori possano creare file sorgenti.

Il metodo init riceve in ingresso il parametro relativo all'utente, e si aspetta che venga correttamente invocato: l'inizializzazione di un generico FILE può ricevere come parametro attuale un generico utente, ma l'inizializzazione di un SORGENTE deve ricevere come parametro attuale un programmatore. Questo è un esempio di *ridefinizione co-variante dei parametri di ingresso*. Si noti l'impossibilità di controllare a tempo di compilazione la correttezza dell'invocazione del metodo, se è possibile che un oggetto possa migrare dinamicamente dalla classe UTENTE alla classe PROGRAMMATORE.

Infine, il parametro di uscita del metodo viene anch'esso ridefinito, poiché quando esso è invocato in una sotto-classe restituisce un tipo più specifico. Questo è un esempio di *ridefinizione co-variante del parametro di uscita*, che invece non pone problemi di correttezza dei tipi.

13.2 Basi di dati relazionali a oggetti

Le "basi di dati relazionali a oggetti" (*Object-Relational*, da cui ORDBMS) si caratterizzano per la scelta di un approccio evolutivo e non rivoluzionario rispetto alle basi di dati relazionali; questi sistemi, cioè, si pongono come obiettivo la realizzazione di estensioni compatibili della nozione classica di tabella, che consentono di esprimere la maggior parte dei concetti degli OODBMS. In questo paragrafo, mostriamo il linguaggio utilizzato per definire strutture negli ORDBMS, per differenza rispetto alle basi di dati a oggetti "pure" (OODBMS) fin qui illustrate. Come già osservato, la distanza tra questi due tipi di sistemi non è elevata, specie nel modello dei dati, in cui vengono definiti concetti molto simili. Introdurremo il modello dei dati per poi mostrare alcune caratteristiche del linguaggio di interrogazione, utilizzando come riferimento la versione più recente dello standard SQL. Come detto precedentemente, il nome convenzionale con cui si fa riferimento all'ultima versione dello standard è SQL-3 ed è questo il nome che utilizzeremo in questa trattazione.

13.2.1 Modello dei dati di SQL-3

Il modello dei dati degli ORDBMS è compatibile con il modello dei dati relazionale. Pertanto, nel modello ORDBMS è possibile definire tabelle, quali per esempio la tabella PERSONA, in cui compaiono alcuni dei vincoli di integrità SQL:

```
create table Persona(
        Nome varchar(30) not null,
        Residenza varchar(30),
        CodFisc char(16) primary key)
```

Tuttavia, l'approccio suggerito negli ORDBMS è quello di definire prima il tipo TipoPersona, in modo da renderlo riutilizzabile. L'uso di costruttori di tipo complessi – che estendono significativamente la nozione di dominio presente in SQL – è la prima differenza significativa rispetto alle basi di dati relazionali.

Tipi strutturati I tipi in SQL-3 si distinguono in *tipi distinti* e *tipi strutturati*; i primi corrispondono al meccanismo di definizione dei domini di SQL visto nel Capitolo 4 e consentono di definire un alias per i tipi predefiniti del modello dei dati SQL; i secondi si utilizzano essenzialmente per costruire le strutture delle tuple da inserire nelle tabelle. Pertanto, il medesimo effetto della definizione precedente della tabella PERSONA si ottiene dalle due seguenti definizioni:

```
create type TipoPersona as (
      Nome varchar(30),
      Residenza varchar(30),
      CodFisc char(16))
not final
ref from (CodFisc)

create table Persona of TipoPersona (
    ref is CodFisc derived)
```

Si può osservare come la definizione di TipoPersona termini con due nuovi costrutti. La clausola `not final` specifica che il tipo ammette la definizione di sotto-tipi; la sintassi SQL-3 obbliga la specifica di questa clausola in ogni definizione di tipo. La clausola `ref from (CodFisc)` specifica invece che l'attributo CodFisc costituisce l'attributo da utilizzare per realizzare riferimenti a istanze del tipo, ovvero svolge il ruolo di identificatore per il tipo; la clausola `ref` è obbligatoria tutte le volte che si prevede, come in questo caso, di utilizzare il tipo strutturato per la definizione di tabelle. La tabella PERSONA viene quindi creata come un contenitore di istanze del tipo TipoPersona. La sintassi SQL-3 obbliga a specificare come vengono realizzati i riferimenti alle istanze della tabella; in questo caso la dichiarazione `ref is CodFisc derived` specifica, in modo ridondante, che l'identificatore viene derivato dal valore dell'attributo CodFisc. Una differenza della definizione di PERSONA tramite il tipo è che il tipo TipoPersona può essere utilizzato anche in altre tabelle. È possibile quindi definire:

```
create table Industriale of TipoPersona (
    ref is CodFisc derived)

create table Pilota of TipoPersona (
    ref is CodFisc derived)
```

Si noti la corrispondenza fra le nozioni di oggetto e classe nell'ambito degli OODBMS e di tupla e tabella nell'ambito degli ORDBMS; negli ORDBMS, i termini di oggetto e di tupla possono essere usati in modo intercambiabile.

Come negli OODBMS, è possibile usare costruttori di tipo in modo ortogonale per costruire tipi arbitrariamente complessi. È possibile inoltre usare riferimenti da un tipo a un altro tipo, e quindi creare le premesse per condividere oggetti nella base di dati. Riprendiamo l'esempio illustrato nel Paragrafo 13.1 (con qualche semplificazione e qualche variante dovuta alle caratteristiche di SQL-3) e illustriamo la definizione dei corrispondenti tipi tupla, che includono TipoPersona già visto. Si noti l'uso del costrutto `multiset` [3] e `ref` (per denotare un riferimento da un tipo a un altro).

```
create type TipoStab as (
      Nome varchar(25),
      Citta varchar(7),
      Addetti integer)
not final

create type TipoCostr as (
     Nome varchar(25),
     Presidente ref(TipoPersona),
     Stabilimenti TipoStab multiset )
not final
ref is system generated

create type TipoPartiAuto as (
     Motore char(10),
     Ammortizzatore char(5))
not final

create type TipoAuto as (
     Targa char(7),
     Modello varchar(30),
     Produttore ref(TipoCostr),
     PartiMeccaniche TipoPartiAuto)
not final
ref is system generated
```

Si noti che i tipi TipoStab e TipoPartiAuto vengono usati nell'ambito dei tipi TipoCostr e TipoAuto senza però introdurre il costrutto `ref`, e quindi senza che vengano introdotti oggetti indipendenti. In tal modo, si possono costruire tuple che comprendono al loro interno sotto-tuple, garantendo un'arbitraria complessità strutturale. L'opzione `ref is system generated` specifica che è compito del sistema generare l'identificatore delle istanze del tipo.

[3] Il costruttore di insiemi `setof` non è previsto in SQL-3; versioni successive dello standard lo potranno aggiungere. La prima versione di SQL-3, SQL:1999, presentava solo il costrutto `array`.

A questo punto, è possibile creare tabelle per i soli concetti AUTOMOBILE e COSTRUTTORE, che si uniscono alle tabelle PRESIDENTE e PILOTA, già create.

```
create table Automobile of TipoAuto
( ref is AutomobileId system generated )

create table Costruttore of TipoCostr
( ref is CostrId system generated,
  Presidente with options scope Industriale)
```

Si noti che nella definizione dei tipi tupla TipoAuto e TipoCostr gli attributi AutomobileId e CostrId non compaiono; i nuovi termini rappresentano in effetti gli identificatori delle tuple, presenti in modo implicito nella definizione del tipo. La clausola `ref is AutomobileId system generated` conferma che i valori dell'identificatore d'oggetto sono creati dal sistema. Grazie all'introduzione dei nomi espliciti AutomobileId e CostrId, i valori dell'identificatore possono essere utilizzati nelle interrogazioni come gli altri attributi.

Si può inoltre osservare l'uso del costrutto `with options` che consente di fornire maggiori dettagli e introdurre restrizioni su componenti del tipo quando esso viene utilizzato nell'ambito di una tabella. In questo caso il costrutto viene utilizzato per associare la clausola `scope` all'attributo Presidente, con la quale si pone il vincolo che i valori presenti nell'attributo Presidente debbano essere riferimenti alle tuple appartenenti alla tabella INDUSTRIALE; se tale clausola fosse omessa, i valori dell'attributo Presidente potrebbero essere riferimenti a generici oggetti di tipo TipoPersona, presenti in una qualunque tabella che utilizza tale tipo.

Gerarchie In SQL-3 è possibile definire gerarchie di tipo e di tabelle. Le gerarchie di tipo estendono tipi precedentemente definiti aggiungendo nuove proprietà; per esempio è possibile costruire un tipo TipoAutoStorica aggiungendo a TipoAuto l'attributo AnnoCostr, come segue:

```
create type TipoAutoStorica
     under TipoAuto as (
     AnnoCostr integer)
not final
```

Le gerarchie sulle tabelle presentano una completa analogia con le gerarchie sulle classi viste nel Paragrafo 13.1. Quindi, le sotto-tabelle hanno per tipo un sotto-tipo delle tabelle da cui ereditano, e ogni oggetto (tupla) presente in una sotto-tabella appare come oggetto (tupla) nelle tabelle di livello gerarchicamente superiore. La definizione di AUTOMOBILESTORICA come sotto-tabella di AUTOMOBILE, che richiede comunque una clausola `under` nel contesto della creazione della sotto-tabella, può — dal punto di vista della costruzione del tipo — avvenire in due modi. È possibile far riferimento al sotto-tipo precedentemente definito, come segue:

```
create table AutomobileStorica of TipoAutoStorica
     under Automobile
```

In alternativa, è possibile definire in un unico passo tabella e tipo, con l'indicazione esplicita del tipo di AutomobileStorica, come segue:

```
create table AutomobileStorica of TipoAutoStorica
     under Automobile (AnnoCostr integer)
```

Si osservi come la sintassi SQL-3 richieda di non ripetere nella definizione di sotto-tabelle la clausola che esplicita la gestione dei riferimenti, in quanto anch'essa viene ereditata.

Metodi e funzioni I tipi definiti dall'utente in SQL-3 possono essere esplicitamente etichettati come tipi instanziabili o meno ([not] instantiable). I tipi classificati come non instanziabili non possono essere utilizzati direttamente nella definizione di tabelle; possono essere invece utilizzati come base per la definizione di sotto-tipi (in modo analogo alle classi *astratte* del modello a oggetti del linguaggio Java) o come componenti all'interno della definizione di altri tipi o tabelle. È inoltre possibile corredare i tipi, instanziabili o meno, di un insieme di metodi, che possono essere definiti in SQL-3 oppure in un linguaggio esterno di programmazione. I metodi di SQL-3 sono del tutto analoghi ai metodi discussi nel Paragrafo 13.1.3, e in particolare includono realizzazioni standard per il *costruttore*, gli *accessori* ai vari attributi e i *trasformatori*; è possibile negare agli utenti i privilegi d'accesso ai metodi, che vengono definiti come privilegi speciali, ottenendo l'effetto di incapsulare i dati.

Vediamo per esempio la definizione del tipo non instanziabile TipoPartiAuto, che comprende anche i metodi PotUnitaria e MaggiorPotUnitaria per esprimere, rispettivamente, il calcolo della potenza di ciascun cilindro del motore e un confronto tra l'istanza su cui viene invocato il metodo e un'altra istanza dello stesso tipo.

```
create type TipoPartiAuto as (
     Motore char(10),
     NumCilindri integer,
     Potenza integer,
     Cilindrata integer )
not instantiable
not final
method PotUnitaria ( )
  returns float,
method MaggiorPotUnitaria (
        ParteCfr TipoPartiAuto )
  returns boolean,
method DisegnoDisponibile ( )
  returns boolean
  language C
```

I metodi così definiti possono essere realizzati in SQL-3, oppure in un linguaggio di programmazione esterno; per esempio, il metodo DisegnoDisponibile è definito come esterno. In entrambi i casi, la specifica dei metodi richiede di definirne la *segnatura*, che, come nel Paragrafo 13.1.3, individua i parametri d'ingresso e i parametri

d'uscita, e poi l'*implementazione*. Per la segnatura si usa una notazione funzionale, in cui i parametri d'ingresso sono racchiusi in parentesi; per ogni parametro di ingresso si definisce il nome e il tipo. Il tipo del parametro d'uscita, che è unico (e può essere omesso) viene indicato dopo la clausola `returns`.

```
create instance method PotUnitaria( )
  returns float
  for TipoPartiAuto
  return (cast (Self.Potenza as float) /
          Self.NumCilindri )
```

```
create instance method MaggiorPotUnitaria
                        (ParteCfr TipoPartiAuto)
  returns boolean
  for TipoPartiAuto
  return
   ((Self.PotUnitaria > ParteCfr.PotUnitaria)
    or((Self.PotUnitaria = ParteCfr.PotUnitaria)
        and (Self.Cilindrata > ParteCfr.Cilindrata)))
```

Qui sopra vediamo la definizione dei metodi PotUnitaria e MaggiorPotUnitaria, la cui segnatura è stata specificata nell'ambito della dichiarazione di TipoPartiAuto. Il metodo PotUnitaria restituisce la potenza erogata da ciascun cilindro; MaggiorPotUnitaria confronta invece l'istanza di TipoPartiAuto su cui viene invocato il metodo con un'altra istanza dello stesso tipo che viene passata come parametro. Le due implementazioni sono autoesplicative, in quanto si limitano a semplici computazioni e confronti sui valori degli attributi dei due parametri, raggiunti con una notazione a punto (*dot notation*) per estrarre l'attributo di un oggetto referenziato da una variabile. È da notare come in MaggiorPotUnitaria venga utilizzato il metodo PotUnitaria; si osserva tra l'altro che la sintassi di SQL-3 non impone di usare una coppia di parentesi dopo il nome del metodo, come viene invece fatto in molti sistemi a oggetti per distinguere i metodi dagli attributi statici.

13.2.2 Linguaggio di interrogazione di SQL-3

SQL-3 è pienamente compatibile "all'indietro" con SQL-2, ovvero ogni comando corretto per la sintassi SQL-2 continua a essere un comando valido in SQL-3. Pertanto, in SQL-3 è possibile definire query relazionali "standard" sulle tabelle "standard". Per esempio:

```
select Nome
from Persona
where CodFisc = 'TRE SFN 56D23 S541S'
```

La navigazione lungo i riferimenti fra tipi richiede in SQL-3 l'operatore di dereferenziamento (*dereferencing*); tale operatore consente di accedere da un oggetto sorgente

x a un attributo A di un oggetto y referenziato in x, nel modo seguente: `x -> A`. Il seguente esempio mostra l'uso dell'operatore di dereferenziamento per accedere al valore dell'attributo Nome dell'oggetto della tabella INDUSTRIALE "puntato" dall'oggetto della tabella COSTRUTTORE che è "puntato" a sua volta dall'automobile che soddisfa il predicato `Targa = 'DB123MS'`.

```
select Produttore -> Presidente -> Nome
from Automobile
where Targa = 'DB123MS'
```

In SQL-3 gli attributi di tipo OID possono essere esplicitamente utilizzati nelle interrogazioni, e in particolare possono essere confrontati tramite l'operatore di uguaglianza con i riferimenti a tuple dello stesso tipo. L'ultima interrogazione può essere perciò espressa anche come segue:

```
select Industriale.Nome
from Automobile, Costruttore, Industriale
where Automobile.Targa = 'DB123MS'
  and Automobile.Produttore = Costruttore.CostrId
  and Costruttore.Presidente = Industriale.CodFisc
```

L'interrogazione costruisce un join fra le tabelle AUTOMOBILE, COSTRUTTORE e INDUSTRIALE, in cui l'attributo Produttore di AUTOMOBILE viene confrontato con l'identificatore di COSTRUTTORE e l'attributo Presidente di COSTRUTTORE viene confrontato con l'identificatore di INDUSTRIALE.

13.3 Estensioni tecnologiche per le basi di dati a oggetti

Le basi di dati a oggetti utilizzano la tecnologia dei sistemi di gestione di basi di dati, descritta nei capitoli precedenti, ma introducono alcune estensioni tecnologiche abbastanza significative.

13.3.1 Rappresentazione dei dati e degli identificatori

Un primo problema che si incontra nella gestione degli oggetti è posto dalla rappresentazione di oggetti complessi in memoria di massa, cioè nei cosiddetti *object server*; per memorizzare una gerarchia di classi tramite file di memoria di massa esistono due approcci estremi.

- L'approccio *orizzontale* consiste nel memorizzare ogni oggetto in modo contiguo; in particolare, tutte le istanze di una stessa classe vengono memorizzate all'interno dello stesso file; si avrebbe pertanto, relativamente all'esempio descritto in Figura 13.2, un file ciascuno per le classi AUTOMOBILE, AUTOSPORTIVA, AUTOSTORICA e AUTOSPORTIVASTORICA. Con questo approccio, l'accesso a un singolo oggetto risulta particolarmente efficiente, mentre risulta gravoso l'accesso

in base a specifiche proprietà degli oggetti (per esempio, l'accesso a una generica automobile in base all'attributo Colore richiede l'accesso a quattro file).

- L'approccio *verticale* consiste nel memorizzare contiguamente le medesime proprietà, spezzando un oggetto nelle sue componenti. Questo approccio è anche detto "normalizzato" in quanto potrebbe essere direttamente applicato per il modello relazionale. Abbiamo anche in questo caso un file ciascuno per le quattro classi citate, però per esempio il file *Automobile* contiene l'informazione relativa al Modello di tutti gli oggetti che sono istanza o membri della classe AUTOMOBILE; gli oggetti vengono ricostruiti a partire da riferimenti fra i loro componenti, e l'accesso a tutti gli attributi di un oggetto della classe AUTOSPORTIVASTORICA richiede l'accesso a quattro file.

Queste soluzioni ripercorrono le varie alternative di traduzioni dal modello Entità-Relazione al modello relazionale che caratterizza il progetto logico, visto nel Capitolo 8. La soluzione orizzontale è più coerente con il paradigma a oggetti in quanto gestisce ciascun oggetto in modo unitario, però pone problemi quando un oggetto può essere istanza di più classi. In generale, qualunque soluzione intermedia è accettabile e può essere suggerita dalle esigenze di una particolare applicazione.

La soluzione verticale si adatta anche alla gestione della complessità strutturale (dovuta all'uso dei costruttori di tipo); in questo caso però i riferimenti tra sotto-oggetti devono essere creati all'atto di partizionare un oggetto in parti e non possono utilizzare gli identificatori assegnati agli oggetti. Inoltre, contatori e identificatori opportunamente generati dal sistema sono necessari per gestire insiemi, liste e multi-insiemi.

Negli OODBMS specializzati nella gestione di documenti e dati multimediali, questi vengono tipicamente rappresentati come oggetti binari (i cosiddetti *binary long objects*, o *blob*) e memorizzati su file specifici (uno per oggetto binario).

Un problema caratteristico degli OODBMS è la rappresentazione degli OID, per i quali sono date due soluzioni:

- utilizzare un *indirizzo fisico*, cioè che include l'allocazione fisica (blocco) dell'oggetto in memoria di massa. L'ovvio vantaggio è la rapidità di accesso, lo svantaggio è la criticità della riallocazione di un oggetto, che può essere gestita tramite un indirizzamento indiretto, lasciando cioè nella pagina fisica in cui l'oggetto è inizialmente creato un "puntatore" alla sua posizione corrente;
- utilizzare un *surrogato*, cioè un valore che venga associato univocamente a un oggetto tramite un algoritmo (per esempio, in un sistema distribuito, tramite un contatore di oggetti per ogni nodo del sistema); in genere si predispone poi un indice oppure un meccanismo di hashing che consenta di passare dal surrogato all'indirizzo fisico di ciascun oggetto, in modo da garantire il loro reperimento efficiente. È possibile, anche se meno frequente, utilizzare surrogati tipizzati associati alle varie classi; è necessario in tal caso disporre di distinti contatori per ogni classe.

Gli OID possono essere di dimensioni elevate: in molte implementazioni di sistemi distribuiti raggiungono le dimensioni di 64 byte.

13.3.2 Indici complessi

Nei sistemi a oggetti è importante consentire accessi efficienti lungo i cammini (*path expression*), in modo da consentire l'esecuzione efficiente di query e programmi; per questo scopo vengono sviluppati negli OODBMS gli *indici complessi*.

Supponiamo che, nella base di dati a oggetti rappresentata in Figura 13.2, la nuova classe CITTÀ sia raggiungibile dalla classe STABILIMENTO. Consideriamo un cammino complesso che parta da un oggetto x appartenente alla classe AUTOMOBILE:

```
x.Produttore.Stabilimenti.Città.Nome
```

Per eseguire una query o un programma che utilizza questo cammino, occorre passare dalle automobili alle città a esse collegate, le quali avranno sull'attributo Nome valori di tipo stringa, quali per esempio "Milano" o "Torino". In questo modo si ritrovano gli oggetti di AUTOMOBILE che soddisfano il predicato:

```
X.Produttore.Stabilimenti.Città.Nome = "Torino"
```

Gli indici complessi che possono essere definiti in un OODBMS sono di tipo assai diverso.

- Un'organizzazione *multi-indice* garantisce la presenza di un indice per ciascuna proprietà utilizzata lungo il cammino. Quindi, procedendo a ritroso:
  - un indice dalle costanti di tipo stringa che contengono i nomi di città agli oggetti CITTÀ;
  - un indice dagli oggetti CITTÀ agli oggetti STABILIMENTO;
  - un indice dagli oggetti STABILIMENTO agli oggetti COSTRUTTORE;
  - un indice dagli oggetti COSTRUTTORE agli oggetti AUTOMOBILE.

  Usando i tre indici in un modo che ricorda il metodo *nested-loop* per l'esecuzione dei join (Paragrafo 11.6.2), si ritrovano tutti gli oggetti della classe COSTRUTTORE che soddisfano la condizione.
- Un *indice nidificato* consente il passaggio diretto dai valori che compaiono al termine del cammino alla classe che è all'origine del cammino. Dunque, dalla costante "Torino" si ricavano immediatamente tutti gli oggetti della classe COSTRUTTORE che soddisfano la condizione.
- Infine, un *indice cammino* consente il passaggio dai valori che compaiono al termine del cammino agli oggetti di tutte le classi che compaiono lungo il cammino.

Gli indici così introdotti vengono realizzati con alberi B+ di struttura particolare. Occorre tener conto nella loro gestione delle creazioni e cancellazioni di oggetti intermedi, che possono modificare in modo non banale la struttura degli indici; tecniche abbastanza sofisticate gestiscono al meglio queste operazioni.

13.3.3 Architettura client-server

Anche negli OODBMS gli oggetti vengono memorizzati in server dedicati alla gestione dei dati; poiché le applicazioni delle basi di dati a oggetti sono più recenti,

l'architettura client-server è ancora più diffusa che non in un contesto relazionale. Cambia però il paradigma di interazione con la base di dati, che nella maggioranza dei casi avviene tramite programmi applicativi scritti in un linguaggio tradizionale. Perciò negli OODBMS la programmazione imperativa si sostituisce a un linguaggio associativo di interrogazione, quale è SQL, che nel Paragrafo 16.1 è indicato come uno dei principali motivi del successo delle architetture client-server nel contesto dei sistemi relazionali.

Questo cambio di paradigma porta a una nuova suddivisione di compiti nella architettura client-server, che sta emergendo nei sistemi a oggetti, in cui il sistema client *importa* interi oggetti; in questo modo, i programmi applicativi possono venire eseguiti direttamente nei buffer dei sistemi client.

Si pensi per esempio a un'applicazione di tipo ingegneristico, in cui l'utente progetta un componente meccanico oppure un chip: una volta che l'intero oggetto è caricato sulla sua workstation, l'utente non interagisce più con l'object server per tutta la durata di una sessione di progettazione. Se viceversa l'oggetto restasse nel buffer del server durante la sessione di progettazione, si avrebbero pesanti interazioni tra client e server, con sovraccarico della rete locale.

Per facilitare l'import/export di oggetti, molti sistemi utilizzano in memoria centrale la stessa rappresentazione degli oggetti che esiste in memoria di massa, e quindi trasferiscono dai server ai client intere pagine di memoria. Una tipica ottimizzazione che avviene all'atto del caricamento degli oggetti dalla memoria di massa ai buffer di memoria centrale è la conversione degli OID; questa operazione prende il nome di *pointer swizzling*, ed è giustificata dalla maggiore efficienza e compattezza degli indirizzi di memoria centrale rispetto a quelli di memoria di massa.

Alcuni sistemi mirano esclusivamente a ottimizzare la velocità di accesso, e in tal caso il puntatore viene *riscritto* esattamente al di sopra dell'OID, lasciando la struttura della pagina inalterata. Per ottimizzare il processo di conversione, il passaggio da OID a puntatore può essere posposto al momento in cui un'applicazione utilizza realmente il puntatore (in quanto, in base al principio di località dei dati, ci sono buone probabilità che il riferimento verrà usato anche successivamente). All'atto di ricaricare gli oggetti in memoria di massa è necessario operare la conversione inversa; apposite strutture dati mantengono la corrispondenza fra puntatori in memoria centrale e OID.

13.3.4 Transazioni

Sul fronte transazionale, gli OODBMS offrono, accanto alle classiche proprietà acide delle transazioni, anche meccanismi meno restrittivi, che si basano sull'ipotesi di una possibile cooperazione fra gli utenti degli OODB. Vediamo brevemente i requisiti di alcuni di questi meccanismi.

- Nel controllo di concorrenza, vengono tipicamente definite delle operazioni di *check-out* degli oggetti che consentono di caricare interi oggetti da un server OODBMS alla workstation su cui opera un utente, per manipolarli durante lunghe sessioni di lavoro, consentendo però ad altri utenti di accedere ancora all'oggetto presente sul server. Ovviamente, l'operazione duale di *check-in*, in cui l'oggetto viene riportato sul server, richiede la cooperazione con gli altri progettisti.

- Un altro modo di garantire processi concorrenti cooperativi è basato sull'uso di *versioni*, cioè di varie copie dello stesso oggetto a istanti progressivi. In tal caso, è anche possibile definire relazioni fra oggetti che rappresentino l'allineamento fra differenti versioni. Questo problema è particolarmente critico nel contesto degli ambienti CASE (*Computer-Aided Software Engineering*), in cui si vuole consentire e documentare l'evoluzione dei programmi e garantire che varie versioni allineate costituiscano un'unica unità logica (per esempio, rispetto al processo di compilazione e link).
- In alcune applicazioni possono essere previste *transazioni di lunga durata* (*long transaction*): un esempio è la transazione che descrive il processo di check-out e check-in di un oggetto, che può essere trasferito su una workstation di progettazione per intere giornate.
- Infine, le transazioni possono essere composte da *transazioni complesse* (*nested transaction*); questo approccio, presente anche nei sistemi relazionali, si verifica per esempio quando un processo client deve realizzare un compito unitario colloquiando con vari server distinti che non consentono l'uso di transazioni globali; in tal caso, l'atomicità globale viene ottenuta tramite un coordinamento fra le transazioni acide di basso livello. L'esempio più tipico è l'organizzazione di un viaggio, che richiede di accedere a un sistema di prenotazioni alberghiere, una compagnia di auto-noleggi e ai sistemi di prenotazione di varie compagnie aeree. Ciascuna operazione di inserzione o cancellazione di una prenotazione su di uno dei DBMS è gestita come una transazione acida. Quando però l'impossibilità di prenotare una delle risorse viene rilevata dopo averne prenotata un'altra, l'annullamento della prenotazione già effettuata avviene tramite una *transazione di compensazione*, attivata dal coordinatore della transazione complessa. L'atomicità globale viene garantita, in modo asintotico, dal gestore delle transazioni complesse.

13.4 Basi di dati multimediali

Gli ultimi anni hanno visto crescere l'esigenza di gestire, accanto ai dati alfanumerici, altri dati che rappresentano documenti, immagini, video, audio, che indichiamo genericamente come "dati multimediali"; con il termine "base di dati multimediale" intendiamo quindi un sistema dotato della capacità di memorizzare, interrogare e presentare dati multimediali. È ragionevole discutere le problematiche delle basi di dati multimediali in questo capitolo in quanto è possibile vedere i dati multimediali come particolari tipi di dati, per i quali sono necessarie astrazioni specifiche che ne consentano una gestione efficace. Tuttavia, la capacità di gestire dati multimediali può essere offerta anche dalle basi di dati relazionali.

13.4.1 Tipi di dati multimediali

Iniziamo ad analizzare le caratteristiche dei principali dati multimediali: immagini, audio, video, documenti e annotazioni.

Immagini L'esigenza di associare a un oggetto, spesso descritto tramite attributi alfanumerici, anche una sua immagine è sempre più diffusa. Basi di dati di immagini

sono utilizzate in campo clinico, ove per esempio associano a ciascun paziente le sue radiografie; le polizie di tutto il mondo si scambiano schede segnaletiche dei ricercati, in cui compaiono le immagini del loro volto; le agenzie immobiliari illustrano le case in vendita tramite immagini fotografiche; la descrizione delle località turistiche avviene soprattutto attraverso immagini. La principale difficoltà nella gestione delle immagini è il numero elevato di bit necessari per memorizzarle in formato binario; per ridurre tale numero si usano formati standard (tra cui: GIF, JPEG, TIFF, PNG), che ne consentono la rappresentazione in forma compressa.

Audio I dati audio possono contenere conversazioni, musiche e altri segnali sonori (per esempio, i suoni che vengono generati quando le applicazioni vogliono segnalare degli eventi). Un segnale audio viene rappresentato mediante una serie di valori numerici che rappresentano l'ampiezza del segnale sonoro in una finestra temporale di piccole dimensioni. Per esempio, nella modalità di campionamento usata nei CD audio, si ha un'osservazione ogni 20 microsecondi rappresentata con un valore a 16 bit. Applicazioni che non richiedono la massima fedeltà di riproduzione dei suoni possono usare una frequenza di campionamento più bassa e meno bit per la rappresentazione del valore. Nella maggior parte delle applicazioni si utilizzano delle tecniche di compressione per limitare la dimensione di questi dati, che possono altrimenti diventare molto voluminosi. Le tecniche di compressione possono essere senza perdita di informazioni (*lossless*), o possono comportare una leggera perdita di informazione (*lossy*), tipicamente con un impatto non percettibile da parte di chi ascolta nella qualità del segnale e un guadagno molto significativo in termini di riduzione della dimensione dei dati.

Video I video sono collezioni di immagini (o *frame*) mostrate l'una dopo l'altra da un dispositivo riproduttore. Un video può illustrare un evento storico, una lezione, gli animali in azione nel loro habitat naturale e così via. Se la memorizzazione di immagini pone problemi di spazio, questi sono assai più significativi nel caso di video; basti pensare che un video di 60 minuti può contenere più di 100.000 frame. Gli standard più diffusi per la gestione dei video (MPEG-1, MPEG-2, MPEG-4) consentono qualità diverse ottenute con diversi livelli di compressione; per esempio, MPEG-1 non ha sufficiente qualità per garantire la riproduzione televisiva, MPEG-2 consente tale riproduzione aumentando la qualità del video (ma richiede anche un maggior numero di bit) e MPEG-4, lo standard oggi maggiormente usato, migliora ulteriormente la qualità consentendo anche la riproduzione televisiva ad alta definizione.

Documenti I documenti sono composti da testi e immagini, presentati tramite un preciso formato. Per esempio, la pagina di un giornale, oppure la home page degli autori di questo libro, oppure una lettera d'affari scritta su carta intestata e firmata dall'autore sono esempi di documenti. Le cosiddette "biblioteche digitali" si ripromettono di memorizzare e rendere disponibili via Internet milioni di libri e altri documenti. Per la costruzione di documenti si possono usare linguaggi di markup, quali HTML o XML (si veda il Capitolo 14); questi linguaggi sono in grado di includere in uno spazio bidimensionale immagini e testi, composti secondo il formato prescelto dall'autore del documento (per esempio, per quanto riguarda i testi, con la scelta dei caratteri, l'uso di grassetto e sottolineature, allineamenti e indentazioni ecc.).

Annotazioni Infine, le annotazioni sono testi liberi, che vengono aggiunti ad altri dati multimediali (immagini, documenti, suoni). Le annotazioni consentono di corredare un generico dato multimediale con indicazioni specifiche (per esempio, il proprio gradimento, o l'esigenza di collegare un documento a un altro). Dato che ciascuna annotazione ha carattere personale, nella gestione delle annotazioni assume un ruolo fondamentale l'utente, che può per esempio decidere di arricchire un documento con molte annotazioni personali, stabilendo poi che alcune di esse siano condivisibili – cioè accedibili opzionalmente da altri utenti – e altre siano invece private.

13.4.2 Interrogazioni su dati multimediali

Mentre la codifica dei dati multimediali è un problema generale della multimedialità, l'abilità di interrogare grandi moli di dati multimediali per estrarre dati di interesse è un problema più specifico delle basi di dati multimediali. In questo paragrafo, ci soffermiamo su alcuni classici esempi di interrogazioni che si applicano ai dati multimediali, occupandoci più della loro formulazione che non della descrizione delle tecniche per calcolarne il risultato.

Per esempio, un'interrogazione a un archivio di immagini può richiedere l'estrazione di immagini con certi tratti particolari: tutte le radiografie polmonari in cui compaiono segni di broncopolmonite, oppure l'individuo più simile all'identikit trasmesso dalla polizia e relativo a un rapinatore di banche, o ancora tutti i ritratti rinascimentali di madonne con bambino conservati al museo del Louvre. Se già la ricerca di un'immagine in base alle sue caratteristiche appare difficile, ancor più difficile appare il compito di estrarre segnali audio oppure video in base a pattern particolari (tutte le musiche di Wagner in cui compare un particolare tema, oppure le sequenze video in cui compaiono goal della nazionale di calcio brasiliana), in quanto in tal caso il pattern deve essere riconosciuto operando su molteplici frame o finestre temporali. L'interrogazione dovrà di norma sfruttare inizialmente informazioni di tipo strutturale, associate ai dati multimediali, per estrarre un sotto-insieme di essi; dopodiché sarà possibile valutare, per ciascun dato multimediale estratto, la "somiglianza" rispetto alla richiesta formulata.

In genere, si potrà selezionare un dato multimediale solo in termini probabilistici, estraendo cioè quei dati che soddisfano la richiesta con una probabilità data. Sarà quindi possibile presentare il risultato di un'interrogazione partendo dai dati che hanno maggior probabilità di soddisfare la richiesta, e arrestandosi al di sotto di una certa soglia di probabilità. Tecniche adattative consentono agli utenti di guidare la ricerca, associando ai dati estratti un gradimento che permette di indirizzare la ricerca in modo che evolva in base all'esito dell'interazione con l'utente.

13.4.3 Ricerca di documenti

Il caso di gran lunga più ricorrente di interrogazione su dati multimediali è l'estrazione di documenti che contengono particolari informazioni testuali. Si possono in tal caso utilizzare tecniche ormai collaudate e di grande successo, che fanno riferimento alla disciplina del *ritrovamento delle informazioni* (*Information Retrieval*) su testi; per rendersi conto dell'efficacia di tali metodi, basti pensare alla qualità dei cosiddetti "motori di ricerca", che consentono di estrarre i siti web di interesse a partire da poche parole chiave.

Queste tecniche si basano innanzitutto sulla capacità di estrarre da un testo informazioni utili per poter decidere se esso sia rilevante nell'ambito di una query. Tali informazioni si riducono a una rappresentazione ottimizzata delle principali parole chiave presenti nel testo, con associata un'indicazione relativa alla loro frequenza di occorrenza. Per costruire queste informazioni relativamente a un generico testo è necessario operare come segue:

- escludere dal testo parole irrilevanti (articoli, congiunzioni, preposizioni ecc.), che compaiono con elevata frequenza ma non sono significative;
- ridurre parole simili a un'unica parola chiave. Per esempio, le parole "abita", "abitazione", "abitato", "abitante" sono tutte legate a un unico concetto di "abitare" e quindi sostituibili con quest'ultima parola;
- associare a ogni parola chiave così definita la propria frequenza, definita come il numero di occorrenze della parola rispetto al numero totale di parole presenti nel testo.

A questo punto, la ricerca di un testo che soddisfi l'interrogazione di un utente, espressa tramite espressioni booleane di parole chiave, si riduce a determinare quel testo in cui le parole chiave proposte dall'utente, in una combinazione compatibile con quella richiesta dall'utente, hanno frequenza più elevata. Per meglio definire l'efficacia della ricerca, si definiscono due misure: la *precisione* (*precision*) e il *richiamo* (*recall*). Supponiamo che per ogni documento sia noto, in modo deterministico, se esso è rilevante, cioè se fa parte o meno del risultato dell'interrogazione.

- La precisione indica la percentuale dei documenti rilevanti all'interno dei documenti estratti.
- Il richiamo indica invece la percentuale dei documenti rilevanti estratti rispetto al totale dei documenti rilevanti nella base di dati.

Un buon algoritmo di ricerca deve cercare di tener conto di entrambi questi fattori: deve offrire una buona precisione, in modo da presentare documenti che sono in larga misura rilevanti, ma anche un buon richiamo, in modo da ridurre il rischio di omettere dalla risposta i documenti più rilevanti per l'utente.

Varie tecniche sono disponibili per organizzare l'informazione che descrive un testo, comprimendola e strutturandola in modo tale da rendere più efficace la ricerca. In particolare, è possibile rappresentare più documenti tramite *matrici*, in cui righe e colonne di una matrice rappresentano rispettivamente parole chiave e identificatori dei documenti, e ciascun elemento della matrice rappresenta la frequenza della parola chiave all'interno del documento. In tal caso, l'estrazione dei documenti e la compressione dei dati a essi relativi sono affidate a tecniche di calcolo matriciale.

In alternativa è possibile rappresentare la corrispondenza fra parole chiave e documenti che le contengono tramite *indici invertiti*; gli indici sono organizzati sulle parole chiave, come illustrato nel Capitolo 11, e al loro interno, come foglie, vi sono gli identificatori dei documenti che contengono tali parole chiave. Tramite queste strutture dati è facile estrarre documenti che soddisfano condizioni booleane sulle parole chiave: per esempio l'intersezione di due parole chiave si ottiene semplicemente

facendo l'intersezione degli insiemi di identificatori ottenuti percorrendo l'indice due volte, con le due parole scelte dall'utente.

Altre strutture dati mettono in evidenza per prime le parole chiave a elevata frequenza; per esempio, la *segnatura* di un documento è una rappresentazione compatta delle n parole chiave che compaiono in un documento con frequenza più elevata; tecniche sofisticate consentono di estrarre documenti oppure di decidere se due documenti sono simili in base alla loro segnatura.

13.4.4 Rappresentazione dei dati spaziali

I dati spaziali descrivono l'informazione presente in uno spazio a n dimensioni, per esempio una mappa geografica (bi-dimensionale) o il progetto di un edificio in costruzione (tri-dimensionale). La gestione dei dati spaziali è un ambito applicativo abbastanza specifico che ha però acquisito grande importanza; per questo motivo, essa viene spesso affidata a sistemi dedicati, tra cui presentano la relazione più stretta con il mondo delle basi di dati i cosiddetti *sistemi informativi geografici* (*Geographic Information Systems*, GIS).

Il problema principale della gestione dei dati spaziali è la scelta di una struttura dati che consenta di rispondere a interrogazioni relative alla disposizione dei dati nello spazio. Per esempio: estrarre tutti i punti di una mappa che distano meno di una determinata distanza da un punto particolare; oppure determinare tutte le regioni che sono adiacenti a una regione data; o ancora, determinare tutti i punti di una mappa che si trovano lungo una linea, e potrebbero rappresentare le città collocate lungo il corso di un fiume. Ovviamente, i GIS sono in grado di descrivere non solo la disposizione dei dati nello spazio, ma anche le caratteristiche di ciascun punto, linea, o regione dello spazio; per esempio, i punti che descrivono le *città* saranno dotati, oltre alle proprie coordinate geografiche, anche di altre informazioni, quali per esempio il numero di abitanti e l'altezza sul livello del mare; le regioni saranno invece caratterizzate, per esempio, dal tipo di cultura agricola prevalente, oppure dalla quantità media mensile di pioggia per unità di superficie. Le linee potranno rappresentare fiumi, oppure strade o ferrovie. Tramite i GIS sarà allora possibile esprimere interrogazioni in cui si mescolano aspetti spaziali e non; per esempio, si potranno estrarre tutte le regioni italiane in cui si produce il vino Chianti, oppure le città lombarde con meno di centomila abitanti in cui si trova un castello medioevale.

La gestione efficiente dei dati spaziali richiede di organizzare l'informazione tramite strutture dati speciali, in modo che la contiguità geografica dei dati sia rappresentata nella struttura dati, e che ciò consenta un'esecuzione efficiente delle interrogazioni che sfruttano tale contiguità. In particolare, vari tipi di strutture ad albero consentono di gestire collezioni di punti (per esempio, le città in una mappa geografica) e di rispondere efficientemente a interrogazioni del tipo: estrarre tutte le città che distano meno di una certa distanza da un determinato punto. Ciascun punto viene rappresentato da un nodo dell'albero; ogni nodo comprende le sue coordinate X e Y, un'informazione specifica del nodo e i puntatori ai nodi successori.

Nell'organizzazione ad *albero bidimensionale* (*2-d tree*) ogni nodo ha due successori, rispettivamente destro e sinistro; in questa organizzazione, il nodo radice rappresenta un'intera zona geografica, e ogni nodo suddivide la zona geografica da es-

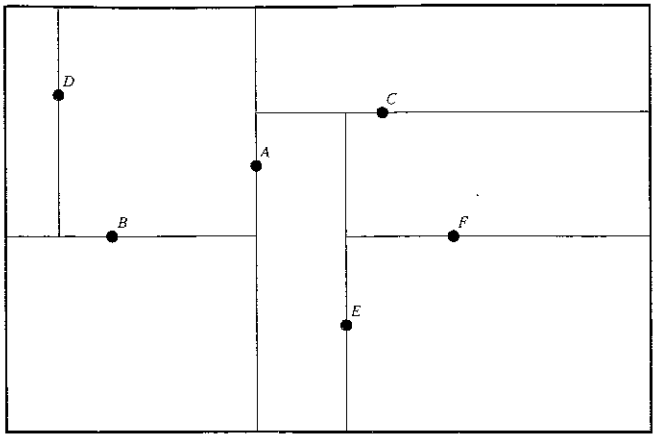

Figura 13.4 Un *2–d tree*.

so rappresentata in due zone tramite una linea, che è orizzontale oppure verticale a seconda che il punto sia a una distanza pari o dispari rispetto alla radice. Nella Figura 13.4, *A* è il nodo radice, *B* e *C* i suoi discendenti, *E* il discendente di *C*, *F* il discendente di *E* e *D* il discendente di *B*. Analizzando la figura si vede che *A* suddivide la figura verticalmente, *B* orizzontalmente, *C* orizzontalmente, *D* verticalmente, *E* verticalmente e *F* orizzontalmente.

Nell'organizzazione a *quad-tree* ogni nodo suddivide la zona geografica da esso rappresentata in quattro zone tramite due linee – orizzontale e verticale – che passano attraverso il punto stesso; quindi, ogni nodo ha quattro successori, che rappresentano i quattro quadranti. Nella Figura 13.5, *A* è il nodo radice, *B*, *C*, *D* e *E* i suoi discendenti, *F* il discendente di *E*. Ogni punto suddivide la regione in cui è collocato in quattro parti.

Vari sistemi commerciali sono dedicati specificamente alla gestione di dati spaziali; tra essi, ArcGIS, dotato di un sottosistema per la gestione di dati spaziali che si integra con i principali DBMS relazionali. L'Open Geospatial Consortium (precedentemente chiamato Open GIS Consortium) ha sviluppato una serie di standard relativi ai formati per lo scambio di dati spaziali, in modo da consentire interoperabilità fra i vari prodotti GIS.

13.5 NoSQL

Concludiamo questo capitolo con la breve presentazione di una tematica che, pur non rientrando propriamente nell'ambito delle basi di dati a oggetti, ne condivide in buona misura le motivazioni. D'altra parte, questa tematica si è sviluppata in modo che non

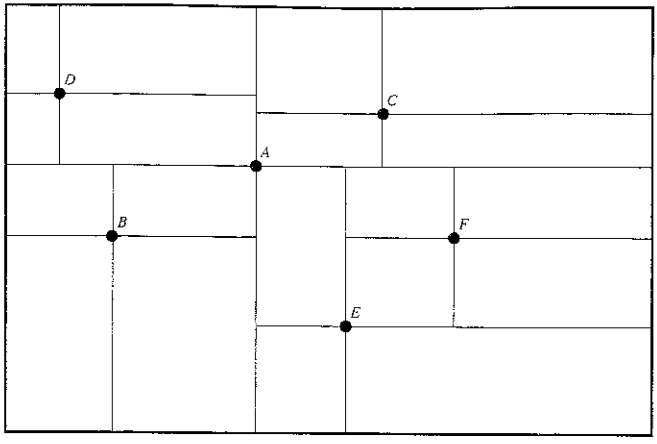

Figura 13.5 Un *quad-tree*.

è esagerato definire tumultuoso, in poco più di cinque anni, nel contesto delle nuove applicazioni nel mondo Web e quindi merita certamente una grande attenzione.

Negli ultimi anni, per esempio nel contesto dei social network o anche in altre applicazioni in ambiente cloud, sono emerse esigenze caratterizzate da due requisiti generali: (I) la presenza di grandi volumi di dati, non ben quantificabili, ma sempre crescenti, e (II) la struttura non regolare dei dati da gestire. Al tempo stesso, l'esigenza è quasi sempre relativa a una gestione relativamente semplice, da due punti di vista: da una parte i dati sono costituiti da collezioni che, pur non avendo una struttura rigida, contengono elementi (che corrispondono di solito a oggetti di qualche applicazione) relativamente omogenei fra loro, almeno da un punto di vista concettuale; dall'altra, i tipi di operazioni, tanto di aggiornamento quanto di interrogazione, sono abbastanza semplici, sulle singole collezioni (in particolare senza correlazioni fra collezioni diverse). Un esempio molto semplice di dati di interesse in questa categoria potrebbe essere quello di un'applicazione del cosiddetto mondo "Web 2.0", con un insieme di utenti che possono pubblicare interventi (come in Twitter) e seguire ciascuno i contributi di altri. Le collezioni di interesse sono relativamente poche (nel semplice esempio, due, gli utenti e gli interventi), ma estremamente voluminose. Inoltre, ciascun aggiornamento dei dati è costituito dall'inserimento di un nuovo utente, dalla modifica di un qualche dato personale di un utente, oppure dalla pubblicazione di un intervento. Tutti i dati di interesse sono identificati da una chiave primaria e correlati attraverso le chiavi stesse. In generale, i dati possono essere solo parzialmente strutturati, in quanto l'utente potrebbe avere un insieme di proprietà non definito a priori e con valori potenzialmente molteplici (per esempio, più indirizzi di posta elettronica, ognuno con un'etichetta specifica).

L'insieme delle caratteristiche sopra sintetizzate, associato al fatto che i DBMS esistenti, tanto relazionali quanto a oggetti, sono estremamente complessi e offrono i loro servizi in modo integrato e difficilmente scorporabile, ha portato allo sviluppo di nuovi sistemi, che concentrano i loro obiettivi sulla scalabilità e sulla gestione di oggetti non rigidamente strutturati, offrendo solo alcune delle funzionalità dei sistemi tradizionali. In effetti, nel giro di pochi anni sono stati sviluppati e utilizzati, in contesti diversi, numerosi sistemi (più di cinquanta), con caratteristiche diverse, ma accomunati dagli obiettivi appena visti. Essi vengono di solito indicati con il termine generico "NoSQL", anche se forse sarebbe più corretto utilizzare il termine "non relazionali" o "non tradizionali."

In questo testo, sia per ragioni di spazio sia per la rapida evoluzione della tecnologia, peraltro molto variegata, non è possibile una presentazione dettagliata dei vari sistemi della famiglia NoSQL. Ci limitiamo quindi ad alcuni spunti di riflessione, che approfondiscono gli aspetti salienti.

13.5.1 Diversificazione dei sistemi

I sistemi di basi di dati, come abbiamo visto nei vari capitoli di questo libro, sono fortemente fondati sul modello relazionale e, soprattutto negli anni Ottanta e Novanta del secolo scorso, si diffuse una certa tendenza a pensare che essi, costituendo una tecnologia solida, potessero essere standardizzati sia dal punto di vista logico (con l'adozione del modello relazionale) sia da quello realizzativo (con le tecniche di gestione delle strutture fisiche, delle transazioni e della distribuzione). In effetti, ci si è presto resi conto del fatto che questo non è facilmente fattibile, né da un punto di vista né dall'altro. A livello logico, per quanto il modello relazionale sia ampiamente diffuso e utilizzato, esso non esaurisce certo il supporto alle esigenze delle applicazioni, come abbiamo visto in questo capitolo relativamente al paradigma a oggetti e come vedremo nei Capitoli 14, con riferimento alla gestione di dati XML, e 19, relativamente alle basi di dati per il supporto alle decisioni. A livello implementativo, come vedremo sempre nel Capitolo 19, i tipi di applicazioni transazionali (OLTP) e analitiche (OLAP), anche quando si basano sul modello relazionale, hanno requisiti talmente contrastanti da rendere necessario ricorrere a soluzioni radicalmente diverse e a sistemi separati. La situazione può essere sintetizzata con lo slogan "one size does not fit all," utilizzata come titolo per un noto articolo [146], che vuole affermare che non esiste, e probabilmente non può esistere, una soluzione universale per i problemi di gestione dei dati. In tale articolo, veniva osservato come vari scenari applicativi, diversi da quello transazionale, potessero giustificare implementazioni differenti dal DBMS tradizionale: oltre al già citato contesto di supporto all'analisi, venivano citati la gestione di "stream" di dati, le reti di sensori, la ricerca su documenti o nel Web e più in generale la gestione di testi e i dati scientifici.

Le più recenti applicazioni Web degli ultimi anni hanno portato ulteriore supporto a queste riflessioni, tanto che sono stati sviluppati numerosi sistemi, spesso con riferimento a specifiche esigenze, leggermente diverse le une dalle altre, ma con in comune la necessità di supportare grande scalabilità per dati relativamente semplici (anche se semistrutturati) e con operazioni pure molto semplici. La scalabilità, come ormai frequente in molte moderne applicazioni, viene gestita attraverso l'utilizzo di

molti processori a basso costo, con reti facilmente riconfigurabili in caso di guasti e partizionamento "orizzontale" dei dati (cioè, come vedremo nel Capitolo 16, allocando i vari oggetti di una collezione su più nodi). Una caratteristica comune, dal punto di vista architetturale, è l'utilizzo di un approccio *shared nothing*, cioè, come vedremo nel Paragrafo 17.1, senza condivisione di memoria di alcun tipo (né disco né RAM). Questo tipo di soluzione è senz'altro quella più facilmente scalabile, se i dati sono partizionabili e distribuibili automaticamente, in modo da bilanciare gli accessi, e le operazioni sono ciascuna su dati che si trovano su un solo nodo (ipotesi di solito soddisfatta, vista la prevalente semplicità delle operazioni). Le differenze fra i vari sistemi e con i sistemi tradizionali fanno riferimento a due aspetti fondamentali, che illustriamo nei due paragrafi seguenti.

13.5.2 Modelli dei dati nei sistemi NoSQL

I vari sistemi NoSQL sviluppati sinora presentano ciascuno una specifica organizzazione dei dati, anche se ci sono alcune similitudini, che permettono di classificarli con riferimento ad alcune categorie. La classificazione più nota al riguardo include tre famiglie, oltre a una con caratteristiche abbastanza diverse (i sistemi di basi di dati a grafo, *graph database systems*), che tralasciamo.

La prima categoria è quella dei sistemi *key-value*, che rappresentano tutti i dati per mezzo di coppie chiave-valore, in cui la chiave (per esempio un dato di tipo stringa), definita dal programma che inserisce i dati, identifica univocamente i singoli elementi, che possono poi avere una struttura diversa l'uno dall'altro e quindi si afferma solitamente che questi sistemi gestiscono basi di dati "senza schema." Ci sono in effetti molti sistemi in questa famiglia, che differiscono da vari punti di vista, quali i costrutti utilizzabili per i singoli elementi, le tecniche di gestione delle transazioni (di cui parleremo brevemente nel prossimo paragrafo). Questi sistemi, tra cui Oracle NoSQL, offrono API per le semplici operazioni di creazione, lettura, aggiornamento ed eliminazione di un oggetto e iteratori. La struttura degli oggetti è trasparente per il sistema ed è lasciata all'applicazione che accede a esso.

I sistemi della seconda famiglia sono chiamati *document store*, in quanto possono memorizzare oggetti con struttura complessa, appunto come i documenti, anche se essi non debbono necessariamente essere "documenti" nel senso stretto del termine. Questi sistemi prevedono la possibilità di organizzare gli oggetti in *collezioni* (così come in una base di dati relazionale abbiamo molte relazioni, ognuna delle quali è una collezione di ennuple) e di definire una struttura complessa per ciascun oggetto (peraltro senza una definizione esplicita di schema). Inoltre, la maggior parte dei sistemi di questa famiglia permettono la definizione di indici secondari sulla base di "attributi", che, a differenza dei loro omonimi relazionali, non sono predefiniti e non hanno necessariamente un tipo. Fra i sistemi più noti di questa famiglia, si possono citare MongoDB e CouchDB.

La terza famiglia è quella dei cosiddetti *extensible record store*, in cui la base di dati è composta di collezioni, che sono chiamate tabelle perché hanno alcune similitudini con quelle del modello relazionale, insieme però ad alcune differenze. Queste tabelle, infatti, possono essere nidificate e non hanno in generale una struttura predefinita, salvo una prima struttura di "famiglie", o gruppi di colonne, che hanno la

funzione di indicare una preferenza per la memorizzazione (le colonne di uno stesso gruppo sono memorizzate insieme). Il primo sistema di questa famiglia fu BigTable, sviluppato da Google per le proprie attività e poi imitato da progetti open source, come HBase e HyperTable.

13.5.3 Gestione delle transazioni

Come abbiamo detto all'inizio della nostra discussione dei sistemi NoSQL, una caratteristica abbastanza diffusa è quella di rilassare i tradizionali requisiti transazionali, costituiti dalle proprietà acide che abbiamo discusso nel Capitolo 12. È stato al riguardo suggerito, in contrasto ad ACID, l'acronimo BASE, per *Basically Available, Soft state, Eventually consistent*. L'idea è che gli aggiornamenti sono eseguiti "prima o poi", ma ci possono essere letture inconsistenti con gli aggiornamenti più recenti. Ciò è dovuto al fatto che, per favorire le prestazioni, si adottano soluzioni che prevedono copie multiple dei dati, che potrebbero non essere aggiornate al momento della lettura. Va detto, peraltro, che le tecniche utilizzate dai vari sistemi sono diverse (dal locking, al controllo multiversione, fino alla rinuncia alle proprietà acide) e che il comportamento è di solito ragionevolmente soddisfacente perché sono rare (se non inesistenti) le transazioni che agiscono su più oggetti. Un argomento che spesso viene citato riguardo alla necessità di rinunciare alla transazionalità è il cosiddetto *CAP theorem*, secondo il quale, in un sistema distribuito, non è possibile soddisfare contemporaneamente le tre proprietà di consistenza (tutti i nodi in un certo istante vedono gli stessi dati), disponibilità (in inglese "availability," ogni nodo risponde alle richieste) e tolleranza alle partizioni (il sistema continua a operare anche in caso di fallimento dei nodi o di impossibilità di comunicazione). Parleremo dei sistemi distribuiti nel Capitolo 16, comunque possiamo anticipare che i sistemi distribuiti tradizionali supportano la consistenza e la disponibilità, ma sostanzialmente non la tolleranza alle partizioni, attraverso il protocollo del commit a due fasi (2PC). I sistemi NoSQL solitamente non implementano tale protocollo perché rallenterebbe troppo le operazioni.

Note bibliografiche

Vari libri di testo sono dedicati specificamente alle basi di dati a oggetti, tra cui il libro di Bertino e Martino *Sistemi di basi di dati orientati agli oggetti: concetti e architetture*, di Addison-Wesley Masson, 1992 [96]. Altri libri su questo argomento sono: *Object Data Management - Object-Oriented and Extended Relational Database Systems* di Cattel [102] e *Object Databases: The Essentials* di Loomis [131]. Il libro *The Story of O2*, curato da Bancilhon *et al.* [93], contiene vari articoli sul sistema O2; il primo articolo riporta il cosiddetto "Manifesto delle basi di dati a oggetti". Una panoramica sulle basi di dati relazionali a oggetti è offerta da [144], mentre il "Manifesto sulle basi di dati della terza generazione" è pubblicato in [147]. La parte a oggetti dello standard SQL-3 è descritta nel testo di Melton [135]. Le basi di dati multimediali sono descritte nel libro di V.S. Subrahmanian *Principles of Multimedia Database Systems* [148], mentre un testo classico sulle strutture dati spaziali è il testo di Samet [140]. Discussioni e confronti sui sistemi SQL sono disponibili su molti siti Web e in alcuni articoli, fra cui quelli di Cattell [103], Cattell e Stonebraker [145], e Atzeni *et al.* [92].

Esercizi

Soluzioni sul sito http://www.pearsonline.it/atzeni

13.1 Definire una classe FIGURAGEOMETRICA e tre sotto-classi che ereditano da essa QUADRATO, CERCHIO e RETTANGOLO. Definire un metodo Area la cui implementazione in FIGURAGEOMETRICA ritorna il valore zero, mentre nelle sotto-classi viene valutata in funzione delle proprietà delle sotto-classi medesime. Mostrare l'invocazione del metodo da parte di un programma che scandisce una lista di figure geometriche di tipo arbitrario.

13.2 Definire il dizionario dati di una base di dati a oggetti. Si suggerisce di introdurre classi e gerarchie di classi relative ai vari concetti di uno schema a oggetti: le classi, i tipi atomici, i tipi strutturati tramite i vari costruttori, le gerarchie di generalizzazione, i metodi con i loro parametri di ingresso e di uscita. Popolare il dizionario dati con dati che descrivono parte dello schema relativo alla gestione delle automobili, descritto in Figura 13.2. Pensare poi a qualche interrogazione che consenta di estrarre, per esempio, l'elenco delle classi e dei metodi con ridefinizione co-variante del tipo di uscita.

13.3 Si consideri il seguente schema di una base di dati a oggetti:

```
add class Città
  type tuple(Nome: string,
             Nazione: string,
             Monumenti: set(Monumenti),
             Hotels: list(Hotel));

add class Hotel
  type tuple(Nome: string,
             Indirizzo: tuple(Via: string,
                              Città: Città,
                              Numero: integer,
                              CAP: string);
             Stellette: integer,
             Attrattive: list(string));

add class Luogo
  type tuple(Nome: string,
             Fotografia: Bitmap,
             Indirizzo: tuple(Via: string,
                              Città: Città,
                              Numero: integer,
                              CAP: string);
             Attrazioni:
                  set(ServiziTuristici));

add class Monumento inherits Luogo
  type tuple(DataCostruzione: date,
             GiorniChiusura: list(string),
             PrezzoAmmissione: integer,
             Architetto: Person);
```

```
add class ServiziTuristici
  type tuple(Nome: string,
             Luoghi: set(Luogo),
             Costo: integer);

add class Teatro inherits Monumento
  type tuple(SerateSpettacoli: list(date))

add class SpettacoloTeatrale
  type tuple(Titolo: string,
             Luogo: Teatro,
             Protagonista: Persona,
             Repliche: set(date));

add class Concerto inherits SpettacoloTeatrale
  type tuple(Protagonista: Direttore,
             Orchestra:set(Musicisti));

add class Persona
  type tuple(Nome: string,
             CodiceFiscale: string,
             Nazione: string);

add class Direttore inherits Persona
  type tuple(AffiliazioneStabile: Teatro);

add class Musicista inherits Persona
  type tuple(Strumenti: set(string));
```

1. Definire i metodi di inizializzazione delle tre classi LUOGO, MONUMENTO e TEATRO, riutilizzando i metodi nello scendere lungo la gerarchia.
2. Quale proprietà dello schema è ridefinita in modo co-variante?
3. Definire la segnatura del metodo di inizializzazione di uno spettacolo e poi raffinare la segnatura in modo co-variante nei parametri di ingresso qualora lo spettacolo sia un concerto.
4. Indicare un esempio di invocazione del metodo definito in precedenza in cui non sia possibile verificarne la correttezza a tempo di compilazione.
5. Descrivere graficamente lo schema precedente nel modo illustrato in Figura 13.2.

13.4 Descrivere lo schema di base di dati a oggetti dell'Esercizio 13.3 usando la sintassi di SQL-3.

13.5 Considerando lo schema SQL-3 introdotto nell'Esercizio precedente, esprimere le seguenti interrogazioni in SQL-3.

1. Estrarre i nomi delle città la cui nazione è il "Liechtenstein".
2. Estrarre il nome dei musicisti che suonano in concerti diretti da Karajan.
3. Estrarre il nome dei monumenti di Londra costruiti nel diciassettesimo secolo e chiusi al lunedì.
4. Estrarre i nomi dei direttori che hanno diretto concerti in teatri diversi da quelli da cui dipendono.

13.6 Facendo riferimento allo schema di base di dati a oggetti dell'Esercizio 13.3, indicare una scelta di indici complessi per gestire in modo efficiente i cammini più usati dalle interrogazioni dell'Esercizio 13.5.

13.7 Costruire una rappresentazione a *2d-tree* e a *quad-tree* della sequenza di punti bidimensionali: $A(5,4)$, $B(3,3)$, $C(6,2)$, $D(2,2)$, $E(4,6)$, $F(1,1)$, $G(7,5)$. Quanti nodi intermedi vi sono, nelle due rappresentazioni, tra A e F oppure tra A e G?

14 Basi di dati per XML

Nel Capitolo 13 abbiamo discusso delle basi di dati a oggetti, una tecnologia per la memorizzazione persistente e per la manipolazione di dati caratterizzati da una struttura più complessa di quella offerta dal modello relazionale.

Con l'avvento del Web, l'evoluzione dei modelli di dati ha fatto un ulteriore passo avanti, che ha portato alla definizione dello standard XML (*eXtensible Markup Language*). La caratteristica particolare dei dati rappresentabili con XML è il fatto di essere *semistrutturati*. Con tale termine si intende un tipo di dato che può rispettare in modo parziale il proprio schema, talora addirittura senza disporre di uno schema definito.

Nel concreto, XML si propone come un mezzo per la definizione di insiemi di tag personalizzati da usarsi per la rappresentazione di dati semistrutturati, specialmente di tipo documentale. Lo standard è definito dal consorzio W3C, l'organismo internazionale preposto alla regolamentazione delle tecnologie Web. La prima versione di XML (1.0) è diventata uno standard W3C nel 1998; la versione attuale dello standard è XML 1.1.

L'idea principale di XML è quella di consentire la definizione di dati con un grado variabile di struttura: un'istanza di documento XML può non avere del tutto una specifica di struttura, ma semplicemente rispettare alcune regole sintattiche per cui risulta *ben formata*; oppure può fare riferimento a una specifica strutturale, che ne descrive i vincoli di composizione. In entrambi i casi, però, il dato è considerato corretto, anche se il tipo di elaborazione che è possibile svolgere su dati privi di schema è inferiore a quello che si può effettuare su documenti XML dotati di struttura. Di norma, un documento XML privo di schema può essere solo formattato graficamente e quindi mostrato a un utente umano; un documento XML dotato di schema può invece essere elaborato facilmente in modo automatico da un programma, per esempio al fine di venire convertito in informazione strutturata e inserito all'interno di un sistema informativo.

Il secondo aspetto fondamentale di XML è la scelta di rappresentare le istanze come documenti contenenti dati e markup, secondo un approccio simile a quello adottato per il linguaggio HTML. Questa scelta dà a XML una natura ibrida, a cavallo tra il modello di rappresentazione dei dati e lo standard per la definizione di documenti; grazie a questa ambivalenza, XML ha rapidamente assunto il ruolo di anello di congiunzione tra il Web (formato da documenti non strutturati) e le basi di dati (contenenti informazioni strutturate). Questa doppia interpretazione si evidenzia anche se si considerano le principali applicazioni di XML.

- Come HTML, XML è utilizzato per la rappresentazione di documenti, sia per il Web sia tradizionali, e per la costruzione di motori di ricerca. Per esempio, alcuni word processor usano XML come formato interno di rappresentazione dei documenti.
- Come gli altri modelli dei dati, XML è usato per costruire basi di dati persistenti, possiede linguaggi di definizione dello schema e di interrogazione e manipolazione dei dati.
- XML è infine utilizzato come formato di interscambio di informazioni tra sistemi in rete, grazie alla sua capacità di rappresentare le informazioni pertinenti a un certo dominio applicativo in modo completamente indipendente dalla piattaforma hardware/software.

Nei prossimi paragrafi illustreremo dapprima HTML, poi presenteremo le caratteristiche di XML e dei linguaggi per la definizione di schemi e interrogazione; mostreremo infine alcune delle applicazioni più rilevanti di questa importante tecnologia.

14.1 HTML

Presentiamo ora le caratteristiche del linguaggio che viene considerato il predecessore di XML. Il linguaggio HTML (abbreviazione di *HyperText Markup Language*) è utilizzato per descrivere i documenti che costituiscono il Web ed è quindi molto diffuso; il Capitolo 18 descrive in modo specifico gli aspetti relativi all'integrazione tra Web e basi di dati.

HTML si basa su una tecnica di rappresentazione dei documenti nota come *markup*. Tale tecnica si contrappone a quella utilizzata da strumenti quali Microsoft Word, in cui l'utente dispone di un'interfaccia grafica che mostra a video il documento nello stesso modo in cui questo verrà prodotto a stampa.

Nelle tecniche di markup, invece, l'utente annota il contenuto testuale del documento con opportuni comandi (detti *tag* o *elementi*), che assegnano proprietà grafiche o strutturali al testo. Il documento marcato deve essere elaborato da un processore al fine di ottenerne la forma visualizzabile o stampabile.

In HTML, i tag sono simboli delimitati da parentesi angolari, quali `<FORM>`, `<TABLE>` e così via. Normalmente, i tag sono utilizzati a coppie, allo scopo di delimitare una porzione di testo; in questo caso, il tag di chiusura si ottiene dal tag di apertura aggiungendo il simbolo `/` al nome del tag, come in `<TABLE>...</TABLE>`.

L'intero documento HTML viene strutturato utilizzando opportuni tag, quali l'elemento `<HTML>` che circonda l'intero documento, l'elemento `<HEAD>` che circoscrive la parte di intestazione non visualizzata, l'elemento `<BODY>` che delimita la parte di contenuto vero e proprio del documento.

Il seguente codice HTML descrive un documento elementare.

```
<HTML>
  <HEAD>
    <TITLE>Un semplice documento</TITLE>
      <META name="keywords" content="HTML, esempio">
  </HEAD>
  <BODY>
  <P align="center">
  Un documento HTML elementare.</P>
  </BODY>
</HTML>
```

Tutto ciò che appare all'interno del tag `HEAD` non viene visualizzato, in quanto parte dell'intestazione del documentò; in particolare, il contenuto del tag `<TITLE>` fornisce il titolo della finestra del browser e il contenuto del tag `<META>` favorisce l'indicizzazione del documento da parte dei motori di ricerca. Invece, il tag `<BODY>` delimita l'effettivo contenuto del documento; nell'esempio, un paragrafo centrato con il testo "Un documento HTML elementare".

L'esempio mostra l'uso degli attributi HTML, quali `name="keywords"` e `align="center"`, che consentono di associare ai tag proprietà che ne precisano il significato.

Le funzioni più importanti di HTML sono svolte dai tag <A> e <OBJECT>, che introducono rispettivamente gli aspetti di ipertestualità e di multimedialità.

Il tag <A> (dove A sta per *àncora*, cioè origine di un collegamento ipertestuale) consente di delimitare una porzione di contenuto e dichiarare che tale porzione funge da rimando verso un altro documento. Il documento referenziato viene specificato precisandone l'URL come valore dell'attributo `href`.

Il tag <OBJECT> permette di inserire oggetti di vario tipo in un documento HTML, quali immagini, musica, video, script. Il tag <OBJECT> permette di specificare il tipo di contenuto e di fornire indicazioni rispetto al modo in cui il contenuto deve essere visualizzato. Per includere immagini viene anche offerto un tag dedicato <IMG>, che può essere considerato una specializzazione di <OBJECT>. Il tag viene inserito singolarmente nel documento come una sorta di segnaposto e gli attributi `data`, `height`, e `width` specificano l'URL dell'oggetto incluso, ed eventualmente la sua altezza e larghezza. La pagina HTML che segue, illustrata anche nella Figura 14.1, dimostra l'uso sia di <A> sia di <OBJECT>.

```
<HTML>
  <HEAD>
    <TITLE>Un documento con immagine e ancora</TITLE>
  </HEAD>
   <BODY>
    <P align="center"><OBJECT data="logo_ateneo.gif"
       type="image/gif"></OBJECT></P>
   <P align="center">Cliccate <A href=
      "http://www.ateneonline.it">QUI</A>
         per accedere alla home page di AteneOnLine.</P>
   </BODY>
</HTML>
```

L'uso del tag <A> trasforma la parola "QUI" in un'àncora ipertestuale che punta alla home page del sito AteneOnLine (`http://www.ateneonline.it`).

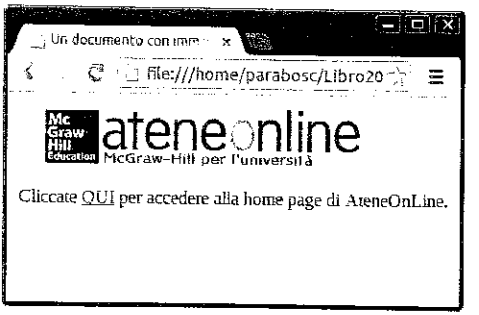

Figura 14.1 Una pagina HTML con immagine e àncora.

Come ultima osservazione su HTML, è utile menzionare un'estensione, aggiunta a partire dalla versione 4.0, per trattare le proprietà stilistiche del documento mediante un vero e proprio linguaggio di presentazione che sostituisce l'uso dei tag di formattazione presenti nelle versioni precedenti di HTML.

Il linguaggio di presentazione *Cascading Style Sheet* (CSS) separa il contenuto del documento dalla modalità di formattazione del testo. Quest'ultima è espressa mediante semplici regole che si applicano a tutti i tag del documento che soddisfano un criterio di selezione; a tali tag le regole CSS applicano attributi grafici che sono interpretati dal browser per la produzione dell'aspetto finale del testo.

Per esempio, la regola CSS presentata di seguito, posta all'interno di un documento HTML, stabilisce le dimensioni e i colori del font per la resa dei tag HTML `<H1>` e `<H2>`, usati per delimitare titoli di primo e secondo livello:

```
<STYLE type="text/css">
    H1 { font-size: 20pt; color: red }
    H2 { font-size: 18pt; color: blue }
</STYLE>
```

L'aspetto più utile di CSS è la possibilità di centralizzare le regole di presentazione in un singolo file di stile esterno alle pagine, referenziato da tutte le pagine di un sito; in tal modo, le regole di formattazione sono scritte una volta sola e possono essere modificate agendo sul file CSS, con la garanzia che le modifiche fatte vengano automaticamente applicate a tutte le pagine che si riferiscono al file di stile.

14.2 Definizione di dati semistrutturati in XML

Lo standard XML detta le regole sintattiche per la costruzione di documenti *ben formati* (*well-formed*), realizzati mescolando tag e contenuto informativo. Un documento XML ben formato deve soddisfare i requisiti seguenti.

- Il documento deve cominciare con una direttiva standard che specifica la versione di XML usata, per esempio: `<?xml version="1.0"?>`.
- Il documento può contenere *elementi*. Ogni elemento può contenere sia testo sia altri elementi. Un elemento dotato di contenuto è delimitato da due tag: uno di apertura e uno di chiusura. Il tag di chiusura deve avere lo stesso nome del tag di apertura, con in più il prefisso "/". Se un tag è privo di contenuto, ha un solo tag, il cui nome deve finire con il simbolo "/".
- Il documento deve avere un elemento radice che racchiude l'intero contenuto. Se un elemento contiene sotto-elementi, il tag di chiusura di ciascun sotto-elemento deve precedere il tag di chiusura del sopra-elemento.
- Ogni elemento può avere uno o più attributi; gli attributi si specificano nel tag di apertura dell'elemento e i valori a essi associati sono racchiusi tra virgolette.

A titolo di esempio, riportiamo di seguito un semplice documento XML ben formato:

```
<?xml version="1.0"?>
<radice>
```

```
  <elemento attributo="valore">
    <sottoelemento>
      ..testo...
    </sottoelemento>
  </elemento>
  <elementovuoto/>
</radice>
```

I documenti XML, oltre che sintatticamente ben formati, possono essere accompagnati da una specifica della loro struttura, analoga a uno schema del documento. Se il documento rispetta la specifica di struttura, si dice *valido* rispetto a tale specifica.

Storicamente, sono state proposte due diverse modalità per la specifica della struttura di documenti XML, una più semplice ma meno espressiva, chiamata *Document Type Definition* (DTD), e una più complessa ma anche più potente, chiamata *XML Schema*.

A scopo di esempio, si consideri il documento XML seguente, che rappresenta una descrizione della struttura dell'edizione 2013 del libro, che copre solo le prime tre parti del volume corrente:

```
<?xml version="1.0"?>
<libro editore="McGraw-Hill" pubblicazione="2013"
        formato="paperback">
 <titolo>Basi di dati: Modelli e
        linguaggi di interrogazione</titolo>
 <autore> Paolo Atzeni</autore>
 <autore> Stefano Ceri </autore>
 <autore> Piero Fraternali </autore>
 <autore> Stefano Paraboschi</autore>
 <autore> Riccardo Torlone </autore>
 <parte numero="1">
  <capitolo> Il modello relazionale </capitolo>
  <capitolo> Algebra e calcolo relazionale </capitolo>
  <capitolo> SQL: Concetti base </capitolo>
  <capitolo> SQL: Caratteristiche evolute </capitolo>
 </parte>
 <parte numero="2">
  <capitolo> Metodologie e modelli per il progetto </capitolo>
  <capitolo> La progettazione concettuale </capitolo>
  <capitolo> La progettazione logica </capitolo>
  <capitolo> La normalizzazione </capitolo>
  <capitolo> Sviluppo di applicazioni per basi di dati </capitolo>
 </parte>
 <parte numero="3">
  <capitolo> Organizzazione fisica e gestione delle interrogazioni
  </capitolo>
  <capitolo> Gestione delle transazioni </capitolo>
 </parte>
 <parte numero="4">
  <appendice> Microsoft Access </appendice>
  <appendice> DBMS open source: Postgres </appendice>
 </parte>
</libro>
```

La struttura del documento XML sopra esemplificato può essere espressa dal seguente DTD, che descrive un insieme di elementi XML utilizzabili per descrivere indici di libri:

```
<!ELEMENT libro (titolo, autore+, parte+)>
<!ELEMENT titolo (#PCDATA)>
<!ELEMENT autore (#PCDATA)>
<!ELEMENT parte (capitolo+|appendice+)>
<!ELEMENT capitolo (#PCDATA)>
<!ELEMENT appendice (#PCDATA)>

<!ATTLIST libro
  editore          CDATA                  #REQUIRED
  pubblicazione    CDATA                  #IMPLIED
  formato          (paperback|hardback)   'paperback'
>
<!ATTLIST parte
  numero           CDATA                  #REQUIRED
>
```

Il DTD riportato come esempio è costituito da un elenco di regole, di due tipi: regole che definiscono gli elementi XML ammissibili e il loro contenuto (introdotte dalla clausola `!ELEMENT`) e regole che specificano gli attributi degli elementi XML, il loro tipo e i vincoli sul loro valore (introdotte dalla clausola `!ATTLIST`).

La prima clausola `!ELEMENT` del DTD riportato definisce l'elemento di nome `libro`, precisandone il modello di contenuto: un elemento `libro` deve contenere un elemento `titolo`, seguito da uno o più elementi `autore` (come richiesto dal simbolo + posto nel modello di contenuto dopo il nome degli elementi), seguiti da uno o più elementi `parte`. Nel modello di contenuto, oltre all'indicatore +, possono apparire il simbolo ?, per indicare zero o un'occorrenza di un sotto-elemento, e il simbolo *, per indicare zero o più occorrenze di un sotto-elemento.

Il modello di contenuto dell'elemento `autore` contiene la stringa `#PCDATA`. Ciò rappresenta l'abbreviazione di *Parsed Character Data* ed esprime il fatto che l'elemento ha contenuto di tipo testuale che può essere analizzato dal parser XML per individuare al suo interno altri elementi ed entità; in alternativa, il tipo `CDATA` rappresenta una componente testuale che non deve essere analizzata dal parser XML e quindi deve essere gestita come una semplice stringa di caratteri. Infine, l'elemento `parte` contiene uno o più capitoli *oppure* una o più appendici, come si evince dall'uso del simbolo |, che rappresenta l'alternativa tra diversi contenuti. Si noti che l'uso dell'alternativa nella definizione del modello dei contenuti di un elemento rende lo schema flessibile: sono validi rispetto al DTD documenti in cui l'elemento `parte` contiene sotto-elementi `capitolo` oppure `appendice` (ma non entrambi).

L'esempio di DTD contiene anche regole per la specifica degli attributi, in particolare degli elementi `libro` e `parte`. L'elemento `libro` ammette tre attributi: l'attributo `editore` è di tipo testo "piatto" (`CDATA`) obbligatorio (`#REQUIRED`); l'attributo `pubblicazione` è dichiarato opzionale (`#IMPLIED`); infine, l'attributo `formato` ammette due valori (`paperback` e `hardback`), con il valore `paperback` assunto come default.

Le regole del DTD di un documento XML possono essere definite all'interno del documento stesso, oppure possono essere contenute in un documento separato, cui il documento XML fa riferimento con una particolare direttiva. L'utilizzo del concetto di DTD per la specifica della struttura dei documenti XML ha il pregio della semplicità, ma impone anche alcune limitazioni:

- i tipi di dati ammessi sono solo testuali. Non è possibile dichiarare tipi interi, reali, date, booleani, o tipi record;
- i vincoli esprimibili mediante il modello di contenuto degli elementi e applicabili al valore degli attributi sono limitati.

Per ovviare a queste limitazioni, XML prevede la possibilità di esprimere la struttura di una classe di documenti mediante una *definizione di schema XML* (*XML Schema Definition*, XSD). Uno schema XML è un particolare documento XML, contenente elementi standard, che specifica la struttura di un insieme di documenti XML. Un XSD consente di definire elementi e attributi, di dichiarare il tipo di ciascun elemento e attributo sulla base di un insieme ampio di tipi predefiniti, di costruire nuovi tipi con l'aiuto di costruttori di tipi complessi e di esprimere vincoli sulla struttura degli elementi e degli attributi più sofisticati di quelli offerti dai DTD.

Come esempio, si riporta di seguito uno schema XML per la rappresentazione di indici di libri, equivalente all'esempio di DTD illustrato in precedenza.

```
<?xml version="1.1"?>
<xs:schema
    xmlns:xs="http://www.w3.org/2001/XMLSchema">

 <xs:element name="libro">
  <xs:complexType>
   <xs:sequence>
    <xs:element name="titolo" type="xs:string"/>
    <xs:element name="autore" type="xs:string"
                maxOccurs="unbounded"/>
    <xs:element ref="parte"  maxOccurs="unbounded"/>
   </xs:sequence>
   <xs:attribute name="editore" type="xs:string"
                 use="required"/>
   <xs:attribute name="pubblicazione" type="xs:date"/>
   <xs:attribute name="formato" default="paperback"/>
    <xs:simpleType>
     <xs:restriction base="xs:string">
      <xs:enumeration value="paperback"/>
      <xs:enumeration value="hardback"/>
     </xs:restriction>
    </xs:simpleType>
   <xs:attribute/>
  </xs:complexType>
```

```
 </xs:element>

 <xs:element name="parte">
  <xs:choice>
   <xs:element name="capitolo" type="xs:string"
               maxOccurs="unbounded"/>
   <xs:element name="appendice" type="xs:string"
               maxOccurs="unbounded"/>
  </xs:choice>
  <xs:attribute name="numero" type="xs:integer"
                use="required"/>
 </xs:element>
</xs:schema>
```

La direttiva posta all'inizio dell'esempio specifica che lo schema XSD è un documento XML valido rispetto al DTD standard che può essere recuperato all'URL `http://www.w3.org/2001/XMLSchema`.

La struttura degli elementi è descritta mediante l'uso del tag `xs:element`, al cui interno sono specificati il modello di contenuto e i vincoli. Per esempio, l'elemento `libro` è dichiarato come un tipo complesso (utilizzando il tag `xs:complexType`) costituito dalla sequenza (tag `xs:sequence`) di tre sotto-elementi e con tre attributi. Il tipo degli elementi e degli attributi non è limitato al solo tipo stringa, ma può estendersi al caso di tipi numerici, date, e così via, come esemplificato dall'attributo `pubblicazione` che è di tipo data (`xs:date`). Tipi enumerativi sono esprimibili a partire da un tipo base, restringendo i valori ammessi mediante l'uso del tag `xs:restriction`. La cardinalità dei sotto-elementi è specificata mediante gli attributi `maxOccurs`, per esprimere il massimo numero di occorrenze di un sotto-elemento, e `minOccurs`, per vincolare il numero minimo di occorrenze. Infine, il costruttore `xs:choice` è utilizzabile per definire elementi il cui modello di contenuto ammette sotto-elementi in alternativa, quali per esempio l'elemento `parte`.

14.3 Interrogazione di dati XML

I dati XML possono essere considerati una fonte di informazione interrogabile dall'utente o da programmi applicativi, nello stesso modo con cui si interrogano i dati contenuti nelle basi di dati relazionali e a oggetti. Per consentire l'interrogazione di dati XML, il consorzio W3C ha avviato nel 1998 un'attività volta alla definizione di un linguaggio di interrogazione per XML. Il gruppo di lavoro incaricato di tale attività ha formulato la proposta di un linguaggio di interrogazione chiamato *XQuery*, che è diventato standard nella versione 1.0 all'inizio del 2007; è attualmente in corso il lavoro per il consolidamento della versione successiva, che verrà chiamata 3.0 e di cui parleremo più avanti. La specifica è disponibile all'URL `http://www.w3c.org/TR/xquery/`.

Le interrogazioni XQuery presuppongono l'uso di un ulteriore linguaggio, denominato *XPath*, che consente di specificare porzioni di un documento XML. XPath

è uno standard antecedente a XQuery, utilizzato anche in un linguaggio di ristrutturazione di documenti XML chiamato XSL, che descriveremo nel Paragrafo 14.3.2. Nel prossimo paragrafo descriviamo gli elementi fondamentali di XPath, mentre nei due successivi illustriamo i linguaggi XSL e XQuery.

14.3.1 XPath

XPath consente la specifica di cammini (path) all'interno di un documento XML, allo scopo di selezionare una porzione del documento. La sintassi di XPath si ispira alla nozione di path già largamente utilizzata nei sistemi operativi per indicare la posizione di una risorsa nel file system. Un'interrogazione XPath ha la forma di una *path expression* e la sua valutazione restituisce l'insieme degli elementi del documento raggiungibili seguendo il percorso specificato dalla path expression. Per esempio, l'espressione `/libro/autore` seleziona tutti gli elementi di tipo `autore` contenuti all'interno dell'elemento `libro`.

Nella scrittura delle espressioni valgono le seguenti convenzioni.

- Se il cammino comincia con il simbolo `/`, rappresenta un cammino *assoluto*, cioè presenta la sequenza di elementi dalla radice del documento fino all'elemento selezionato. Per esempio, l'espressione `/libro/parte/capitolo` seleziona gli elementi che si trovano nel documento seguendo una sequenza che parte dall'elemento `libro`, procede ai sotto-elementi `parte` e termina con i sotto-sotto-elementi `capitolo`.
- Se il cammino comincia con il simbolo `//`, tutti gli elementi che soddisfano il criterio di selezione vengono restituiti, indipendentemente dal cammino usato per raggiungerli. Per esempio, l'espressione `//capitolo` trova tutti gli elementi di tipo `capitolo` all'interno del documento, a qualunque livello essi si trovino.
- Il simbolo `*` denota un elemento qualsiasi. Per esempio, la valutazione dell'espressione `/libro/*` seleziona tutti gli elementi contenuti all'interno dell'elemento `libro`, mentre l'espressione `/*/*/*` seleziona tutti gli elementi al terzo livello di annidamento (capitoli e appendici, nell'esempio del Paragrafo 14.2).

XPath consente anche di esprimere condizioni logiche al fine di selezionare i cammini usati per identificare gli elementi del risultato. Sintatticamente, le condizioni sono poste tra parentesi quadre, inserite all'interno del cammino. Per esempio, l'espressione `/libro/parte[1]` seleziona il primo elemento `parte` nell'elemento `libro`; l'espressione `/libro/autore[text()='Stefano Ceri']` seleziona invece l'elemento autore il cui contenuto è "Stefano Ceri". Nella scrittura delle condizioni è possibile fare uso degli operatori aritmetici e di comparazione e dei connettivi logici `and` e `or`, nonché di un certo numero di funzioni predefinite, quali funzioni numeriche, booleane, di manipolazione di stringhe e di conteggio. Per esempio `/libro/parte[count(capitolo)>2]` seleziona tutte le parti con più di due capitoli, mentre l'applicazione dell'espressione `/libro/parte[capitolo[contains(text(),'XML')]]` estrae le parti con un capitolo il cui nome contiene la stringa XML.

Le path expression possono anche fare riferimento agli attributi degli elementi, utilizzando il carattere `@` come prefisso del nome dell'attributo: l'espressione

`//@editore` seleziona tutti gli attributi di nome `editore`, mentre l'espressione `/libro/parte[@numero='2']/capitolo` seleziona i capitoli della parte che ha 2 come valore dell'attributo `numero`.

14.3.2 XSL

Il linguaggio XPath permette soltanto di estrarre porzioni di un documento. Per elaborazioni complesse sono necessari i linguaggi più potenti XSL (*eXtensible Stylesheet Language*)[1] e XQuery (presentato nel prossimo paragrafo), i quali permettono di ristrutturare contenuti XML in modo arbitrario.

XSL è un linguaggio a regole basato sul concetto di *pattern-matching*. Una regola XSL si applica a un documento XML in due passi: dapprima la regola seleziona una porzione di documento che obbedisce a una certa struttura (*pattern*) e successivamente trasforma la porzione di documento selezionata in un altro documento di output, dotato della struttura e dei contenuti desiderati. Diverse regole XSL possono essere raggruppate in un file, che prende il nome di *foglio di stile* (*stylesheet*) e costituisce un vero e proprio programma per la trasformazione e ristrutturazione di documenti.

XSL è l'unione di tre sotto-linguaggi:

- *XPath*: è il sotto-linguaggio che consente la specifica della porzione di documento su cui la regola XSL agisce;
- *XSL Transformations* (XSLT): è la parte che consente di scrivere le regole di trasformazione documentale;
- *XSL Formatting Objects* (XSL-FO): è un linguaggio per la specifica della resa dei documenti in modo astratto e indipendente dalla piattaforma di presentazione.

Uno degli usi più frequenti di XSL è la trasformazione di documenti XML in pagine HTML, allo scopo di consentirne la pubblicazione su Web. A titolo di esempio, mostriamo le regole XSL che traducono il documento di esempio riportato nel Paragrafo 14.2 nella pagina HTML mostrata nella Figura 14.2.

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL
                /Transform" version="1.0">
  <!-- Regola principale, si applica a elementi di tipo
       libro-->
  <xsl:template match="libro">
   <HTML>
    <HEAD>
      <TITLE><xsl:value-of select="titolo"/></TITLE>
    </HEAD>
    <BODY>
      <P><H1><xsl:value-of select="titolo"/></H1></P>
```

[1]XSL è uno standard antecedente a XQuery, definito dal consorzio W3C, come già XML e le diverse versioni di HTML. La prima raccomandazione relativa a XSL risale al 1999.

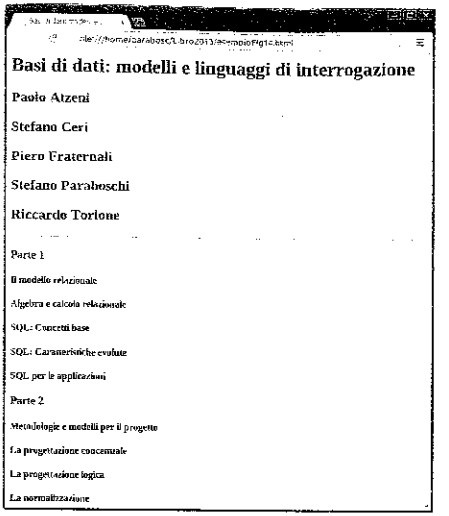

Figura 14.2 Una pagina HTML prodotta mediante una trasformazione XSL applicata al documento XML che descrive l'indice del libro.

```
      <xsl:apply-templates select="autore"/><HR/>
      <xsl:apply-templates select="parte"/>
    </BODY>
   </HTML>
  </xsl:template>

  <!-- Regola che si applica a elementi di tipo autore-->
  <xsl:template match="autore">
    <P><H2> <xsl:value-of select="text()"/></H2></P>
  </xsl:template>

  <!-- Regola che si applica a elementi di tipo parte-->
  <xsl:template match="parte">
    <P><H3>Parte <xsl:value-of select="@numero"/></H3></P>
    <xsl:apply-templates select="capitolo"/>
  </xsl:template>

  <!-- Regola che si applica a elementi di tipo capitolo
       o appendice-->
  <xsl:template match="capitolo|appendice">
    <P><H4><xsl:value-of select="text()"/></H4></P>
  </xsl:template>
</xsl:stylesheet>
```

Per meglio capire la trasformazione operata dalle regole XSL, riportiamo anche il codice HTML da esse prodotto:

```
<HTML>
 <HEAD>
  <TITLE>Basi di dati: modelli e linguaggi
         di interrogazione</TITLE>
 </HEAD>
 <BODY>
 <P><H1>Basi di dati: modelli e linguaggi
         di interrogazione</H1></P>
 <P><H2> Paolo Atzeni</H2></P>
 <P><H2> Stefano Ceri</H2></P>
 <P><H2> Piero Fraternali</H2></P>
 <P><H2> Stefano Paraboschi</H2></P>
 <P><H2> Riccardo Torlone</H2></P><HR>
 <P><H3>Parte 1</H3></P>
 <P><H4>Il modello relazionale </H4></P>
 <P><H4>Algebra e calcolo relazionale </H4></P>
 <P><H4>SQL: Concetti base </H4></P>
 <P><H4>SQL: Caratteristiche evolute </H4></P>
 <P><H4>SQL per le applicazioni </H4></P>
 <P><H3>Parte 2</H3></P>
 <P><H4>Metodologie e modelli per il progetto </H4></P>
 <P><H4>La progettazione concettuale </H4></P>
 <P><H4>La progettazione logica </H4></P>
 <P><H4>La normalizzazione </H4></P>
 <P><H4>Sviluppo di applicazioni per basi
        di dati </H4></P>
 <P><H3>Parte 3</H3></P>
 <P><H4>Organizzazione fisica e gestione
        delle interrogazioni </H4></P>
 <P><H4>Gestione delle transazioni </H4></P>
 <P><H3>Parte 4</H3></P>
 <P><H4>Microsoft Access </H4></P>
 <P><H4>DBMS open source: Postgres </H4></P>
 </BODY>
</HTML>
```

Il foglio di stile XSL contiene quattro regole, ciascuna delimitata dal tag `xsl:template`. Ogni regola include un attributo `match`, contenente la path expression XPath che determina la porzione di documento cui si applica la regola. La prima regola seleziona l'elemento `libro` e quindi l'intero contenuto del documento (`match=libro`); la seconda regola si applica alle istanze dell'elemento `autore`, la terza regola si applica alle istanze dell'elemento `parte` e la quarta regola alle istanze dell'elemento `capitolo`.

All'interno del tag `xsl:template`, ogni regola contiene il testo e il markup HTML da inserire nel documento di output, dentro il quale possono essere annidate chiamate ad altre regole che producono ulteriori porzioni di output. Tali chiamate sono rappresentate mediante l'uso di tag XSLT, caratterizzati dal prefisso `xsl:`.

La prima regola produce la parte esterna del codice HTML del documento restituito in output. All'interno del tag `TITLE`, la clausola `<xsl:value-of select=titolo/>` inserisce il contenuto estratto dall'elemento `titolo` annidato all'interno dell'elemento `libro`. In questo modo, il titolo del libro appare anche come intestazione della finestra del browser. La medesima clausola è ripetuta all'interno del tag `BODY` del documento HTML, in modo da stampare il titolo, formattato usando il tag di titolazione di primo livello (`H1`). Successivamente, la clausola `<xsl:apply-templates select=autore/>` produce l'invocazione ricorsiva delle eventuali regole che si applicano all'elemento `autore`. In particolare, il foglio di stile contiene una regola che si applica all'elemento autore e inserisce nell'output il contenuto testuale dell'elemento (mediante la clausola `<xsl:value-of select=text()/>`), posizionandolo all'interno di un paragrafo formattato con un titolo di secondo livello (`H2`).

Le rimanenti due regole producono la lista delle parti, ciascuna formattata mediante un titolo di terzo livello e contenente la lista dei suoi capitoli e appendici, formattati con un titolo di quarto livello.

14.3.3 XQuery

L'altro linguaggio proposto dal W3C per selezionare e trasformare documenti XML è XQuery. Rispetto a XSL, XQuery si distingue per il fatto che pone l'enfasi sulla specifica di interrogazioni, rispetto alla riorganizzazione di un documento ai fini della presentazione (l'obiettivo originale di XSL).

Un'interrogazione XQuery consta di cinque parti.

- La clausola `for`, per la dichiarazione di variabili che permettono di iterare sugli elementi di un documento.
- La clausola `let`, per la dichiarazione di nuove variabili, eventualmente correlate a quelle introdotte nella clausola `for`.
- La clausola `where`, per l'espressione di predicati che filtrano gli elementi che compongono il risultato dell'interrogazione.
- La clausola `order by`, per imporre un ordinamento nella sequenza che compone il risultato.
- La clausola `return`, per definire il contenuto e la struttura del risultato.

Per questa forma particolare, le interrogazioni XQuery sono anche dette espressioni FLWOR (*For-Let-Where-OrderBy-Return*). Tale acronimo è pronunciato come la parola inglese "flower".

Presentiamo le caratteristiche del linguaggio procedendo con una sequenza di esempi. L'interrogazione seguente esemplifica la sintassi della clausola `for`.

Interrogazione XQuery 1: estrarre l'elenco degli autori del libro descritto nel documento XML `libro.xml`.

```
for $author in doc("libro.xml")//autore
return $author
```

La clausola `for` valuta la path expression `doc(libro.xml)//autore`, che estrae in sequenza tutti gli elementi di tipo `autore` all'interno del documento `libro.xml`. La variabile `$author`, dichiarata nella clausola `for`, itera all'interno della sequenza di autori, assumendo a ogni iterazione il valore di un elemento diverso. La successiva clausola `return` specifica che il risultato dell'interrogazione coincide con il valore della variabile `$author`. Pertanto, se si valuta l'interrogazione sul documento di esempio descritto nel Paragrafo 14.2, si ottiene come risultato il frammento XML seguente, prodotto dall'iterazione di `$author` sugli autori:

```
<autore>Paolo Atzeni</autore>
<autore>Stefano Ceri</autore>
<autore>Piero Fraternali</autore>
<autore>Stefano Paraboschi</autore>
<autore>Riccardo Torlone</autore>
```

Le espressioni `for` possono essere annidate, come dimostra l'esempio seguente.

Interrogazione XQuery 2: estrarre i capitoli di tutte le parti del libro descritto nel documento XML `libro.xml`.

```
for $part in doc("libro.xml")//parte
    for $chapter in $part/capitolo
return $chapter
```

Si noti come la definizione della variable `$chapter` faccia uso del valore della variabile `$part`. L'interpretazione della doppia clausola `for` è che, per ogni valore di `$part`, si deve considerare ogni valore che viene assunto da `$chapter`, restituendo quindi la sequenza dei diversi valori di `$chapter`.

Per quanto riguarda la clausola `return`, segnaliamo che essa può generare singoli elementi, foreste ordinate di elementi, o semplici valori testuali. L'espressione che compare dopo la parola chiave `return` può contenere *costruttori*, valori costanti, riferimenti a variabili definite nelle clausole `for` e `let`, o ulteriori espressioni annidate. Per esempio, per racchiudere il contenuto di ogni elemento restituito dalla query precedente in una nuova coppia di tag `<CapitoloBD>...</CapitoloBD>`, è sufficiente strutturare la clausola `return` come segue.

Interrogazione XQuery 3: estrarre i capitoli del libro descritto nel documento XML `libro.xml`, racchiudendo ciascuno di essi all'interno di un elemento `<CapitoloBD>`.

```
for $part in doc("libro.xml")//parte
    for $chapter in $part/capitolo
return <CapitoloBD> {$chapter/text()} </CapitoloBD>
```

La funzione `text()` estrae il contenuto testuale di un elemento. Il risultato prodotto dalla query è il seguente:

```
<CapitoloBD>Il modello relazionale</CapitoloBD>
<CapitoloBD>Algebra e calcolo relazionale</CapitoloBD>
<CapitoloBD>SQL: Concetti base</CapitoloBD>
...
```

L'espressione `<CapitoloBD>...</CapitoloBD>` prende il nome di *costruttore di elementi*; il risultato viene specificato a partire da elementi XML fissi e dal valore di variabili o espressioni.

La presenza delle parentesi graffe è necessaria per individuare un contesto di valutazione dell'espressione. Se la clausola `return` nell'interrogazione precedente fosse stata scritta così:

```
return <CapitoloBD> $chapter/text() {$chapter/text()}
       </CapitoloBD>
```

il risultato prodotto dalla query sarebbe stato il seguente:

```
<CapitoloBD> $chapter/text() Il modello relazionale
</CapitoloBD>
<CapitoloBD> $chapter/text() Algebra e calcolo relazionale
</CapitoloBD>
<CapitoloBD> $chapter/text() SQL: Concetti base
</CapitoloBD>
...
```

L'esempio mostra che l'espressione all'esterno della parentesi viene interpretata come una stringa costante, mentre quella all'interno viene valutata come un'espressione del linguaggio e produce per ogni capitolo il contenuto testuale. L'uso delle parentesi graffe riattiva la valutazione XQuery, che si era sospesa con la costruzione esplicita di elementi XML.

La clausola `let` permette di introdurre nuove variabili, cui viene assegnato come valore l'intero risultato della valutazione dell'espressione di assegnamento. Di norma quindi la variabile dichiarata nella clausola `let` rappresenta un insieme di valori.

Interrogazione XQuery 4: estrarre, divisi per parte e racchiusi in un elemento `<CapitoloBD>`, i titoli dei capitoli del libro descritto nel documento XML `libro.xml`.

```
for $part in doc("libro.xml")//parte
let $chapters := $part/capitolo
return <CapitoloBD> {$chapters/text()} </CapitoloBD>
```

Il risultato prodotto dalla query è il seguente:

```
<CapitoloBD> Il modello relazionale
             Algebra e calcolo relazionale
             SQL: Concetti base
             SQL: Caratteristiche evolute
             SQL per le applicazioni
```

```
</CapitoloBD>
<CapitoloBD> Metodologie e modelli per il progetto
             La progettazione concettuale
             La progettazione logica
             La normalizzazione
             Sviluppo di applicazioni per basi di dati
</CapitoloBD>
<CapitoloBD> Organizzazione fisica e gestione
             delle interrogazioni
             Gestione delle transazioni
</CapitoloBD>
```

La differenza tra le interrogazioni 3 e 4 è evidente: nell'interrogazione 3, la variabile `$chapter` ha come valore un singolo capitolo e la valutazione della clausola `for` più interna inserisce nel risultato, iterando, tutti i capitoli della parte associata alla variabile `$part`, che "scorre" tutte le parti del libro grazie all'iterazione della clausola `for` più esterna; nell'interrogazione 4, la variabile `$chapters` ha come valore *l'insieme* dei capitoli della parte del libro collegata alla variabile `$part`.

La clausola `where` ha un ruolo simile alla clausola omonima di SQL, come dimostra l'esempio seguente.

Interrogazione XQuery 5: estrarre la lista dei soli libri editi da McGraw-Hill in formato paperback dal documento `catologo.xml`.

```
for $book in doc("catalogo.xml")//libro
where $book/@editore="McGraw-Hill"
  and $book/@formato="paperback"
return $book
```

Grazie al fatto che XQuery include XPath, è spesso possibile formulare le query senza fare uso della clausola `where`, utilizzando opportune path expression con predicato. Per esempio, la query precedente può essere espressa nel modo seguente.

Interrogazione XQuery 6: estrarre la lista dei soli libri editi da McGraw-Hill in formato paperback (utilizzando come predicato una path expression).

```
for $book in doc("catalogo.xml")//libro
                          [@editore="McGraw-Hill"
                          and @formato="paperback"]
return $book
```

Come esempio dell'applicazione della clausola `order by` consideriamo la seguente query.

Interrogazione XQuery 7: estrarre i libri in catalogo, ordinandoli in base al titolo.

```
for $book in doc("catalogo.xml")//libro
order by $book/titolo
return $book
```

Come esempio di applicazione delle funzioni aggregate, mostriamo un'interrogazione più complessa, che effettua una sorta di join tra due insiemi di elementi XML (libri e autori) e sfrutta una funzione aggregata.

Interrogazione XQuery 8: estrarre l'elenco degli autori presenti nel documento `catalogo.xml` che hanno scritto almeno 5 libri.

```
for $author in  doc("catalogo.xml")//autore
let $books := doc("catalogo.xml")//libro[autore = $author]
where count($books) > 5
return $author
```

Un problema presentato dalla query precedente è il fatto che nel risultato gli autori selezionati compariranno una volta per ciascuno dei libri di cui sono autori. Per evitare ciò, è possibile utilizzare il costrutto `distinct-values` (analogo alla clausola `distinct` di SQL), che elimina i duplicati dalla sequenza di valori che compare come argomento.

Interrogazione XQuery 9: estrarre, senza duplicati, l'elenco degli autori presenti nel documento `catalogo.xml` che hanno scritto almeno 5 libri.

```
for $author in distinct-values(doc("catalogo.xml")//autore)
let $books := doc("catalogo.xml")//libro[autore = $author]
where count($books) > 5
return $author
```

Riepilogando, l'interpretazione di espressioni FLWOR segue il modello descritto graficamente in Figura 14.3. L'espressione che compare sulla destra dell'operatore `in` nella clausola `for` produce di norma una sequenza di valori. La clausola `for` assegna in modo iterativo ognuno di questi valori alla variabile dichiarata nella clausola; per esempio, nell'interrogazione 3 la variabile `$part` avrà associati i 2 elementi `<parte>` presenti in `"libro.xml"` (li chiamiamo sinteticamente P_1 e P_2). Se vi sono clausole `for` nidificate, queste produrranno per l'insieme di variabili dichiarate nelle clausole un insieme di ennuple di valori; per esempio, nell'interrogazione 3 la coppia di variabili (`$part`,`$chapter`) darà luogo a 9 coppie di frammenti XML, composte da un elemento `<parte>` e un elemento `<capitolo>` (chiamando C_i il capitolo i-esimo, le 9 coppie saranno (P_1,C_1), (P_1,C_2), (P_1,C_3), (P_1,C_4), (P_1,C_5), (P_2,C_6), (P_2,C_7), (P_2,C_8), (P_2,C_9)). La clausola `let` assegna l'intero risultato di un'espressione a una nuova variabile, senza alterare la cardinalità delle ennuple di valori associati alle variabili, ma estendendo ciascuna ennupla con nuove componenti. La clausola `where` realizza un filtraggio, valutando un predicato separatamente su ciascuna ennupla ed eliminando quelle che non lo soddisfano. La clausola `order by` permette di specificare un ordinamento sulle ennuple. Infine, la clausola `return` costruisce il risultato, operando una volta su ciascuna ennupla.

Interrogazioni XQuery avanzate Mostriamo ora alcuni esempi di query relativamente sofisticate. L'obiettivo non è quello di descrivere in modo completo e

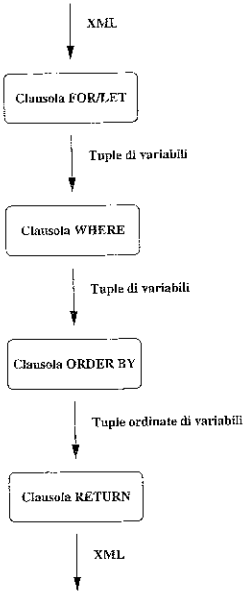

Figura 14.3 L'interpretazione di espressioni XQuery.

formale il linguaggio (si rimanda invece ai libri disponibili sull'argomento o alla ricca documentazione disponibile in Internet), bensì illustrare il potere espressivo e la flessibilità del linguaggio XQuery. Facciamo sempre riferimento allo schema XML di descrizione dei libri presentato nel Paragrafo 14.2.

Interrogazione XQuery 10: estrarre i titoli e gli anni di pubblicazione di tutti i libri pubblicati da McGraw-Hill dopo il 2001, in ordine alfabetico sul titolo.

```
<bib>
 {
  for $book in doc("catalogo.xml")//libro
  where $book/@editore="McGraw-Hill"
    and $book/@pubblicazione > 2001
  order by $book/titolo
  return <libro>
           {$book/@pubblicazione}
           {$book/titolo}
         </libro>
 }
</bib>
```

Osserviamo che la query compare all'interno di un costruttore `<bib> ... </bib>`; la clausola `for` è quindi contenuta all'interno di un costruttore che produce un unico elemento XML denominato `bib`. Il fatto che la struttura completa FLWOR possa comparire all'interno del nuovo elemento `bib` dipende dal fatto che il linguaggio XQuery realizza un paradigma di tipo funzionale, in cui una query specifica una costruzione di dati XML a partire da dati XML forniti in input; è così possibile concatenare diverse query tra loro e inserire interrogazioni XQuery direttamente all'interno di elementi XML, riutilizzando in modo estremamente flessibile i costrutti del linguaggio.

Interrogazione XQuery 11: estrarre il titolo e gli autori di ogni libro nel catalogo, presentando i dati relativi a ciascun libro all'interno di un elemento XML `risultato`.

```
<risultati>
 {
  for $book in doc("catalogo.xml")//libro
  return <risultato>
           {$book/titolo}
           {$book/autore}
         </risultato>
 }
</risultati>
```

L'aspetto particolare che distingue questa query dalla precedente è la presenza dell'espressione `{$book/autore}` all'interno della clausola `return`, che permette alla query di recuperare l'intero insieme di autori di ciascun libro.

Interrogazione XQuery 12: estrarre per ogni autore del catalogo il nome dell'autore e l'elenco dei titoli di tutti i libri da lui scritti, raggruppati in un elemento `libri`, con gli autori ordinati alfabeticamente in base al nome.

```
<risultati>
 {
  let $author := doc("catalogo.xml")//autore
  for $name in distinct-values($author/text())
  order by $name
  return
   <risultato>
     <autore> {$name} </autore>
     { for $book in doc("catalogo.xml")//libro
       where $book/autore/text()=$name
       return $book/titolo
     }</risultato>
 }
</risultati>
```

La query in questo caso individua al livello esterno i nomi degli autori, senza duplicati. Quindi, per ogni nome, ricerca all'interno del catalogo i titoli dei libri e con essi costruisce una sequenza di titoli.

Interrogazione XQuery 13: estrarre, per ogni libro con almeno un autore, il titolo e i primi due autori, aggiungendo un elemento vuoto `et-al` se il libro ha più di due autori.

```
<risultati>
 {
  for $book in doc("catalogo.xml")//libro
  where count($book/autore) > 0
  return <libroCompatto>
          {$book/titolo}
          {for $author in $book/autore[position()<=2]
             return $author }
          {if (count($book/autore) > 2)
            then <et-al/>
            else () }
         </libroCompatto>
 }
</risultati>
```

La query fa uso della funzione predefinita `position()`, che restituisce la posizione dell'elemento all'interno della sequenza dei figli. Inoltre, utilizza la struttura di controllo `if-then-else` per rappresentare l'alternativa.

Interrogazione XQuery 14: estrarre una lista piatta degli elementi `parte`. Al posto della struttura originale, ogni parte deve presentare un singolo attributo `numCapitoli` che rappresenta il numero di capitoli all'interno di ciascuna parte del libro.

```
<listaParti>
 {
  for $part in doc("libro.xml")//parte
  let $chapters := $part/capitolo
  return <parte
          numCapitoli="{ count($chapters)}"/>
 }
</listaParti>
```

La query mostra un esempio in cui la clausola `return` costruisce valori per gli attributi degli elementi, utilizzando le funzioni predefinite (in questo caso la funzione aggregata di conteggio `count`). Il risultato della query è il seguente.

```
<listaParti>
  <parte numCapitoli="4"/>
  <parte numCapitoli="5"/>
  <parte numCapitoli="2"/>
  <parte numCapitoli="0"/>
</listaParti>
```

Comandi di aggiornamento Il comitato di standardizzazione di XQuery si è interessato molto presto all'estensione del linguaggio con comandi di aggiornamento, per renderlo una soluzione completa per la gestione di dati XML, in modo analogo alla presenza all'interno di SQL dei comandi di `insert`, `update` e `delete`. I comandi di aggiornamento risiedono però in una specifica separata denominata XQuery Update, che ha raggiunto lo stato di W3C Recommendation con la versione 1.0 nel 2011. La specifica è disponibile all'URL `http://www.w3c.org/TR/xquery-update-10/`.

I comandi permettono di realizzare sugli elementi operazioni di inserimento, di cancellazione, di sostituzione, di ridenominazione e di costruzione di copie di elementi esistenti. L'inserimento di nuovi elementi avviene con il comando `insert`, che specifica dove inserire i nuovi elementi e quale struttura questi devono avere, prestando attenzione anche alla posizione dei nuovi elementi rispetto a quelli esistenti. La cancellazione è gestita con il comando `delete`, che ha una struttura relativamente semplice in quanto deve solo specificare la porzione di dati XML da rimuovere. La sostituzione viene espressa con il comando `replace`, che richiede di specificare quali elementi rimuovere e cosa inserire al loro posto; il comando può operare a livello di elementi o a livello di valori. La ridenominazione interviene invece a livello di struttura, cambiando il valore di un tag XML, ed è specificata tramite il comando `rename`. Infine, la costruzione di copie di elementi esistenti, caratterizzati da una nuova identità, viene realizzata tramite un comando `copy` che richiede il supporto dei costrutti `modify` e `return`.

Esiste quindi una varietà di modi abbastanza estesi in cui esprimere operazioni che applicano modifiche a una struttura XML esistente. Anche in questo caso rimandiamo ai documenti di specifica per una descrizione dettagliata della sintassi e ci limitiamo qui a mostrare esempi di ciascun comando.

Interrogazione XQuery 15: esprimere con XQuery Update il comando che inserisce una nuova appendice intitolata "DB2 Universal Database" nella quarta parte del libro come ultimo elemento.

```
insert node
  <appendice>DB2 Universal Database</appendice>
  as last
  into doc("libro.xml")//parte[@numero="4"]
```

Interrogazione XQuery 16: esprimere con XQuery Update il comando che rimuove il quarto capitolo della seconda parte.

```
delete node doc("libro.xml")//parte[@numero="2"]
                              /capitolo[position()=4]
```

Interrogazione XQuery 17: esprimere con XQuery Update il comando che sostituisce l'elemento che descrive l'appendice intitolata "Microsoft Access" con un elemento che descrive l'appendice intitolata "Microsoft Access 2013", cambiando l'identità dell'elemento.

```
replace node doc("libro.xml")//appendice
                          [@text()="Microsoft Access"]
  with <appendice>Microsoft Access 2013</appendice>
```

Interrogazione XQuery 18: esprimere con XQuery Update il comando che sostituisce il valore del titolo dell'appendice "Microsoft Access" con il valore "Microsoft Access 2013", mantenendo l'identità dell'elemento.

```
replace value of node doc("libro.xml")//appendice
                          [@text()="Microsoft Access"]
  with "Microsoft Access 2013"
```

Interrogazione XQuery 19: esprimere con XQuery Update il comando che ridenomina l'elemento che descrive il primo autore con un nuovo tag `PrimoAutore`. Si noti che così facendo si rende il documento non più valido rispetto al DTD corrente.

```
rename node doc("catalogo.xml")//autore[position()=1]
  as "PrimoAutore"
```

Interrogazione XQuery 20: esprimere con XQuery Update il comando che costruisce una copia degli elementi autori e inserisce in ciascuno di essi un elemento `royalty` che assegna a ciascun autore una percentuale determinata dal numero di autori del libro.

```
for $book in doc("catalogo.xml")//libro
for $author in $book/autore
return
  copy $newAuthorNode := $author
  modify insert node <royalty>
                     { (1 div count($book/autore)) * 100}
                     </royalty>
        into $newAuthorNode
  return $newAuthorNode
```

Applicando il comando all'elemento descritto precendentemente, si ottiene il seguente risultato. Si noti che il comando non modifica il documento, bensì restituisce una collezione di nuovi elementi.

```
<autore>Paolo Atzeni<royalty>20</royalty>
</autore>
<autore>Stefano Ceri<royalty>20</royalty>
</autore>
<autore>Piero Fraternali<royalty>20</royalty>
</autore>
<autore>Stefano Paraboschi<royalty>20</royalty>
</autore>
<autore>Riccardo Torlone<royalty>20</royalty>
</autore>
```

Estensioni di XQuery 3.0 Nel 2013 il comitato di standardizzazione del W3C ha rilasciato al livello di *Last Call Working Draft* la versione 3.0 del linguaggio XQuery. Si prevede che questa specifica raggiungerà in seguito il livello di *Recommendation*. È bene notare come si sia passati direttamente dalla versione 1.0 alla versione 3.0, con l'obiettivo di rendere la numerazione dello standard XQuery coerente con quella di altri standard strettamente legati a esso, come XSLT e XPath. La specifica è disponibile all'URL `http://www.w3.org/TR/xquery-30/`.

Vi sono un certo numero di cambiamenti rispetto alla versione 1.0. Illustriamo brevemente i due che appaiono più interessanti: l'aggiunta di una clausola `group by` e il supporto per l'opzione `allowing empty` nel `for` nidificato. Entrambi presentano una naturale corrispondenza con costrutti del linguaggio SQL che sono stati presentati in precedenza. XQuery 3.0 presenta poi una serie di costrutti abbastanza complessi per gestire interrogazioni su flussi (*stream*) di dati XML, per i quali rimandiamo alla specifica del linguaggio.

Per quanto riguarda la clausola `group by`, il comportamento si ispira al linguaggio SQL, ma la diversa struttura dei dati XML conduce ad alcune differenze. La clausola `group by` ha come argomento la chiave di raggruppamento, in base alla quale gli elementi oggetto della valutazione verranno aggregati. In SQL si prevede che per ogni gruppo di tuple venga prodotta una sola tupla, applicando funzioni aggregate; qui invece, in corrispondenza di ciascun valore della chiave di raggruppamento verrà mantenuta una sequenza che contiene tutti gli elementi associati a quel valore della chiave di raggruppamento.

Interrogazione XQuery 21: estrarre per ciascun editore il numero di libri in catalogo e la tiratura complessiva dei volumi.

```
for $book in doc("catalogo.xml)//libro
let $publisher := $book/editore
group by $publisher
return
  <editoreLibri nome="{$publisher}">
  Numero di libri: {count($book)}
  Tiratura complessiva: {sum($book/@copieVendute)}
  </editoreLibri>
```

Si può osservare che la query precedente poteva anche essere formulata utilizzando i costrutti illustrati di XQuery 1.0, in un modo probabilmente meno diretto e leggibile rispetto a quanto offerto dalla clausola `group by`.

La clausola `allowing empty` è invece analoga al costrutto di outer join illustrato per SQL nel Paragrafo 4.3.2. Usando questa clausola in un `for` nidificato è possibile mantenere in considerazione nel risultato anche elementi della scansione esterna cui non corrispondono elementi nel contesto interno. Contribuiscono quindi al risultato tutti gli elementi considerati al livello più esterno.

Interrogazione XQuery 22: estrarre, per ciascun libro, il titolo e il numero di capitoli di ogni parte, considerando anche i libri che non presentano parti al loro interno.

```
for $book in doc("catalogo.xml)//libro
  for $part allowing empty in $book/parte
  return
    <libro>
      Titolo: {$book/titolo}
      Numero di capitoli nella parte: {count($part
      /capitolo)}
    </libro>
```

14.4 Gestione di dati XML con DBMS relazionali

Nei paragrafi precedenti si è visto come i documenti XML possono essere dotati di schema e interrogati, presentando caratteristiche analoghe ai dati contenuti in una base di dati relazionale o a oggetti.

Tuttavia queste somiglianze non devono indurre a credere che i documenti XML siano direttamente utilizzabili per la costruzione di applicazioni complesse quali i sistemi informativi. Alcune delle caratteristiche essenziali per la costruzione di un sistema informativo non sono garantite dall'uso di semplici documenti XML come deposito di informazioni. Si pensi, per esempio, all'uso degli indici per velocizzare l'esecuzione delle interrogazioni, alla gestione degli accessi concorrenti, alle transazioni, al mantenimento della consistenza dei dati e ai meccanismi di ripristino da guasti.

Pertanto, per utilizzare XML come formato di rappresentazione dei dati in applicazioni complesse è necessario combinare tale standard con le capacità tipiche dei sistemi di gestione delle basi di dati.

In questo paragrafo mostriamo come i DBMS relazionali e a oggetti vengono estesi allo scopo di consentire la gestione di dati XML. Nel paragrafo successivo, invece, illustreremo brevemente una diversa classe di sistemi, chiamati DBMS XML nativi, che coniugano le funzionalità di un DBMS con il formato di rappresentazione dei dati XML, senza fare uso di tecnologia relazionale ma sfruttando modelli logici e strutture fisiche di memorizzazione concepite espressamente per i dati XML.

La scelta tra i due tipi di soluzione dipende in gran parte dalla natura dei dati XML e dal tipo di applicazioni che si devono costruire. Se i dati XML sono semplici e ben strutturati e utilizzano XML solo come un formato intermedio di rappresentazione, senza sfruttarne tutte le caratteristiche più complesse, allora l'uso di basi di dati relazionali con supporto XML è solitamente preferibile. Se invece i dati XML sono complessi e poco strutturati e le applicazioni hanno come obiettivo principale l'elaborazione di documenti, allora la scelta di un DBMS XML nativo può essere conveniente.

L'architettura generale di un sistema relazionale o a oggetti esteso per il supporto di dati XML è descritta nella Figura 14.4.

I dati sono gestiti all'interno della base di dati nel formato relazionale o a oggetti e opportune interfacce consentono l'inserimento e l'estrazione di documenti XML. Tali interfacce possono essere di due tipi.

- *Interfacce di programmazione* (*Application programming interface*, API): sono interfacce basate su librerie concepite per agevolare l'inserimento di documenti XML nella base di dati e l'estrazione di dati relazionali in formato XML. La

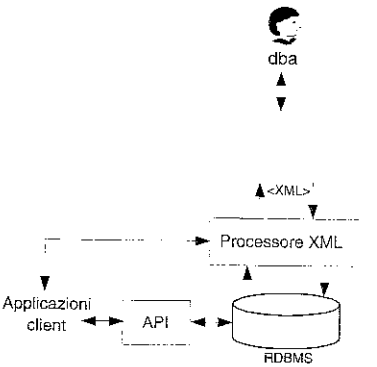

Figura 14.4 Architettura delle basi di dati relazionali con supporto XML.

trasformazione da formato relazionale a XML e viceversa è essenzialmente a carico di un programma applicativo che sfrutta le funzioni di libreria per una più agevole interazione con la base di dati. Eventualmente tale trasformazione può essere agevolata dalla presenza di uno schema di conversione predefinito tra il modello logico relazionale e il formato XML. A titolo di esempio, si possono menzionare l'interfaccia ADO.NET di Microsoft e le librerie XML di Oracle. Per esempio, ADO.NET fornisce automaticamente la rappresentazione dei risultati di un'interrogazione SQL in XML, usando uno schema XML predefinito.

- *Interfacce di interrogazione*: sono interfacce che sfruttano linguaggi di interrogazione dichiarativi per l'inserimento e l'estrazione dei dati XML e processori dedicati alla conversione delle interrogazioni o degli aggiornamenti su dati XML in operazioni equivalenti su dati relazionali. Al momento attuale, le basi di dati relazionali estese con supporto XML offrono principalmente linguaggi per l'estrazione dei dati, mentre l'aggiornamento è ancora svolto in prevalenza mediante interfacce di tipo programmativo. I linguaggi adottati dai sistemi commerciali sono estensioni proprietarie di SQL, che includono primitive per la formattazione del risultato di un'interrogazione in XML. Alcuni dei principali venditori di sistemi di gestione delle basi di dati si sono consorziati per definire un'estensione del linguaggio SQL, denominata SQL/XML, divenuta poi uno standard per l'estrazione di documenti XML da basi di dati relazionali.

La gestione di dati XML con tecnologia relazionale presuppone una qualche forma di adattamento (mapping) tra la struttura dei documenti XML, espressa per esempio come DTD o XSD, e quella delle tabelle relazionali o classi presenti all'interno della base di dati. Tale mapping può essere realizzato in diversi modi.

- Custom: il DBMS non prevede un mapping predefinito tra DTD o XSD e catalogo relazionale. Il progettista della base di dati deve quindi definire lo schema relazio-

nale più opportuno per accogliere i dati XML e il programmatore applicativo deve deserializzare i dati XML e inserirli nelle relazioni della base di dati e, viceversa, costruire documenti XML a partire dai risultati di una o più interrogazioni SQL.

- Mapping canonico: esiste una regola di rappresentazione dei dati relazionali in XML indipendente dal particolare documento, DTD o schema XML. Tale mapping solitamente implica l'uso di un insieme di elementi XML prefissato (per esempio `<catalog>`, `<table>`, `<row>`) mediante i quali è possibile rappresentare qualunque struttura relazionale in XML e quindi convertire automaticamente i risultati delle interrogazioni SQL in XML. La conversione di documenti in tabelle o resta a carico del programmatore, oppure può avvalersi di una trasformazione documentale XSL, che porta il documento dal formato originale a quello canonico, e di comandi per l'inserimento diretto di dati XML in formato canonico in tabelle relazionali di schema predefinito. Si noti che in quest'ultimo approccio le relazioni utilizzate per memorizzare documenti XML sono sempre le stesse e non dipendono dallo schema del documento XML.
- Mapping basato sullo schema del documento: si definiscono regole generali per la trasformazione dei vari costrutti XML in costrutti relazionali equivalenti; per esempio:
  - ciascun elemento XML con attributi, contenuto testuale o misto produce una relazione con lo stesso nome dell'elemento, dotata delle colonne necessarie alla rappresentazione degli attributi e del contenuto testuale;
  - l'annidamento tra elementi viene rappresentato con relazioni innestate o relazioni piatte collegate da chiavi esterne.

  Si noti che, a differenza del mapping canonico, il mapping basato sullo schema del documento associa diversi schemi relazionali a diversi schemi XML.

Qualunque sia il tipo di mapping effettuato, l'uso di una base di dati relazionale per la memorizzazione di documenti XML solitamente non preserva tutte le caratteristiche del documento originale. Per esempio, l'ordine di apparizione dei sotto-elementi nel modello di contenuto di un elemento può essere perso una volta che il documento è stato trasformato in un insieme di tuple relazionali.

Concludiamo questa breve illustrazione della gestione di dati XML con basi di dati relazionali accennando alle caratteristiche di SQL/XML, l'estensione del linguaggio SQL per la trattazione di dati XML che è ora uno standard all'interno di SQL-3. SQL/XML costituisce la *Part 14* di SQL-3; la prima proposta è apparsa nella versione 2003 dello standard (SQL:2003-14), con successivi aggiornamenti (SQL:2006-12, SQL:2008-14, SQL:2011-14) che hanno seguito l'evoluzione di XQuery. Il punto di partenza di SQL/XML è l'introduzione di un nuovo tipo SQL atto a rappresentare dati in formato XML. Tale tipo, chiamato semplicemente XML, può essere usato per definire colonne, variabili o parametri SQL. I valori di colonne di tipo XML possono essere costruiti a partire dai contenuti di tabelle relazionali utilizzando opportuni operatori nella clausola `select` di un'interrogazione SQL. SQL/XML prevede diversi operatori, tra cui:

- `xmlelement`: costruisce un singolo elemento XML dotato di contenuto e/o attributi;

- `xmlforest`: costruisce un insieme di elementi XML a partire da valori passati come argomento;
- `xmlconcat`: costruisce un insieme di elementi XML concatenando singoli elementi XML;
- `xmlagg`: si utilizza in congiunzione al costrutto SQL `group by` e consente di produrre liste innestate di elementi XML conformi ai gruppi indotti dalla clausola di raggruppamento dell'interrogazione SQL.

A titolo di esempio si riporta una semplice interrogazione SQL/XML che costruisce una colonna di valori di tipo XML a partire dai contenuti della tabella IMPIEGATO con attributi Matr e Nome.

```
select I.Matr,
       xmlelement (name "Imp",
          xmlattributes (I.Matr, I.Nome as "Cognome"))
          as Risultato
from Impiegato I;
```

Il risultato della valutazione dell'interrogazione è la tabella a due colonne riportata di seguito:

Matr	Risultato
1001	`<Imp Matr="1001" Cognome="Smith"/>`
1206	`<Imp Matr="1206" Cognome="Martin"/>`
...	...

La seconda colonna della tabella contiene gli elementi XML costruiti dall'operatore `xmlelement`. Si tratta di elementi XML privi di contenuto e con due attributi, costruiti con la funzione `xmlattributes`.

14.5 Le basi di dati XML native

Nel paragrafo precedente abbiamo mostrato come i DBMS relazionali e il linguaggio SQL consentano di immagazzinare in modo persistente e interrogare dati in formato XML.

Un approccio alternativo alla gestione dei dati in formato XML è costituito dalle cosiddette *basi di dati XML native*, cioè da sistemi per la gestione di basi di dati concepiti espressamente per XML.

Le basi di dati XML native offrono funzioni per memorizzare in modo persistente, interrogare e trasformare documenti XML. Per far ciò, utilizzano un modello logico dei dati specifico per la rappresentazione dei documenti, in grado di esprimere la struttura gerarchica degli elementi che costituiscono un documento XML. Un siffatto modello logico consente la rappresentazione delle caratteristiche tipiche di XML, quali le relazioni di inclusione tra elementi e sotto-elementi, l'ordinamento presente tra i sotto-elementi di un elemento e il contenuto testuale di attributi ed elementi. Grazie

alla loro specializzazione, le basi di dati XML native sono indicate nei casi in cui i dati XML da immagazzinare sono particolarmente complessi, per esempio perché fanno uso di caratteristiche avanzate di XML, quali istruzioni processabili, commenti e definizioni di entità, oppure nei casi in cui i dati XML sono particolarmente destrutturati, per esempio a causa dell'assenza di un DTD o di uno schema XML.

La caratteristica distintiva delle basi di dati XML native è la presenza di un *modello logico dei documenti XML*, utilizzato come riferimento per la memorizzazione e l'interrogazione dei dati XML. Diversamente dalle basi di dati relazionali, però, le basi di dati XML native non adottano tutte lo stesso modello logico. Alcuni prodotti si basano su standard o proposte di standard collegati a XML, quali per esempio il modello di dati associato allo standard DOM (*Document Object Model*), *XPath data model* e *XML Information Set*, che propongono modelli alternativi della struttura dei documenti. Esistono inoltre prodotti che utilizzano modelli logici del tutto proprietari.

Al livello fisico, i DBMS XML adottano le più svariate tecnologie per la memorizzazione persistente e l'accesso ai dati XML. Gli approcci prevalenti possono essere classificati in due categorie principali:

- *memorizzazione basata sul testo*: i documenti vengono memorizzati in forma testuale, sfruttando il file system o un DBMS relazionale che supporti il tipo CLOB (*Character Large Object*). Grazie all'indicizzazione del contenuto testuale, la memorizzazione basata su testo consente alte prestazioni nel caso di interrogazioni orientate all'estrazione di interi documenti o di frammenti identificabili mediante accesso all'indice; invece, questa forma di memorizzazione appesantisce l'esecuzione di interrogazioni che richiedono la ristrutturazione dei documenti, a causa della necessità di ricostruire il risultato a partire da un testo con una diversa struttura;
- *memorizzazione basata sul modello logico*: i documenti vengono memorizzati sfruttandone il modello logico, per esempio associando a ogni elemento del modello logico una tabella relazionale, una classe persistente o una struttura dati proprietaria. Le prestazioni di questa tecnica di memorizzazione dipendono dalle specifiche modalità di mappatura del modello logico nelle strutture fisiche, che determinano il numero di operazioni di accesso ai dati necessarie per costruire il risultato dell'interrogazione.

Una caratteristica particolare delle basi di dati XML native è il supporto alla definizione di *collezioni di documenti*. Una collezione è un insieme di documenti raggruppati secondo qualche criterio di omogeneità. A differenza di una relazione nel modello logico relazionale, le cui istanze hanno tutte rigorosamente la stessa struttura, i documenti di una collezione possono avere diversa conformazione, fino a giungere al caso di collezioni di documenti XML completamente eterogenei, privi di DTD o XSD. In tal caso, le collezioni servono essenzialmente per raggruppare documenti correlati, per esempio al fine di effettuare interrogazioni su sottoinsiemi dei dati definiti dall'utente.

Come le basi di dati relazionali e quelle a oggetti, le basi di dati XML native consentono la formulazione di interrogazioni e aggiornamenti dei dati. Al momento, i sistemi commerciali forniscono una varietà di linguaggi di interrogazione, quali XPath, XQuery e XSL, descritti precedentemente, e vari altri linguaggi proprietari, alcuni dei quali ispirati a SQL. In futuro, si prevede una convergenza verso il linguaggio XQuery, che si auspica possa assumere il ruolo che ha avuto SQL per le basi di dati relazionali.

Note bibliografiche

Una trattazione sistematica di XML è contenuta nel libro di Abiteboul, Suciu e Buneman [88]. Il testo inquadra XML nel contesto più ampio dei dati semistrutturati e contiene un'esposizione approfondita dei fondamenti teorici da cui XML e i linguaggi di interrogazione per dati semi-strutturati prendono spunto. XML e XSL sono trattati nei testi [99, 124, 143], con un taglio spiccatamente pratico e applicativo. Esistono già un certo numero di libri dedicati a XQuery, tra cui [136, 149]. Una fonte di informazioni attendibile e aggiornata è il sito del Consorzio W3C, che dedica all'interrogazione dei dati XML un'intera sezione (`http://www.w3.org/XML/Query`).

Le basi di dati XML sono l'oggetto principale del libro [105], in cui si illustrano le caratteristiche di diversi sistemi commerciali, sia basi di dati XML native sia sistemi relazionali estesi con supporto XML, e si discutono in generale gli aspetti architetturali della memorizzazione di contenuti XML nelle basi di dati relazionali. La specifica di SQL/XML è disponibile presso l'ISO. I Web Service sono discussi in modo approfondito nel libro [90], che parte dalla definizione di Web Service e discute le caratteristiche architetturali e i principali linguaggi e protocolli proposti in questo settore.

Esercizi

Soluzioni sul sito http://www.ateneoniine.it/atzeni

14.1 Dato il seguente DTD:

```
<!ELEMENT listino (articolo+)>
<!ELEMENT articolo (descrizione,variante+)>
<!ELEMENT descrizione (#PCDATA)>
<!ELEMENT variante (nome,prezzo,qta_disponibile?)>
<!ATTLIST listino
  data-inizio      CDATA              #REQUIRED
  data-fine        CDATA              #REQUIRED
  valuta           (euro|dollaro)     'euro'>

<!ATTLIST articolo
  codice           CDATA              #REQUIRED
>
```

scrivere un documento XML valido rispetto a tale DTD.

14.2 Dato il DTD dell'Esercizio 14.1, scrivere uno schema XML a esso equivalente. Dove opportuno, usare le caratteristiche di XML Schema che consentono una specifica più precisa della struttura dei documenti.

14.3 Dato il seguente documento XML:

```
<elenco-ordini>
 <ordine stato="aperto" data="1-1-2002">
  <cliente categoria="A">
    <nome>Rossi</nome>
    <indirizzo>Via Verdi 1, Milano</indirizzo>
    <indirizzo-spedizione>Via Gracchi 12, Milano
    </indirizzo-spedizione>
```

```
    </cliente>
    <linea-ordine prodotto="12" qta="1" prezzo="120"/>
    <linea-ordine prodotto="13" prezzo="340"/>
    <linea-ordine prodotto="67" qta="10" prezzo="330"/>
  </ordine>
  <ordine stato="chiuso" data="12-10-2002">
    <cliente categoria="B">
      <nome>Verdi</nome>
      <indirizzo>Via Rossi 1, Bergamo</indirizzo>
    </cliente>
    <linea-ordine prodotto="33" qta="11" prezzo="100"/>
    <linea-ordine prodotto="13" prezzo="55"/>
  </ordine>


  <ordine stato="aperto" data="2-1-2003">
    <cliente categoria="C">
      <nome>Neri</nome>
      <indirizzo>Via Pisani 31, Como</indirizzo>
    </cliente>
    <linea-ordine prodotto="12" qta="1" prezzo="120"/>
    <linea-ordine prodotto="44" prezzo="340"/>
    <linea-ordine prodotto="55" qta="6" prezzo="234"/>
  </ordine>
```

scrivere un DTD rispetto al quale il documento sia valido nell'ipotesi che l'attributo `prezzo` sia obbligatorio, l'attributo `categoria` sia opzionale, l'attributo `qta` abbia come default il valore 1, l'attributo `stato` possa valere "aperto" o "chiuso". Scrivere poi uno schema XML rispetto al quale il documento sia valido sotto le stesse ipotesi.

14.4 Dato il documento dell'Esercizio 14.4, scrivere in XSL l'interrogazione che estrae tutti i clienti che hanno ordinato il prodotto n. 12 e la quantità totale di tale prodotto da essi ordinata.

Scrivere la stessa interrogazione in XQuery.

14.5 Dato il seguente documento XML:

```
<dizionario>
 <termine id='1'> parola
  <fonetica> [pa-ro'-la] </fonetica>
  <etimologia>Lat. tardo parabola(m) 'parabola'</etimologia>
  <lemma genere='s.f.'>
  <definizione>complesso di suoni articolati (o anche un solo
    suono) che esprime un significato;  ...</definizione>
  <definizione informale='s'> unita' logica minima di
    informazione, costituita da un gruppo di caratteri  in
    un'unica cella di memoria alla quale si accede con
    un unico indirizzo</definizione>
  </lemma>
 </termine>
 <termine id='2' sinonimi='4,8'>   buono
  <fonetica> [buo'-no] </fonetica>
  <etimologia>Lat. bonu(m)</etimologia>
  <lemma genere='agg'>
```

```
    <definizione> conforme al bene; onesto, moralmente
      positivo</definizione>
    <definizione> tranquillo, calmo; composto,
      silenzioso</definizione>
    </lemma>
    <lemma genere='s.m.'>
    <definizione> documento che da' diritto, all'atto della
      presentazione, alla riscossione di una somma, al ritiro di
      una merce, alla prestazione di un servizio </definizione>
    </lemma>
  </termine>
... <dizionario>
```

scrivere una trasformazione XSL che estragga dal dizionario tutti gli aggettivi e produca un documento HTML così composto: la frase "Elenco degli aggettivi" deve comparire sia come titolo del documento HTML sia come primo paragrafo; successivamente deve seguire la lista degli aggettivi presenti nel dizionario: ogni termine deve essere rappresentato in corsivo (rappresentato in HTML dal tag <TT>), seguito dall'etimologia (se esiste), dal numero di definizioni trovate e da una serie di paragrafi contenenti le diverse definizioni.

14.6 Dato il seguente schema relazionale:

AUTORE (Codice, Nome, Cognome)
SCRIVE (Codice,ISBN)
LIBRO (ISBN, Titolo, Editore, Prezzo)

scrivere un'interrogazione SQL/XML che produca un tabella con una sola colonna contenente valori di tipo XML. Ogni tupla deve contenere un frammento XML simile a quello seguente:

```
<libro ISBN="8838660085" editore="McGraw-Hill"
       numero_autori="4">
  Basi di dati: modelli e linguaggi</libro>
```

15

Basi di dati semantiche, SPARQL e linked open data

L'aggiunta di aspetti semantici ai modelli delle basi di dati ha caratterizzato l'evoluzione delle basi di dati fin dalle sue origini. Solo 4 anni dopo la pubblicazione del modello relazionale, Jean-Raymond Abrial nel 1974 ha proposto un modello semantico che in affetti era una prima versione del modello a oggetti [89], mentre un anno dopo Peter Chen ha proposto il modello Entità-Relazione che abbiamo utilizzato nella Parte 2 come strumento per la progettazione concettuale dei dati. Si tratta dei precursori di un vastissimo numero di modelli che, per le loro caratteristiche, classifichiamo come *modelli semantici*, e che nella parte seconda del libro abbiamo chiamato *modelli concettuali*. L'uso dei modelli semantici dei dati può essere assai diverso.

- Possono essere utilizzati durante la fase iniziale della progettazione della base di dati, per descrivere fedelmente la realtà applicativa; gli schemi realizzati con i modelli semantici vengono poi trasformati in schemi compatibili con i modelli dei sistemi commerciali, che offrono minori capacità espressive, in modo da poter sfruttare la loro maggior robustezza e efficienza. Questo approccio alla progettazione è usato nella Parte 2 di questo libro ed è comune a molte altre tecniche di progettazione di sistemi sofware.
- In altri casi, i modelli semantici sono usati come strumento per consentire la rappresentazione e gestione dei dati e la formulazione di interrogazioni ad alto livello, in cui si sfruttino conoscenze semantiche più ricche di quelle contenute nella base di dati; diverse trasformazioni consentono di interpretare l'interrogazione e di esprimerla nel linguaggio di query dei sottostanti sistemi di gestione dei dati, in modo da costruire la risposta alle interrogazioni.
- Infine, ci sono vari sistemi che supportano direttamente i modelli semantici. In questo caso, gli utenti possono usare il modello semantico per esprimere le interrogazioni, e il sistema è capace di interpretarle ed eseguirle usando tecniche semantiche, per esempio di inferenza, che sono realizzate all'interno del sistema.

Le tre opzioni nascondono tutte interessantissimi problemi di ricerca e hanno tutti grandi possibilità di utilizzo. Per comprendere in questo quadro le varie tecnologie descritte dal corso, si tenga presente che Datalog, presentato nel Capitolo 3 come un linguaggio formale per esprimere interrogazioni, è di fatto un linguaggio capace di fare inferenze, calcolando i predicati definiti in modo intensionale a partire dal database estensionale, e che quindi si presta alla descrizione di aspetti semantici, quali le ontologie oppure i vincoli di integrità [101]; d'altra parte, l'uso di ontologie per riformulare interrogazioni su basi di dati, per esempio di tipo biologico o medico, è diventato sempre più efficace e diffuso. Pertanto, mentre si può dire che l'uso dei modelli semantici ha prevalso per le attività di design, anche il loro uso per interrogare direttamente i dati o per meglio comporre le interrogazioni e interpretare i risultati sta caratterizzando l'attuale sviluppo della gestione dei dati.

Una spinta importante per lo sviluppo degli aspetti semantici direttamente nei sistemi di gestione dei dati si è verificato negli anni ottanta, con la costruzione dei *database a oggetti*, trattati nel Capitolo 13; in questo caso, la caratteristica semantica evidenziata è la complessità strutturale degli oggetti che costituiscono la base di dati, capaci di avere organizzazioni interne complesse e di relazionarsi attraverso ricche relazioni di natura semantica.

La spinta più forte a uno sviluppo della semantica nella gestione dei dati si accompagna alla nascita del *Semantic Web*, proposto da Tim Berners-Lee nella primavera del 1999. Già il Web aveva rivoluzionato il mondo delle basi di dati, portando alla nascita di linguaggi semi-strutturati (tra cui XML, descritto nel Capitolo 14) che meglio si adattano alla descrizione di dati con una conformazione irregolare. Il Semantic Web introduce una nuova sfida, quella di far interagire autonomamente programmi sul Web in base alla descrizione delle loro caratteristiche, viste come *annotazioni semantiche*.

Per realizzare il Semantic Web si è posto quindi il problema di disporre di un modello adatto a esprimere annotazioni, e poi di costruire linguaggi con una capacità espressiva crescente, in grado di inferire sempre più informazioni dalle annotazioni, con l'obiettivo di sostituire la programmazione delle applicazioni con ragionamenti che gli oggetti computazionali possono svolgere quando vengono messi a confronto (composti) per svolgere un compito. Anche se la sfida del Semantic Web non è stata per ora vinta, essa ha prodotto una vorticosa innovazione; in particolare, un aspetto inizialmente sottovalutato, e cioè la costruzione di raccolte *aperte* e *connesse* (i cosiddetti *linked open data*), ha assunto un'importanza crescente, di fatto aprendo la strada a tantissime, interessanti applicazioni delle tecnologie delle basi di dati semantiche.

Con queste premesse, in questo capitolo ci concentreremo sulle tecnologie semantiche per le basi di dati proposte dal Semantic Web. Introdurremo dapprima il modello dei dati RDF, per poi introdurre il linguaggio di interrogazione SPARQL, usato per interrogare basi di dati RDF; concluderemo descrivendo lo scenario dei *linked open data* e loro possbili applicazioni.

15.1 Il modello dei dati RDF

Il modello RDF (*Resource Description Framework*), proposto dal W3C, esprime le informazioni sotto forma di insiemi di triple che possono essere interpretate come soggetto, predicato e oggetto di una breve frase; l'insieme di tutte le triple costituisce un grafo. Un esempio di tripla RDF è la frase:

```
< Verdi HaComposto Traviata >
```

ove `Verdi` è il soggetto, `HaComposto` è il verbo e `Traviata` è l'oggetto.

Un'*istanza RDF* è un insieme di triple RDF (s, p, o); le triple possono essere rappresentate graficamente mediante un grafo nel quale, per ogni tripla (s, p, o), esiste un arco con etichetta `'p'` tra un nodo con etichetta `'s'` e un nodo con etichetta `'o'`; un'istanza RDF non deve avere necessariamente uno schema di riferimento, ma è possibile definire, almeno parzialmente, metadati mediante una notazione chiamata RDFS, come vedremo nel prossimo paragrafo. In genere, le etichette usate in RDF possono appartenere a un vocabolario o un'ontologia predefinita, e ciò può consentire di derivare ulteriore conoscenza. Ogni vocabolario (namespace) utilizzato in un'istanza RDF è introdotto da un'istruzione `prefix` che consente di estrarre le etichette presenti nel vocabolario. Per esempio:

```
@prefix o: <http://www.polimi.it/ceri/opera> .
```

Questa istruzione rende disponibili all'interno dell'istanza RDF i termini del vocabolario presente presso `<http://www.polimi.it/ceri/opera>`, purché tali termini siano preceduti dal prefisso `o:`. Nel definire un'istanza RDF, si possono indicare più vocabolari.

I dati rappresentabili in RDF sono di tre categorie.

- Gli IRI (*Internationalized Resource Identifier*) sono stringhe che identificano univocamente tutte le risorse disponibili sul Web in base a una notazione standard,

proposta dal W3C;[1] talvolta essi offrono anche un metodo per accedere alla risorsa (si parla in tal caso di URL, *Uniform Resource Locator*).

- I *letterali* sono valori che descrivono le risorse presenti nell'istanza RDF. È possibile assegnare ai letterali un tipo; per esempio, le clausole `xsd:integer` o `xsd:date` poste dopo un letterale (una stringa delimitata da apici o da virgolette) lo fanno interpretare rispettivamente come un numero intero oppure una data. I letterali inclusi tra apici sono implicitamente di tipo stringa, mentre numeri senza apici, con segno, sono implicitamente di tipo intero.
- I *blank nodes* sono identificatori usati per rappresentare risorse anonime, che non corrispondono a risorse Web. I blank nodes vengono estratti da un vocabolario particolare, denotato con il singolo carattere di *underscore* (_); quindi, sintatticamente, i blank nodes sono preceduti dalla stringa `_:`.

Nelle triple RDF il primo termine (soggetto) può essere un IRI o un blank node, il secondo termine (predicato) è un IRI, e il terzo termine (oggetto) può essere un IRI, un letterale o un blank node. Nel seguito, rappresentiamo 5 triple in cui si assume che le etichette dei predicati (`HaTitolo`, `RappresentataIn`, `HaNome`, `HaLuogo`, `HaSito`) siano prelevate dal vocabolario `opera` introdotto in precedenza; l'indirizzo del sito della Scala di Milano è una URL, mentre `_:o1` e `_:p1` sono blank node. Si noti che `_:o1` corrisponde a un oggetto (un'opera) caratterizzato da una proprietà (il titolo Traviata) e `_:p1` corrisponde a un oggetto (un teatro di Milano) caratterizzato da tre proprietà (nome, luogo, sito). In generale, i blank nodes consentono di rappresentare in modo univoco risorse in un'istanza RDF senza che sia necessario attribuire loro una URI.

```
_:o1 o:HaTitolo 'Traviata' ;
     o:RappresentataIn _:p1 .
_:p1 o:HaNome 'TeatroAllaScala' ;
     o:HaLuogo 'Milano' ;
     o:HaSito <http://www.teatroallascala.org> .
```

Viene adottata una versione semplificata della notazione *N3*,[2] che utilizza la punteggiatura per sintetizzare le parti comuni delle triple. Quando due triple condividono il soggetto si usa il separatore `';'`; quando condividono sia soggetto sia predicato, si usa il separatore `','`. La rappresentazione grafica di queste triple è mostrata in Figura 15.1.

Oltre alla notazione N3, è possibile utilizzare il formato XML per la definizione di un'istanza RDF; tale notazione prende il nome di *RDF/XML* ed è stata standardizzata dal W3C. La rappresentazione in RDF/XML dell'istanza RDF appena discussa è mostrata in Figura 15.2.

Nella Figura 15.3 è descritta un'istanza RDF relativa al *Ring*, la tetralogia di Wagner offerta alla Scala di Milano nel Giugno 2013.

[1] Il termine IRI ha recentemente sostituito negli standard RDF e SPARQL del W3C il termine URI (Universal Resurce Identifier), generalizzandolo.

[2] Comunemente anche chiamata Turtle (Terse RDF Triple Language).

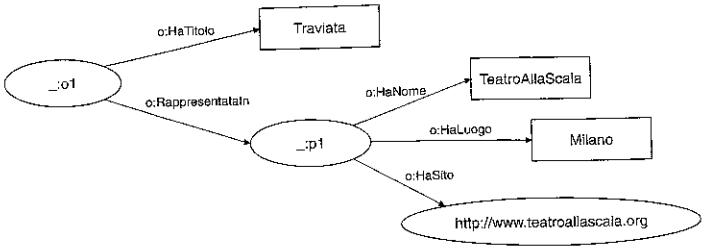

Figura 15.1 Un grafo RDF.

15.2 RDF Schema e OWL

Il modello dei dati fin qui utilizzato è un modello utile a esprimere conoscenza elementare in forma di triple, ma non possiede di per se la capacità di classificare risorse sulla base delle loro caratteristiche (per esempio, dire che Verdi e Wagner sono elementi della classe Compositori) né caratteristiche semantiche più evolute che consentano di derivare proprietà di dati. Viceversa, la comunità del Web semantico ha costruito vari linguaggi che consentono di esprimere aspetti semantici progressivamente più espressivi (e anche più difficili da valutare).

La prima estensione di RDF è *RDF Schema* (RDFS), che è una raccomandazione del W3C dal 2004; RDFS aggiunge a RDF costrutti per descrivere *metadati*, ovvero struttura e proprietà di istanze RDF. Utilizzando la terminologia tipica delle basi di dati, RDFS costituisce cioè lo strumento per definire lo *schema* di un'istanza RDF. Utilizzando invece la terminologia tipica del Web semantico, RDFS consente la definizione di *vocabolari* per la descrizione di un insieme di risorse e delle relazioni esistenti tra di esse.

```
<?xml version="1.0"?>
<rdf:RDF xmlns:o="http://www.polimi.it/ceri/opera"
         xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:ceri="http://www.polimi.it/ceri/">
  <rdf:Description>
    <ceri:operaHaTitolo>Traviata</ceri:operaHaTitolo>
      <ceri:operaRappresentataIn>
        <rdf:Description>
          <ceri:operaHaNome>TeatroAllaScala</ceri:operaHaNome>
          <ceri:operaHaLuogo>Milano</ceri:operaHaLuogo>
          <ceri:operaHaSito rdf:resource="http://www.teatroallascala.org"/>
        </rdf:Description>
      </ceri:operaRappresentataIn>
  </rdf:Description>
</rdf:RDF>
```

Figura 15.2 La rappresentazione in RDF/XML dell'istanza RDF rappresentata graficamente in Figura 15.1.

```
@prefix o: <http://www.polimi.it/ceri/opera> .

_:Wagner o:nome "Richard Wagner";
         o:natoIn "Germania" .

_:Ring o:compostoDa _:Wagner .
_:Ring o:prodottoDa "Scala Di Milano" .
_:Ring o:condottoDa "Daniel Baremboim" .
_:Ring o:messoInScenaDa "Guy Cassiers" .
_:Ring o:haTitolo "Ring".

_:Ring o:comprende _:OroDelReno,
                   _:Valchiria,
                   _:Sigfrido,
                   _:Crepuscolo.

_:OroDelReno o:haTitolo "Oro Del Reno" ;
             o:dura 150 ;
             o:haPersonaggio "Wotan" .

_:Valchiria o:haTitolo "Valchiria" ;
            o:dura 310 ;
            o:haAtti 3 ;
            o:haPersonaggio "Brunilde" ;
            o:haPersonaggio "Wotan" ;
            o:haPersonaggio "Sieglinde" .

_:Sigfrido o:haTitolo "Sigfrido" ;
           o:dura 320 ;
           o:haAtti 3 ;
           o:haPersonaggio "Brunilde" ;
           o:haPersonagio "Sigfrido" .

_:Crepuscolo o:haTitolo "Crepuscolo degli Dei" ;
             o:dura 360 ;
             o:haAtti 3 ;
             o:haPrologo "si" ;
             o:haPersonaggio "Brunilde" ;
             o:haPersonaggio "Sigfrido";
             o:haPersonaggio "Waltraute" .
```

Figura 15.3 Istanza RDF relativa al Ring di Wagner.

I principali costrutti messi a disposizione da RDFS sono le *classi* e le *proprietà*. Una classe RDFS consente di definire insiemi di risorse di un'istanza RDF che hanno caratteristiche comuni. Una proprietà RDFS consente invece di definire il soggetto e l'oggetto dei predicati in un'istanza RDF.

Se per esempio vogliamo introdurre nell'istanza RDF in Figura 15.3 il concetto di *compositore* ed esprimere il fatto che Wagner è un membro di questa classe, possiamo estendere l'istanza RDF con le seguenti istruzioni.

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix o: <http://www.polimi.it/ceri/opera>.
o:Compositore rdf:type rdfs:Class.
_:Wagner rdf:type o:Compositore.
```

Le prime due definizioni servono a dichiarare i vocabolari di RDF e di RDFS. La prima tripla definisce la classe `Compositore` (risorsa del nostro vocabolario) mediante la risorsa predefinita `class` del vocabolario di RDFS e il predicato predefinito `type` del vocabolario di RDF (che in notazione N3 può essere semplicemente denotato dalla lettera `a`). La tripla successiva dichiara, con lo stesso predicato `type` di RDF, che la risorsa anonima `Wagner` è un'istanza della classe `Compositore`. In effetti, la seconda tripla implica la prima, che può quindi essere omessa.

Per ogni proprietà RDFS va definito il tipo del soggetto (con il predicato `domain` di RDFS) e dell'oggetto (con il predicato `range` di RDFS). Se per esempio vogliamo caratterizzare i predicati `nome` e `compostoDa` dell'istanza RDF in Figura 15.3, possiamo aggiungere le istruzioni che seguono.

```
o:nome a rdfs:Property;
       rdfs:domain o:Compositore;
       rdfs:range rdfs:Literal.

_:Ring a o:Opera.
o:compostoDa a rdfs:Property;
             rdfs:domain o:Opera;
             rdfs:range o:Compositore.
```

La prima tripla dichiara che la risorsa del nostro vocabolario `nome` è una proprietà RDFS. Questa definizione è in effetti implicita dalla presenza della tripla `_:Wagner o:nome 'Richard Wagner'` presente nell'istanza RDF e può quindi essere omessa. Le triple che seguono ne definiscono il dominio (un compositore) e il codominio (una stringa, con la risorsa RDFS predefinita `string`). Si tratta in pratica della definizione di un attributo della classe `Compositore`. Le triple successive dichiarano che la risorsa anonima `Ring` è un membro della classe `Opera` (implicitamente definita) e che il predicato `compostoDa` lega le opere ai loro compositori. Si tratta in pratica della definizione di una relazione tra le classi `Opera` e `Compositore`.

In RDFS è inoltre possibile definire relazioni di generalizzazione tra classi. Per esempio, possiamo dichiarare che il concetto di *artista* generalizza quello di *compositore* con l'istruzione che segue, che fa uso del predicato predefinito `subClassOf` di RDFS.

```
o:Artista rdf:type rdfs:Class.
o:Compositore rdfs:subClassOf o:Artista.
```

Alla stessa maniera, è possibile definire proprietà che sono casi particolari di altre proprietà mediante il predicato `subPropertyOf` di RDFS.

Per i linguaggi RDF e RDFS è stata definita una semantica precisa che specifica in maniera univoca la natura e il comportamento di ciascun elemento dei loro vocabolari. La semantica di RDF stabilisce, per esempio, che ogni oggetto di una tripla che usa il predicato `rdf:type` è una classe. La semantica di RDFS definisce invece, per esempio, la proprietà di *ereditarietà* in base alla quale, se si stabilisce che i compositori sono artisti e che gli artisti sono persone, si deduce che i compositori sono

persone; e se le persone possiedono come proprietà una città e una data di nascita, anche i direttori d'orchestra hanno queste proprietà. Questi meccanismi di inferenza vengono chiamati *entailments*.

Una maggior ricchezza semantica rispetto a RDFS è offerta dal *Web Ontology Language* (OWL), nelle sue versioni *OWL Lite*, *OWL DL* e *OWL Full*. Ciascuna di queste versioni aggiunge altri costrutti a quelli di RDFS, via via più complessi e generali passando dalla versione Lite alla versione Full. Questi costrutti consentono di definire metadati più evoluti di quelli visti finora, tra cui: relazioni particolari tra classi (per esempio la disgiunzione); cardinalità di partecipazione alle proprietà (simili alle cardinalità del modello Entità-Relazione); caratteristiche speciali delle proprietà stesse (per esempio simmetria e transitività). Le varie versioni dispongono di tecniche di entailment via via più complesse per calcolare, a partire da un insieme di triple che usano i costrutti del linguaggio, tutte le ulteriori triple che da esse derivano. In particolare: OWL Lite pone delle restrizioni sui costrutti OWL utilizzabili che in pratica garantiscono che queste inferenze possono essere calcolate efficientemente; OWL DL estende OWL Lite garantendo solo che tutte le inferenze possono essere calcolate in un tempo finito; infine OWL Full fornisce il massimo dell'espressività senza però garantire la computabilità completa degli entailment.

15.3 Il linguaggio SPARQL

SPARQL (*Simple Protocol and RDF Query Language*) è un linguaggio di interrogazione con caratteristiche molto simili a SQL, proposto dal W3C per interrogare dati descritti in RDF; dal nome del linguaggio si intuisce che lo standard include un protocollo per interrogare una o più istanze RDF (che in questo contesto si chiamano anche *endpoint SPARQL*), inviando loro una richiesta in SPARQL e ricevendo il risultato in un formato XML. SPARQL si adatta al contesto d'uso tipico di RDF, in cui non è dato per scontato che chi interroga conosca le caratteristiche di un grafo RDF presente presso un endpoint; per questo motivo, alle classiche forme di interrogazione (introdotte dalla clausola `SELECT`) si affiancano altre forme: `DESCRIBE`, per ottenere una descrizione delle risorse presenti presso un endpoint; `ASK`, per sapere se specifici termini sono disponibili nell'endpoint; `COSTRUCT`, per costruire un nuovo grafo RDF a partire da un'interrogazione.

La versione 1.0 del linguaggio è una raccomandazione W3C dal 2008; successivamente, nel marzo 2013, il W3C ha approvato la versione 1.1 del linguaggio, che ne aumenta notevolmente il potere espressivo; i sistemi per la gestione di interrogazioni SPARQL stanno progressivamente adeguandosi alle caratteristiche di SPARQL 1.1. La descrizione del linguaggio tiene conto dell'evoluzione del linguaggio: la prima parte descrive le caratteristiche di SPARQL 1.0, la seconda parte prende in esame le estensioni di SPARQL 1.1.

15.3.1 Triple e Graph Patterns

Sintatticamente, una query SPARQL 1.0 è composta da cinque parti.

1. La definizione opzionale `PREFIX`, che consente di introdurre uno o più vocabolari utilizzabili nell'interrogazione.

2. La definizione del risultato prodotto dalla query, che è una delle quattro clausole `SELECT`, `DESCRIBE`, `ASK` e `CONSTRUCT`.
3. La definizione opzionale degli endpoint consultati, tramite le clausole `FROM` e `FROM NAMED`.
4. La definizione della parte centrale della query, introdotta dalla clausola `WHERE`, che come vedremo include vari triple patterns.
5. La definizione opzionale di modificatori, che includono i costrutti `ORDER BY`, `LIMIT` e `OFFSET`.

È interessante notare che le clausole `FROM` e `FROM NAMED` consentono di far riferimento a vari endpoint SPARQL; la clausola `FROM` indica semplicemente una IRI da cui leggere i dati, la clausola `FROM NAMED` indica una URI in cui è presente un grafo RDF che viene anche associato a un nome. A valle dell'esecuzione di varie istruzioni `FROM` e `FROM NAMED`, i grafi RDF referenziati dalle URI vengono logicamente assemblati, costruendo un *grafo di default*, che non è associato ad alcun nome, e vari *grafi nominati*, cioè grafi cui è associato un nome, utilizzabili nella query.

La parte centrale di un'interrogazione SPARQL, introdotta dalla clausola `WHERE`, consente di esprimere condizioni di pattern matching tra la query stessa e il grafo RDF su cui la query opera. Elemento centrale del pattern matching è un triple pattern, che sintatticamente è una tripla (come in RDF) nelle cui posizioni è possibile far comparire, in aggiunta a IRI, letterali e blank nodes, anche *variabili*. Le variabili in SPARQL sono sintatticamente riconoscibili in quanto introdotte dal simbolo `?`. Per esempio, un triple pattern è:

```
< ?o1 o:HaPersonaggio 'Sigfrido' > .
```

Questo triple pattern, una volta valutato sull'istanza RDF rappresentata in Figura 15.3, fa sì che la variabile `?o1` venga legata ai blank node `_:Sigfrido` e `_:Crepuscolo`.

Vari triple pattern possono essere combinati assieme, formando un graph pattern; vediamone l'uso nel contesto di una query SPARQL sintatticamente completa. La query Q1, illustrata in Figura 15.4, esprime l'interrogazione *estrai i nomi delle opere che hanno Wotan come personaggio*. Essa usa il vocabolario `opera`, disponibile presso `<http://www.polimi.it/ceri/opera>`, i cui termini devono essere introdotti dal prefisso `o:`. La query si riferisce all'istanza RDF letta dalla IRI `<http://www.polimi.it/ceri/ring.rdf>` e proietta nel risultato tutti i match relativi alla variabile `?t`, introdotta nella clausola `WHERE`. Nella clausola compare un graph pattern costituito da due triple pattern, legati dalla variabile `?o`. Nel primo triple pattern si richiede che il predicato sia `o:haPersonaggio` e che il personaggio sia il letterale `'Wotan'`; in questo modo, la variabile `?o` si lega ai blank nodes `_:OroDelReno` e `_:Valchiria`. A partire da questo legame, la variabile `?t` viene poi legata ai letterali `'Oro Del Reno'` e `'Valchiria'`, e questi valori sono mostrati nel risultato.

Si noti come, nei graph pattern, ogni letterale corrisponda a una selezione con predicato di uguaglianza, mentre l'uso della stessa variabile in due triple pattern corrisponde a una condizione di join; per questo motivo, i graph pattern sono equivalenti a espressioni algebriche congiuntive che utilizzano selezione e join. Si noti inoltre che le variabili relative a blank nodes potrebbero essere *tipizzate*, cioè potrebbero prendere

```
PREFIX o: <http://www.polimi.it/ceri/opera>
SELECT ?t
FROM <http://www.polimi.it/ceri/ring.rdf>
WHERE {
     ?o o:haPersonaggio 'Wotan' .
     ?o o:haTitolo ?t  .
}
```

?t
'Oro del Reno'
'Valchiria'

Figura 15.4 Interrogazione Q1 e suo risultato.

la forma `_:v` a indicare che nelle prime posizioni dei due triple pattern ci si aspetta lo stesso blank node; la forma indicata nella query è equivalente e preferibile per la sua maggior semplicità.

Vediamo ora query con graph pattern più complessi. La query Q2, illustrata in Figura 15.5, esprime l'interrogazione *estrai i titoli delle opere e i nomi dei personaggi che hanno gli stessi personaggi che compaiono in Valchiria.* Si noti che la query così formulata estrarrebbe anche tutti i personaggi di Valchiria; per evitarlo va aggiunto che le opere estratte siano *diverse da Valchiria*. La query ha quattro variabili, `?o1`, `?o2`, `?p`, `?t`; il graph pattern impone che: `?o2` si leghi all'opera intitolata Valchiria; `?o1` abbia un titolo (ergo è un blank node corrispondente a un'opera) e che sia diverso dall'opera cui è legata `?o2`; `?o1` e `?o2` abbiano lo stesso personaggio `?p`. Il risultato mostra i legami che vengono costruiti per le variabili `?t` e `?p` dal pattern matching con l'istanza RDF mostrata in Figura 15.3.

La query Q3, rappresentata in Figura 15.6, è una variazione sul tema; in questo caso si cercano tutte le *coppie di opere che hanno un personaggio in comune*; nel graph pattern viene meno la condizione di selezione sul titolo di Valchiria. Il risultato estrae triple di legami per le variabili `?t1`, `?t2` e `?p`; si noti che ogni coppia di opere compare due volte in quanto si lega sia alla variabile `?o1` sia alla variabile `?o2`.

```
PREFIX o: <http://www.polimi.it/ceri/opera>
SELECT ?t, ?p
FROM <http://www.polimi.it/ceri/ring.rdf>
WHERE {
     ?o1 o:haPersonaggio ?p .
     ?o2 o:haPersonaggio ?p .
     ?o1 != ?o2 .
     ?o1 o:haTitolo ?t .
     ?o2 o:haTitolo 'Valchiria' .
}
```

?t	?p
'Oro del Reno'	'Wotan'
'Sigfrido'	'Brunilde'
'Crepuscolo degli Dei'	'Brunilde'

Figura 15.5 Interrogazione Q2 e suo risultato.

```
PREFIX o: <http://www.polimi.it/ceri/opera>
SELECT ?t1, ?t2, ?p
FROM <http://www.polimi.it/ceri/ring.rdf>
WHERE {
      ?o1 o:haPersonaggio ?p .
      ?o2 o:haPersonaggio ?p .
      ?o1 != o2 .
      ?o1 o:haTitolo ?t1 .
      ?o2 o:haTitolo ?t2 .
}
```

?t1	?t2	?p
'Oro del Reno'	' Valchiria'	' Wotan'
'Valchiria'	'Oro del Reno'	'Wotan'
'Valchiria'	'Sigfrido'	'Brunilde'
'Sigfrido'	'Valchiria'	'Brunilde'
'Valchiria'	"Crepuscolo degli Dei'	'Brunilde'
'Crepuscolo degli Dei'	'Valchiria'	'Brunilde'
'Sigfrido'	'Crepuscolo degli Dei'	'Brunilde'
'Crepuscolo deli Dei'	'Sigfrido'	'Brunilde'
'Sigfrido'	'Crepuscolo degi Dei'	'Sigfrido'
'Crepuscolo degli Dei'	'Sigfrido'	'Sigfrido'

Figura 15.6 Interrogazione Q3 e suo risultato.

15.3.2 Clausole FILTER, UNION, OPTIONAL

I graph pattern possono essere estesi utilizzando alcune clausole opzionali. La clausola `FILTER` applica alle tuple di binding che risultano dalla valutazione della clausola `WHERE` un predicato; solo le tuple che soddisfano il predicato fanno parte del risultato finale. Per esempio, riprendendo la query Q2 , la query Q4 di Figura 15.7 filtra le *opere la cui durata è superiore a 300 minuti*. In tal caso, è necessario aggiungere al graph pattern la variabile `?d`, che esprime la durata in minuti (un numero intero), e poi filtrare in base alla condizione `?d > 300`.

```
PREFIX o: <http://www.polimi.it/ceri/opera>
SELECT ?t, ?p
FROM <http://www.polimi.it/ceri/ring.rdf>
WHERE {
      ?o1 o:haPersonaggio ?p .
      ?o2 o:haPersonaggio ?p .
      ?o1 != ?o2 .
      ?o1 o:haTitolo ?t.
      ?o2 o:haTitolo 'Valchiria' .
      ?o2 o:haDurata ?d .
      FILTER (?d > 300)
}
```

?t	?p
Sigfrido	Brunilde
Crepuscolo degli Dei	Brunilde

Figura 15.7 Interrogazione Q4 e suo risultato.

```
PREFIX o: <http://www.polimi.it/ceri/opera>
SELECT ?t
FROM <http://www.polimi.it/ceri/ring.rdf>
WHERE {
   { ?o o:haTitolo ?t .
     ?o o:haPersonaggio ?p .
     FILTER (?p = 'Wotan') .}
UNION
   { ?o o:haTitolo ?t .
     ?o o:dura ?m .
     FILTER (?m > 350). }
}
```

?t
'Oro del Reno'
'Valchiria'
'Caduta degli Dei'

Figura 15.8 Interrogazione Q5 e suo risultato.

Gli operatori utilizzabili all'interno di una clausola `FILTER` sono i comparatori e gli operatori aritmetici per costruire predicati semplici, i connettivi logici AND (`&&`), OR (`||`), NOT (`!`) tra predicati, e alcuni predicati unari specifici di SPARQL: `bound(?x)` che è vero se `?x` è legata a un valore, `isIRI(?x)`, `isLiteral(?x)` e `isBlank(?x)`, che sono veri se `?x` è legata rispettivamente a una IRI, un letterale o un blank node. Quindi, i filtri consentono di esprimere predicati di selezione e di join con comparatori arbitrari. Per esempio, un filtro `?t1 < ?t2` applicato ai bindings della query Q3 consente di evitare di ripetere due volte i titoli delle opere nel risultato, in quanto tutte le stringhe sono ordinate lessico graficamente.

Per completare l'analogia con i linguaggi algebrici, SPARQL prevede all'interno della clausola `WHERE` la clausola `UNION`, in cui è possibile fare l'unione dei binding prodotti separatamente da due graph patterns. Si veda per esempio la query Q5, rappresentata in Figura 15.8, in cui si estraggono *le opere che hanno Wotan come personaggio oppure durano più di 350 minuti*.

Vediamo ora la clausola `OPTIONAL`, che introduce una caratteristica tipica di SPARQL.[3] In SPARQL i dati non sono necessariamente conformi a uno schema, conseguentemente certe informazioni potrebbero essere assenti; la clausola `OPTIONAL` consente di definire ulteriori graph patterns rispetto a quelli della clausola `WHERE` senza vincolare le variabili che vengono ivi introdotte ad assumere un valore preciso; nel caso un valore manchi, tali variabili vengono legate al valore `NULL`. Per esempio, la query Q6, rappresentata in Figura 15.9, estrae *titolo, durata, numero di atti e presenza di prologo di tutte le opere*. Tenuto conto che non tutte le opere sono suddivise in atti o hanno un prologo, le ultime due variabili vengono introdotte nella clausola `OPTIONAL`; il risultato prevede per queste variabili alcuni valori `NULL`.

[3] Il lettore è invitato a confrontare il significato della clausola `OPTIONAL` con quello di `LEFT` e `RIGHT JOIN` nella SQL.

```
PREFIX o: <http://www.polimi.it/ceri/opera>
SELECT ?t, ?d, ?a, ?p
FROM <http://www.polimi.it/ceri/ring.rdf>
WHERE {
     ?o o:haTitolo ?t .
     ?o o:haDurata ?d .
     OPTIONAL {
           ?o o:haAtti ?a .
           ?o o:haPrologo ?p.  }
}
```

?t	?d	?a	?p
'Oro del Reno'	150	NULL	NULL
'Valchiria'	310	3	NULL
'Sigfrido'	320	3	NULL
'Crepuscolo degli Dei'	360	3	'si'

Figura 15.9 Interrogazione Q6 e suo risultato.

15.3.3 Negazione e modificatori

Abbiamo visto che SPARQL 1.0 consente di esprimere le operazioni algebriche di selezione, proiezione, join e unione; ha quindi un potere espressivo che include l'algebra relazionale senza differenza. Il linguaggio non prevede esplicitamente la differenza, però è possibile ottenere il suo potere espressivo tramite meccanismi un po' tortuosi. Innanzitutto, ricordiamo che la negazione non è riconducibile all'uso del comparatore di disuguaglianza. Per esempio, consideriamo la query Q7, descritta in Figura 15.10. La query estrae tutte le opere, in quanto il suo significato è *estrai il titolo di tutte le opere che hanno qualche personaggio diverso da Sigfrido*. Quindi l'opera `'Crepuscolo degli Dei'` è presente nel risultato, pur avendo Sigfrido tra i suoi personaggi.

La query *estrai le opere in cui non è presente il personaggio Sigfrido* non è invece esprimibile tramite le sole clausole `WHERE` e `FILTER`. Per esprimerla, si ricorre a un

```
PREFIX o: <http://www.polimi.it/ceri/opera>
SELECT ?t
FROM <http://www.polimi.it/ceri/ring.rdf>
WHERE {
     ?o o:haTitolo ?t
     ?o o:haPersonaggio ?p
     FILTER {
           ?p != 'Sigfrido'}
}
```

?t
'Oro del Reno'
'Valchiria'
'Sigfrido'
'Caduta degli Dei'

Figura 15.10 Interrogazione Q7 e suo risultato.

```
PREFIX o: <http://www.polimi.it/ceri/opera>
SELECT ?t
FROM <http://www.polimi.it/ceri/ring.rdf>
WHERE {
     ?o o:haTitolo ?t
     OPTIONAL {
           ?o haPersonaggio ?p .
           FILTER (?p = 'Sigfrido'}}
     FILTER (! BOUND(?p))
}
```

?t
'Oro del Reno'
'Valchiria'

Figura 15.11 Interrogazione Q8 e suo risultato.

artificio illustrato dalla query Q8 in Figura 15.11; in primo luogo, si estraggono tutte le opere che hanno Sigfrido presente opzionalmente; in tal caso, la variabile `?p` risulta legata a Sigfrido oppure a `NULL`. Si applica successivamente un filtro che esclude tutti i pattern in cui `?p` è `NULL`, estraendo le due opere in cui Sigfrido non compare.

I modificatori sono introdotti da clausole opzionali che si applicano al risultato prodotto della clausola `WHERE`. Anche la clausola `SELECT` può essere considerata un modificatore, in quanto proietta i legami presenti nel risultato del matching su un sottoinsieme delle variabili. I modificatori sono:

- clausola `ORDER BY`, che ordina i legami del risultato in modo ascendente (clausola `ASC()`, usata come default) oppure discendente (clausola `DESC()`);
- clausola `DISTINCT`, che elimina i duplicati dal risultato, e la sua variante `REDUCED`, che elimina solo duplicati che si presentano consecutivamente nel risultato;
- clausola `LIMIT`, che indica il numero massimo di tuple nel risultato, e la clausola `OFFSET`, che indica il numero di tuple da omettere nel risultato.

15.3.4 Forme CONSTRUCT, ASK e DESCRIBE

Oltre alla `SELECT`, esistono tre altre forme di query. La `CONSTRUCT` consente di costruire un nuovo grafo RDF, con un meccanismo simile a quello delle `VIEW` in SQL. Per esempio, la query illustrata in Figura 15.12 consente di costruire nuove tuple RDF a partire dall'istanza RDF di Figura 15.4. La query estrae legami per le variabili `?t` e `?p`, in particolare le coppie in cui il primo elemento è il nome `'Ring'` e il secondo elemento è un personaggio di una delle quattro opere che lo compongono, eliminando i duplicati. Dopodiché questa forma di query costruisce, per ogni coppia, una tripla RDF il cui predicato è il nuovo termine `haPersonaggio`. Il grafo RDF risultante è illustrato in Figura 15.12; si noti che vengono conservati i duplicati.

La query `ASK` consente di valutare se una query ha un risultato non nullo, cioé se alle variabili della query viene associata almeno una tupla di binding. Per esempio, la prima delle due query `ASK` di Figura 15.13 ha valore `false`, mentre la seconda ha valore `true`.

```
PREFIX o: <http://www.polimi.it/ceri/opera>
CONSTRUCT  { ?t haPersonaggio ?p  . }
FROM <http://www.polimi.it/ceri/ring.rdf>
WHERE {
      ?r o:haTitolo ?t .
      ?r o:comprende ?o .
      ?o o:haPersonaggio ?p .
}

 'Ring' haPersonaggio 'Wotan'  .
 'Ring' haPersonaggio 'Brunilde'  .
 'Ring' haPersonaggio 'Wotan'  .
 'Ring' haPersonaggio 'Sieglinde'  .
 'Ring' haPersonaggio 'Brunilde'  .
 'Ring' haPersonaggio 'Sigfrido'  .
 'Ring' haPersonaggio 'Brunilde'  .
 'Ring' haPersonaggio 'Sigfrido'  .
 'Ring' haPersonaggio 'Waltraute'  .
```

Figura 15.12 Query `CONSTRUCT` che costruisce nuove triple RDF.

```
PREFIX o: <http://www.polimi.it/ceri/opera>
ASK
FROM <http://www.polimi.it/ceri/ring.rdf>
WHERE {
      ?o o:haTitolo 'Walkiria'  .
      ?o o:haPrologo ?p .
}

PREFIX o: <http://www.polimi.it/ceri/opera>
ASK
FROM <http://www.polimi.it/ceri/ring.rdf>
WHERE {
      ?o o:haTitolo 'Crespuscolo degli Dei'  .
      ?r o:haPrologo ?p .
}
```

Figura 15.13 Due query `ASK`, la prima ha per risultato `false` e la seconda `true`.

```
PREFIX o: <http://www.polimi.it/ceri/opera>
DESCRIBE ?o
FROM <http://www.polimi.it/ceri/ring.rdf>
WHERE {
      ?o o:haTitolo 'Walkiria'  .
}
```

Figura 15.14 Query `DESCRIBE` relative alle risorse che soddisfano una query.

Infine, la query `DESCRIBE` consente di estrarre tutta l'informazione conosciuta relativamente alle risorse che soddisfano una query; la forma della risposta è definita dall'endpoint SPARQL. Nell'esempio di Figura 15.14, viene richiesto di descrivere le proprietà relative all'opera che ha per titolo `Valchiria`; la risposta comprenderà le proprietà che si possono intuire a partire dall'istanza RDF di Figura 15.4, tra esse durata, numero di atti e personaggi.

15.3.5 Aggregazione, negazione e sotto-query in SPARQL 1.1

La definizione di SPARQL 1.0 difettava di alcune caratteristiche tipiche dei linguaggi di query (e in particolare di SQL); tali aspetti sono stati recentemente introdotti in SPARQL 1.1. In questa sezione, vediamo l'aggregazione, la negazione e le sotto-interrogazioni.

L'aggregazione introduce le clausole GROUP BY e HAVING, descritte nelle query Q9 e Q10. In particolare, la query Q9 illustrata in Figura 15.15 calcola la durata totale delle quattro opere che formano il Ring, con un raggruppamento (per opera) e l'uso della funzione aggregata SUM. Come in SQL, sono disponibili le funzioni aggregate COUNT, SUM, MIN, MAX e AVG, cui si aggiungono la SAMPLE (per estrarre un valore arbitrario da un insieme di valori) e GROUP_CONCAT (per concatenare stringhe).

La query Q10, illustrata in Figura 15.16, estrae i personaggi che compaiono in due o più opere; in questo caso il raggruppamento viene fatto in base ai personaggi, vengono poi contate le opere, e selezionate (tramite la clausola HAVING) quei personaggi che hanno un contatore maggiore o uguale a 2.

```
PREFIX o: <http://www.polimi.it/ceri/opera>
SELECT ((SUM ?d) AS ?DurataDelRing)
FROM <http://www.polimi.it/ceri/ring.rdf>
WHERE {
     ?r o:haTitolo 'Ring' .
     ?r o:comprende ?o .
     ?o o:haDurata ?d .
}
GROUP BY ?r
```

?DurataDelRing
1140

Figura 15.15 Interrogazione Q9 e suo risultato.

```
PREFIX o: <http://www.polimi.it/ceri/opera>
SELECT (?p, (COUNT ?o) AS ?n)
FROM <http://www.polimi.it/ceri/ring.rdf>
WHERE {
     ?r o:haTitolo 'Ring' .
     ?r o:comprende ?o .
     ?o o:haPersonaggio ?p .
}
GROUP BY ?p
HAVING ( COUNT(?o) > 2 )
```

?p	?n
'Wotan'	2
'Brunilde'	3
'Sigfrido'	2

Figura 15.16 Interrogazione Q10 e suo risultato.

```
PREFIX o: <http://www.polimi.it/ceri/opera>
SELECT ?t
FROM <http://www.polimi.it/ceri/ring.rdf>
WHERE {
      ?o o:haPersonaggio ?p .
      ?o o:haTitolo ?t .
      MINUS { ?o haPersonaggio 'Sigfrido' .
              ?o haTitolo ?t . }
}

PREFIX o: <http://www.polimi.it/ceri/opera>
SELECT ?t
FROM <http://www.polimi.it/ceri/ring.rdf>
WHERE {
      ?o o:haTitolo ?t .
      FILTER (NOT EXISTS  ?o haPersonaggio 'Sigfrido')
}
```

Figura 15.17 Due modi alternativi per esprimere l'interrogazione Q8 in SPARQL 1.1.

Per quanto riguarda la negazione, SPARQL 1.1 introduce l'operatore binario `MINUS`, che completa la copertura dell'algebra relazionale, e la clausola `NOT EXISTS`, che ricorda le sotto-query di SQL. La query Q8 può essere espressa in modo più naturale in SPARQL 1.1 usando questi due costrutti, come illustrato in Figura 15.17. L'operatore `MINUS` calcola la differenza tra i binding prodotti dalla query principale e i binding prodotti dalla sotto-query introdotta dall'operatore. L'espressione `NOT EXISTS` all'interno della clausola `FILTER` filtra gli elementi del risultato prodotto dalla valutazione della `WHERE` che non hanno matching con il graph pattern presente nella clausola stessa. Si noti che SPARQL 1.1 consente l'uso degli operatori `EXISTS` e `NOT EXISTS` all'interno della clausola `FILTER`.

SPARQL 1.1 consente la costruzione di sotto-query; come in SQL, le sotto-query possono essere arbitrariamente nidificate formando una struttura ad albero, che viene valutata dal basso verso l'alto; solo le variabili che vengono proiettate da una sotto-query di livello inferiore sono visibili nelle query di livello superiore. Per esempio, la query Q11, illustrata in Figura 15.18, calcola quanti sono i personaggi che compaiono

```
PREFIX o: <http://www.polimi.it/ceri/opera>
SELECT ((sum ?p) AS ?n)
FROM <http://www.polimi.it/ceri/ring.rdf>
WHERE
      {  SELECT ?p
         WHERE { ?r o:haTitolo 'Ring' .
                 ?r o:comprende ?o .
                 ?o o:haPersonaggio ?p . }
         GROUP BY ?p
         HAVING ( COUNT(?o) > 2 ) }
```

?n
3

Figura 15.18 Interrogazione Q11 e suo risultato.

più di due volte nell'istanza RDF di Figura 15.3, nidificando due query aggregate (in particolare, inserendo la query Q10 all'interno di una query che ha struttura simile alla query Q9).

15.3.6 Aspetti avanzati di SPARQL 1.1

SPARQL 1.1 ha introdotto anche alcuni aspetti che estendono in modo significativo e caratteristico il potere espressivo del linguaggio; vediamo brevemente le nozioni di *entailment* e di property path. Nella prossima sezione verrà poi descritto l'operatore `SERVICE`, che consente forme sofisticate di interoperabilità.

Gli *entailment regimes* sono particolari contesti di valutazione che interpretano le triple come appartenenti a istanze RDFS oppure in OWL, e conseguentemente estendono la valutazione delle query includendo nel risultato tutte le triple che sono implicate dalla particolare istanza in base agli entailments definiti per il linguaggio che si sta adottando. Si supponga per esempio che l'istanza RDF in Figura 15.3 sia estesa che le seguenti definizioni RDFS:

```
_:Wagner a o:Compositore.
o:Compositore rdfs:subClassOf o:Artista.
o:Artista rdfs:subClassOf o:Persona.
```

Un'interpretazione della query Q12 in Figura 15.19 che si basa sul semplice pattern matching tra la query stessa e il grafo RDF su cui la query opera restituisce l'insieme vuoto, perché non esiste nessuna tripla nell'istanza RDF che dichiara l'appartenenza di una risorsa alla classe delle persone. Questo tipo di interpretazione viene chiamata *simple entailment*. Viceversa, l'interpretazione basata sull'entailment RDFS restituisce `Richard Wagner` perché le regole di inferenza definite per questo linguaggio estendono l'istanza RDF con due nuove triple:

```
_:Wagner a o:Artista.
_:Wagner a o:Persona.
```

Permettono cioè di inferire, per la proprietà di ereditarietà di RDFS, che se tutti i compositori sono artisti e tutti gli artisti sono delle persone, allora anche Wagner è una persona. Per una definizione accurata degli entailment regimes presenti nello standard SPARQL 1.1 si rimanda il lettore agli specifici documenti del W3C, descritti nella parte bibliografica del capitolo.

```
prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX o: <http://www.polimi.it/ceri/opera>
SELECT ?n
FROM <http://www.polimi.it/ceri/ring.rdf>
WHERE
      {  ?p o:nome ?n .
         ?p rdf:type o:Persona . }
```

Figura 15.19 Interrogazione Q12.

```
SELECT ?n
WHERE
     {  ?x haNome ?n .
        ?x (amico)* ?y .
        ?y haNome 'Giorgio' . }
```

Figura 15.20 Interrogazione Q13.

Vediamo ora i *property path*, che estendono SPARQL nella direzione della gestione della ricorsione, che abbiamo già incontrato nei linguaggi logici e a oggetti. Supponendo che lo schema RDF comprenda un predicato `amico`, che collega fra loro blank nodes corrispondenti a persone, e supponendo che tali blank nodes abbiano una relazione `haNome` con un letterale che indica per ciascuna persona il suo nome, è possibile in SPARQL 1.1 scrivere l'interrogazione rappresentata in Figura 15.20, che percorre transitivamente la relazione `amico` a partire dalla persona di nome Giorgio tramite un property path; sintatticamente, il path utilizza la stella di Kleene per rappresentare la ricorsione. SPARQL 1.1 consente varie altre opzioni per descrivere path, molto simili al linguaggio XPath per XML (si veda la sezione 14.3.1).

15.3.7 SPARQL e interoperabilità

Fino a ora, abbiamo valutato le caratteristiche di SPARQL come linguaggio di interrogazione, ma in realtà SPARQL ha un importante ruolo anche come strumento di integrazione di differenti fonti dati. Innanzitutto, notiamo come sia semplice, tramite istruzioni `PREFIX`, importare definizioni di standard, vocabolari e ontologie; per esempio, in Figura 15.21 si mostra come sia possibile importare in un'istanza RDF (o in una query SPARQL) elementi sintattici da RDF e RDFS, le categorie e proprietà di DBpedia, e i termini delle ontologie Skos, Geo e Yago.

Mentre le clausole `PREFIX` consentono di importare definizioni, le clausole `FROM` e `FROM NAMED`, già discusse nel Paragrafo 15.3.1, consentono di importare instanze RDF. Supponiamo uno scenario in cui siano presenti tre istanze RDF, pubblicate presso i dipartimenti degli autori di questo libro, ciascuna contenente i rapporti di ricerca pubblicati dal dipartimento; per esempio, la Figura 15.22 illustra un frammento dell'istanza RDF pubblicata al DEIB del Politecnico di Milano, in cui sono presenti tre ricercatori e due report.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX dbcat: <http://dbpedia.org/resource/Category:>
PREFIX dbpprop: <http://dbpedia.org/property/>
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
PREFIX geo:  <http://www.w3.org/2003/01/geo/wgs84_pos#>
PREFIX yago: <http://dbpedia.org/class/yago/>
```

Figura 15.21 Tipico uso dei `Prefix` nelle istanze RDF oppure nelle query SPARQL .

```
_:p1 haNome 'Stefano Ceri' ;
     haWebSite 'http://dbgroup.como.polimi.it/ceri/' .
_:p2 haNome 'Piero Fraternali' ;
     haWebSite 'http://home.deib.polimi.it/fraterna/' .
_:p3 haNome 'Paolo Atzeni'
     haWebSite 'www.dia.uniroma3.it/~atzeni/' .
_:r1 scrittoDa \_:p1,
                _:p3;
     haTitolo 'Genomic Data Management' ;
     haData '2013 August 31' ^^ rdf.date .
_:r1 scrittoDa \_:p1,
                _:p2;
     haTitolo 'Web Data Management' ;
     haData '2013 October 1' ^^ rdf.date .
```

Figura 15.22 Frammento di istanza RDF pubblicata presso il DEIB del Politecnico di Milano che contiene informazioni sui rapporti di ricerca.

La query Q14, illustrata in Figura 15.23, importa i dati dalle tre istanze RDF, tramite la clausola `FROM NAMED` che associa a ogni istanza RDF importata un nome (coincidente con il suo IRI). La clausola `GRAPH` all'interno della clausola `WHERE`, definita in SPARQL 1.0, consente di valutare la query su una particolare istanza RDF, oppure di determinare le istanze RDF che producono ciascun binding. Per esempio, tramite la clausola `GRAPH` nella query Q14, la variabile `?g` è legata al nome dell'istanza RDF che produce i binding presenti nel risultato.

In SPARQL 1.1, all'interno della clausola `WHERE`, la keyword `SERVICE` indica che una sotto-query va rivolta a uno SPARQL endpoint remoto. Nella query Q15, tramite la query `SERVICE`, viene interrogato l'endpoint presso l'Università Roma Tre, estraendo i nomi degli autori del report intitolato `'Data Mapping'`; tra di essi assumiamo che vi sia `'Paolo Atzeni'`, e corrispondentemente viene legata la variabile `?n`, esportata dalla sotto-query. Dopodiché, dato che tale valore compare anche come autore del report `'Genomic Data Management'`, memorizzato

```
SELECT ?g, ?n
FROM NAMED <http://www.deib.polimi.it/reports>
FROM NAMED <http://www.dia.uniroma3.it/reports>
FROM NAMED <http://www.ingegneria.unibg.it/reports >
WHERE  {
    GRAPH ?g
    { ?r scrittoDa ?p .
      ?p haNome ?n .
      ?r haTitolo 'Genomic Data Management' . }
}
```

?g	?n
'http://www.deib.polimi.it/reports'	'Paolo Atzeni'
'http://www.deib.polimi.it/reports'	'Stefano Ceri'

Figura 15.23 Interrogazione Q14 e suo risultato.

```
SELECT ?n
FROM NAMED <http://www.deib.polimi.it/reports>
WHERE {
      ?r scrittoDa ?p .
      ?p haNome ?n .
      ?r haTitolo 'Genomic Data Management' .
      SERVICE <http://www.dia.uniroma3.it/reports>
            {  ?p1 haNome ?n .
               ?r scrittoDa ?p1 .
               ?r haTitolo 'Data Mapping' . }
}
```

?n
'Paolo Atzeni'

Figura 15.24 Interrogazione Q15 e suo risultato.

presso l'endpoint del DEIB al Politecnico di Milano, questo nome è estratto dalla query; la query e il suo risultato sono illustrati in Figura 15.24.

15.4 Linked e open data

Il termine *linked data* si riferisce a un insieme di risorse disponibili sul Web, descritte tramite triple RDF e collegate fra loro tramite riferimenti (link). Pubblicare risorse sotto forma di linked data vuol dire aderire a una *buona pratica* di pubblicazione, in cui è sufficiente che i dati siano processabili automaticamente e che il loro significato sia ben definito, usando le IRI per identificare le risorse, lo standard HTTP per trasferire dati tra diversi nodi del Web, e gli standard RDF, RDFS e OWL per descrivere le triple. I linked data, elencati sul sito `www.linkeddata.org`, sono in continua crescita. Sono disponibili varie rappresentazioni dei linked data sotto forma di rete, in cui i nodi rappresentano le varie risorse disponibili sul Web e gli archi rappresentano i collegamenti tra le risorse; lo spessore degli archi è proporzionale al numero di legami presenti. Si veda in Figura 15.25 una rappresentazione dei linked data pubblicata nel 2011.

Le principali risorse presenti nei linked data possono essere classificate in base al loro dominio di riferimento.

- Innanzitutto, fanno parte dei linked data alcune *risorse interdominio*, tra cui DBpedia, un dataset periodicamente estratto in modo automatico dai nodi di Wikipedia; altre raccolte interdominio sono Freebase, Yago e OpenCyc.
- I *dati di tipo geografico* includono i nodi Geonames (vari milioni di nomi di luoghi) e LinkedGeoData, derivato dal progetto OpenStreetMap.
- I *dati relativi ai media* sono stati sviluppati a partire dalla pubblicazione di programmi e musiche della BBC inglese; l'esempio della BBC è poi stato seguito da altre organizzazioni, tra cui il New York Times e la Reuters.
- I *dati relativi all'istruzione e alla ricerca* includono per esempio i dati messi a disposizione dall'American Library of Congress e dall'Open Library. Tra i primi esempi di linked data ricordiamo DBLP, il sito dedicato alle pubblicazioni di ri-

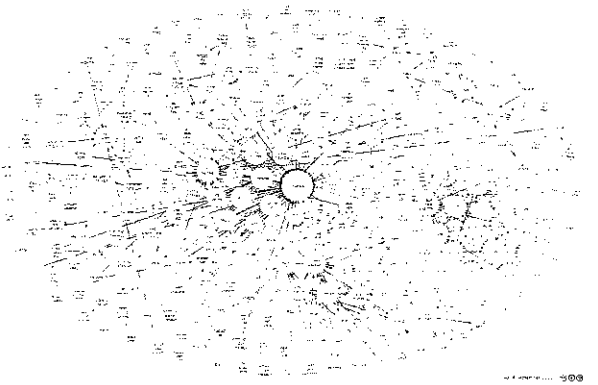

Figura 15.25 Rappresentazione dei linked data al settembre 2011, fornita da Richard Cyganiak e Anja Jentzsch.

cerca inizialmente dei due settori dei database e della programmazione logica, poi esteso a coprire tutta la ricerca informatica.

- I *dati relativi alle scienze della vita*, attualmente indispensabili allo sviluppo della genetica, e che includono la Gene Ontology (che descrive i geni), UniProt (che descrive le proteine), Kegg (l'enciclopedia di Kyoto dei geni e dei genomi) e PubMed (che descrive le pubblicazioni scientifiche relative alla medicina).
- I *dati relativi al commercio*, tra cui l'ontologia GoodRelations che descrive i vari aspetti del commercio elettronico ed è stata adottata per la descrizione delle proprie offerte da molti siti di commercio elettronico.

Quest'ultimo dominio ci offre lo spunto per una considerazione generale: una delle principali caratteristiche dei linked data è la loro capacità di descrivere siti e pagine Web in modo semantico, consentendo ai motori di ricerca una migliore capacità di selezione e descrizione delle risorse sul Web.

Un ulteriore dominio importante dei linked data è costituito dai dati amministrativi; nel loro caso, si parla comunemente di *open data*, per sottolineare il fatto che questi dati sono di dominio pubblico e la loro conoscenza aumenta la *trasparenza* delle amministrazioni nei riguardi dei cittadini. Tra i primi a pubblicare open data, gli Stati Uniti (www.data.gov) nel 2009, sotto la spinta dell'amministrazione Obama, e la Gran Bretagna (www.data.gov.uk). Il fenomeno della pubblicazione dei dati aperti, talvolta non connessi alla rete dei linked data, è in continua evoluzione, e citiamo per esempio i siti dei comuni di Milano[4] e di Roma.[5]

[4] www.dati.comune.milano.it/
[5] www.dati.comune.roma.it/

Note bibliografiche

Il termine *Semantic Web* è stato introdotto nel 2001 da Tim Berners Lee per designare un Web di dati processabili in modo automatico dai computer [94]; nel 2006, riconoscendo che alcuni dei problemi più ardui del Semantic Web rimanevano in gran parte irrisolti [141], la comunità del Semantic Web ha iniziato a ridirigere i propri sforzi verso la definizione di vocabolari e ontologie compatibili e comunicanti fra loro, introducendo i linked data e gli open data.

La definizione dei principali standard per il Semantic Web è opera del *World Wide Web Consortium* (W3C), fondato nel 1994 e che vede tra i suoi membri numerose organizzazioni: il processo di standardizzazione attuato dal W3C passa dalla definizione di *Working Draft* (WD), *Candidate Recommendation* (CR), *Proposed Recommendation* (PR) e *Recommendation* (REC). Questo capitolo è basato sulle REC relative a RDF (febbraio 1999),[6] RDFS (febbraio 2004),[7] OWL (febbraio 2004),[8] SPARQL 1.0 (gennaio 2008)[9] e SPARQL 1.1 (marzo 2013).[10]

La definizione di SPARQL 1.1 pone vari problemi di natura teorica tuttora in corso di esplorazione, tra cui la realizzazione efficiente degli entailment regimes e l'uso della clausola SERVICES nel contesto di query federate [100]. In questo capitolo non abbiamo preso in considerazione alcuni aspetti importanti dell'attuale evoluzione di SPARQL, tra cui la gestione degli stream di dati [111] e l'integrazione nel linguaggio SPARQL 1.1 degli update, che è divenuta una REC nel marzo 2013.[11]

Vari libri di testo descrivono il Semantic Web, tra di essi ricordiamo il testo in italiano di Dellavalle, Celino e Cerizza [110]; una descrizione più estesa di SPARQL si trova anche nel Capitolo [112], che comprende anche le descrizioni di varie applicazioni di SPARQL.

Esercizi

Soluzioni sul sito http://www.ateneonline.it/atzeni

15.1 Descrivere in RDF le seguenti specifiche facendo uso della notazione N3:

- Roma Tre è un'Università.
- Politecnico di Milano è un'Università.
- La Sapienza è un'Università.
- Roma Tre e La Sapienza hanno sede a Roma, Italia.
- Politecnico di Milano ha sede a Milano, Italia.
- Paolo Atzeni e Riccardo Torlone sono professori di Roma Tre.
- Marco Rossi, Anna Bianchi, Livio Bruni sono studenti di Roma Tre.
- Basi di dati è un corso di Roma Tre e del Politecnico.
- Basi di dati per il Web è un corso del Politecnico.
- Paolo Atzeni insegna il corso di Basi di dati.

[6] http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/
[7] http://www.w3.org/TR/rdf-schema/
[8] http://www.w3.org/TR/owl-features/
[9] http://www.w3.org/TR/rdf-sparql-query/
[10] http://www.w3.org/TR/sparql11-query/
[11] http://www.w3.org/TR/sparql11-update/

- Anna Bianchi ha sostenuto l'esame di Basi di dati.
- Stefano Ceri insegna il corso di Basi di dati e di Basi di dati per il Web.

Esprimere le seguenti interrogazioni SPARQL e indicare il loro risultato.

- I docenti che insegnano Informatica a Roma Tre.
- Gli studenti che hanno sostenuto un esame del professore Nino Vani.
- Tutti i corsi che sono offerti a Roma Tre oppure al Politecnico.
- I corsi offerti sia a Roma Tre, sia alla Sapienza e sia al Politecnico.
- I corsi di Roma Tre che non sono offerti a La Sapienza.
- Le città nelle quali hanno sede più di un'Università.

15.2 Descrivere le seguenti affermazioni in RDF.

- Unicredit è una banca.
- Ghizzoni lavora in Unicredit.
- BNL è una banca.
- Roma12 è un'agenzia di Unicredit.
- Roma12 ha sede in Via Giotto, 15 a Roma.
- Franco Dotti e Laura Pini sono impiegati presso l'agenzia Roma12.
- Marco Rossi, Anna Bianchi e Livio Bruni hanno un conto presso l'agenzia Roma12.
- Il conto di Anna Bianchi ha un saldo di 1200 Euro.

Esprimere le seguenti interrogazioni SPARQL e indicare il loro risultato.

- Nomi e indirizzi delle agenzie Unicredit di Roma.
- Nome e saldo dei correntisti della banca Unicredit.
- Gli impiegati di un'agenzia che non hanno un conto presso la stessa agenzia.
- Le città nelle quali hanno sede agenzie di banche diverse.

15.3 Si consideri la seguente istanza RDF:

```
@prefix dc: <http://purl.org/dc/elements/1.1/> .
@prefix : <http://example.org/book/> .
@prefix ns: <http://example.org/ns#> .
_:l1 dc:titolo "SPARQL Tutorial" .
_:l1 dc:autore DellaValle
_:l1 ns:prezzo 42 .
_:l2 dc:titolo "The Semantic Web" .
_:l2 ns:prezzo 23 .
_l2: dc:autore DellaValle
_l2: dc:autore Cellino
```

Esprimere le seguenti interrogazioni SQL.

- Calcolare il prezzo medio dei libri.
- Estrarre i libri che non sono stati scritti da Cellino.
- Estrarre i libri che sono stati scritti da DellaValle e non da Cellino.
- Costruire un grafo RDF che esprima per ogni autore l'elenco dei libri scritti e il relativo prezzo.
- Chiedere se è vero che DellaValle ha scritto SPARQL Tutorial.

Costruire i risultati delle seguenti query:

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX ns: <http://example.org/ns#>
SELECT ?t ?p
WHERE { ?x dc:titolo ?t .
```

```
        ?x ns:prezzo ?p .
        FILTER (?p < 30 ) }

PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX ns: <http://example.org/ns#>
SELECT ?t ?p
WHERE { ?x dc:titolo ?title .
        OPTIONAL { ?x ns:prezzo ?p .
                   FILTER (?p < 30 ) }
```

15.4 Discutere in maniera sintetica il supporto che il Web semantico può offrire al problema dell'integrazione di informazioni disponibili in rete.

16 Architetture distribuite

Questo capitolo affronta la descrizione delle architetture distribuite per basi di dati. Una prima considerazione da farsi è che, in pratica, la quasi totalità delle applicazioni delle basi di dati si colloca naturalmente in un contesto distribuito, cioè caratterizzato dalla presenza di una rete. Ciò premesso, vediamo innanzitutto di caratterizzare le differenti tipologie di architetture distribuite, per poi affrontare ciascuna di esse separatamente.

L'architettura distribuita più semplice ma anche più diffusa utilizza il paradigma *client-server*; questa architettura è basata sulla tecnologia dei server per la gestione dei dati e rientra nel panorama vastissimo delle architetture distribuite per basi di dati, in quanto la gestione del dialogo con l'utente è affidata a macchine *client* dedicate; una rete collega gli elaboratori che svolgono il ruolo di client agli elaboratori che svolgono il ruolo di server.

Problemi diversi vengono posti dalle *basi di dati distribuite*; questo contesto è caratterizzato dalla presenza di almeno due server che possono anche svolgere transazioni autonomamente, però in qualche caso devono interagire; tale interazione avviene a livelli di progressiva complessità. Vedremo che la maggior difficoltà è posta dal garantire le proprietà "acide" delle transazioni (secondo quanto definito nel Paragrafo 5.6.2), specie quando le transazioni svolgono azioni di scrittura su due o più server. Per gestire queste situazioni, si introduce il protocollo di *commit a due fasi*, che costituisce uno dei protocolli più completi e interessanti nel contesto della distribuzione dei dati, per la naturale interazione di aspetti legati alla distribuzione, all'atomicità e alla persistenza. Vedremo anche un altro tipico protocollo che descrive la *rilevazione dei deadlock* in un contesto distribuito.

Recentemente, è emersa una tecnologia di servizio per la realizzazione di applicazioni distribuite: quella della *replicazione dei dati*. Con questo termine si intende la capacità di costruire copie di collezioni di dati, esportandole da un nodo a un altro di un sistema distribuito, in modo da massimizzare la disponibilità dei dati e anche aumentarne l'affidabilità. Prodotti specifici sono in grado di garantire la gestione di copie in modo trasparente rispetto ai programmi che aggiornano gli originali.

In un quadro tecnologico caratterizzato dalla necessità di far interagire fra loro sistemi e prodotti diversi, assumono inoltre grande importanza i problemi di portabilità e di interoperabilità.

- La *portabilità* denota la possibilità di trasportare programmi da un ambiente a un altro (ed è quindi una proprietà tipica del *tempo di compilazione*).
- L'*interoperabilità* denota la possibilità di far interagire sistemi eterogenei (ed è quindi una proprietà tipica del *tempo di esecuzione*).

Per ottenere portabilità e interoperabilità assumono grande importanza gli *standard*; in particolare, la portabilità dipende dagli standard relativi ai linguaggi (innanzitutto, SQL), mentre l'interoperabilità dipende dagli standard relativi ai protocolli di accesso ai dati. Abbiamo già discusso nel Capitolo 10 gli standard ODBC e DBC per far colloquiare fra loro basi di dati eterogenee; in questo capitolo vedremo il protocollo X-Open DTP, una versione standardizzata del protocollo di commit a due fasi che assicura la possibilità di far interagire in una stessa transazione DBMS diversi.

A partire dalla fine degli anni Novanta, le architetture dei sistemi distribuiti hanno risentito in modo straordinario della presenza di un nuovo paradigma unificante per accedere ai loro contenuti: il *World Wide Web*. È possibile considerare tutta l'informazione accessibile su Internet come un'enorme, diversificata, eterogenea, talvolta inaffidabile base di dati, interrogabile tramite i motori di ricerca, che offrono semplici meccanismi di interrogazione. Abbiamo poi visto il nascere (e il decadere) di molte tecnologie dedicate a estrarre contenuti dalle basi di dati e a presentarli nel contesto di pagine Web (le cosiddette pagine *generate dinamicamente*); più in generale, il Web ha introdotto numerose innovazioni nella tecnologia e nella progettazione dei sistemi informativi. Tali aspetti sono descritti separatamente nel Capitolo 18.

16.1 Architettura client-server

Il paradigma client-server è un modello di interazione tra processi software, ove i processi interagenti si suddividono tra *client*, che richiedono servizi, e *server*, che offrono servizi. L'interazione client-server richiede perciò una precisa definizione di una *interfaccia di servizi*, che elenca i servizi messi a disposizione dal server. Il processo client è tipicamente dedicato a interagire con l'utente finale; esso svolge un ruolo attivo, in quanto genera autonomamente richieste di servizi. Invece, il processo server è reattivo: esso svolge una computazione solo a seguito di una richiesta da parte di un qualunque client. Normalmente, un processo client può richiedere in sequenza alcuni (pochi) servizi a vari processi server, mentre ogni processo server risponde a (molte) richieste da parte di molti processi client. Per ora ipotizziamo che ogni client rivolga tutte le sue richieste a un solo server, e che tali richieste siano identificate come appartenenti a una stessa transazione, inizializzata alla prima richiesta fatta dal client. Tratteremo invece l'interazione di un client con vari server nel prossimo paragrafo, relativo alle basi di dati distribuite.

Non è necessario che i processi server e client siano allocati su macchine diverse: la distinzione fra processi client e server è un ottimo paradigma per la costruzione del software indipendentemente dall'allocazione dei processi. Tuttavia nella gestione dei dati l'uso di elaboratori diversi per la parte client e server è ormai molto diffuso. Vari motivi spingono verso l'uso di architetture client-server per basi di dati.

- Le funzioni di client e server sono ben identificate nel contesto delle basi di dati. Esse corrispondono a una decomposizione ideale delle competenze e professionalità: il programmatore applicativo ha la responsabilità di gestire il software relativo al client rispondendo a esigenze specifiche, mentre l'amministratore della base di dati ha la responsabiltà sul server che è condiviso da varie applicazioni, e deve organizzare la base di dati così che essa garantisca prestazioni ottimali a tutti i processi client.
- Oltre alla decomposizione funzionale dei processi e dei compiti, nelle basi di dati l'utilizzo di elaboratori diversi per client e server è particolarmente conveniente. L'elaboratore dedicato al client deve essere adatto all'interazione con l'utente; si usa spesso un personal computer, dotato di strumenti di produttività (posta elettronica, word processing, spreadsheet, accesso a Internet, workflow management) che sono tipici dell'automazione d'ufficio. Tra questi strumenti, spesso mascherate da una "interfaccia amichevole", vi sono le applicazioni che richiedono l'uso della base di dati. L'elaboratore dedicato al server è dimensionato in funzione dei

servizi che deve offrire e del carico transazionale, e gestisce ovviamente ampi buffer di memoria centrale e dispositivi di memoria di massa efficienti e robusti, con un'elevata capacità di svolgere operazioni di ingresso/uscita.

- Il linguaggio SQL, diffuso in tutte le basi di dati relazionali, offre un paradigma di programmazione ideale per identificare la "frontiera dei servizi". Le interrogazioni SQL vengono infatti formulate dal client e inviate al server; i risultati dell'interrogazione vengono calcolati dal server e restituiti al client. Sulla rete viaggia così un'informazione compatta: l'interrogazione può essere sottomessa come un'invocazione di un servizio remoto oppure come una stringa di caratteri. Il server esegue le interrogazioni, estraendo dati utili a partire dai dati presenti nei file, e confeziona il risultato dell'interrogazione; esegue inoltre eventuali comandi di modifica dei dati, contenuti nell'interrogazione. Quindi, sulla rete viaggia solo l'informazione utile al processo client, che rappresenta una frazione assai piccola dell'informazione estratta dalla memoria di massa del server e transitata nel canale di ingresso/uscita. In aggiunta, la standardizzazione, portabilità e interoperabilità del linguaggio SQL consentono di costruire applicazioni che coinvolgono sistemi differenti.

L'architettura client-server si adatta sia a interrogazioni compilate staticamente sia a interrogazioni con SQL dinamico. Con un processo statico (*compile and store*), le interrogazioni vengono sottomesse al server una volta e poi richiamate molte volte; meccanismi quali i *cursori*, descritti nel Capitolo 10, consentono di richiamare le interrogazioni dai processi client, con chiamate a procedure e/o servizi remoti. Con un processo dinamico (*compile and go*), le interrogazioni vengono trasmesse sotto forma di stringhe di caratteri che vengono compilate ed eseguite dal server. In entrambi i casi, l'ottimizzatore e i metodi di accesso risiedono sul server, che quindi include tutti i meccanismi che abbiamo illustrato nei Capitoli 11 e 12.

In genere, un server può ricevere un'interrogazione parametrica, in cui cioè il processo client assegna valori ad alcuni parametri di ingresso e poi richiama l'esecuzione di un'interrogazione o procedura. Spesso, il server che gestisce tali richieste è *multi-threaded*: dal punto di vista del sistema operativo, si comporta cioè come un unico processo che opera dinamicamente per conto di differenti transazioni; ciascuna unità di esecuzione del processo server per conto di una transazione è detto *thread*. Questa organizzazione, illustrata in Figura 16.1, consente di gestire i server come processi permanentemente attivi che controllano una *coda di ingresso*, su cui vengono accodate le richieste dei client, e una *coda di uscita*, su cui vengono accodati i risultati delle richieste. L'uso di un modello multi-threaded è più conveniente rispetto ad associare a ciascuna transazione un processo dedicato, perché i tempi di allocazione, deallocazione e commutazione per i thread sono molto minori che per i processi.

I server possono gestire le code o direttamente o tramite altri processi, denominati *dispatcher*, il cui compito è distribuire le richieste ai processi server e restituire le risposte al processo client. In alcuni casi, i dispatcher possono definire dinamicamente il numero di processi server attivi in funzione del numero di richieste ricevute; in tal caso, si dice che è disponibile una *classe di server*, che contiene un numero arbitrario di processi server tra loro indistinguibili. Questo fenomeno assomiglia alla

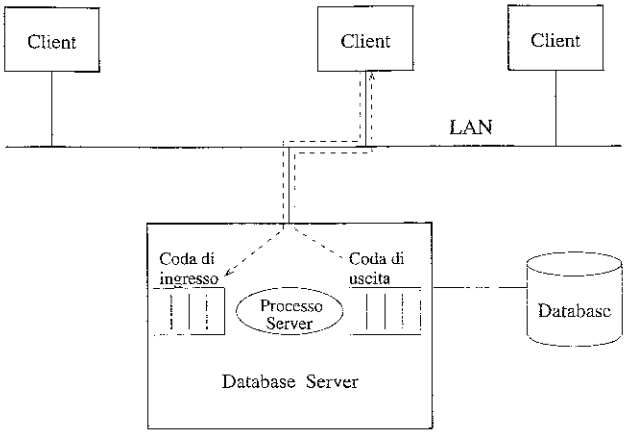

Figura 16.1 Architettura client-server.

gestione delle code in uscita da un grande magazzino, ove il numero di casse aperte varia dinamicamente in funzione del numero di clienti presenti alle casse stesse.

L'architettura illustrata fino a ora viene denominata architettura a due livelli (*two-tier*) in quanto in essa sono presenti un client, con funzioni sia di interfaccia utente sia di gestione dell'applicazione, e un server dedicato alla gestione dei dati. Recentemente si è diffusa anche un'architettura alternativa, detta a tre livelli (*three-tier*), in cui è presente un secondo server, detto *server applicativo*, responsabile di gestire la logica applicativa comune a più client. In questa architettura il client è assai semplice e si occupa del solo interfacciamento con l'utente finale (si dice che il client è leggero, dall'inglese *thin client*). Il client invia richieste al server applicativo, e quest'ultimo dialoga con il server per la gestione dei dati secondo le modalità illustrate in precedenza. Questa architettura è utilizzata soprattutto nell'ambito delle applicazioni su Internet, in quanto consente la realizzazione dei client tramite un semplice browser, e verrà pertanto descritta nel Capitolo 18.

16.2 Basi di dati distribuite

Abbiamo visto che in un'architettura client-server una transazione coinvolge al più un server; quando viceversa le transazioni coinvolgono più server, parliamo di *base di dati distribuita*. In questo paragrafo intraprendiamo lo studio delle basi di dati distribuite da un punto di vista funzionale, studiando cioè il modo in cui un utente può specificare interrogazioni distribuite. Nel prossimo paragrafo ci concentreremo viceversa sugli aspetti tecnologici, e vedremo come è necessario far evolvere la tecnologia dei server per consentire il loro uso in una base di dati distribuita.

16.2.1 Applicazioni delle basi di dati distribuite

Le motivazioni che portano allo sviluppo di soluzioni distribuite nella gestione dei dati sono di tipo pragmatico; esse rispondono all'esigenza di adeguare la gestione dei dati alla gestione dell'impresa, che è strutturalmente distribuita. La gestione distribuita dei dati si contrappone a una gestione centralizzata tipica dei grandi centri di calcolo, dominante fino alla metà degli anni Ottanta, e consente l'elaborazione e il controllo dei dati negli ambienti ove essi sono generati e maggiormente usati. Sul piano tecnologico, abbiamo assistito recentemente al progressivo prevalere dei sistemi distribuiti su quelli centralizzati, giustificato da un gran numero di motivazioni; tra di esse, la maggior flessibilità, modularità e resistenza ai guasti. I sistemi distribuiti possono essere configurati aggiungendo e modificando progressivamente gli elaboratori che li compongono, con una flessibilità e modularità ben maggiore dei sistemi basati sull'uso di "mainframe"; pur essendo maggiormente esposti ai guasti per la loro maggior complessità strutturale, sono infatti capaci di rispondere ai guasti con un degrado delle prestazioni, evitando però un blocco completo.

Una prima classificazione delle basi di dati distribuite considera il tipo di DBMS e di rete utilizzato. Quando tutti i server utilizzano DBMS dello stesso produttore (eventualmente uniformi anche a livello di versione) la base di dati si dice *omogenea*, altrimenti si dice *eterogenea*. Una base di dati distribuita può utilizzare una *rete locale* (LAN, *Local Area Network*) oppure una *rete geografica* (WAN, *Wide Area Network*). Questa classificazione introduce una corrispondente classificazione di soluzioni applicative.

Ovviamente, i sistemi omogenei su LAN corrispondono a soluzioni tecnologicamente più semplici, presenti in un gran numero di applicazioni (per esempio, di tipo gestionale). Tuttavia anche le situazioni più complesse vengono utilizzate da applicazioni indispensabili nella società contemporanea. Per esempio, i sistemi per applicazioni bancarie possono essere realizzati sia su LAN sia su WAN, a seconda della complessità e diffusione delle filiali della banca sul territorio.

Anche la soluzione eterogenea è molto diffusa; per esempio, molti sistemi informativi intersettoriali integrati presenti nelle aziende sono eterogenei. L'eterogeneità origina in alcuni casi dal fatto che ciascun settore aziendale sviluppa in maniera autonoma le proprie applicazioni e in particolare sceglie in modo autonomo il software per la gestione dei dati; si scopre poi a livello direzionale che i settori aziendali devono interagire, ma a questo punto il sistema informativo risultante è eterogeneo. Per evitare questa situazione, moltissime aziende impongono delle linee guida tassative a livello direzionale per quanta riguarda il tipo di DBMS da usare da parte di qualunque applicazione. Nonostante la presenza di queste linee guida, può comunque essere necessario affrontare scenari con architetture eterogenee, per esempio quando si devono realizzare applicazioni per la cooperazione tra aziende diverse, oppure quando le aziende sono coinvolte in fusioni e acquisizioni.

Questa "dinamica" aziendale giustifica la forte preferenza che viene data da chi disegna l'architettura del sistema informatico verso l'uso di *architetture aperte*, cioè in grado di interagire a tutti i livelli con sistemi di altri produttori, contrapposte alle cosiddette *architetture proprietarie*, in cui hardware e software sono forniti da un solo venditore, capace di garantire compatibilità solo all'interno di una famiglia di prodotti.

16.2.2 Autonomia locale e cooperazione

Da un punto di vista astratto, una base di dati distribuita può essere considerata come un'unica raccolta di dati; i database server garantiscono ai programmi applicativi un accesso transazionale a queste risorse, offrendo all'utente applicativo esattamente lo stesso tipo di interazione che vi sarebbe con un sistema centralizzato. Tuttavia, è importante notare che in una base di dati distribuita ciascun server ha una sua capacità di gestire applicazioni in modo *autonomo*, e garantire l'autonomia dei server è uno dei principali obiettivi nella gestione di questi sistemi.

La motivazione che spinge a costruire le basi di dati distribuite non è perciò quella di massimizzare l'interazione e la necessità di trasmettere in rete; viceversa, un progetto accurato della distribuzione e allocazione dei dati dovrebbe garantire che il maggior numero possibile di applicazioni si svolgano in modo autonomo su di un solo server, in modo da non dover incorrere nei maggiori costi di esecuzione e nella maggior criticità che è tipica delle applicazioni distribuite.

16.2.3 Frammentazione e allocazione dei dati

La frammentazione dei dati consente di organizzare i dati stessi in modo tale da garantire una loro distribuzione efficiente e ben organizzata; questa tecnica è applicabile solo se la distribuzione dei dati viene pianificata a priori.

Consideriamo una relazione R; la sua frammentazione consiste nel determinare un certo numero di frammenti R_i, ottenuti applicando a R operazioni algebriche. La frammentazione può essere di due tipi, orizzontale e verticale.

- Nella *frammentazione orizzontale*, i frammenti R_i sono insiemi di tuple con lo stesso schema della relazione R; ciascun frammento orizzontale può essere interpretato come il risultato di una selezione applicata alla relazione R.
- Nella *frammentazione verticale*, i frammenti R_i hanno uno schema ottenuto come sottoinsieme dello schema di R; ciascun frammento verticale può essere interpretato come il risultato di una proiezione applicata alla relazione R.

La frammentazione è corretta se valgono le seguenti proprietà:

- *completezza*: ogni dato della relazione R deve essere presente in un qualche suo frammento R_i;
- *ricostruibilità*: la relazione deve essere interamente ricostruibile a partire dai suoi frammenti.

Normalmente i frammenti orizzontali sono disgiunti, cioè non hanno tuple in comune; viceversa, i frammenti verticali includono la chiave primaria definita per R, in modo da garantire la ricostruibilità (si ricordino a questo proposito le *decomposizioni senza perdita* già discusse nel Capitolo 9). Si noti che la frammentazione può essere formalmente definita tramite operazioni sulle relazioni; in altri termini, la frammentazione esprime proprietà *logiche* dei dati.

Vediamo un esempio di frammentazione orizzontale e verticale. Si consideri la relazione:

IMPIEGATO(Empnum, Nome, Dipnum, Stipendio, Tasse)

- La sua frammentazione orizzontale si ottiene supponendo le tuple di IMPIEGATO suddivise in più frammenti da operazioni di selezione:

$$\text{IMPIEGATO1} = \sigma_{\text{Empnum} \leq 3}\ \text{IMPIEGATO}$$
$$\text{IMPIEGATO2} = \sigma_{\text{Empnum} > 3}\ \text{IMPIEGATO}$$

  La ricostruzione della relazione a partire dai suoi frammenti richiede un'operazione di unione:

$$\text{IMPIEGATO} = \text{IMPIEGATO1} \cup \text{IMPIEGATO2}$$

- La sua frammentazione verticale si ottiene supponendo le tuple di IMPIEGATO suddivise in più frammenti da operazioni di proiezione che includano in ciascun frammento la chiave primaria della relazione:

$$\text{IMPIEGATO1} = \pi_{\text{Empnum,Nome}}\ (\text{IMPIEGATO})$$
$$\text{IMPIEGATO2} = \pi_{\text{Empnum,Dipnum,Stipendio,Tasse}}\ (\text{IMPIEGATO})$$

  La ricostruzione della relazione a partire dai suoi frammenti richiede un'operazione di equi-join a pari valore di chiave (join naturale):

$$\text{IMPIEGATO} = \text{IMPIEGATO1} \bowtie \text{IMPIEGATO2}$$

I due esempi di frammentazione orizzontale e verticale precedentemente discussi sono illustrati nelle Figure 16.2, 16.3 e 16.4.

Ogni frammento R_i viene implementato tramite un file fisico e installato presso un server; si dice che i frammenti vengono *allocati* sui server. Quindi, la relazione è presente in modo virtuale (come una vista), mentre i frammenti sono effettivamente memorizzati. Ovviamente, anche le relazioni non frammentate vengono allocate su un server. Lo *schema di allocazione* contiene il mapping dai frammenti o da intere relazioni ai server che li memorizzano, consentendo il passaggio da una descrizione logica a una descrizione fisica dei dati. Tale mapping può essere:

- *non ridondante*, quando ciascun frammento o relazione viene allocato esattamente su un server;
- *ridondante*, quando qualche frammento o relazione viene allocato su più di un server.

Empnum	Nome	Dipnum	Stipendio	Tasse
1	Roberto	Produzione	37 000	12 000
2	Giovanni	Amministrazione	35 000	11 000
3	Anna	Produzione	53 000	21 000
4	Carlo	Marketing	35 000	11 000
5	Alfredo	Amministrazione	37 000	12 000
6	Paolo	Direzione	83 000	35 000
7	Giorgio	Marketing	42 000	14 000

Figura 16.2 Tabella di esempio per la frammentazione.

Empnum	Nome	Dipnum	Stipendio	Tasse
1	Roberto	Produzione	37 000	12 000
2	Giovanni	Amministrazione	35 000	11 000
3	Anna	Produzione	53 000	21 000

Primo frammento orizzontale

Empnum	Nome	Dipnum	Stipendio	Tasse
4	Carlo	Marketing	35 000	11 000
5	Alfredo	Amministrazione	37 000	12 000
6	Paolo	Direzione	83 000	35 000
7	Giorgio	Marketing	42 000	14 000

Secondo frammento orizzontale

Figura 16.3 Esempio di frammentazione orizzontale.

Empnum	Nome
1	Roberto
2	Giovanni
3	Anna
4	Carlo
5	Alfredo
6	Paolo
7	Giorgio

Primo frammento verticale

Empnum	Dipnum	Stipendio	Tasse
1	Produzione	37 000	12 000
2	Amministrazione	35 000	11 000
3	Produzione	53 000	21 000
4	Marketing	35 000	11 000
5	Amministrazione	37 000	12 000
6	Direzione	83 000	35 000
7	Marketing	42 000	14 000

Secondo frammento verticale

Figura 16.4 Esempio di frammentazione verticale.

16.2.4 Livelli di trasparenza

La distinzione tra frammentazione e allocazione consente di individuare vari livelli di trasparenza nelle applicazioni, cioè vari modi di scrivere applicazioni, dal più astratto e indipendente dalla distribuzione dei dati al più concreto e dipendente dalla loro allocazione fisica. I livelli di trasparenza più significativi sono tre: trasparenza di frammentazione, di allocazione e di linguaggio; vi è poi la possibilità di una totale assenza di trasparenza, quando non esiste un linguaggio comune per accedere ai dati presenti nel sistema, e i programmatori devono quindi rivolgersi a ciascun server con uno specifico "dialetto".

Vediamo un esempio di questi quattro livelli di trasparenza. Consideriamo la tabella che descrive i fornitori di prodotti per un'impresa:

$$\text{FORNITORE}(\underline{\text{Fnum}}, \text{Nome}, \text{Città})$$

frammentata orizzontalmente in due frammenti relativi alle città di Milano e Roma, supposte le uniche città in cui l'impresa opera:

$$\text{FORNITORE1} = \sigma_{\text{Città}='Milano'}(\text{FORNITORE})$$
$$\text{FORNITORE2} = \sigma_{\text{Città}='Roma'}(\text{FORNITORE})$$

Roma dispone di una base di dati replicata, in quanto il secondo frammento è allocato su due nodi *roma*1 e *roma*2. Pertanto, l'allocazione dei frammenti è:

FORNITORE1@`ditta.milano.it`
FORNITORE2@`ditta.roma1.it`
FORNITORE2@`ditta.roma2.it`

Consideriamo poi la semplice applicazione che richiede al suo utente un numero di fornitore e ne restituisce il nome. Codifichiamo tale applicazione in SQL ai vari livelli di trasparenza.

- *Trasparenza di frammentazione*: a questo livello, il programmatore non si deve preoccupare del fatto che la base di dati sia o meno distribuita e frammentata; la sua interrogazione è identica a quella che verrebbe scritta per una relazione non frammentata.

```
procedure Query1(:fnum int,:nome varchar(20))
    select Nome into :nome
    from Fornitore
    where Fnum = :fnum;
```

- *Trasparenza di allocazione*: a questo livello, il programmatore conosce la struttura dei frammenti, ma non deve indicarne l'allocazione. In particolare, se il sistema ammette frammenti replicati (come nel nostro esempio), il programmatore non deve neppure indicare quale copia viene scelta per l'accesso (questa proprietà è denominata *trasparenza di replicazione*). Si ipotizza che il parametro `:empty` assuma valore *true* se la prima interrogazione SQL non ritrova alcun valore; si noti quindi uno "stile sequenziale" del programma, che esplora i frammenti uno dopo l'altro.

```
procedure Query2(:fnum int,:nome varchar(20))
begin
    select Nome into :nome
    from Fornitore1
    where Fnum = :fnum;
    if :empty then
          select Nome into :nome
          from Fornitore2
          where Fnum = :fnum;
end;
```

- *Trasparenza di linguaggio*: a questo livello, il programmatore deve indicare nella sua interrogazione sia la struttura dei frammenti sia la loro allocazione; per esempio, deve scegliere di accedere al frammento memorizzato al nodo *roma1*. È avvantaggiato però dalla presenza di un solo linguaggio di interrogazione in cui esprimere l'interrogazione.

```
procedure Query3(:fnum int,:nome varchar(20))
begin
    select Nome into :nome
    from Fornitore1@ditta.milano.it
    where Fnum = :fnum;
    if :empty then
          select Nome into :nome
          from Fornitore2@ditta.roma1.it
          where Fnum = :fnum;
end;
```

- *Assenza di trasparenza:* in questo caso, ciascun DBMS accetta un proprio "dialetto" SQL, in quanto il sistema è eterogeneo e i DBMS non riconoscono un linguaggio di interoperabilità comune. Come nel caso precedente, il programmatore deve indicare in modo esplicito i frammenti e la loro allocazione nel sistema.

Si noti che gli ultimi due livelli caratterizzano le interrogazioni che vengono effettivamente eseguite in un contesto omogeneo oppure eterogeneo; interrogazioni sottoposte da un utente al livello di trasparenza più elevato (di frammentazione o di allocazione) vengono trasformate durante la fase di ottimizzazione, introducendo nella loro specifica la scelta di specifici frammenti e la loro allocazione su specifici nodi.

Questa applicazione può essere resa più efficiente pur di utilizzare il parallelismo oppure di richiedere maggior informazione circa il frammento cui indirizzare la richiesta. Ci occuperemo del parallelismo nel Paragrafo 17.1; la seconda strategia è illustrata nel programma seguente. In questo esempio si richiede all'utente l'indicazione sulla città del fornitore e ciò consente di indirizzare la richiesta al frammento che contiene il dato di interesse, con una soluzione che si colloca al livello di *trasparenza di allocazione*. Si noti che questo approccio introduce un forte elemento di rigidità: se in un futuro la frammentazione dovesse cambiare, la logica applicativa dovrebbe essere rivista.

```
procedure Query4(:fnum int,
    :nome varchar(20),:citta varchar(15));
begin
    case :citta of
    "Milano":
          select Nome into :nome
          from Fornitore1
          where Fnum = :fnum;
    "Roma":
          select Nome into :nome
          from Fornitore2
          where Fnum = :fnum;
end;
```

16.2.5 Classificazione delle transazioni

Uno schema di classificazione delle transazioni basato sulla composizione degli statement SQL che compongono una transazione è stato proposto per i DBMS della IBM e successivamente adottato da vari altri costruttori. In questo schema, ogni client fa riferimento a un solo DBMS e la distribuzione della transazione avviene a opera di questo DBMS. Il client può ovviamente indirizzare transazioni *locali* al DBMS, cioè transazioni composte da interrogazioni le cui tabelle sono tutte allocate su quel DBMS; può poi indirizzare transazioni *non locali*, che cioè coinvolgono dati memorizzati su altri DBMS. Queste transazioni possono essere più o meno complesse.

- Le *richieste remote* (*remote request*) sono transazioni di sola lettura (costituite cioè da un numero arbitrario di interrogazioni `select`) indirizzate a un solo DBMS remoto.
- Le *transazioni remote* (*remote transaction*) sono transazioni costituite da un numero qualsiasi di comandi SQL (`select`, `insert`, `delete`, `update`) dirette a un solo DBMS remoto.
- Le *transazioni distribuite* (*distributed transaction*) sono transazioni rivolte a un numero generico di DBMS ma in cui ciascun comando SQL fa riferimento a dati memorizzati su di un solo DBMS.
- Le *richieste distribuite* (*distributed request*) sono transazioni arbitrarie, costituite da un numero arbitrario di comandi SQL, in cui ciascuna query può far riferimento a dati distribuiti su qualunque DBMS. Si noti che questa classificazione fa riferimento a interrogazioni SQL al livello di trasparenza di linguaggio; quindi, tutte le query al livello di trasparenza di frammentazione sono classificabili come richieste distribuite.

Questo schema, nonostante adotti alcuni termini in modo poco naturale, è diventato un modello di riferimento per la classificazione. È importante l'individuazione di livelli progressivi di complessità nell'interazione fra DBMS. Nel primo caso il DBMS remoto può essere solo interrogato; nel secondo caso è possibile operare scritture (ma ciascuna transazione scrive su di un solo DBMS); nel terzo caso è possibile includere

in una transazione scritture su più nodi ma ciascuna interrogazione SQL è indirizzata a uno specifico DBMS; nell'ultimo caso l'interrogazione SQL deve essere distribuita su più nodi. Come vedremo, il terzo caso richiede l'uso del protocollo di commit a due fasi e il quarto caso richiede, in aggiunta, la disponibilità di un ottimizzatore per interrogazioni distribuite.

Un esempio tipico di transazione è il trasferimento di un importo fra due conti correnti, descritti tramite la relazione:

CONTOCORRENTE(CCNum,Nome,Saldo)

Assumiamo che la relazione sia frammentata in modo tale che tutti i conti inferiori al numero 10000 siano allocati sul primo frammento e tutti i conti superiori siano allocati sul secondo frammento; un trasferimento di 100 euro dal conto corrente numero 3154 al numero 14878 comporta così due operazioni di modifica, una per frammento; questo è un esempio di transazione distribuita scritta al livello di trasparenza di allocazione.

```
begin transaction
      update ContoCorrente1
      set Saldo = Saldo - 100
      where CCNum = 3154;

      update ContoCorrente2
      set Saldo = Saldo + 100
      where CCNum = 14878;
      commit work;
end transaction
```

I correntisti 3154 e 14878 (e ovviamente anche la banca) desiderano che la transazione sia eseguita correttamente su entrambi i nodi; in alternativa, è accettabile che non venga eseguita nessuna delle due modifiche. È assolutamente inaccettabile il caso in cui una delle due modifiche venga eseguita ma non l'altra.

16.3 Tecnologia delle basi di dati distribuite

Nei paragrafi precedenti, abbiamo visto alcune caratteristiche delle applicazioni di una base di dati distribuita; vediamo ora come realizzare tali applicazioni dal punto di vista tecnologico. Dapprima ci poniamo il problema di capire in che modo la tecnologia delle basi di dati debba essere estesa per tenere in conto la distribuzione dei dati; vedremo che alcuni sottosistemi non pongono alcun problema mentre altri divengono assai più critici. Anticipiamo che la distribuzione dei dati non influisce su due proprietà acide delle transazioni: consistenza e persistenza.

- La *consistenza* delle transazioni non dipende dalla distribuzione dei dati, in quanto i vincoli di integrità descrivono solo proprietà locali a un DBMS. Questo è più che altro un limite dell'attuale tecnologia dei DBMS, poiché vincoli di integrità potrebbero benissimo far riferimento a dati distribuiti, ma di fatto non esistono meccanismi che consentano di specificarli o verificarli.

- In modo del tutto analogo, la *persistenza* non è un problema che dipende dalla distribuzione dei dati, perché ciascun sistema garantisce la persistenza anche in presenza di guasti dei dispositivi tramite meccanismi di dump e backup locali.

Invece, altri problemi caratterizzano la tecnologia delle basi di dati distribuite; ci occupiamo prima dell'ottimizzazione delle interrogazioni, poi del controllo di concorrenza, infine dell'atomicità di transazioni distribuite.

16.3.1 Ottimizzazione di interrogazioni distribuite

L'ottimizzazione delle interrogazioni è richiesta solo quando un utente può sottomettere a un DBMS una richiesta distribuita, in cui cioè dati allocati su più DBMS sono coinvolti nella stessa interrogazione. Il DBMS cui viene sottomessa l'interrogazione è responsabile della cosiddetta "ottimizzazione globale"; decide cioè la decomposizione dell'interrogazione in tante sotto-interrogazioni, ciascuna rivolta a uno specifico DBMS. Una strategia di esecuzione distribuita consiste nell'esecuzione coordinata di vari programmi sui vari DBMS e nello scambio di dati fra di essi, che si rende necessario quando i risultati di una sotto-interrogazione vengono usati come operandi in un'altra sotto-interrogazione. Pertanto, tra i fattori di costo di un'interrogazione distribuita vi è anche la *quantità di dati trasmessi in rete*.

L'ottimizzatore globale decide il piano più conveniente esaminando, come nel Paragrafo 11.6, un albero delle alternative. Come nei sistemi centralizzati, è necessario scegliere l'ordine delle operazioni e il loro metodo di esecuzione; in aggiunta, si deve anche definire la strategia di esecuzione delle operazioni i cui operandi sono allocati su nodi distinti, definendo quindi una strategia di trasmissione dei dati e di allocazione dei risultati; quando i dati sono ridondanti è necessario anche scegliere la particolare copia dei dati utilizzata per l'accesso. Come negli ottimizzatori centralizzati, è possibile associare a un nodo foglia un costo, che questa volta prevede il contributo di tre componenti:

$$C_{total} = C_{I/O} \times n_{I/O} + C_{cpu} \times n_{cpu} + C_{tr} \times n_{tr}$$

I due nuovi elementi n_{tr} e C_{tr} misurano, rispettivamente, la quantità di dati trasmessi in rete e il costo di trasmissione unitario, e si aggiungono ai costi di elaborazione e di ingresso/uscita descritti nel Paragrafo 11.6. L'importanza di questo terzo fattore è cambiata nel corso del tempo: agli inizi degli anni Ottanta, date le ridotte prestazioni relative alla trasmissione dati, si riteneva che il costo di trasmissione fosse così importante da prevalere su tutti gli altri, e in pratica l'ottimizzazione di interrogazioni distribuite si basava esclusivamente sulla riduzione dei tempi e costi di trasmissione. Questa semplificazione del problema attualmente non è più giustificata, dato che la trasmissione dati ha, specie su una rete locale veloce come una Gigabit Ethernet, prestazioni paragonabili a quelle dei dispositivi di ingresso/uscita.

Il lancio dell'interrogazione da parte di un processo client scatena tutti i processi necessari all'esecuzione dell'interrogazione presso i vari server. L'esecuzione dei metodi di accesso avviene localmente ai vari DBMS.

16.3.2 Controllo di concorrenza

Il controllo della concorrenza in ambito distribuito potrebbe riservare gravi difficoltà, ma la teoria del controllo di concorrenza ci riserva una sorpresa gradevole, sotto forma di proprietà che confermano la validità dei metodi di locking a due fasi e di timestamping.

Vediamo innanzitutto di impostare il problema. In un sistema distribuito, una transazione t_i può eseguire varie sotto-transazioni t_{ij}, ove il secondo indice denota il nodo del sistema su cui la sotto-transazione opera. Per esempio, una transazione distribuita t_1 che opera su due oggetti x e y allocati sui nodi 1 e 2 si presenta, dal punto di vista dei suoi accessi concorrenti, nel modo seguente:

$$t_1 \quad r_{11}(x)\, w_{11}(x)\, r_{12}(y)\, w_{12}(y)$$

Osserviamo che la serializzabilità locale presso gli scheduler non è una garanzia sufficiente per la serializzabilità. Infatti, supponiamo di avere una seconda transazione t_2 che opera anch'essa sui dati x e y, ma accede a essi in ordine opposto:

$$t_2 \quad r_{22}(y)\, w_{22}(y)\, r_{21}(x)\, w_{21}(x)$$

Può accadere che le due transazioni eseguano ai nodi 1 e 2 presentando due schedule S_1 e S_2, come segue:

$$S_1: \quad r_{11}(x)\, w_{11}(x)\, r_{21}(x)\, w_{21}(x)$$

$$S_2: \quad r_{22}(y)\, w_{22}(y)\, r_{12}(y)\, w_{12}(y)$$

Questi due schedule sono localmente serializzabili (addirittura, sono localmente seriali); se però osserviamo il loro grafo di conflitti globale, costruito come nel Paragrafo 12.2.4, scopriamo un ciclo tra t_1 e t_2, in quanto:

- sul nodo 1, t_1 precede t_2 ed è in conflitto con t_2;
- sul nodo 2, t_2 precede t_1 ed è in conflitto con t_1.

Quindi, le due esecuzioni non sono conflict-serializzabili (CSR). È facile costruire esempi di applicazioni in cui due transazioni t_1 e t_2 che agiscano nel modo descritto dagli schedule S_1 e S_2 portino alla perdita di update o alla presenza di update fantasmi.

Serializzabilità globale La *serializzabilità globale* di due transazioni distribuite richiede l'esistenza di un *unico schedule seriale* S, che coinvolga tutte le transazioni del sistema, e che sia equivalente a tutti gli schedule locali S_i; per ogni nodo i, deve cioè accadere che la proiezione $S[i]$ di S contenente le sole azioni che avvengono sul nodo i sia equivalente allo schedule S_i. Questa proprietà è difficile da verificare nel caso si usino scheduler che applicano direttamente la view-equivalenza o la conflict-equivalenza, ma risulta immediatamente verificata qualora si usino il locking a due fasi o il metodo di timestamping. Valgono infatti le proprietà elencate di seguito.

- Se ciascuno scheduler della base di dati distribuita usa su ciascun nodo il metodo di locking a due fasi stretto e quindi svolge l'azione di commit in modo atomico in un istante in cui tutte le sotto-transazioni ai vari nodi detengono tutte le risorse, gli schedule risultanti sono globalmente serializzabili in base ai conflitti.
- Se un insieme di sotto-transazioni distribuite acquisisce un unico timestamp e usa tale timestamp nelle sue richieste a tutti gli scheduler che usano il controllo di concorrenza basato su timestamp, gli schedule risultanti sono globalmente seriali in base all'ordinamento indotto dai timestamp.

Metodo di Lamport per assegnare i timestamp Per garantire il buon funzionamento del controllo di concorrenza basato su timestamp, è necessario che ciascuna transazione possa acquisire un timestamp che corrisponda all'istante di tempo in cui la transazione distribuita deve sincronizzarsi con le altre transazioni; per questo si può usare il *metodo di Lamport* per assegnare timestamp che riflettano le relazioni di precedenza fra eventi in un sistema distribuito. Con questo metodo, un timestamp è caratterizzato da due gruppi di cifre; le meno significative identificano un nodo, le più significative identificano gli eventi che accadono in ciascun nodo, e possono essere ottenute a partire da un contatore locale, incrementato a ogni evento. In aggiunta, ogni qual volta due nodi comunicano scambiandosi un messaggio, i timestamp si sincronizzano: l'evento ricezione deve avere un timestamp successivo all'evento di invio. Ciò è ottenuto incrementando opportunamente il contatore locale sul nodo ricevente.

La Figura 16.5 descrive un certo numero di eventi su 3 nodi di un sistema distribuito, e mostra un assegnamento di timestamp ottenuto con il metodo di Lamport; si noti che ciascun evento è caratterizzato da un timestamp distinto e che i timestamp riflettono l'ordinamento degli eventi indotto dagli scambi di messaggi, denotati da frecce tratteggiate.

Rilevazione distribuita dei deadlock Infine, trattiamo i problemi relativi alla rilevazione dei deadlock. Quando i deadlock coinvolgono transazioni distribuite, la loro rilevazione è ovviamente più complessa, in quanto i deadlock possono essere

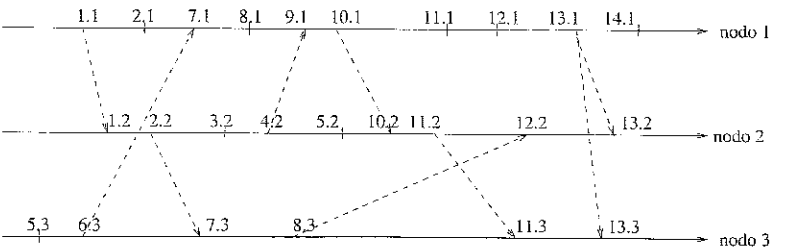

Figura 16.5 Esempio di assegnamento di timestamp con il metodo di Lamport.

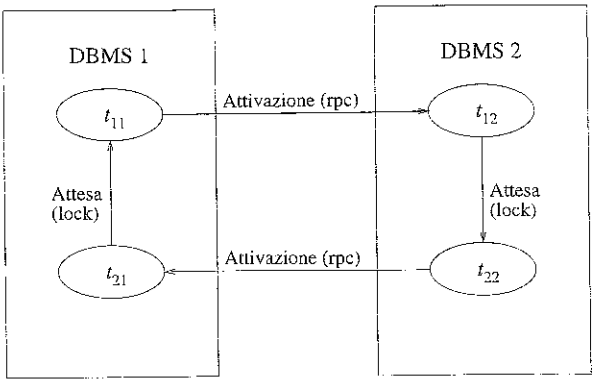

Figura 16.6 Esempio di un deadlock distribuito.

dovuti a situazioni di attesa circolare che si verificano su due o più nodi; ovviamente, la soluzione semplice di usare timeout vale indipendentemente dal contesto in cui si creano i deadlock, e perciò viene usata nella maggioranza dei sistemi. Esiste però un algoritmo elegante e abbastanza efficiente che rileva i deadlock; viene descritto nel seguito perché è un esempio abbastanza caratteristico di protocollo asincrono e strutturalmente distribuito.[1]

Assumiamo un modello di base di dati distribuita in cui le transazioni vengono decomposte in sotto-transazioni; in questo contesto, è possibile che le sotto-transazioni si attivino in modo sincrono, e in tal caso quando la sotto-transazione t_{11} attiva la sotto-transazione t_{12} (per esempio tramite una chiamata a procedura remota), t_{11} si mette in attesa della terminazione di t_{12}. Quindi, questo modello prevede due distinti tipi di attesa: due sotto-transazioni della stessa transazione possono essere in attesa su DBMS distinti perché l'una attende la terminazione dell'altra, e due sotto-transazioni diverse sullo stesso DBMS possono attendersi in quanto una blocca un dato cui l'altra vuole accedere.

Un esempio di deadlock distribuito è illustrato in Figura 16.6; il deadlock è dovuto alle seguenti condizioni di attesa:

1. t_{11} attende t_{12}, attivata tramite una chiamata a procedura remota;
2. t_{12} attende una risorsa controllata da t_{22};
3. t_{22} attende t_{21}, attivata tramite una chiamata a procedura remota;
4. infine, t_{21} attende una risorsa controllata da t_{11}.

Le condizioni di attesa visibili a ciascun DBMS sono caratterizzabili tramite condizioni di precedenza. Al DBMS 1, sappiamo che t_{21} è attivata da un DBMS remoto

[1] Questo algoritmo è implementato nella versione distribuita del sistema IBM DB2.

e che attende t_{11}, la quale a sua volta ha attivato una sotto-transazione su un DBMS remoto. Abbiamo cioè:

$$E_2 \rightarrow t_{21} \rightarrow t_{11} \rightarrow E_2$$

Simmetricamente, al DBMS 2 si osservano le precedenze:

$$E_1 \rightarrow t_{12} \rightarrow t_{22} \rightarrow E_1$$

Più in generale, una qualunque condizione di attesa in cui una sotto-transazione t_i, attivata da un DBMS remoto, attende anche transitivamente a causa della situazione dei lock un'altra transazione T_j, che a sua volta attende una sotto-transazione remota, viene schematizzata con una *sequenza di attesa*:

$$(1) \quad E_{\text{in}} \rightarrow t_i \rightarrow t_j \rightarrow E_{\text{out}}$$

L'algoritmo per la rilevazione del deadlock distribuito è a sua volta un algoritmo distribuito, che viene periodicamente attivato ai vari DBMS del sistema; quando è attivato, esso analizza la situazione di attesa al suo DBMS e comunica con altre istanze dello stesso algoritmo le sequenze di attesa. Per evitare che lo stesso deadlock venga riconosciuto su più DBMS, l'algoritmo prevede che un messaggio (1), contenente sequenze di attesa, venga scambiato "in avanti", cioè verso il DBMS ove è attiva la sotto-transazione attesa da t_j, e venga inviato solo se, per esempio, $i > j$, ove i e j sono gli identificatori delle transazioni.

L'algoritmo si attiva periodicamente e opera come segue.

1. I messaggi contenenti sequenze di attesa provenienti da altri DBMS remoti vengono letti e l'informazione presente nei messaggi viene composta con il grafo di attesa locale. Ogni transazione è rappresentata nel grafo d'attesa da un solo nodo, indipendentemente dal numero di sotto-transazioni di cui è composta; i nodi corrispondenti a transazioni remote vengono comunque aggiunti al grafo d'attesa locale.
2. Viene svolta una ricerca locale di deadlock; situazioni di stallo vengono risolte forzando l'abort di una delle transazioni coinvolte nel deadlock.
3. Vengono poi computate le sequenze di attesa, e inviate ai DBMS remoti "in avanti", secondo la regola espressa in precedenza. Al termine della trasmissione, l'algoritmo si disattiva.

La Figura 16.7 illustra l'applicazione di questo algoritmo a una base di dati distribuita. Si assuma una condizione iniziale di attesa caratterizzata dalle sequenze di attesa (Figura 16.7a):

$$S_1: \quad E_3 \rightarrow t_3 \rightarrow t_1 \rightarrow E_2$$

$$S_2: \quad E_1 \rightarrow t_1 \rightarrow t_2 \rightarrow E_3$$

$$S_3: \quad E_2 \rightarrow t_2 \rightarrow t_3 \rightarrow E_1$$

In queste condizioni, dato che le sequenze di attesa devono essere scambiate "in avanti", solo il DBMS 1 può trasmettere la sua sequenza di attesa al DBMS 2, ove è in esecuzione la sotto-transazione attivata da t_1. La Figura 16.7b illustra la situazione

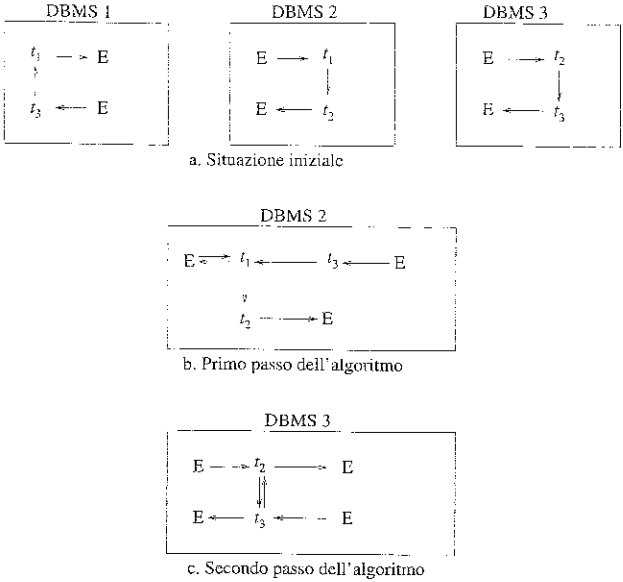

Figura 16.7 Rilevazione di un deadlock distribuito.

che si verifica al DBMS 2 dopo aver ricevuto la condizione di attesa S_1. Viene così rilevata una nuova condizione di attesa:

$$S_4: \quad E_3 \rightarrow t_3 \rightarrow t_2 \rightarrow E_3$$

inviata al DBMS 3, ove è in esecuzione la sotto-transazione attivata da t_2. La Figura 16.7c illustra la situazione che si genera al DBMS 3 dopo aver ricevuto la condizione di attesa S_4; finalmente, il deadlock costituito dalla coppia $< t_2, t_3 >$ è riconosciuto (si noti che il deadlock originario coinvolgeva anche t_1); una delle due transazioni è scelta per il rollback, che coinvolge tutte le sue sotto-transazioni su tutti i nodi, e il deadlock è risolto.

16.3.3 Atomicità di transazioni distribuite

Infine, ci occupiamo dell'atomicità delle transazioni distribuite. Per garantire l'atomicità è necessario che tutti i nodi che partecipano a una transazione giungano alla stessa decisione circa la transazione (commit o abort); è perciò necessario eseguire protocolli particolari, detti *protocolli di commit*, che consentano a una transazione di raggiungere correttamente la decisione di commit e abort.

Sfortunatamente, garantire l'atomicità è molto problematico per la presenza di molteplici cause di guasto. Nel seguito, proponiamo una classificazione dei guasti, prima di dedicarci ai protocolli di commit nel prossimo paragrafo.

Un sistema distribuito è soggetto alle *cadute di un nodo*, che possono avvenire su ogni elaboratore; il guasto su un nodo può essere soft oppure hard, come discusso nel Capitolo 12. In aggiunta alle cadute di nodi, è anche possibile che vengano *persi dei messaggi*, che lasciano l'esecuzione di un protocollo in situazione di incertezza. In genere, proprio per garantire l'avanzamento di un protocollo, ciascun messaggio del protocollo è seguito da un messaggio di risposta (detto "ack" come abbreviazione del termine inglese *acknowledgement*); tuttavia, la perdita di uno dei due messaggi (quello primario o di ack) lascia il mittente nell'incertezza se il messaggio primario sia stato ricevuto o no, e quindi se il destinatario del messaggio abbia ricevuto o no l'informazione relativa.

Dato che i messaggi possono essere persi, i protocolli di commit pongono un tempo limite alla ricezione del messaggio di ack, trascorso il quale il mittente decide di procedere comunque con il protocollo. In questo caso, è possibile che il timeout scatti anche in assenza di una reale perdita di messaggi, ma semplicemente perché uno dei due elaboratori che scambiano il messaggio è sovraccarico e quindi risulta troppo lento nel processare i messaggi. Infine, è possibile che qualche collegamento della rete si interrompa; in tal caso, oltre alla perdita di messaggi, si può verificare un altro fenomeno: il partizionamento della rete in due sotto-reti che non comunicano fra loro. Questo guasto può causare ulteriori problemi, in quanto è possibile che una transazione risulti contemporaneamente attiva in più di una sotto-rete.

Abbiamo così analizzato tutte le cause di guasto nei sistemi distribuiti; ci dedichiamo ora al protocollo di commit a due fasi.

16.4 Protocollo di commit a due fasi

Il protocollo di commit a due fasi ricorda, nelle sue linee essenziali, un matrimonio. Come in un matrimonio, la decisione di due persone viene raccolta e ratificata da una terza persona, che celebra il matrimonio. Perché il matrimonio abbia luogo, è necessario che entrambi i partecipanti esprimano la loro volontà di sposarsi; il celebrante, durante la prima fase, raccoglie il desiderio di sposarsi, espresso separatamente dai due partecipanti, per poi in una seconda fase dar loro notizia che il matrimonio è avvenuto. L'analogia con il matrimonio è completata assegnando il ruolo di promesso sposo a ciascun server che partecipa a una transazione; in questo contesto, i server vengono denominati *resource manager* (RM). Il celebrante (o coordinatore) è viceversa associato a un processo, detto *transaction manager* (TM); come in alcune culture, il numero dei partecipanti al matrimonio non è limitato a due, ma è viceversa arbitrario.

Il protocollo di commit a due fasi si svolge tramite un rapido scambio di messaggi fra TM e RM; per rendere il protocollo resistente ai guasti, RM e TM scrivono alcuni nuovi record nei loro log.

16.4.1 Nuovi record nel log

I nuovi record nel log scritti durante il protocollo di commit a due fasi estendono i tipi di record visti nel Paragrafo 12.1.2; sia TM sia RM sono dotati di un proprio log. Il TM scrive:

- il record di `prepare`, che contiene l'identità dei processi RM (cioè il loro identificativo di nodo e di processo). Continuando l'analogia con il matrimonio, questo record corrisponde alle partecipazioni, che vengono scritte prima di un matrimonio;
- il record di `global commit` o di `global abort`. Anticipando il protocollo, notiamo che l'istante di tempo in cui il TM scrive nel suo log il record di `global commit` esprime la decisione, atomica e persistente, di portare a termine con successo l'intera transazione, su tutti i nodi in cui opera, rendendo visibile il suo stato finale;
- il record di `complete`, che è scritto alla conclusione del protocollo di commit a due fasi.

Il processo RM rappresenta una sotto-transazione; come nel contesto centralizzato, ogni RM scrive un record di `begin` all'inizio, seguito da vari record di `insert`, `delete` e `update` che rappresentano le operazioni svolte dalla sotto-transazione. Per quanto concerne il protocollo di commit a due fasi, sull'RM vi è un solo nuovo record:

- il record di `ready`, che indica l'irrevocabile disponibilità di partecipare al protocollo di commit a due fasi contribuendo a una decisione di commit. Su tale record viene scritto anche l'identificativo (numero di nodo e di processo) del TM.

Si noti che il partecipante, una volta scritto il record `ready`, perde ogni autonomia decisionale; l'esito finale della transazione sarà deciso dal TM. Inoltre, un partecipante può scrivere il record `ready` solo quando si trova in uno stato affidabile; quindi, deve aver bloccato con opportuni lock tutte le risorse cui fa accesso e deve aver seguito le regole di WAL e di precedenza al commit nella gestione del suo log.

16.4.2 Protocollo in assenza di guasti

In assenza di guasti, il protocollo di commit a due fasi consiste in una rapida sequenza di scritture sul log e di scambi di messaggi tra TM e RM; nella comunicazione con gli RM, il TM può utilizzare meccanismi di *broadcast*, cioè che trasmettono lo stesso messaggio a molti nodi; deve poi essere in grado di collezionare risposte provenienti da vari nodi. Altrimenti, il TM utilizza una comunicazione seriale con tutti i sistemi RM.

La *prima fase* del protocollo si articola nel modo seguente.

1. Il TM scrive il record di `prepare` nel proprio log e invia un messaggio di `prepare` per informare tutti gli RM dell'inizio del protocollo; imposta poi un timeout, che scatterà in presenza di un eccessivo ritardo nella ricezione di messaggi di risposta dagli RM.

2. Gli RM che sono in uno stato affidabile attendono il messaggio di `prepare`; non appena il messaggio arriva, scrivono sul proprio log il record di `ready` e trasmettono al TM un messaggio di `ready`, che indica la scelta favorevole di partecipazione al commit. Se invece un RM non è in uno stato affidabile, perché si è verificato un guasto di transazione, invia un messaggio `not-ready` e termina il protocollo; però è anche possibile che l'RM realizzi un abort della sotto-transazione in modo autonomo, disfandone gli effetti, e termini prima dell'inizio del protocollo; come vedremo, l'effetto globale di questa situazione sulla transazione è un `global abort`.
3. Il TM colleziona i messaggi di risposta dagli RM. Se da tutti gli RM riceve un messaggio positivo, scrive sul suo log un record di `global commit`; se invece uno o più messaggi sono negativi oppure il timeout scatta senza che il TM abbia ricevuto tutte le risposte, il TM scrive sul suo log un record di `global abort`.

La *seconda fase* del protocollo si articola nel modo seguente.

1. Il TM trasmette la sua decisione globale agli RM; imposta poi un timeout, che scatterà in presenza di un eccessivo ritardo nella ricezione di messaggi di risposta dagli RM.
2. Gli RM che sono in uno stato `ready` attendono il messaggio di decisione; non appena il messaggio arriva, scrivono sul proprio log il record di `commit` oppure di `abort`, che vanno interpretati localmente; inviano poi al TM un messaggio di `acknowledgement` (ack). A questo punto, l'implementazione del commit o dell'abort procede su ciascun server come descritto nel capitolo precedente; in particolare, le pagine modificate dalla transazione vengono scritte nella base di dati dal buffer manager.
3. Il TM colleziona tutti i messaggi di ack dagli RM coinvolti nella seconda fase. Se tutti gli ack arrivano regolarmente, scrive sul suo log il record di `complete`; se invece il timeout scatta senza che il TM abbia ricevuto tutti gli ack, il TM imposta un altro timeout e ripete la trasmissione verso tutti gli RM da cui non ha ricevuto un ack. Questa sequenza si ripete fintantoché tutti gli RM non rispondono inviando il loro ack.

Quindi, l'assenza di comunicazione fra TM e RM durante la prima fase provoca un abort globale, mentre l'assenza di comunicazione fra TM e RM durante la seconda fase provoca una ripetizione delle trasmissioni, fino all'avvenuto ripristino delle condizioni di funzionamento normale. È infatti necessario comunicare a tutti gli RM in stato `ready` l'esito finale delle transazioni. Il protocollo è schematizzato nella Figura 16.8, che descrive lo scambio di messaggi e le scritture sul log del TM e di uno degli RM.

Un RM in stato di `ready` perde la propria autonomia e si affida alla decisione del TM; questa situazione è particolarmente pericolosa, in quanto un guasto del TM durante il protocollo lascia il processo RM in uno stato di incertezza; in questo stato, le risorse acquisite dal processo (tramite meccanismi di lock) sono bloccate. Si corre cioè il rischio di bloccare grandi porzioni della base di dati per lunghi periodi di tempo. L'intervallo di tempo che intercorre tra la scrittura sul log del record `ready` e

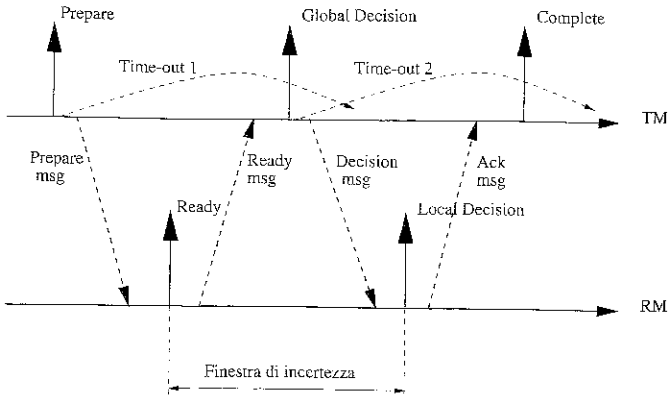

Figura 16.8 Protocollo di commit a due fasi.

la scrittura del record `commit` o `abort` è detto *intervallo di incertezza*; il protocollo è progettato in modo da ridurre tale intervallo al minimo.

In effetti, l'intero insieme delle azioni svolte da processi *client*, *server* e *TM* è descritto in Figura 16.9 per uno specifico RM (ma ovviamente gli RM devono essere almeno due per caratterizzare un contesto distribuito). Il processo client lancia il processo RM e attende che esso svolga il suo compito applicativo (per esempio, cambiare il saldo di un conto corrente); lancia poi, in serie oppure in parallelo, altri processi server su altri nodi. Si noti che nel modello descritto in Figura 16.9 è il client a coordinare un'esecuzione distribuita; alternativamente, il client può indirizzare la sua richiesta a un solo RM e quest'ultimo può lanciare altri processi RM, svolgendo il ruolo di coordinatore della transazione. In entrambi i casi, il client oppure il processo RM coordinatore attendono che tutti gli RM abbiano terminato la loro ela-

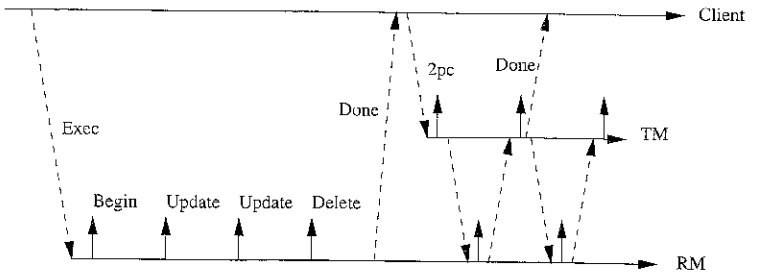

Figura 16.9 Protocollo di commit a due fasi nel contesto di una transazione.

borazione, e poi lanciano il protocollo di commit a due fasi, interagendo con il TM. A questo punto, tutti gli RM sono in attesa di partecipare al commit ricevendo il primo messaggio di `prepare`; il protocollo di commit avviene in modo molto rapido, in quanto sia RM sia TM si limitano a scrivere record nel log e inviare messaggi, minimizzando la finestra di incertezza.

In gran parte delle applicazioni, caratterizzate dall'esecuzione ripetitiva di transazioni che coinvolgono gli stessi elaboratori, i processi client, RM e TM inizializzano i loro collegamenti stabilendo dei meccanismi di comunicazione permanente (sessioni), all'interno dei quali far avvenire molte transazioni; in tal caso, il contesto in cui le transazioni distribuite operano è ottimale dal punto di vista dell'efficienza e robustezza delle trasmissioni. Il concetto di sessione fa riferimento ai protocolli di comunicazione fra processi ed esula dagli scopi del presente testo.

16.4.3 Protocolli di ripristino

Affrontiamo ora le possibili cause di errore che possono compromettere la correttezza del protocollo di commit a due fasi.

Caduta di un partecipante La caduta di un partecipante comporta la perdita del contenuto dei buffer e può quindi lasciare la base di dati in uno stato inconsistente. Come nel Paragrafo 12.1.4, lo stato di quei partecipanti che sono "incerti" (in quanto presenti nel record di `checkpoint` oppure perché il loro inizio segue l'ultimo checkpoint) si deduce leggendo il contenuto del log, supposto memorizzato su memoria stabile. Il protocollo di *ripresa a caldo* ci indica come comportarci in due casi, nei quali il fatto che una transazione sia distribuita oppure centralizzata è del tutto irrilevante:

- quando l'ultimo record scritto nel log è un record che descrive un'azione oppure un record di `abort`, le azioni vanno *disfatte*;
- quando l'ultimo record scritto nel log è un `commit`, le azioni vanno *rifatte*.

Quindi, l'unico caso critico aggiuntivo, dovuto alla presenza di transazioni distribuite, si verifica per quelle transazioni in cui l'ultimo record scritto nel log è di `ready`. In tal caso, il partecipante è *in dubbio* circa l'esito della transazione. Durante il protocollo di recovery a caldo, vengono collezionati in un insieme (detto *READY*) gli identificativi delle transazioni in dubbio; per ciascuna di queste transazioni è necessario richiedere l'esito finale della transazione medesima al processo master; ciò può avvenire per effetto di una richiesta diretta (richiesta di *remote recovery*) dal nodo RM al nodo TM. Alternativamente, l'informazione può essere trasferita dal TM all'RM, a seguito della ripetizione della seconda fase del protocollo (che a intervalli periodici ri-invia la decisione) oppure a seguito di un'esplicita richiesta di effettuare la recovery (come nel protocollo X-Open, che verrà descritto nel seguito).

Caduta del coordinatore La caduta del coordinatore avviene durante la trasmissione dei messaggi e comporta la loro eventuale perdita; è possibile al solito ricavare dal log informazione sullo stato di avanzamento del protocollo, ma non informazione

su quali messaggi siano stati correttamente inviati. Lo stato del TM è caratterizzato dai tre casi elencati di seguito.

- Quando l'ultimo record nel log è un `prepare`, la caduta del TM può effettivamente aver posto alcuni RM in situazione di blocco. Il loro recovery, guidato dal TM, avviene normalmente decidendo per un `global abort`, scrivendo tale record nel log, e poi svolgendo la seconda fase del protocollo. Alternativamente, il TM può ripetere anche la prima fase, sperando che tutti gli RM siano ancora in attesa in stato di `ready`, per poter così decidere un `global commit`.
- Quando l'ultimo record nel log è una `global decision`, la caduta del TM può aver causato una situazione in cui alcuni RM sono stati correttamente avvertiti della decisione e altri sono viceversa rimasti in stato di blocco. In questo caso, il TM deve ripetere la seconda fase, ridistribuendo la decisione a tutti gli RM.
- Quando l'ultimo record nel log è una `complete`, la caduta del coordinatore non ha effetto sulla transazione.

Si noti che la ripetizione della seconda fase, dovuta sia allo scadere dei timeout durante il funzionamento normale, sia al recovery dopo la caduta dell'RM o del TM, può portare un partecipante a ricevere molteplici volte la stessa decisione relativa alla stessa transazione. In tal caso, il partecipante può tranquillamente ignorare la decisione, ma deve comunque rispondere con un `ack`, in modo da far completare il processo di recovery.

Perdita di messaggi e partizionamenti della rete Infine, analizziamo i casi di perdite di messaggi e di partizionamento della rete:

- le perdite di un messaggio `prepare` o del successivo messaggio `ready` non sono distinguibili dal TM; in entrambi i casi scatta un timeout sulla prima fase, e viene presa una decisione di `global abort`;
- le perdite di un messaggio decisione o del successivo messaggio `ack` non sono anch'esse distinguibili; in entrambi i casi scatta un timeout nella seconda fase, e quest'ultima viene ripetuta;
- un *partizionamento della rete* non provoca problemi ulteriori, in quanto la transazione avrà successo solo se il TM e tutti gli RM appartengono, durante le fasi critiche del protocollo, alla stessa partizione.

16.4.4 Ottimizzazioni del commit a due fasi

Il protocollo che abbiamo visto è abbastanza oneroso; in particolare, abbiamo fino a ora assunto che tutte le scritture nel log fossero *sincrone* (effettuate cioè con un'operazione di *force*) in modo da garantirne la persistenza. In effetti, alcune varianti del protocollo consentono di evitare la scrittura sincrona di alcuni record nel log, pur di garantire un meccanismo di scelta per default sull'esito della transazione in caso di caduta del TM. È possibile cioè che il TM, in assenza di informazione circa alcuni partecipanti, indichi per default che questi partecipanti hanno preso una decisione di commit o abort.

Si costruiscono così due varianti del protocollo, dette di *commit presunto* oppure di *abort presunto*. Descriviamo nel seguito il protocollo di abort presunto (*presumed abort*), che per i suoi vantaggi è adottato dalla maggioranza dei DBMS commerciali.

Protocollo di abort presunto Il protocollo di abort presunto si basa sulla seguente regola:

> *A ogni richiesta di* remote recovery *da parte di un partecipante in dubbio sulla cui transazione il TM non abbia informazione viene restituita la decisione di* `abort`.

Come conseguenza, è possibile evitare alcune scritture sincrone di record nel log del TM; in particolare, si può evitare di scrivere con la primitiva *force* i record di `prepare` e `global abort`; in caso di perdita di questi record a causa di una caduta del TM, il TM, seguendo il comportamento di default, darebbe infatti una identica risposta durante la ripresa della transazione. Inoltre, anche il record di `complete` non è critico per l'algoritmo: in alcuni sistemi esso è addirittura omesso, e in genere una sua perdita causa la ripetizione della seconda fase. In conclusione, i soli record di `ready` e `commit`, nel log dell'RM, e `global commit`, nel log del TM, devono essere scritti in modo sincrono, richiedendone la scrittura tramite la primitiva di *force*. Tali record possono essere scritti con operazioni di gruppo, come indicato nel Paragrafo 12.1.3, per ottenere prestazioni più efficienti.

Ottimizzazione "sola lettura" Un'ulteriore ottimizzazione del protocollo di commit a due fasi si verifica quando un partecipante, pur essendo coinvolto in questo protocollo, scopre durante l'esecuzione di aver svolto solo operazioni di lettura ma nessuna operazione di scrittura. In tal caso, il partecipante non deve influenzare l'esito finale della transazione, e di fatto può disinteressarsene. Si noti che i partecipanti per i quali è noto a priori un ruolo di sola lettura possono essere esclusi dal protocollo. L'ottimizzazione del protocollo di un partecipante che scopra di essere di "sola lettura" consiste nel rispondere un messaggio `read-only` (sola lettura) al messaggio di `prepare`, proveniente dal coordinatore. Il partecipante non svolge a quel punto alcuna operazione di scrittura sul log e sospende la esecuzione del protocollo dopo aver spedito il messaggio. Dopo aver ricevuto la risposta di `read-only`, il coordinatore ignora il partecipante nella seconda fase del protocollo.

16.4.5 Altri protocolli di commit

Il principale problema del protocollo di commit a due fasi è la possibilità che un RM resti bloccato a causa della caduta del TM. Per evitare questo problema, sono stati pensati altri protocolli, a tre o quattro fasi, che consentono di evitare il blocco introducendo però una maggior complessità di esecuzione; questi protocolli non hanno avuto grande diffusione ma è interessante discutere brevemente l'idea di fondo che li caratterizza.

Protocollo di commit a quattro fasi Il protocollo di commit a quattro fasi è stato ideato negli anni Novanta dalla *Tandem*, un costruttore di architetture per la gestione di dati basato sull'uso di risorse replicate per ottenere elevata affidabilità. In questo protocollo, il processo TM viene anch'esso replicato da un processo di *backup*, posto su di un nodo diverso. A ogni fase del protocollo, il TM prima informa

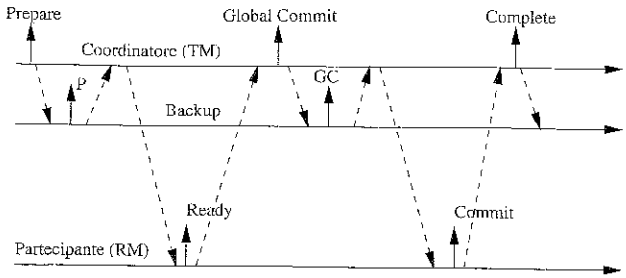

Figura 16.10 Protocollo di commit a quattro fasi.

il backup delle sue decisioni e poi comunica con gli RM, come illustrato in Figura 16.10; in questo modo, il backup conosce esattamente lo stato della transazione e può subentrare al TM in caso di caduta del nodo su cui opera il TM. Quando un backup diviene TM, per prima cosa attiva un suo backup cui comunica le informazioni relative al suo stato, poi continua l'esecuzione delle transazioni in corso.

Protocollo di commit a tre fasi Il protocollo di commit a tre fasi, viceversa, non è stato implementato con successo; l'idea di base è di introdurre una terza fase nel protocollo standard, come illustrato in Figura 16.11. Quindi, dopo che l'RM registra sul log lo stato di `ready`, il TM che dispone di una risposta positiva da tutti gli RM registra uno stato di *pre-commit*, che viene registrato anche dall'RM; quando poi il TM dispone della successiva risposta positiva da tutti gli RM, registra finalmente una decisione di commit. L'aggiunta di una fase nel protocollo consente di reagire alla caduta del TM tramite la scelta di uno qualunque dei partecipanti, il quale può sostituire il TM. Questo partecipante può decidere l'esito della transazione guardando nel suo log; infatti:

- se il partecipante trova nel log un `ready`, sa che nessuno dei partecipanti al protocollo può essere andato oltre la condizione di *pre-commit*, dalla quale può ricevere la decisione di *abort*; quindi, il partecipante registra una decisione di *global abort* e la comunica agli altri partecipanti;

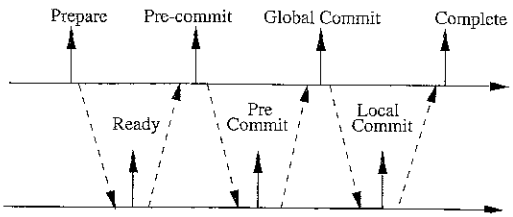

Figura 16.11 Protocollo di commit a tre fasi.

- se invece il partecipante trova nel log un `pre-commit`, sa che tutti gli altri partecipanti sono perlomeno nello stato di *ready*, dunque può prendere la decisione di *commit*; quindi, il partecipante registra una decisione di *global commit* e la comunica agli altri partecipanti.

Il protocollo di commit a tre fasi ha però gravi inconvenienti, che lo rendono inutilizzabile in pratica. In primo luogo, esso allunga la finestra di incertezza, e perciò rende la condizione di blocco più probabile. In secondo luogo, l'atomicità può essere persa qualora si verifichi una partizione della rete e vengano scelti *due o più* partecipanti come coordinatori per terminare il protocollo (uno in ciascuna partizione). Per risolvere anche questo problema, occorre assicurarsi che la decisione di commit o abort venga presa da un numero sufficiente di partecipanti tale da garantire che una diversa decisione non possa essere presa nelle altre partizioni, tramite meccanismi basati sull'uso di votazioni e di quorum.

Paxos Commit Recentemente, il concetto di protocollo di commit è stato rivisitato nel contesto più generale degli algoritmi di consenso, necessari in presenza di guasti in un sistema distribuito affinché un gruppo di nodi raggiungano un accordo su di un valore (per esempio, relativo all'identità di un nodo "coordinatore"). Nel caso di un algoritmo di commit, si tratta di raggiungere un consenso su una variabile che può assumere due valori (commit o rollback).

In particolare, il protocollo "Paxos" propone una tecnica per costruire consenso che suddivide i nodi in tre categorie: proponenti, accettori, riceventi; il protocollo opera assumendo che i messaggi possano essere persi ma che non possano essere alterati, e garantisce che tutti i riceventi giungano asintoticamente allo stesso valore purché i nodi abbiano guasti temporanei e tutti i messaggi inviati vengano effettivamente ricevuti entro un tempo predefinito. Il protocollo prevede tre fasi, di cui la prima serve a eleggere o confermare il coordinatore, la seconda fase serve a scegliere un valore tra gli accettori, e la terza propaga tale valore ai riceventi. La prima fase del protocollo è in grado di riconoscere la presenza di più coordinatori, ciascuno inconsapevole dell'altro, e di risolvere questo problema eleggendone uno; assumendo che vi siano $2F + 1$ nodi accettori, questa fase tollera F guasti.

Questa caratteristica rende il protocollo di consenso interessante anche per risolvere il problema del commit distribuito. La versione "Paxos" del protocollo di commit prevede la presenza di N riceventi e $2F + 1$ nodi accettori, di cui però solo $F + 1$ coinvolti in ogni operazione di commit; per ragioni di efficienza, si usano gli altri accettori solo in caso di malfunzionamenti. Per coerenza con i protocolli di commit descritti precedentemente, assumiamo che i riceventi abbiano il ruolo di partecipante (RM), siano distinti dagli accettori, e il loro ruolo sia sempre passivo. Il disegno del protocollo inoltre trascura la scrittura del record di decisione locale e la registrazione su memoria stabile del suo completamento. In Figura 16.12 si mostra il funzionamento del protocollo. Uno degli accettori è il coordinatore, che manda N messaggi di `prepare` ai partecipanti; ogni partecipante, dopo aver registrato in memoria stabile il consenso al commit, comunica in risposta il suo stato di `ready` al coordinatore e agli altri F accettori; ciascun accettore comunica poi il proprio stato e quello dei partecipanti al coordinatore, tramite altri F messaggi. A questo punto, coordinatore

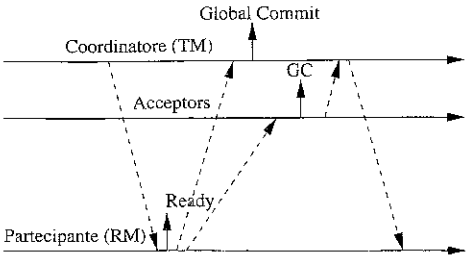

Figura 16.12 Protocollo di commit Paxos.

e accettori sono $F + 1$ nodi che conoscono lo stato della transazione, e il protocollo risulta non bloccante di fronte ai guasti di F di loro; il protocollo di consenso viene usato nel caso vi sia una caduta del coordinatore per eleggerne un altro. Il numero di passi è minimizzato, in quanto il protocollo di commit Paxos richiede in assenza di guasti $(N + 1) \cdot F$ messaggi e F scritture su memoria stabile in più rispetto a una versione standard del protocollo di commit a due fasi che usi le stesse assunzioni circa le trasmissioni e i ruoli dei nodi; il commit Paxos degenera al commit standard per $F = 0$.

Questo protocollo di per sé non risolve il problema del partizionamento della rete, in quanto si assume che ogni trasmissione di messaggio venga prima o poi portata a termine; viene però garantita la possibilità di eseguire transazioni senza blocco in presenza di F nodi guasti, il che viene garantito in modo totalmente distribuito, senza cioè che vi siano nodi critici. La definizione dell'algoritmo è recente, ma si prevede che esso avrà utilizzazioni commerciali nell'ambito dei DBMS. Già esistono significative applicazioni di Paxos nell'ambito delle infrastrutture informatiche di maggiori dimensioni.

16.4.6 Interoperabilità del commit a due fasi: X-Open DTP

I protocolli di commit illustrati fino a ora consentono varie ottimizzazioni e varie soluzioni per quanto concerne le procedure di ripristino o la gestione delle situazioni di blocco, ed esistono perciò varie realizzazioni proprietarie del protocollo, cioè disponibili nel DBMS distribuito di uno specifico prodotto commerciale; tali versioni proprietarie consentono di realizzare transazioni distribuite su sistemi omogenei (con lo stesso DBMS) ma non gestiscono transazioni eterogenee, in cui cioè sono coinvolti DBMS differenti. Per questo scopo, è stata realizzata una versione standard del protocollo di commit a due fasi.

Il protocollo X-Open DTP (*Distributed Transaction Processing*) garantisce l'interoperabilità di computazioni transazionali su DBMS di venditori differenti. Questo protocollo è stato definito nei primi anni Novanta ed è tuttora un elemento cruciale per la gestione di transazioni distribuite in ambiente eterogeneo. L'architettura di X-Open DTP assume la presenza di un processo client, vari processi RM e un processo TM,

che interagiscono come descritto in Figura 16.9 (già discussa). Il protocollo consta di due interfacce principali:

- l'interfaccia fra client e TM, chiamata *TM-interface*;
- l'interfaccia fra TM e RM, chiamata *XA-interface*.

L'aspetto rilevante dello standard è che i costruttori di DBMS, per garantire che i loro server siano accessibili dai TM, devono garantire la disponibilità della XA-interface. Per questo motivo, vari sistemi relazionali (per esempio IBM DB2, Oracle Server o Microsoft SQLServer) hanno, in aggiunta a una realizzazione proprietaria del protocollo di commit a due fasi utilizzata per realizzare applicazioni relazionali omogenee, anche un'implementazione dell'interfaccia XA, utilizzata per realizzare applicazioni transazionali eterogenee. Lo standard X-Open è adottato da vari prodotti specializzati nella gestione di transazioni, tra cui DTC (*Distributed Transaction Coordinator*, prodotto dalla Microsoft) e *Tuxedo* (prodotto dalla BEA), che forniscono il componente TM.

Le caratteristiche principali dello standard X-Open DTP sono le seguenti.

- Lo standard prevede che gli RM siano totalmente passivi; tutto il controllo del protocollo è concentrato nel TM, che attiva le funzioni degli RM, rese disponibili come una libreria di primitive richiamabili remotamente.
- Il protocollo realizza un commit a due fasi con le ottimizzazioni di *abort presunto* e *sola lettura*, descritte in precedenza.
- Il protocollo prevede *decisioni euristiche*, che in presenza di guasti consentono l'evoluzione della transazione sotto il controllo dell'operatore. Tali decisioni euristiche possono causare una perdita di atomicità, e in tal caso il protocollo garantisce che i processi client vengano notificati.

La TM-interface consta delle seguenti procedure:

- `tm_init` e `tm_exit`, per iniziare e terminare il dialogo client-TM;
- `tm_open`, cui segue l'apertura di una sessione del TM con i vari processi RM. La sessione consente di stabilire un supporto stabile alla comunicazione TM-RM, che può essere usato da molteplici transazioni. La sessione viene chiusa a fronte della richiesta della primitiva `tm_term` da parte del client;
- `tm_begin`, per iniziare una transazione;
- `tm_commit`, per richiedere un commit globale.

La XA-interface consta delle seguenti procedure:

- `xa_open` e `xa_close`, per inizializzare e concludere una sessione tra TM e vari RM;
- `xa_start` e `xa_end`, per far partire una nuova transazione RM e per completarla;
- `xa_precom`, per richiedere all'RM di svolgere la prima fase del protocollo di commit. Il processo RM può rispondere positivamente alla chiamata solo se garantisce la sua affidabilità e la scrittura del record `ready` nel suo log;

- `xa_commit` e `xa_abort`, per comunicare la decisione globale relativa alla transazione;
- `xa_recover`, per iniziare una procedura di ripristino che si rende necessaria dopo un'eventuale caduta di un processo (TM o RM);
- `xa_forget`, per consentire a un RM di dimenticare transazioni decise in modo euristico.

Una tipica interazione fra client, TM e RM è illustrata in Figura 16.13.

Quando un RM è bloccato per la caduta del TM, un operatore può imporre una *decisione euristica* (in genere l'abort), consentendo così il rilascio delle risorse. La procedura di ripristino è guidata dal TM, che in un istante successivo al guasto chiama l'RM. A questo punto, il processo RM consulta il proprio log e indica tre insiemi di transazioni:

- le transazioni *in dubbio*;
- le transazioni decise con un *commit euristico*;
- le transazioni decise con un *abort euristico*.

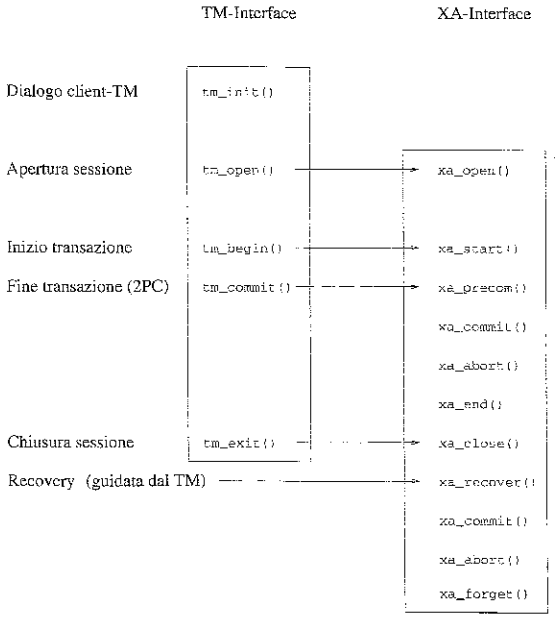

Figura 16.13 Interazione fra client, TM e server con il protocollo X-Open DTP.

Il processo TM comunica alle transazioni in dubbio il loro esito effettivo (commit o abort) e verifica in base al proprio log se le decisioni euristiche siano in conflitto con quelle comunicate al client; se ciò accade, notifica il client informandolo dell'inconsistenza. In ogni caso, le transazioni decise in modo euristico vengono poi dimenticate dal RM a seguito di una primitiva di `xa_forget`, inviata dal TM.

16.5 Basi di dati replicate

La replicazione dei dati è un servizio essenziale per la realizzazione di molte applicazioni distribuite. Questo servizio è garantito da specifici prodotti, detti *replicatori dei dati* (*data replicator*), che consentono la creazione di copie di tabelle o sottoinsiemi di tabelle in un generico contesto distribuito.

La funzione principale di un replicatore dei dati è di mantenere l'allineamento tra le copie. In genere, per ogni dato esiste una *copia principale* e varie *copie secondarie*, e gli aggiornamenti vengono propagati dalla copia principale alle secondarie in modo asincrono, senza cioè utilizzare il protocollo di commit a due fasi. La propagazione può essere *incrementale*, quando dalla copia principale alla copia secondaria viene inviata una descrizione delle variazioni che vengono ri-applicate alla copia secondaria; alternativamente, l'intera copia secondaria viene periodicamente ri-materializzata completamente a partire dall'intera copia primaria. Il replicatore dei dati realizza queste funzioni in modo trasparente, senza cioè alterare le applicazioni che operano sulla copia principale.

L'uso della replicazione rende un sistema meno sensibile ai guasti, in quanto se una copia principale non è disponibile (perché il corrispondente sistema è guasto), è possibile perlomeno accedere a una sua copia. Una tipica architettura distribuita con dati replicati è illustrata in Figura 16.14; questa architettura è stata introdotta per la gestione di applicazioni finanziarie (compravendita di titoli) in cui l'eventuale non disponibilità del sistema informatico causa una perdita economica enorme. L'architettura prevede la presenza di due centri speculari: ciascuno gestisce l'intera base di dati, che per metà è copia principale e per metà è copia secondaria. Le transazioni vengono inviate alla copia principale e da questa redirette, in modo differito, alla copia secondaria. Ciascun "punto di accesso" al sistema è collegato a entrambi i centri. In caso di guasto che coinvolga l'intero centro, il sistema è capace di commutare quasi istantaneamente tutte le transazioni sull'altro centro, che è dimensionato per "reggere" bene l'intero carico applicativo. Quando il problema è risolto, il gestore della replicazione ripristina la correttezza dei dati in modo trasparente nonché il funzionamento dei due centri.

Inoltre, come abbiamo visto nell'introdurre la nozione di memoria stabile (Paragrafo 12.1.1), la ridondanza è uno dei metodi per garantire la persistenza delle informazioni a fronte di guasti. Esistono alcuni sistemi informativi particolarmente critici che usano la replicazione dei dati come forma sofisticata di backup; per esempio, i sistemi informativi di alcune banche californiane, collocati in un'area sismica, hanno tutti i loro dati replicati in un sistema identico, collocato però in una regione differente. In questo caso, il sistema-copia non è in grado di reggere il normale carico applicativo.

Replicazione, distribuzione e frammentazione dei dati possono essere combinate; per esempio, il sistema informativo che descriveva i vari componenti dell'ar-

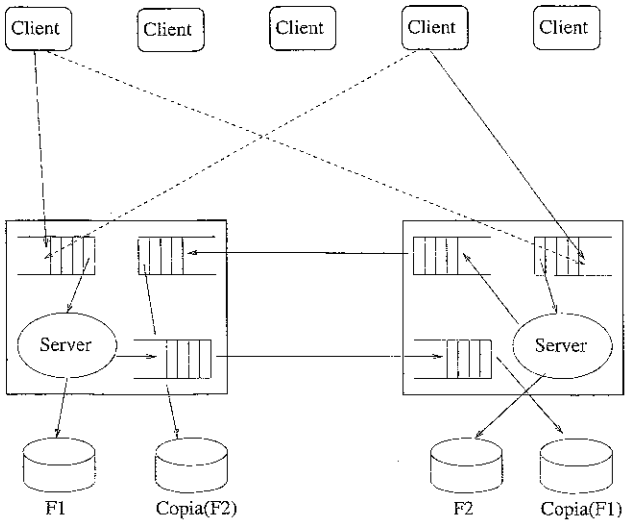

Figura 16.14 Esempio di architettura con dati replicati.

chitettura Tandem è stato realizzato, verso la metà degli anni Ottanta, componendo queste tre tecnologie. La Tandem disponeva di una decina di stabilimenti distribuiti in varie parti del mondo, ciascuno specializzato nel realizzare una parte specifica dell'architettura del calcolatore (tastiere, schermi, CPU-box e così via). La cosiddetta "distinta-base" delle parti disponibili in azienda è stata modellata tramite un opportuno insieme di tabelle; queste tabelle sono state frammentate in modo da riflettere la distribuzione fisica del processo di costruzione dei componenti e allocate in modo ridondante, disponendo la copia principale del frammento presso il nodo responsabile del processo produttivo relativo ai dati contenuti nel frammento. Con questa configurazione, illustrata in Figura 16.15, le modifiche vengono sempre dirette alla copia principale; il gestore della replicazione colleziona un insieme (batch) di modifiche e le applica agli altri frammenti in modo asincrono.

16.5.1 Nuove funzionalità dei gestori della replicazione

Alcuni prodotti per la replicazione dei dati trattano anche il problema, assai complesso, di garantire la cosiddetta *replicazione simmetrica*, in cui cioè le modifiche possono essere effettuate su qualunque copia, con una situazione "alla pari" (*peer-to-peer*) fra le copie. In questo caso, chiaramente, è possibile introdurre conflitti, in quanto due copie della stessa informazione vengono gestite in modo concorrente *senza* controllo di concorrenza; si verificano cioè le anomalie descritte nel Para-

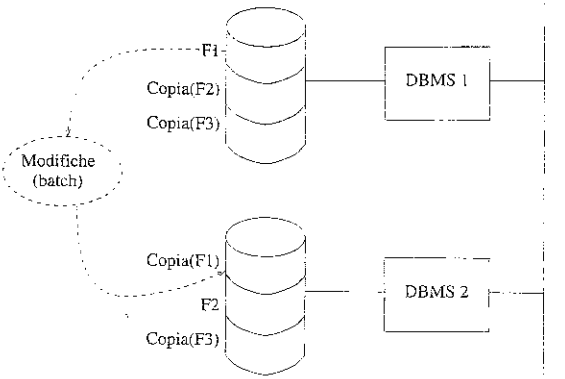

Figura 16.15 Sistema informativo della Tandem.

grafo 12.2.2. Per controllare questo fenomeno, vengono sviluppate tecniche capaci perlomeno di rivelare le anomalie a posteriori e segnalarle a un gestore della base di dati, affinché le inconsistenze possano essere gestite in un modo che dipende dalla specifica applicazione.

Questa problematica ha assunto nuovo impulso nel contesto di sistemi distribuiti *mobili*, in cui il collegamento con la base di dati può essere discontinuo. È il caso di compagnie di distribuzione e vendita al minuto, i cui agenti possono collegarsi sporadicamente con la casa madre per caricare la disponibilità di merci e scaricare gli ordini ricevuti; gli agenti si spostano per recarsi sul luogo dei clienti e utilizzano personal computer mobili come macchine client. In questo caso, un venditore può rimanere disconnesso dalla base di dati per molte ore, accettando transazioni sulla sua copia dei dati, che vanno "riconciliate" con la copia principale al momento in cui il venditore si ricollega a essa.

16.6 Cooperazione di basi di dati eterogenee e autonome

Le architetture viste fino a ora consentono di sfruttare i vantaggi dovuti alla distribuzione e alla replicazione tenendo in gran parte nascosti questi aspetti sia all'utente finale sia al programmatore; quest'ultimo infatti può programmare le applicazioni prescindendo in larga parte da queste caratteristiche. Architetture di questo tipo consentono una progettazione "centralizzata" della base di dati, vista come risorsa aziendale, utilizzando le dimensioni della distribuzione, del parallelismo e della re-

plicazione solo in fase di realizzazione del sistema, essenzialmente per migliorare le sue caratteristiche di funzionalità e di prestazione.

In un contesto reale accade però molto spesso che queste condizioni vengano meno, per esempio per la presenza di soluzioni già realizzate (i cosiddetti sistemi *legacy* che vengono "ereditati" dal passato); in altri casi, le esigenze applicative stesse sono tali da non giustificare un'integrazione delle basi di dati, mentre è necessario realizzare applicazioni che usano dati provenienti da più sistemi distinti. Si parla in questo caso di applicazioni che nascono dalla *cooperazione* fra sistemi, facendo uso di servizi applicativi messi a disposizione e gestiti da soggetti diversi.

Nella maggior parte dei casi, la cooperazione è centrata sui processi: i sistemi offrono l'un l'altro servizi, attraverso scambio di messaggi, informazioni, documenti, o innescamento di attività, senza però mettere a disposizione dati remoti. Questo argomento non è caratteristico di un libro di testo sulle basi di dati e sarà brevemente trattato nel Capitolo 18, parlando dei "Web Service". La cooperazione basata sui processi si basa tipicamente sull'uso di moduli di sistema, detti *wrapper*, che "incapsulano" le caratteristiche dei sistemi e le rendono richiamabili tramite funzioni di alto livello, nascondendo del tutto i dati e la loro organizzazione.

Nel seguito, concentriamo invece l'attenzione sulla *cooperazione basata sui dati*, in cui si vogliono rendere visibili i dati di sistemi eterogenei e autonomi. Si tratta di un caso meno frequente ma pur sempre significativo; un vantaggio di questo approccio è la maggior libertà d'azione lasciata all'applicazione, che può accedere ai dati in un modo qualsiasi invece che doversi adeguare ai processi realizzati dal wrapper. I sistemi che prendiamo in considerazione sono realizzati con tecnologie diverse e gestiti da enti e soggetti diversi; in genere, ciascun sistema dovrà continuare a soddisfare le richieste di utenti locali, pur concorrendo a eseguire funzionalità globali.

Autonomia e eterogeneità non facilitano lo sviluppo dei progetti di cooperazione, e vengono viste come ostacoli al processo stesso, da rimuovere attraverso opportune iniziative di standardizzazione (per esempio degli ambienti operativi) o di razionalizzazione (per esempio integrazione delle basi di dati). In molti casi, però, mancano autorità di coordinamento fra i vari soggetti; spesso le esigenze di ciascuno dei sistemi componenti variano nel tempo, e quindi obiettivi eccessivamente ambiziosi di integrazione e omogeneizzazione sono destinati a generare faticosi processi di allineamento.

La realizzazione di architetture per la cooperazione centrate sui dati richiede di effettuare le operazioni di traduzione di query e conversione di formati dei dati necessarie per far parlare sistemi diversi. Le soluzioni possibili oscillano fra due estremi: fornire un modulo di sistema che si occupa completamente di questo aspetto, offrendo una visione di alto livello della cooperazione, oppure far gestire direttamente dal client, tramite i protocolli di interoperabilità (ODBC e JDBC), le attività di integrazione. Tali soluzioni verranno descritte nel seguito.

Inoltre, un'ulteriore possibilità di accesso a basi di dati autonome ed eterogenee viene offerta dai *data warehouse*, che collezionano informazioni estratte da sistemi differenti e offrono ai loro utenti una visione integrata; in questo caso però i dati non sono "attuali", ma vengono prelevati dalle varie basi di dati e caricati nel data warehouse con un processo di estrazione tipicamente periodico. Questa terza possibilità d'accesso verrà illustrata nel Capitolo 19.

Soluzione con multidatabase La prima soluzione, rappresentata in Figura 16.16, introduce un modulo di sistema detto *multidatabase*; in questa soluzione le singole basi di dati partecipanti continuano a essere utilizzate dai rispettivi utenti (programmi o utenti finali). Ai singoli sistemi accedono anche moduli, chiamati *mediatori*, che trasformano e filtrano gli accessi, mostrando solo la porzione di base di dati che si desidera esportare, e la mettono a disposizione di un *sistema multidatabase*, che realizza l'integrazione e mette a disposizione degli utenti una visione integrata, come se la base di dati fosse unica. In particolare, il sistema multidatabase ha un proprio modello dei dati (tipicamente relazionale) e offre un proprio linguaggio di interrogazione (tipicamente SQL).

In genere, i sistemi multidatabase consentono di interrogare le basi di dati sottostanti, ma non di svolgere modifiche. Questa limitazione non dipende tanto dalla tecnologia (abbiamo visto che, pur con qualche complessità, si possono utilizzare protocolli standard di commit), quanto piuttosto da esigenze applicative. Infatti, ciascun DBMS che concorre all'applicazione cooperativa è un ambiente autonomo, e come tale non disponibile alla possibilità di accettare modifiche attivate e controllate all'esterno del sistema.

I sistemi multidatabase sono moduli software capaci di trasformare interrogazioni e dati; in particolare, possono contenere dei meccanismi per scegliere da quale DBMS estrarre i dati prioritariamente, oppure per definire politiche da seguire in presenza di conflitti sui valori dei dati, cioè quando lo stesso dato è presente, con valori differenti, in DBMS diversi.

I principali DBMS attualmente sul mercato prevedono moduli molto articolati, con una pluralità di funzionalità, che permettono in buona sostanza di realizzare soluzioni come quella illustrata qui. In particolare sono disponibili funzionalità per:

- accesso (ove possibile trasparente, cioè con istruzioni SQL) a dati esterni al DBMS, gestiti da altri DBMS o da strumenti software diversi, quali gestori di documenti

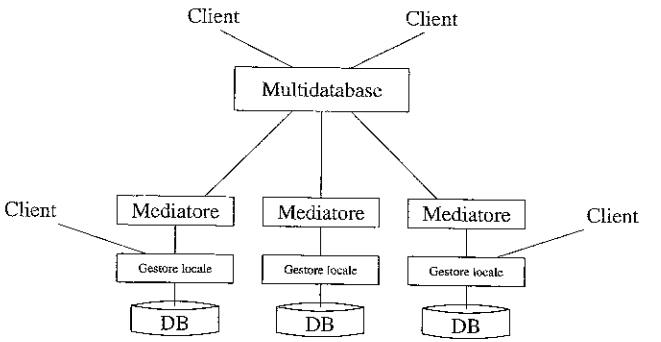

Figura 16.16 Architettura di un sistema multidatabase.

XML, fogli elettronici, applicazioni per la posta elettronica o gestori di file di tipo legacy;
- ottimizzazione delle interrogazioni e gestione delle transazioni sulle sorgenti (ove possibile, con la dovuta disponibilità di funzioni sulle sorgenti stesse);
- conversione e filtraggio dei dati, operazioni spesso indicate con la sigla *ETL* (per *Extract, Transform, Load*) di particolare interesse soprattutto nel contesto del data warehousing, come vedremo nel Capitolo 19;
- integrazione anche attraverso processi e Web Service;
- gestione di dati replicati, con funzionalità per la pianificazione della propagazione degli aggiornamenti;
- gestione di comunicazione asincrona fra i vari componenti, attraverso scambi di messaggi, con componenti che vanno sotto il nome di *Message Oriented Middleware* MOM) e che possono anche prevedere gestori di code e di bilanciamento del carico.

Integrazione dal lato del client La soluzione architetturale che si contrappone ai multidatabase vede l'attività di integrazione svolta solamente dalla parte del client, come illustrato in Figura 16.17. La differenza sostanziale rispetto al caso precedente è costituita dal fatto che in questa architettura non compare un "gestore globale", e le integrazioni sono realizzate esplicitamente dall'applicazione (client) che interagisce con i vari sistemi, utilizzando le interfacce applicative offerte da ciascuno di essi. Conversioni di query e formati devono avvenire sul client e in modo specifico per la particolare applicazione; pertanto, l'architettura ha un basso grado di trasparenza e deve essere programmata completamente usando gli standard di interoperabilità descritti nel Paragrafo 9.2.

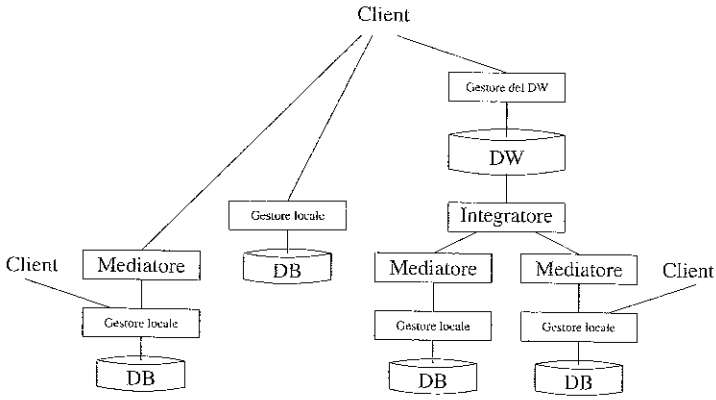

Figura 16.17 Architettura con integrazione dal lato del client.

Nell'esempio, vengono integrate tre fonti: una base di dati esterna, una base di dati locale e un data warehouse (che a sua volta utilizza due fonti di informazioni). L'esempio illustra che sono in effetti possibili molte alternative e molte tipologie intermedie rispetto alle due architetture che abbiamo discusso.

Note bibliografiche

Anche questo capitolo fa riferimento ai testi *Transaction Processing Systems*, di Gray e Reuter, e *Distributed Databases: Principles and Systems*, di Ceri e Pelagatti, citati nei Capitoli 11 e 12. Le applicazioni delle basi di dati distribuite sono descritte nell'articolo: "Distributed Databases: Four Case Studies", di Gray e Anderton [119]; gli algoritmi di commit a due fasi, e in particolare le loro ottimizzazioni, la standardizzazione in X-Open e il loro uso nell'ambito dei sistemi commerciali, sono diffusamente descritti nell'articolo: "Two-phase Commit Organizations in a Commercial Distributed Environment", di Samaras, Bitton, Citron e Mohan [139]; il protocollo di commit Paxos è dovuto a Gray e Lamport [121]; l'algoritmo di rilevazione distribuita dei deadlock è di Obermark [137] e gli orologi di Lamport sono definiti nel classico articolo "Time, Clocks and the Ordering of Events in a Distributed System", di Lamport [129].

Le tematiche relative alla cooperazione di basi di dati possono essere approfondite nei testi di Brodie e Stonebraker [15] e Kim [127], e negli articoli di Bernstein [95] e Sheth e Larson [142].

Esercizi

Soluzioni sul sito http://www.ateneonline.it/atzeni

16.1 Si consideri la base di dati:

PRODUZIONE(NumeroSerie,TipoParte,Modello,Qta,Macchina)
PRELIEVO(NumeroSerie,Lotto,Cliente,Venditore,Ammontare)
CLIENTE(Nome,Città,Indirizzo)
VENDITORE(Nome,Città,Indirizzo)

Progettare la frammentazione orizzontale delle tabelle PRODUZIONE e PRELIEVO in base al tipo di parte (che assume quattro valori: "Tastiera", "Schermo", "CPU" e "Cablaggio"), prevedendo quattro stabilimenti di produzione disposti a Milano, Torino, Roma e Napoli, e delle tabelle CLIENTE e VENDITORE in base a tre bacini di vendita, centrati su Torino, Milano e Roma; si supponga che le vendite siano distribuite per bacini geografici (quindi, clienti di Milano sono serviti solo da venditori di Milano; si assuma che il bacino di vendita di Roma comprenda anche Napoli) e che ciascuna area geografica abbia una propria base di dati (cioè che sia disponibile una base di dati a Milano, Torino, Roma e Napoli). Esprimere a livello di trasparenza di frammentazione, di allocazione e di linguaggio le interrogazioni di seguito elencate.

1. Determinare la quantità disponibile del prodotto 77Y6878.
2. Determinare i clienti che hanno comprato qualche lotto dal rivenditore Bianchi, che ha ufficio a Roma.
3. Determinare le macchine utilizzate per la produzione dei pezzi di tipo "Tastiera" venduti al cliente Rossi.

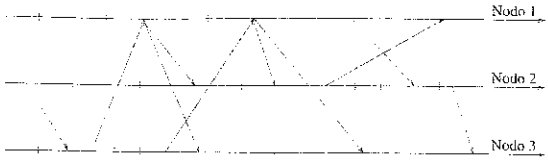

Figura 16.18 Esercizio 16.2.

4. Modificare l'indirizzo del rivenditore Rossi, che si trasferisce da "Via Po 45" di Milano a "Viale Trastevere 150" di Roma.
5. Calcolare la somma degli importi degli ordini ricevuti a Milano, Torino e Roma (si noti che anche le funzioni aggregate sono distribuibili).

Inventare poi, ipotizzando di usare il DBMS di "Milano", una richiesta remota, una transazione remota, una transazione distribuita e una richiesta distribuita.

16.2 Assegnare i timestamp agli eventi descritti in Figura 16.18 con il metodo di Lamport, e indicare quali eventi sono pseudo-simultanei, cioè sono eventi che non possono essere ordinati in base ai messaggi scambiati.

16.3 Date le condizioni di attesa illustrate in Figura 16.19, determinare con l'algoritmo di ricerca distribuita le condizioni di deadlock, con due diverse ipotesi di condizione di attesa relative al nodo 4.

16.4 Descrivere come si modifica il protocollo di ripresa a caldo tenendo presente che alcune sotto-transazioni distribuite possono essere in stato `ready`.

16.5 Applicare il protocollo di ripresa a caldo dopo la caduta di un nodo assumendo un algoritmo di commit a due fasi, a fronte del seguente input (ove $r(t_i)$ indica la presenza di un record `ready`):

$b(t_1)$, $b(t_2)$, $b(t_3)$, $i(t_1, o_1, a_1)$, $d(t_2, o_2, b_2)$, $b(t_4)$, $r(t_1)$, $u(t_4, o_3, b_3, a_3)$, $c(t_1)$, $ck(t_2, t_3, t_4)$, $b(t_5)$, $b(t_6)$, $u(t_5, o_5, b_5, a_5)$, $r(t_5)$, $b(t_7)$, $u(t_7, o_6, b_6, a_6)$, $b(t_8)$, $u(t_6, o_1, b_7, a_7)$, $a(t_7)$, $r(t_6)$, guasto

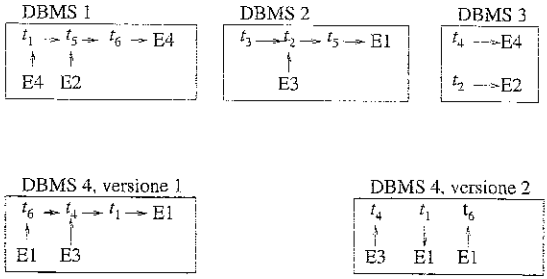

Figura 16.19 Esercizio 16.3.

16.6 Descrivere il protocollo di ripresa a caldo dopo la caduta di un nodo assumendo un algoritmo di commit a tre fasi, a fronte del seguente input (ove $PC(Ti)$ indica la presenza di un record `pre-commit`):

$b(t_1)$, $b(t_2)$, $b(t_3)$, $i(t_1, o_1, a_1)$, $d(t_2, o_2, b_2)$, $b(t_4)$, $r(t_1)$, $u(t_4, o_3, b_3, a_3)$, $pc(t_1)$, $c(t_1)$, $ck(t_2, t_3, t_4)$, $b(t_5)$, $b(t_6)$, $u(t_5, o_5, b_5, a_5)$, $r(t_5)$, $b(t_7)$, $u(t_7, o_6, b_6, a_6)$, $u(t_6, o_3, b_7, a_7)$, $b(t_8)$, $pc(t_5)$, $a(t_7)$, $r(t_6)$, guasto

16.7 Descrivere un esempio di comportamento di base di dati replicata in cui si verifichi un disallineamento fra i dati.

16.8 Descrivere un esempio di comportamento di base di dati con replicazione simmetrica in cui si verifichi una inconsistenza dei dati.

17 Basi di dati parallele e architetture per il cloud

Il parallelismo è una dimensione estremamente importante della tecnologia delle basi di dati e destinata ad assumere un ruolo centrale nell'evoluzione della tecnologia. A livello di singolo sistema di calcolo, osserviamo che le moderne architetture informatiche sono sempre caratterizzate dalla presenza di un certo numero di unità di elaborazione, in numero crescente nel tempo. Aldilà dell'ambito di un singolo calcolatore, le architetture dei moderni impianti informatici vedono la presenza di un gran numero di apparecchiature collegate da reti ad alte prestazioni che condividono un ricco insieme di risorse. La costruzione di sistemi per la gestione di basi di dati ad alte prestazioni e scalabili non può prescindere dalla definizione di modalità per sfruttare la capacità di elaborazioni di molteplici dispositivi. Dal punto di vist'architetturale, il parallelismo è possibile sia con architetture multiprocessore a memoria condivisa (*shared memory*, detti anche sistemi *Symmetric Multi-Processing*, SMP), sia con architetture multiprocessore con memorie separate (*shared nothing*, normalmente chiamati *cluster*), ovviamente con soluzioni tecniche e livelli di costo assai differenti.

A livello più alto, una delle evoluzioni recenti della tecnologia informatica è stato lo sviluppo di architetture per il *cloud computing*. Queste architetture da un lato offrono delle soluzioni che adattano tecniche e metodi che caratterizzano da diverso tempo il mondo delle basi di dati, d'altra parte offrono delle opportunità assai interessanti per la realizzazione di sistemi per la gestione di grandi basi di dati con caratteristiche di affidabilità ed elasticità che sono assai interessanti e immediatamente disponibili anche in contesti di piccole e medie dimensioni che avrebbero difficoltà ad avere accesso a infrastrutture di questo tipo se non esistesse un mercato di servizi cloud.

In questo capitolo tratteremo dapprima le caratteristiche del parallelismo facendo riferimento a classici sistemi multiprocessore e a cluster. Analizzeremo poi le caratteristiche principali delle architetture per il cloud computing, introducendo il paradigma MapReduce. Analizzeremo infine le caratteristiche del sistema Apache Pig, che rappresenta una soluzione per l'elaborazione dei dati in un contesto cloud.

17.1 Parallelismo

Il motivo del successo del parallelismo nelle basi di dati è che le computazioni svolte da una base di dati si prestano a essere eseguite in parallelo con grande efficienza. Per esempio, una scansione completa di una grande base di dati può essere svolta tramite n scansioni, ciascuna relativa a una frazione della base di dati; a patto di separare il file corrispondente in n parti uguali, allocate su dispositivi differenti, si ottengono tempi di risposta che approssimano il valore ideale di $(1/n)$ rispetto al tempo di risposta iniziale. In generale, la gestione dei dati avviene tramite operazioni ripetitive, che si prestano a essere parallelizzate con prestazioni ideali.

Vi sono, in particolare, due tipologie di parallelismo.

- Il parallelismo si dice *inter-query* quando si eseguono interrogazioni diverse in parallelo. In tal caso, il carico cui il DBMS è sottoposto è tipicamente caratterizzato da molteplici transazioni molto semplici, che però vengono presentate con una frequenza assai elevata (anche migliaia di transazioni al secondo). Questo parallelismo è particolarmente utile quando il DBMS gestisce transazioni in linea (è cioè un sistema OLTP).
- Il parallelismo si dice *intra-query* quando si eseguono parti della stessa interrogazione in parallelo; in tal caso, il carico cui il DBMS è sottoposto è tipicamente caratterizzato da poche interrogazioni assai complesse, quindi è opportuno suddividere l'interrogazione in varie sotto-interrogazioni parziali, da affidarsi a vari processori. Questo parallelismo è cruciale quando il DBMS gestisce transazioni per l'analisi dei dati (è cioè un sistema OLAP), ma porta comunque benefici in tutti gli scenari applicativi.

In entrambi i casi, il parallelismo consente che a ciascun processore venga indirizzata una frazione del carico. Nel parallelismo inter-query, tipicamente caratterizzato da un forte carico transazionale ma da un limitato numero di servizi offerti, il parallelismo viene introdotto moltiplicando il numero di processi server e allocando su ciascun processore un numero ideale di questi processi. In molti casi, le interrogazioni vengono raccolte da un processo *dispatcher*, il cui unico compito è di ridirigere ciascuna interrogazione verso uno dei processi server, per esempio mirando a pareggiare il carico su ciascun processore, oppure verso il processore che può fare accesso in modo ottimale ai dati coinvolti nell'interrogazione.

Il parallelismo intra-query è tipicamente caratterizzato da interrogazioni complesse, che coinvolgono cioè molti operatori e si valutano su basi di dati di grosse dimensioni; in genere, si dedica a una determinata interrogazione un insieme ben definito di processi, e talvolta le interrogazioni si eseguono una dopo l'altra, utilizzando l'intero sistema multiprocessore per ciascuna interrogazione. Per sfruttare il parallelismo intra-query, l'ottimizzatore deve individuare una decomposizione dell'interrogazione in sotto-interrogazioni e prevedere le modalità di coordinamento e sincronizzazione fra di esse. Le sotto-interrogazioni possono limitarsi all'esecuzione distribuita di una specifica operazione (per esempio: scansione, sort, join) oppure possono essere più complesse, svolgendo catene di operazioni.

17.1.1 Parallelismo e frammentazione dei dati

Il parallelismo è normalmente associato alla *frammentazione dei dati*: i frammenti vengono distribuiti su dischi distinti, che possono essere associati a certi processori. Questo tipo di frammentazione è spesso chiamato *partizionamento*; noi usiamo il termine frammentazione per mettere in evidenza il legame con i concetti espressi nel Capitolo 16. Per esempio, consideriamo una base di dati relativa alla gestione dei conti correnti bancari, caratterizzata dalle seguenti relazioni:

CONTOCORRENTE(CCNum,Nome,Saldo)
MOVIMENTO(CCNum,Data,Progr,Causale,Ammontare)

Supponiamo che le tabelle siano frammentate in base a intervalli predefiniti di numero di conto corrente, e ogni coppia di frammenti relativi allo stesso intervallo di valori sia associata a un processore (per esempio, in quanto allocato su una porzione del file system abbinata al processore medesimo). Questa frammentazione può essere *statica*, cioè permanente, oppure *dinamica*, realizzata cioè poco prima di rispondere a una specifica interrogazione; nel secondo caso, occorre includere nel costo dell'interrogazione anche la distribuzione iniziale dei dati sui dischi del sistema parallelo, che evidentemente vengono caricati a ogni esecuzione delle interrogazioni a partire da dati memorizzati persistentemente su altri supporti. Qualora la memoria centrale sia sufficiente, il caricamento sui dischi può essere evitato.

Una tipica interrogazione OLTP è la richiesta del saldo di uno specifico correntista:

```
procedure CercaSaldo(:cc-num,:saldo);
    select Saldo into :saldo
    from ContoCorrente
    where CCNum = :cc-num;
end procedure;
```

Una tipica interrogazione OLAP è la richiesta dei conti correnti che hanno svolto, nel 2014, transazioni per ammontare complessivo superiore a 50.000 euro:

```
procedure Totale2014();
    select CCNum, sum(Ammontare)
    from ContoCorrente join Movimento on
         ContoCorrente.CCNum = Movimento.CCNum
    where Data >=1/1/2014
    and Data <=31/12/2014
    group by CCNum
    having sum(Ammontare) >50000
end procedure;
```

In genere, le interrogazioni OLTP possono essere indirizzate verso specifici frammenti in base ai loro predicati di selezione; le interrogazioni OLAP devono spesso essere eseguite su tutti i frammenti in parallelo.

Si noti che la frammentazione e allocazione dei dati relativi ai conti correnti e ai movimenti consente l'esecuzione del *join distribuito* tra i frammenti: è possibile cioè svolgere dapprima il join tra i frammenti di ugual indice, che contengono numeri di conti correnti distribuiti sullo stesso intervallo di valori, per poi eseguire le altre operazioni dell'interrogazione. L'esecuzione di n join su frammenti di dimensione $(1/n)$ è ovviamente preferibile all'esecuzione di un unico join che coinvolga le intere tabelle; per questo motivo, l'esecuzione di join distribuiti è essenziale per il parallelismo intra-query. La distribuzione dinamica dei dati è necessaria ogniqualvolta la frammentazione iniziale non consente l'esecuzione distribuita dei join presenti nell'interrogazione.

17.1.2 Speed-up e scale-up

Gli effetti del parallelismo vengono tipicamente descritti da due curve, dette di *speed-up* e *scale-up*. La curva di *speed-up* caratterizza solo il parallelismo inter-query e

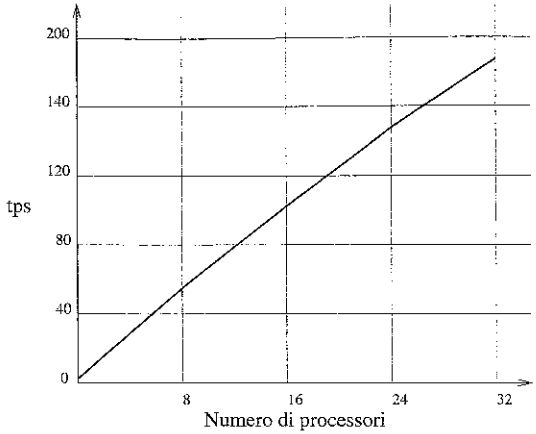

Figura 17.1 Speed-up in un sistema parallelo.

misura il crescere delle prestazioni, misurate in *tps* (transazioni al secondo), al crescere del numero di processori. La Figura 17.1 illustra una situazione ideale, in cui le prestazioni crescono quasi linearmente al crescere dei processori; i sistemi OLTP garantiscono prestazioni vicine a quelle ideali, e in effetti molti costruttori sono in grado di mostrare curve di speed-up pressoché lineari.

La curva di *scale-up* caratterizza sia il parallelismo inter-query sia il parallelismo intra-query e misura il costo di una singola transazione al crescere del numero di processori. La Figura 17.2 illustra una situazione ideale, in cui i costi rimangono quasi identici al crescere dei processori (si dice in tal caso che il sistema "scala" in modo ideale); nei sistemi OLTP l'aumento dei processori consente di gestire un numero maggiore di transazioni al secondo e risponde quindi a un accentuato carico transazionale, mentre nei sistemi OLAP l'aumento dei processori è spesso associato a un aumento della numerosità dei dati, che fa crescere la complessità delle interrogazioni. Sia i sistemi OLTP sia i sistemi OLAP garantiscono prestazioni assai vicine a quelle ideali, e in effetti molti costruttori sono in grado di mostrare curve di scale-up pressoché costanti.

17.1.3 Benchmark delle transazioni

Le curve di speed-up e di scale-up hanno introdotto un problema generale, e cioè quello di valutare le prestazioni delle architetture per basi di dati, siano esse centralizzate, distribuite o parallele. La misura delle prestazioni di un sistema richiede che esistano specifiche oggettive e precise delle transazioni e delle condizioni di carico in cui operare le misure; tali specifiche prendono il nome di *benchmark*. Dopo una lunga competizione fra vari enti per la proposta di uno standard, si è imposto

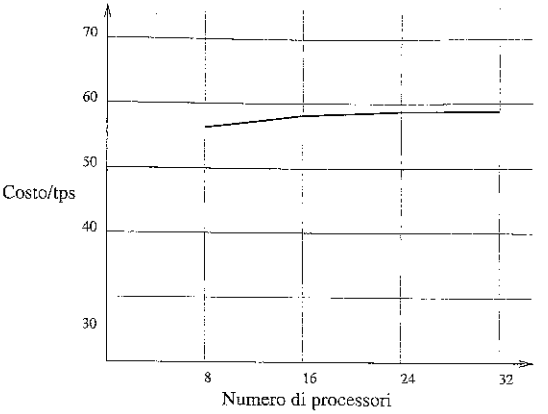

Figura 17.2 Scale-up in un sistema parallelo.

il TPC (*Transaction Processing Performance Council*), un consorzio che raccoglie i principali produttori di DBMS e di sistemi transazionali.

Nell'ambito del consorzio TPC sono stati definiti diversi benchmark. I benchmark correnti più significativi sono chiamati TPC-E (per applicazioni di tipo OLTP), TPC-H (per applicazioni di tipo OLAP) e TPC-W (per applicazioni in cui la base di dati viene utilizzata in supporto a un'architettura Web, come si vedrà nel Capitolo 18). Nelle specifiche di un benchmark si indica:

- il codice delle transazioni. Per esempio, nel TPC-C si prevede un tipico carico di lavoro OLTP, usando come esempio di riferimento un'applicazione di gestione ordini;
- la dimensione della base di dati e il metodo da usare per generare i dati in modo casuale;
- la distribuzione degli arrivi delle transazioni, in modo da caratterizzare il carico transazionale in termini di tps;
- le modalità di misurazione e di "certificazione" (*audit*) della validità dei benchmark.

17.2 Le architetture per il cloud computing

Le tecniche per l'uso del parallelismo nelle basi di dati sono state sviluppate inizialmente avendo come scenario di riferimento quello di un impianto informatico di tipo SMP o cluster controllato direttamente dal proprietario dei dati, oppure ospitato presso fornitori di servizi con il cosiddetto *hosting*. Lo sviluppo del mondo del *cloud computing* ha portato alla ribalta uno scenario molto più ampio per l'applicazione di queste tecniche.

Il cloud computing può essere visto come una forma sofisticata di hosting, in cui un fornitore mette a disposizione tutte le risorse hardware, di rete e software neces-

sarie e inoltre consente all'utente di gestire la propri'architettura attraverso interfacce remote, tipicamente Web. Il cloud computing è quindi definibile come un modello che consente l'accesso tramite rete internet a un insieme condiviso di risorse (hardware, software di base e applicativo, connessione in rete, spazio di memoria persistente) che possono essere allocate e rilasciate in modo semplice, senza necessità di interazione fisica con il fornitore. Un aspetto critico di questo scenario è proprio la possibilità di disporre in modo facile di un gran numero di unità di elaborazione, che possono essere sfruttate per gestire grandi quantità di dati solo grazie alla distribuzione attenta del carico di lavoro su diversi nodi.

Prima di considerare gli aspetti specifici legati al parallelismo, analizziamo le varie caratteristiche che distinguono il cloud computing dalle normali architetture dei sistemi informativi.

- *Self-service su richiesta*: gli elementi dell'architettura possono essere definiti dall'utente in base alle proprie esigenze e allocati in modalità self-service, grazie a semplici interfacce Web.
- *Accesso via rete*: la gestione dell'architettura avviene da remoto: si possono comandare la creazione e configurazione degli elementi architetturali, l'installazione delle applicazioni, la messa online e lo spegnimento.
- *Misurazione del servizio*: le risorse dell'architettura vengono affittate secondo un modello di pagamento che presuppone il monitoraggio continuo della quantità e qualità delle risorse prese in affitto, siano esse capacità di rete, macchine virtuali o licenze software.
- *Elasticità*: la configurazione dell'architettura non è immutabile, ma può essere aggiornata in qualunque momento, sia manualmente dall'utente sia da regole che monitorano il carico delle applicazioni e reagiscono alle variazioni. Per esempio, è possibile definire regole che aumentano il numero di macchine virtuali quando il numero di richieste concorrenti o i tempi di risposta superano una soglia massima.
- *Condivisione delle risorse*: le risorse dell'infrastruttura cloud non sono normalmente riservate all'utente ma possono essere condivise tra molte applicazioni, anche di utenti diversi. Per economia di gestione, spesso le infrastrutture cloud sono costituite da moltissime macchine fisiche tutte identiche, che vengono virtualizzate mediante software di gestione appositi in modo da emulare l'ambiente di esecuzione richiesto dal client.

Il modello del cloud computing può essere realizzato in modi diversi, a seconda di dove risiedano le risorse e del grado di condivisione.

- *Cloud privata*: questa configurazione utilizza il modello del cloud computing per realizzare un'infrastruttura proprietaria, quale l'infrastruttura di calcolo di un'impresa o di un'organizzazione. Per esempio, il CERN di Ginevra, l'istituzione dove è nato il World Wide Web, utilizza una cloud privata di grandi dimensioni e distribuita in più paesi per memorizzare ed elaborare l'enorme quantità di dati prodotti dagli esperimenti di collisione.
- *Cloud comunitaria*: l'infrastruttura di cloud appartiene a un'istituzione comunitaria, che ne garantisce l'uso agli aderenti. Un esempio è l'infrastruttura federale di

cloud computing degli Stati Uniti, che mette a disposizione le proprie risorse alle agenzie federali.

- *Cloud pubblica*: l'infrastruttura cloud è esposta al pubblico generale dei clienti, che possono prendere in affitto le risorse. Tra i fornitori di servizi di cloud pubblica più usati ci sono: Amazon Web Service, Microsoft Windows Azure, Google Cloud Platform, Salesforce, Cloudbees e Rackspace.
- *Cloud ibrida*: a questa categoria appartengono le soluzioni che fanno uso di più modelli contemporaneamente. Un caso tipico è una cloud privata che utilizzi risorse aggiuntive prese da una cloud pubblica per fare fronte a picchi di domanda.

Dal punto di vista del fornitore, un'infrastruttura cloud si presta alla realizzazione di modelli di servizio differenti.

- *Cloud Software as a Service (SaaS)*: il fornitore mette a disposizione tutto ciò che è necessario all'uso di un pacchetto applicativo. Il cliente noleggia l'uso di un'applicazione finita. Salesforce.com esemplifica questo modello di offerta e mette a disposizione dei clienti un sistema gestionale affittabile su cloud.
- *Cloud Platform as a Service (PaaS)*: il fornitore mette a disposizione le risorse hardware e il software di base. Il cliente usa la piattaforma risultante per installare le proprie applicazioni. Cloudbees esemplifica questo modello di offerta e mette a disposizione dei clienti una piattaforma completa per lo sviluppo di applicazioni Web in Java.
- *Cloud Infrastructure as a Service (IaaS)*: il fornitore mette a disposizione soltanto le risorse hardware. Il cliente usa tali risorse per installare il software di base e le proprie applicazioni. Amazon Web Services e RackSpace esemplificano questo modello di offerta.

La Figura 17.3 riassume i caratteri distintivi del cloud computing, secondo uno schema proposto dall'agenzia federale americana NIST (*National Institute of Standards and Technology*).

17.2.1 Programmazione distribuita per il cloud con Hadoop e MapReduce

La disponibilità a costi limitati di infrastrutture cloud composte da molti nodi ha portato alla ribalta tecniche e ambienti di programmazione distribuita in grado di trarre vantaggio dal cloud computing.

Un ambiente di programmazione distribuito di questo genere molto usato è *Apache Hadoop*, un progetto della Apache Software Foundation per la costruzione di un insieme di risorse per la programmazione distribuita, specialmente utile per l'elaborazione di grandi moli di dati.

Hadoop comprende vari moduli, tra cui quelli connessi specificamente con l'elaborazione dei dati sono illustrati nella Figura 17.4:

- *Hadoop Distributed File Sytem (HDFS)*: si tratta di un file system distribuito scritto in Java ed eseguibile in un cluster di macchine. HDFS usa un protocollo di

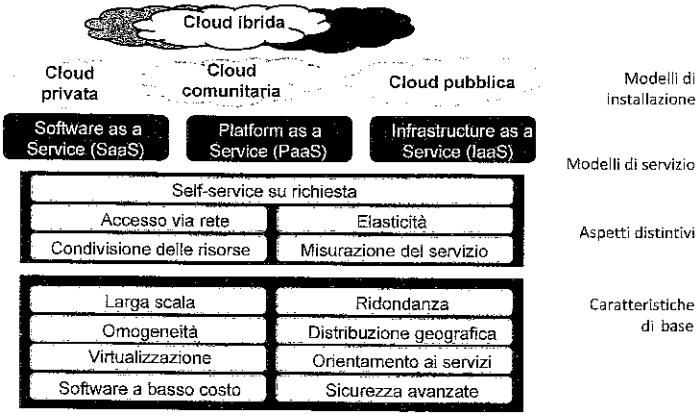

Figura 17.3 L'architettura del cloud computing secondo il NIST.

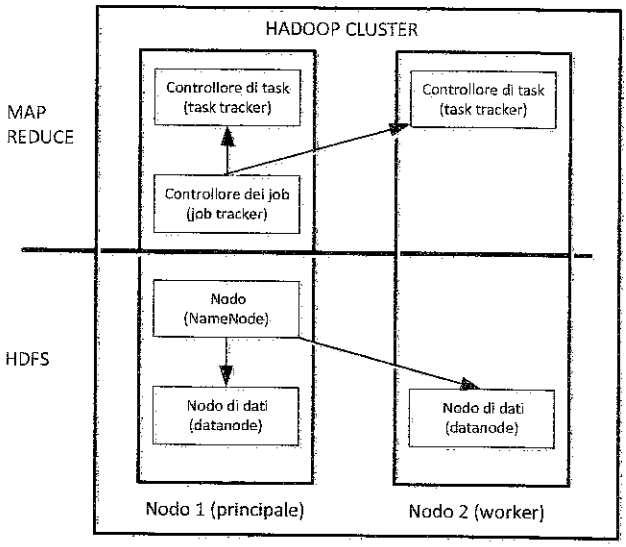

Figura 17.4 Architettura di computazione distribuita con Hadoop.

distribuzione dei dati basato su TCP-IP ed è in grado di memorizzare file anche di grandi dimensioni (nell'ordine dei terabyte) frammentandoli sui nodi e gestendo l'affidabilità mediante la replicazione dei dati. Un cluster Hadoop contiene un nodo principale (detto *NameNode*) e diversi nodi controllati (detti nodi *worker*). Il nodo principale gestisce la distribuzione di blocchi di dati sui diversi nodi controllati.

- *Hadoop MapReduce*: fornisce un modello di elaborazione parallela, ispirato all'omonimo progetto originalmente sviluppato da Google. L'elaborazione è governata dal motore MapReduce, che gestisce i lavori (*job*) tramite un processo, detto *job tracker*, allocato sul nodo principale del cluster; il job tracker assegna unità di lavoro (*task*) a diversi processi (*task tracker*), residenti sullo stesso nodo o su nodi controllati. L'ambiente di esecuzione si occupa di schedulare i task, allocare i task ai nodi dove sono presenti i dati, controllare la terminazione dei task e rischedulare quelli che hanno fallito.

Un job MapReduce definisce una sessione di elaborazione dei dati. Tipicamente un job suddivide i dati in ingresso in un gran numero di pacchetti indipendenti che vengono elaborati in parallelo da task di tipo *Map*. Il motore MapReduce raccoglie gli output dei task di tipo Map e li fornisce come input a task di tipo *Reduce*.

La programmazione delle funzioni da utilizzarsi nei task di tipo Map e Reduce si basa su un formato unico dei dati: coppie chiave-valore. I tipi di dati che possono essere usati come chiave o valore sono molteplici: si richiede unicamente che possano essere serializzati, cioè convertiti in un formato adatto alla trasmissione in rete. I tipi usabili come chiavi devono inoltre permettere la comparazione dei valori, al fine di consentire l'ordinamento delle chiavi.

A titolo di esempio, mostriamo la definizione in Java di due classi, che realizzano rispettivamente la funzione di Map e di Reduce per contare le parole presenti in un testo.

La classe `Map` implementa la divisione del testo in parole tramite il metodo `map()`. Questo accetta in input una chiave (`chiave`) e un testo (`Testo`), scandisce il testo e inserisce in un parametro collettore dell'output (`output`) tante coppie quante sono le parole del testo. Ciascuna coppia abbina a una parola semplicemente il numero 1.

```
public static class Map extends MapReduceBase
  implements Mapper<LongWritable, Text, Text, IntWritable>
    {private final static IntWritable uno = new
    IntWritable(1); private Text parola = new Text();
    public void map(LongWritable chiave, Text testo,
                    OutputCollector<Text, IntWritable> output,
                    Reporter reporter) throws IOException
      {String linea = testo.toString();
      StringTokenizer tokenizer = new StringTokenizer(linea);
      while (tokenizer.hasMoreTokens())
        {parola.set(tokenizer.nextToken());
        output.collect(parola, uno);
      }
    }
}
```

Il risultato dell'esecuzione dei task di tipo Map è quindi una (anche enorme) collezione di parole associate al numero 1, che denota semplicemente l'occorrenza della parola nel testo; tali occorrenze possono essere distribuite su più nodi del cluster. La funzione di Reduce procede al conteggio, sommando le occorrenze, come illustra il codice Java seguente:

```
public static class Reduce extends MapReduceBase
  implements Reducer<Text, IntWritable, Text, IntWritable>
    {public void reduce(Text chiave, Iterator<IntWritable>
                        valori, OutputCollector<Text,
                        IntWritable> output, Reporter
                        reporter) throws IOException {
      int somma = 0;
      while (valori.hasNext()) {
        somma += valori.next().get();
      }
      output.collect(chiave, new IntWritable(somma));
  }
}
```

Il metodo `Reduce` riceve in input la chiave corrispondente a una delle parole estratte dal testo, associata a una collezione di valori interi (`valori`) corrispondenti alle occorrenze di tale parola; il metodo produce in output la parola associata alla somma delle sue occorrenze. Si noti come il framework aggreghi per valore della chiave le singole coppie prodotte dalla funzione `map` creando il parametro di ingresso `valori` ricevuto dalla funzione `reduce`.

17.3 Pig e Pig Latin

Descriviamo brevemente in questo paragrafo le caratteristiche principali del sistema *Apache Pig* e del linguaggio di interrogazione a esso associato *Pig Latin*. Si tratta del frutto di un'iniziativa dell'Apache Software Foundation, che integra il paradigma MapReduce con i normali approcci alla gestione dei dati. L'attenzione nel disegno di Pig è rivolta a gestire una computazione sui dati nel contesto di un sistema cloud, integrando parallelismo e distribuzione. Si può notare che il linguaggio Pig Latin è spesso considerato un esempio della famiglia dei sistemi NoSQL, che abbiamo trattato nel Paragrafo 13.5. In effetti il sistema si basa su un modello dei dati non relazionale e usa un linguaggio diverso da SQL per la scrittura di interrogazioni.

17.3.1 Caratteristiche di base

Il sistema Pig rappresenta uno degli elementi dell'architettura Hadoop, sviluppata dall'Apache Software Foundation e descritta nel Paragrafo 17.2.1. L'uso di Pig prevede infatti che sia già configurato e installato l'ambiente Hadoop. Entrambi gli elementi di Hadoop che abbiamo descritto presentano una forte relazione con Pig: il file system distribuito HDFS rappresenta il supporto di default per il caricamento dei dati e il salvataggio dei risultati della computazione; il paradigma di computazione

MapReduce è utilizzato per l'esecuzione delle query. Il ruolo di Pig è di fornire un'infrastruttura per facilitare le operazioni di calcolo e trasformazione sui dati, sfruttando il parallelismo in modo quanto più possibile automatico e trasparente all'utente.

Come detto sopra, Pig Latin è il linguaggio per la rappresentazione delle interrogazioni che viene riconosciuto da Pig. La gestione delle interrogazioni prevede che ogni richiesta Pig Latin venga compilata in un opportuno programma che utilizza le primitive di Hadoop. È possibile usare i servizi del sistema sia in modalità interattiva sia mediante la scrittura di interrogazioni che da programma vengono passate direttamente all'ambiente di esecuzione, in modo analogo a quanto avviene con SQL. L'interprete interattivo, simile alle shell dei normali DBMS reazionali, si chiama `grunt` e offre un insieme di comandi che facilitano lo sviluppo di interrogazioni e l'analisi del comportamento del sistema.

17.3.2 Modello dei dati

Il modello dei dati utilizzato da Pig presenta alcune somiglianze con i modelli relazionali a oggetti, con molte delle caratteristiche descritte nel Capitolo 13, senza però il supporto per l'importante concetto di gerarchia di ereditarietà. I dati sono organizzati in relazioni, che rappresentano collezioni di tuple in cui ciascuna tupla contiene un elenco ordinato di campi. Queste relazioni differiscono dall'analogo del modello relazionale per vari aspetti, comuni a molti sistemi relazionali a oggetti: la relazione può contenere elementi ripetuti (in effetti una relazione viene descritta come un multiinsieme (*bag*) di tuple); esiste un identificatore implicito per ciascuna tupla, ma non esiste una chiave primaria esplicita; i campi possono rappresentare valori strutturati.

Esempio Pig Latin 1: caricare con il comando LOAD una tabella STUDNOTEL, a partire da un file StudentiSenzaTelefono che descrive nome, anno di nascita, città e corso di laurea di un insieme di studenti.

```
StudNoTel = LOAD 'StudentiSenzaTelefono'
                AS (Nome:chararray, AnnoNascita:int,
                Citta: chararray, Corso:chararray)
```

Il contenuto della tabella può essere illustrato tramite l'invocazione del comando DUMP. Negli esempi faremo spesso uso di questo comando, che permette di mostrare in una sessione interattiva il contenuto di una tabella. Il comando DUMP di norma non farà parte dei programmi Pig Latin che dovranno essere eseguiti dal sistema in modalità non interattiva.

```
DUMP StudNoTel;
(Carlo, 1998, Milano, Informatica)
(Antonio, 1993, Roma, Meccanica)
(Paola, 1995, Bergamo, Informatica)
```

Come è consuetudine, l'operatore di deriferimento, rappresentato da un semplice punto (la cosiddetta *dot notation*), permette di accedere alle componenti interne di una

tupla. Per accedere ai campi si possono utilizzare sia i nomi degli attributi, quando specificati, sia una notazione posizionale che associa la posizione zero al primo attributo. Considerando la struttura dell'esempio precedente, supponendo che una variabile `S` rappresenti la generica tupla, sarà possibile fare riferimento all'attributo AnnoNascita utilizzando l'espressione `S.AnnoNascita`, oppure `S.$1` con la notazione posizionale.

Nell'esempio precedente sono stati utilizzati i tipi `chararray` e `int`, con il consueto significato di rappresentazione di array di caratteri e di valori interi. L'insieme dei tipi predefiniti di Pig Latin è paragonabile a quanto offerto dai moderni linguaggi di programmazione. Quando il tipo non viene esplicitato, il sistema assegna di default il tipo `bytearray`. Vengono poi offerti i costruttori di tipo `tuple`, `bag` e `map`; `tuple` permette di definire domini associati a un elenco ordinato di campi; `bag` associa al dominio una collezione di tuple con possibili duplicati, e `map` rappresenta un insieme di coppie chiave-valore. Come nei sistemi relazionali, è previsto che ogni dominio possa assumere il valore nullo, che viene gestito all'interno dei predicati di confronto con una logica a tre valori, in modo analogo a SQL. Mostriamo ora l'uso del costruttore di tuple e la modalità di accesso alle componenti interne nell'ambito di una query.

Esempio Pig Latin 2: caricare nella tabella STUD una descrizione degli studenti che presenta il recapito telefonico degli studenti come un record strutturato.

```
Stud = LOAD 'Studenti' AS
       (Nome:chararray, AnnoNascita:int, Corso:chararray,
        Telefono:tuple(Prefisso: chararray,
        Numero:chararray));

DUMP Stud;
(Carlo, 1998, Milano, Informatica, (0521, 555431))
(Antonio, 1993, Roma, Meccanica, (06, 55512345))
(Paola, 1995, Bergamo, Informatica, (035, 2052356))
```

Costruiamo quindi una semplice interrogazione che scandisce le tuple ed estrae solo i numeri di telefono.

```
NumeriTel = FOREACH Stud GENERATE Telefono.Prefisso,
                                  Telefono.Numero;

DUMP NumeriTel;
(0521, 555431)
(06, 55512345)
(035, 2052356)
```

Rispetto ai DBMS relazionali, Pig ammette maggiore flessibilità relativamente al rispetto dello schema dei dati. Se una tupla presenta valori non compatibili con lo schema della tabella o alcuni attributi sono mancanti, Pig cerca di norma di gestire l'incoerenza mediante l'assegnamento di un valore nullo all'attributo. L'obiettivo è quello di minimizzare il numero di eccezioni ed evitare che si possano causare interruzioni nell'elaborazione di interrogazioni fortemente distribuite.

17.3.3 Query semplici

Le interrogazioni Pig Latin fanno uso di un insieme di operatori che corrispondono in modo abbastanza diretto agli operatori dell'algebra relazionale. La sintassi del linguaggio differisce da SQL, ponendo maggiore enfasi su una rappresentazione procedurale della query. Come già segnalato, l'obiettivo di fondo del linguaggio è di consentire una facile integrazione tra le caratteristiche del modello dei dati e le modalità di elaborazione MapReduce.

Mostriamo una query che svolge un'operazione di proiezione. La struttura FOREACH prevede di scandire le tuple di una tabella e con GENERATE produrre nuove tuple che costruiscono la tabella Pig che rappresenta il risultato dell'interrogazione.

Esempio Pig Latin 3: considerando la tabella STUD creata nell'esempio precedente, estrarre il nome degli studenti e il numero di telefono, rappresentando tutte le proprietà allo stesso livello.

```
NomeStudENumTel = FOREACH Stud GENERATE Nome,
                        Telefono.Prefisso, Telefono.Numero;

DUMP NomeStudENumTel;
(Carlo, 0521, 555431)
(Antonio, 06, 55512345)
(Paola, 035, 2052356)
```

Illustriamo ora l'operatore FILTER, che realizza la selezione delle tuple che soddisfano un certo predicato. La sintassi per la rappresentazione dei predicati è abbastanza ricca, in linea con le caratteristiche dei moderni linguaggi di programmazione.

Esempio Pig Latin 4: estrarre gli studenti che hanno un prefisso telefonico pari a 035.

```
Stud035 = FILTER Stud BY Telefono.Prefisso='035';

DUMP Stud035;
(Paola, 1995, Bergamo, Informatica, (035, 2052356))
```

Nel prossimo esempio si mostra un'interrogazione che combina gli operatori di selezione e proiezione. La sintassi di Pig Latin prevede che le interrogazioni che fanno uso di diversi operatori tengano distinti i diversi passi, con l'introduzione di un nome per la tabella che contiene ogni risultato intermedio e il suo riutilizzo nei passi successivi. È da notare che l'ambiente di esecuzione, mediante una cosiddetta ottimizzazione *multi-query*, analizza l'intera sequenza di comandi Pig Latin al fine di individuare una strategia di esecuzione efficiente, la quale può benissimo operare la computazione in un ordine diverso da quello specificato, purché venga restituito lo stesso risultato finale.

Esempio Pig Latin 5: estrarre nome e corso di studio degli studenti che hanno un nome che contiene la lettera 'l'.

```
StudConL = FILTER Stud BY Nome matches '.*l.*';
StudConLNomeCorso = FOREACH StudConL GENERATE Nome, Corso;
```

```
DUMP StudConLNomeCorso;
(Carlo, Informatica)
(Paola, Informatica)
```

Le espressioni regolari in Pig Latin utilizzano la sintassi del linguaggio Java; nell'esempio visto sopra, il punto rappresenta un generico carattere e l'asterisco indica che l'elemento può apparire un numero arbitrario di volte.

La combinazione di informazioni conservate in tabelle diverse viene gestita tramite l'operatore di JOIN, che corrisponde direttamente all'operatore di join del'algebra relazionale (il prodotto cartesiano è pure disponibile, mediante l'operatore CROSS, ma entrano in gioco anche qui i limiti alla sua applicabilità che abbiamo osservato per l'algebra relazionale).

Esempio Pig Latin 6: caricare una tabella che descrive i corsi di laurea e poi estrarre i nomi degli studenti e il numero di esami del loro corso di laurea.

```
CorsiDiLaurea = LOAD 'Corsi' AS
        (NomeCorso:chararray, NumEsami:int,
         NumCfu:int, Sede:chararray);

DUMP CorsiDiLaurea;
(Informatica, 20, 180, Milano)
(Meccanica, 19, 180, Roma)
(InformaticaUM, 12, 120, Dalmine)

StudJoinCorso = JOIN Stud BY Corso,
                     CorsiDiLaurea BY NomeCorso;
StudConNumEsami = FOREACH StudJoinCorso
                    GENERATE Nome, NumEsami;

DUMP StudConNumEsami;
(Carlo, 20)
(Antonio, 19)
(Paola, 20)
```

Una caratteristica della sintassi dell'operatore JOIN è la possibilità di specificare immediatamente se il join deve essere realizzato assumendo un certo partizionamento sull'attributo di join, eventualmente fornendo un'indicazione rispetto a quale tra i diversi algoritmi di join deve essere utilizzato per l'esecuzione e quale grado di parallelismo adottare.

Esempio Pig Latin 7: calcolare il join tra la tabella che descrive gli studenti e quella che descrive il corso di laurea, utilizzando per la computazione un *merge join*, partizionando i dati in base a una classe PARTIZIONA e usando un grado di parallelismo pari a 3.

```
JOIN Stud BY Corso, CorsiDiLaurea BY NomeCorso
  USING 'merge'
  PARTITION BY Partiziona
  PARALLEL 3;
```

Il codice Java della classe **Partiziona** può essere il seguente:

```
public class Partiziona
  extends Partitioner <PigNullableWritable, Writable> {
     //@Override
    public int getPartition(PigNullableWritable chiave,
                            Writable valore,
                            int numPartizioni) {
            return (chiave.hashCode()) % numPartizioni;
        }
    }
```

Quest'ultimo esempio dimostra l'attenzione che viene dedicata dal linguaggio alla considerazione degli aspetti di livello fisico e all'integrazione con le architetture di esecuzione distribuite. Il partizionamento dei dati sui diversi nodi è una delle caratteristiche centrali della computazione MapReduce e l'ambiente Pig consente la definizione relativamente facile di proprie funzioni di partizionamento.

Completiamo l'analisi degli operatori di base illustrando esempi relativi agli operatori di unione e differenza. Pig Latin offre direttamente l'operatore `UNION`. L'aspetto particolare dell'operatore `UNION` è la possibilità di svolgere l'unione anche di tabelle con formato diverso (ciò dovrebbe però essere evitato nelle applicazioni). Al contrario dell'operatore binario `union` di SQL, i duplicati vengono mantenuti nel risultato.

Esempio Pig Latin 8: estrarre in un unico elenco i nomi dei corsi di laurea offerti dall'università e quelli seguiti dagli studenti.

```
CorsiLaurea = FOREACH CorsiDiLaurea GENERATE NomeCorso;
CorsiStudenti = FOREACH Stud GENERATE Corso;
ListaCorsiConRipetizioni = CorsiLaurea UNION CorsiStudenti;

DUMP ListaCorsiConRipetizioni;
(Informatica)
(Meccanica)
(InformaticaLM)
(Informatica)
(Meccanica)
(Informatica)
```

Per quanto riguarda la differenza tra relazioni, nella versione corrente Pig Latin non offre un operatore specifico. La valutazione della differenza richiederà tipicamente di costruire interrogazioni che sfruttano altri operatori. Per esempio, la differenza tra due relazioni può essere calcolata partendo dal join esterno. Il join `LEFT` mantiene tutte le tuple del primo operando; selezionando solo le tuple del risultato del join `LEFT` che presentano valori nulli in corrispondenza delle tuple della seconda relazione, si ottengono proprio le tuple della prima relazione che non hanno un elemento corrispondente nella seconda. Per questa computazione può essere utile fare uso dell'operatore `DISTINCT`, che elimina i duplicati delle tuple.

Esempio Pig Latin 9: estrarre i corsi di laurea che non sono associati a studenti.

```
CorsiStudenti = FOREACH Stud GENERATE Corso;
CorsiStudUnici = DISTINCT CorsiStudenti;
CorsiConStudenti = JOIN CorsidiLaurea BY NomeCorso,
                        CorsiStudUnici BY Corso;
CorsiEStudenti = JOIN CorsidiLaurea LEFT BY NomeCorso,
                      CorsiStudUnici BY Corso;
CorsiSenzaStudenti = FILTER CorsiEStudenti BY Corso IS NULL;

DUMP CorsiEStudenti;
(Informatica, Informatica)
(Meccanica, Meccanica)
(InformaticaLM, )

DUMP CorsiSenzaStudenti;
(InformaticaLM, )
```

17.3.4 Query con ordinamenti e raggruppamenti

L'operatore che permette di ordinare le tuple di una tabella è l'operatore ORDER BY. L'applicazione dell'operatore garantisce che l'accesso agli elementi della tabella sia ordinato nelle fasi immediatamente successive all'esecuzione dell'operazione, ma l'ordinamento non diventa una proprietà stabile dei dati.

Esempio Pig Latin 10: ordinare l'elenco degli studenti in base all'anno di nascita.

```
StudOrd = ORDER StudNoTel BY AnnoNascita;

DUMP StudOrd;
(Antonio, 1993, Roma, Meccanica)
(Paola, 1995, Bergamo, Informatica)
(Carlo, 1998, Milano, Informatica)
```

L'operatore RANK rappresenta una variante dell'operatore di oridnamento che può essere d'aiuto nei contesti nei quali serve avere a disposizione un'indicazione stabile della posizione relativa delle tuple rispetto a un certo criterio di ordinamento. L'operatore realizza un ordinamento e poi aggiunge a ciascuna tupla nella prima posizione un attributo numerico il cui valore rappresenta la posizione relativa della tupla. Valori uguali saranno associati allo stesso indice.

Esempio Pig Latin 11: ordinare l'elenco degli studenti in base al corso di laurea, tenendo traccia in ogni tupla della posizione rispetto a questo ordinamento.

```
StudRanked = RANK StudNoTel BY Corso;

DUMP StudRanked;
(1, Carlo, 1998, Milano, Informatica)
(1, Paola, 1995, Bergamo, Informatica)
(3, Antonio, 1993, Roma, Meccanica)
```

La possibilità offerta dal modello dei dati di gestire strutture complesse permette di realizzare interrogazioni con raggruppamento caratterizzate da una struttura relativamente snella. Pig Latin offre due diversi nomi per l'operatore che realizza il raggruppamento, `GROUP` e `COGROUP`; l'indicazione dei progettisti del linguaggio è di usare `GROUP` quando si opera su una sola tabella e `COGROUP` quando l'operatore di raggruppamento viene applicato a più tabelle. Il comportamento base dell'operatore prevede di utilizzare l'attributo specificato dalla clausola `BY` (il primo attributo della tabella se la clausola `BY` viene omessa) come attributo di raggruppamento e di nidificare i rimanenti attributi, raggruppando tutte le tuple che sono associate allo stesso valore in una struttura di tipo *bag*. Lo schema del risultato ha `group` come nome del primo attributo e il nome della tabella per l'altro attributo.

Esempio Pig Latin 12: estrarre una tabella con i nomi dei corsi di laurea e associati a ciascun corso gli studenti iscritti a esso.

```
StudGByCorsi = GROUP StudNoTel BY Corso;

DUMP StudGByCorsi;
(Informatica, {(Carlo, 1998, Milano, Informatica),
               (Paola, 1995, Bergamo, Informatica)})
(Meccanica,{(Antonio, 1993, Roma, Meccanica)})
```

L'operazione di raggruppamento su più tabelle restituisce una collezione di tuple in cui ciascuna tupla presenta il valore della chiave e poi una sequenza di collezioni (*bag*) di tuple, che rappresentano gli elementi che provengono da ciascuna tabella. Se una tabella non presenta valori in corrispondenza di un valore della chiave, nel risultato comparirà una collezione vuota.

Esempio Pig Latin 13: estrarre una tabella con i nomi delle città e per ciascuna di esse gli studenti e i corsi di laurea associati.

```
StudCorsiGByCitta = COGROUP StudNoTel BY Citta,
                            CorsiDiLaurea BY Sede;

DUMP StudCorsiGByCitta;
(Milano,{(Carlo, 1998, Milano, Informatica)},
         {(Informatica, 20, 180, Milano)})
(Roma,{(Antonio, 1993, Roma, Meccanica)},
      {(Meccanica, 19, 180, Roma)})
(Bergamo,{(Paola, 1995, Bergamo, Informatica)},{})
(Dalmine,{},{(InformaticaLM, 12, 120, Dalmine)})
```

Pig Latin mette a disposizione le classiche funzioni aggregate `AVG`, `COUNT`, `SUM`, `MIN` e `MAX`. La loro applicazione può avvenire su qualunque struttura nidificata. Il caso più tipico d'uso si ha nell'ambito delle query con raggruppamento.

Esempio Pig Latin 14: estrarre per ciascun corso di laurea il numero degli studenti iscritti.

```
StudGByCorsi = GROUP Stud BY Corso;
CorsiNumStudenti = FOREACH StudGByCorsi
                        GENERATE group, COUNT(Stud);

DUMP CorsiNumStudenti;
(Informatica, 2)
(Meccanica, 1)
```

17.3.5 Integrazione con MapReduce

Pig può anche essere configurato per operare in modalità locale, ma la modalità che permette di sfruttare i vantaggi di Pig è quella distribuita che si integra con un'installazione di Apache Hadoop. Il cluster Hadoop può essere realizzato in una configurazione che è pienamente sotto il controllo dell'utente, oppure può basarsi sull'uso delle risorse di un cloud provider. Per quanto riguarda l'interazione con il file system, oltre all'uso di HDFS, Pig permette di sfruttare in modo nativo Amazon S3 per l'acquisizione dei dati e il salvataggio dei risultati.

L'integrazione con il paradigma MapReduce riguarda l'intero funzionamento di Pig. Lo sviluppatore deve assumere che ogni operatore venga eseguito in modo distribuito, partizionando le richieste sui diversi nodi di calcolo. Pig usa l'ottimizzazione *multi-query*, considerando l'intero insieme di comandi e cercando di mettere a fattore comune e riutilizzare computazioni intermedie che possono essere utili per produrre risultati diversi. Ciò spesso porta a configurazioni in cui le fasi di Map e Reduce vengono svolte in modo flessibile, al fine di migliorare le prestazioni del sistema.

Un obiettivo dell'ottimizzatore di Pig è quello di generare un numero di task MapReduce realtivamente limitato, in modo da minimizzare i tempi di setup, che in Hadoop possono essere molto significativi (dell'ordine di decine di secondi). Un altro vantaggio dell'avere un numero ridotto di task è legato al fatto che quando si combinano più task Hadoop in sequenza, la computazione di un task può partire solo quando il task precedente è completato. Se più operatori possono essere raggruppati in un solo task, ciò porterà normalmente a un minore numero di vincoli nell'esecuzione, a un miglior sfruttamento delle risorse di calcolo e a una riduzione del tempo di risposta del sistema.

La computazione sui dati avviene distribuendo le tuple sui diversi worker Hadoop in base al valore delle chiavi. Un aspetto delicato è la distribuzione dei valori delle chiavi, che è molto spesso non uniforme (cioè *skewed*) e che tende a sovraccaricare i nodi che hanno la responsabilità di gestire i valori più frequenti. L'ottimizzatore di Pig cerca di ridurre questo problema costruendo dinamicamente una stima della distribuzione dei valori e allocando a più nodi worker i valori più frequenti.

In molti scenari l'impatto maggiore sul tempo di esecuzione della query è dato dal trasferimento dei dati tra i vari nodi del cluster, piuttosto che dalle risorse computazionali richieste per le fasi di Map e Reduce. Per migliorare le prestazioni, l'ottimizzatore di Pig considera anche l'applicazione di tecniche di compressione sui

dati, usando algoritmi che presentano un buon bilanciamento tra tempo di esecuzione e grado di compressione.

Confrontato con gli ottimizzatori dei sistemi relazionali, l'ottimizzatore di Pig presenta alcune caratteristiche avanzate, come quelle che abbiamo appena descritto, ma anche alcuni limiti. La versione attuale dell'ottimizzatore non fa uso di un modello dei costi; inoltre affida comunque una buona parte di responsabilità allo sviluppatore, il quale deve fornire le giuste indicazioni, come si può vedere dalla sintassi dell'operatore di join.

Lo sviluppatore ha la possibilità di intervenire in modo rilevante sul comportamento di Pig mediante la definizione di *User Defined Functions* (UDF). Ciò avviene tramite l'operatore `REGISTER`, cui è possibile passare come argomento delle proprie classi Java, arricchendo l'insieme di funzioni che possono essere eseguite durante l'elaborazione dei dati. Queste funzioni devono essere scritte facendo riferimento al paradigma MapReduce.

17.3.6 Apache Hive

Pig rappresenta un elemento di una famiglia di soluzioni realizzate dalla Apache Software Foundation e focalizzate sul paradigma MapReduce. L'elemento distintivo di tutte queste soluzioni è la strett'integrazione con l'ambiente Hadoop. Apache Hive è un altro elemento di questa famiglia e presenta anch'esso un forte legame con il mondo delle basi di dati, offrendo dei servizi per lo svolgimento di attività di analisi su grandi quantità di dati strutturati. L'esempio applicativo di riferimento è l'analisi di record di log di grandi dimensioni, ma lo strumento è adattabile a vari altri scenari. Come Pig, anche Hive è associato a un linguaggio per la rappresentazione di interrogazioni, chiamato *Hive QL*. L'obiettivo di questo paragrafo è quello di fornire una illustrazione molto superficiale delle caratteristiche di Hive QL.

Rispetto a Pig Latin, Hive QL mantiene un legame molto più stretto con SQL. Può in effetti essere considerato un dialetto di SQL. Per esempio, la creazione di una tabella che mostri per ogni corso di laurea il numero degli studenti iscritti può essere realizzata nel modo seguente.

```
INSERT OVERWRITE TABLE NumIscritti
 SELECT S.Corso, count(*)
 FROM Studenti S
 GROUP BY C.NomeCorso
```

Hive QL ammette varianti sintattiche che vedono un diverso ordine delle clausole rispetto a quanto prescritto da SQL. Per esempio, se la stessa tabella deve essere la base per la costruzione di più tabelle, è possibile presentare una sola clausola `FROM` cui fanno riferimento i diversi comandi di inserimento, nel modo seguente.

```
FROM Studenti S
 INSERT OVERWRITE TABLE NumIscritti
  SELECT S.Corso, count(*)
  GROUP BY C.NomeCorso
```

```
INSERT OVERWRITE TABLE NumEsamiMedio
 SELECT S.Corso, avg(NumEsami)
 GROUP BY C.NomeCorso
```

Il linguaggio ha quindi una forte natura dichiarativa, rispetto all'impostazione più procedurale di Pig Latin. Il modello dei dati e le modalità di esecuzione delle query di Hive fanno riferimento ai concetti e strumenti per l'analisi dei dati che verranno descritti nel Capitolo 19. Per quanto riguarda l'integrazione con il paradigma MapReduce, le considerazioni che abbiamo presentato rispetto ad Apache Pig si applicano anche a Hive. Lo strumento è pensato per operare in un contesto in cui si devono manipolare grandi quantità di dati e in cui il parallelismo diventa un fattore critico per ridurre i tempi di elaborazione. L'elaborazione delle query richiede di svolgere una fase di analisi e attivazione dei servizi del cluster Hadoop che può essere relativamente onerosa e che rende questo strumento inadatto quando si devono eseguire interrogazioni su tabelle di piccole dimensioni, in quanto il tempo di setup avrebbe un peso eccessivo. Lo sviluppatore dovrà quindi valutare con molta attenzione se lo scenario applicativo si presta a sfruttare i vantaggi che derivano dall'uso di questi strumenti.

Note bibliografiche

Una trattazione interessante sul parallelsimo nelle basi di dati si può trovare in un lavoro di Dewitt e Gray [113]. Sono anche interessanti i manuali dei diversi sistemi, che illustrano le funzionalità di ciascun DBMS su questo fronte. Per quanto riguarda le architetture cloud, è un ottimo punto di partenza il documento rilasciato dal NIST [134] da cui è stato tratto lo schema illustrato in Figura 17.3.

Sul fronte del paradigma MapReduce, l'articolo di Dean e Ghemawat [109] che ha introdotto i concetti nella comunità scientifica è disponibile in rete. Per le soluzioni Pig e Hive prodotte dall'Apache Software Foundation e i relativi linguaggi di interrogazione, le migliori risorse sono rappresentate dalla documentazione disponibile liberamente sul sito dei corrispondenti progetti. Esistono anche volumi della O'Reilly dedicati a questi sistemi [150, 116]. Le problematiche di disegno dell'ottimizzatore di Pig sono trattate in un articolo di Gates, Dai e Nair [117].

Esercizi

Soluzioni sul sito http://www.ateneonline.it/atzeni

17.1 Descrivere uno schema di esecuzione delle seguenti interrogazioni che massimizzino il parallelismo inter-query:

1. estrarre la somma della quantità delle produzioni, raggruppate in base a tipo e modello delle parti;
2. estrarre la media delle parti vendute dai venditori, raggruppate in base a tipo e modello delle parti.

18

Architetture dei sistemi informativi su Web

La rapida diffusione della rete Internet e la sua evoluzione costituiscono uno dei fenomeni più significativi di tutto il settore dell'informatica. La tecnologia che più delle altre ha fatto registrare una crescita e un'accettazione oltre ogni aspettativa è quella del *World Wide Web* (spesso chiamato semplicemente *Web* o abbreviato con le iniziali WWW). Il Web, nella sua accezione originale, può essere definito come un sistema su scala planetaria per la distribuzione e l'accesso a documenti ipertestuali e multimediali codificati sotto forma di pagine HTML, caratterizzato da una grande facilità d'uso, flessibilità ed economicità.

In seguito alla sua enorme diffusione, il Web si è però trasformato. Dal suo ruolo iniziale di applicazione per la consultazione di documenti distribuiti su più siti, il Web è diventato una piattaforma per lo sviluppo di sistemi informativi di ogni genere, accessibili non solo da PC ma anche dai più svariati terminali mobili, attraverso il *browser*, cioè un programma client universale in grado di collegarsi a qualsiasi applicazione che pubblichi la propria interfaccia utente sotto forma di pagine Web.

I *sistemi informativi su Web* (*Web Information Systems*, abbreviato con WIS) presentano una serie di problematiche comuni ai sistemi informativi tradizionali, quali la capacità di supportare lo svolgimento dei processi aziendali e di gestire lo scambio di dati richiesto da tali processi. I dati da scambiare sono spesso gestiti almeno in parte da un DBMS, per cui risulta estremamente importante lo studio delle modalità con cui integrare la tecnologia delle basi di dati e quella del Web.

Al tempo stesso, i WIS presentano anche differenze significative rispetto ai sistemi informativi tradizionali. Innanzitutto, il tipo di informazioni che viene offerto è molto vario: il Web è nato soprattutto per gestire informazioni testuali e multimediali (immagini, audio, video), con struttura piuttosto flessibile e irregolare (come detto nel Capitolo 14, questa esigenza è alla base dell'evoluzione delle basi di dati per la gestione di dati semistrutturati). Al contrario, i sistemi informativi tradizionali operano soprattutto su dati con una struttura relativamente semplice, usando la tecnologia delle basi di dati relazionali.

Un'altra caratteristica dei WIS è il fatto che essi tipicamente offrono agli utenti una ricca varietà di primitive per la consultazione delle informazioni, ma non per l'aggiornamento. L'interazione dell'utente è guidata dai suoi stessi interessi e non segue percorsi predefiniti; al tempo stesso le modifiche che un utente può fare sui dati remoti sono minime, e corrispondono a pochi tipi di richieste, normalmente predefinite (per esempio l'inserimento di un ordine di acquisto).

Infine, le interfacce utente possono variare molto: si pensi alla differenza tra un'interfaccia grafica progettata per una rete locale ad alta velocità dove l'utente si collega per mezzo di un PC dotato di uno schermo di grandi dimensioni e un'interfaccia pensata per l'accesso con un dispositivo mobile dotato di schermo piccolo e un collegamento in cui il traffico deve essere limitato.

Tutte queste differenze, insieme alla maggiore complessità degli obiettivi delle applicazioni, che sono in molti casi offerte all'intero pubblico degli utenti di Internet, richiedono un ripensamento del processo di sviluppo delle applicazioni, sia per quanto riguarda le componenti da sviluppare ex-novo, sia per quanto riguarda l'integrazione di moduli software pre-esistenti all'interno di un'applicazione Web.

In questo capitolo, dopo un richiamo iniziale dei concetti fondamentali dell'architettura del World Wide Web, utile per rendere la trattazione più comprensibile, affronteremo le tecnologie disponibili per realizzare sistemi informativi su Web. Cominceremo con l'introdurre l'architettura base a tre livelli (il client, il Web server e il DBMS) e mostreremo in successione quattro tecnologie, *Common Gateway Interface* (CGI), *Java servlet*, *scripting a lato server* e *Model-View-Controller*, che consentono di costruire dinamicamente le pagine Web a partire da contenuti estratti dalle basi di dati. Descriveremo un'evoluzione dell'architettura a tre livelli che ha lo scopo di aumentare le prestazioni e la scalabilità: l'architettura con *application server*; esemplificheremo quindi lo sviluppo di componenti Java nell'application server tramite la piattaforma *Enterprise Java Beans*. Amplieremo quindi la visione dei sistemi informativi su Web al tema dell'integrazione di applicazioni eterogenee su Web, descrivendo il concetto di *Web Service*, uno strumento per definire servizi applicativi facilmente riutilizzabili e integrabili mediante gli standard aperti del Web. Dopo la panoramica delle architetture a lato server, descriveremo l'evoluzione a lato client, parlando di Rich Internet Application (RIA), JavaScript, AJAX e HTML 5, con un accenno speciale alla memorizzazione dei dati persistenti al lato client. Infine affronteremo gli aspetti metodologici relativi alla progettazione di applicazioni Web centrate sull'uso di basi di dati, che sono indipendenti dall'architettura prescelta per il rilascio delle applicazioni.

18.1 Tecnologie di base

Questo paragrafo ha l'obiettivo di riepilogare brevemente i concetti relativi a Internet e al World Wide Web che risultano importanti ai fini della comprensione dei paragrafi successivi.

18.1.1 Internet

Internet può essere definita come una federazione di reti che comunicano attraverso lo stesso insieme di protocolli, quelli della famiglia TCP/IP (*Transmission Control Protocol/Internet Protocol*). Una situazione molto diffusa è quella che vede reti locali di calcolatori collegate attraverso dispositivi chiamati *router* (speciali calcolatori dedicati all'interconnessione di reti) e linee digitali ad altre reti. Tale schema di connessione si ripresenta anche su scala più ampia, secondo un meccanismo a scatole cinesi: per esempio varie reti locali formano la rete di una università, le differenti reti universitarie sono interconnesse per costituire la rete di tutte le università italiane e così via.

Ciascun calcolatore della rete Internet ha un indirizzo numerico (chiamato *indirizzo IP*) rappresentato da una sequenza di quattro valori compresi tra 0 e 255, per esempio `193.204.161.147`; se pubblico, l'indirizzo è univoco in tutta la rete mondiale. A ciascun calcolatore può essere anche associato un nome simbolico, costituito da identificatori separati da punti (per esempio `www.dia.uniroma3.it`), che viene di solito utilizzato nelle applicazioni al posto del meno intuitivo indirizzo numerico.

Grazie alla struttura degli indirizzi e alle capacità di instradamento delle informazioni dei router, l'effettiva topologia della rete risulta trasparente all'utente (sia esso una persona o un programma): qualunque nodo pubblico risulta accessibile mediante l'indirizzo IP, senza che sia necessario conoscerne la posizione all'interno dello schema complessivo della rete. Da un punto di vista logico, ciascun calcolatore su Internet può connettersi direttamente a ogni altro nodo pubblico. La comunicazione tra

due nodi avviene attraverso una serie complessa di regole, organizzate secondo una gerarchia che coinvolge diversi protocolli, fra cui i già citati TCP e IP; il dettaglio del funzionamento di questi protocolli esula però dagli scopi di questo testo.

Una caratteristica di molte applicazioni che operano sulla rete Internet è l'utilizzo del paradigma client/server, che abbiamo già discusso nel Paragrafo 16.1. Il client gestisce l'interazione con l'utente e trasforma le richieste dell'utente in opportune richieste al server. Il server esegue le operazioni richieste, fornendo al client le risposte appropriate. Per ciò che ci interessa in questo capitolo, un server è schematizzabile come un particolare modulo software che offre un insieme di funzionalità predefinite (in particolare i servizi collegati al Web), fruibili mediante protocolli applicativi di alto livello (in particolare il protocollo HTTP). Tali protocolli a loro volta si appoggiano sui servizi di più basso livello forniti da TCP/IP.

18.1.2 Il World Wide Web

Il World Wide Web è l'insieme di risorse accessibili sulla rete Internet tramite l'uso di applicazioni per lo scambio di *ipertesti multimediali*, utilizzando un particolare protocollo applicativo denominato HTTP (*HyperText Transfer Protocol*).

Un *ipertesto* è un documento con una struttura non sequenziale, costituito da varie porzioni fra loro collegate, al fine di favorire la consultazione a seconda delle esigenze, senza la rigidità della struttura fisica sequenziale.

A titolo di esempio, vediamo come il contenuto di un libro possa essere organizzato in forma ipertestuale. Un possibile ipertesto potrebbe essere dotato di una struttura gerarchica (Figura 18.1) al cui vertice c'è una descrizione generale (per esempio la prefazione), da cui si accede ai vari capitoli, ognuno dei quali è ancora organizzato con una breve introduzione e vari collegamenti verso i paragrafi. All'interno del testo di ciascun paragrafo ci possono essere riferimenti a un altro paragrafo (come in un libro tradizionale ci sono rimandi tra pagine o paragrafi) o alla bibliografia. Tali collegamenti consentono una lettura non lineare del libro, in cui il lettore può saltare da un punto a un altro del testo, seguendo i collegamenti.

Una struttura ipertestuale come quella descritta può essere consultata in modo efficace con l'ausilio di un sistema informatico che permetta la *navigazione* dei col-

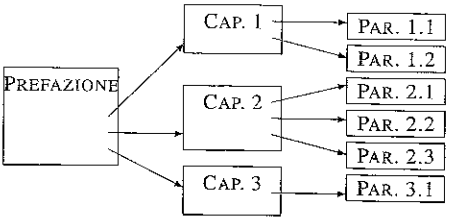

Figura 18.1 Un ipertesto con struttura gerarchica.

legamenti, per esempio attraverso l'uso del mouse che consenta l'accesso immediato alla porzione di testo referenziata dal collegamento ipertestuale selezionato.

Il concetto di ipertesto può essere generalizzato ed esteso in vari modi. In primo luogo, i collegamenti ipertestuali possono essere utilizzati per associare non soltanto le parti di uno stesso documento ma anche documenti differenti, predisposti da persone diverse in momenti diversi. Dal punto di vista concettuale, non c'è differenza fra i legami interni a un documento e quelli verso altri documenti.

Inoltre, i documenti possono essere non solo testuali, ma anche *multimediali*, cioè contenenti risorse non testuali quali immagini, filmati, suoni: si usa in questo caso il termine *ipertesto multimediale*, spesso abbreviato con *ipermedia*.

Infine, qualora si utilizzi la rete Internet come mezzo di pubblicazione e consultazione, l'insieme di documenti che costituiscono l'ipermedia può essere distribuito su nodi diversi (si veda una schematizzazione in Figura 18.2).

In sintesi, possiamo dire che il World Wide Web è un *ipertesto multimediale distribuito*, formato da componenti *autonomi*, in quanto collega documenti e risorse di vario genere prodotti e mantenuti da persone diverse su tutta la rete Internet.

I documenti multimediali pubblicati su Web possono essere realizzati in modo "statico", cioè come file di formato opportuno memorizzati nel file system del server; oppure, come vedremo in questo capitolo, i documenti possono anche essere prodotti dinamicamente, mediante programmi invocabili via Web capaci di costruire sul momento un documento a partire da parametri inviati dal browser. Risulta quindi possibile costruire documenti a partire da informazioni giacenti in una base di dati e utilizzare il browser come interfaccia verso il sistema informativo.

Le componenti tecnologiche del Web che permettono di realizzare l'interscambio di ipertesti multimediali sono descritte nei prossimi sottoparagrafi, e sono: il protocollo HTTP, la tecnica di indirizzamento basata sul concetto di URL e il linguaggio HTML (presentato nel Paragrafo 14.1).

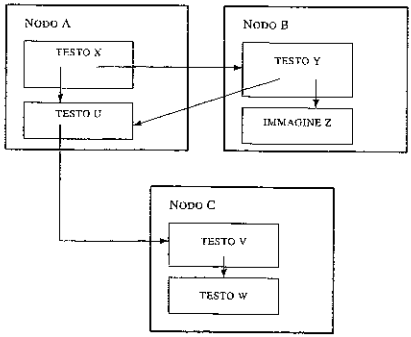

Figura 18.2 Un ipertesto multimediale distribuito.

18.1.3 Il protocollo HTTP

L'interscambio di documenti su Web è regolato da un particolare protocollo client-server, chiamato HTTP (*HyperText Transfer Protocol*). Tale protocollo governa l'interazione tra uno o più client e un server, utilizzando TCP/IP come protocollo di comunicazione di rete.

Il ruolo di client è svolto da un programma denominato *browser*, il cui compito principale è quello di inviare richieste di risorse al server e visualizzare tali risorse in modo da permetterne la consultazione da parte dell'utente. I browser più noti sono quelli delle famiglie *Microsoft Internet Explorer*, *Mozilla Firefox*, *Google Chrome* e *Apple Safari*.

Il server è un particolare sistema software, denominato *Web server*, capace di ricevere richieste di risorse da uno o più browser, reperire le risorse e restituirle al client che ne ha fatto richiesta.

HTTP è stato inventato da Tim Berners-Lee e Robert Cailliau, due ricercatori del CERN di Ginevra (Conseil Européen pour la Recherche Nucléaire), all'interno di un progetto denominato World Wide Web (WWW) avviato nel 1990 allo scopo di costruire un sistema client-server per la distribuzione di manualistica ipertestuale e multimediale sulla rete TCP/IP del CERN.

Il funzionamento di HTTP nella sua versione originale (1.0) prevede un ciclo di interazione tra client e server costituito da quattro fasi, schematizzato nella Figura 18.3:

1. *richiesta di connessione*: il browser (che ha il ruolo di client) contatta il server HTTP al fine di verificarne l'esistenza e la disponibilità; la richiesta prevede l'instaurazione di una connessione secondo il protocollo TCP/IP;
2. *accettazione della connessione*: il server (se disponibile) risponde e la connessione TCP/IP viene così attivata;
3. *richiesta*: il client invia al server HTTP un messaggio, con i dettagli della risorsa richiesta ed eventuali parametri;
4. *risposta*: il server comunica al client l'esito della richiesta e, in caso positivo, i risultati; il server chiude la connessione, senza mantenere una memoria che possa essere riutilizzata nelle richieste successive.

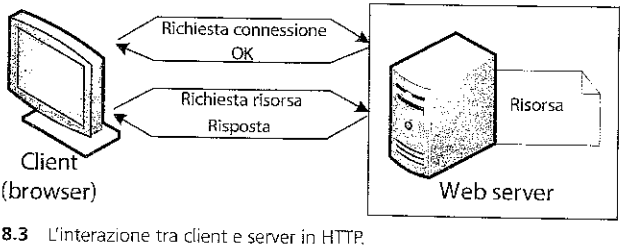

Figura 18.3 L'interazione tra client e server in HTTP.

Il protocollo HTTP ha come scopo principale di consentire al client di emettere richieste di risorse verso il server. Allo scopo di identificare le risorse, HTTP prevede un meccanismo di indirizzamento standard, denominato Uniform Resource Locator (URL). Ogni risorsa disponibile su Web ha un URL distinto, strutturato come segue:

```
http: // <host> [ : <porta> ] [ <path> [ ? <query> ]]
```

Dopo il prefisso `http://` che identifica il protocollo, un URL contiene l'indirizzo IP o simbolico del server (`host`), eventualmente seguito da un numero di porta che consente l'identificazione della particolare porta a cui il server risponde alle richieste, seguito da un pathname (`path`) che identifica la risorsa richiesta nel file system del server, seguito opzionalmente da una serie di parametri detta *query string* (`query`). Per esempio l'URL:

```
http://www.w3c.org/Protocols/Classic.html
```

denota la risorsa rappresentata dalla pagina `Classic.html` memorizzata nel direttorio `Protocols` del server `www.w3c.org`[1].

Quando l'utente attiva un link all'interno di una pagina Web o scrive un URL nel browser, quest'ultimo invia una *richiesta HTTP* al server.

Una richiesta HTTP è una stringa strutturata formata da tre parti: la linea di richiesta (*request line*), uno o più campi opzionali (*header*) e un corpo (*body*), anch'esso opzionale.

La linea di richiesta contiene l'URL della risorsa, il *metodo HTTP* utilizzato dal client per comunicare con il server e la versione del protocollo HTTP usata dal browser. Due sono i metodi principali utilizzabili dal client: *GET* e *POST*; essi differiscono principalmente per le dimensioni dei parametri inviabili al server e per le modalità tecniche con cui tali parametri sono allegati alla richiesta.

A seguito della ricezione di una richiesta HTTP, il server risponde con l'invio di una *risposta HTTP*. La risposta è formata dalla successione di una linea di stato (*status line*), che include la versione di protocollo usata dal server e un codice contenente l'esito della richiesta (per esempio `HTTP/1.1 200 OK`), una serie di campi opzionali (*header*) e un corpo (*body*), che contiene la risorsa restituita al client.

I campi opzionali contenuti nella richiesta e nella risposta assolvono una funzione importante in HTTP, in quanto permettono al client e al server di scambiarsi informazioni utili, quali, per esempio, l'insieme di caratteri e la lingua utilizzata dal client, oppure la durata massima per cui la risorsa restituita dal server può essere tenuta nella memoria cache del client per rispondere a future richieste senza interpellare il server.

La versione successiva di HTTP (1.1) ha apportato varie migliorie alla versione originale, tra cui la più importante è la possibilità di effettuare *connessioni multirichiesta*, in cui il client e il server sfruttano la connessione TCP/IP stabilita all'inizio della sessione interattiva (si veda la Figura 18.3) per effettuare vari cicli di richiesta e risposta, in modo da risparmiare sui tempi di connessione e ridurre i tempi di risposta.

[1] Tale pagina contiene i principali documenti che descrivono l'evoluzione storica di HTTP a partire dalle sue origini.

Nonostante ciò, è opportuno sottolineare alcuni aspetti caratteristici di HTTP, che permangono nella versione 1.1 e rendono questo protocollo diverso dai protocolli client-server precedenti, influenzando le modalità di sviluppo delle applicazioni Web.

- *HTTP non ha memoria* (si dice che è *stateless*): nel caso di operazioni complesse, realizzate attraverso operazioni multiple, il server non è in grado di mantenere informazioni sulle operazioni già effettuate e sul loro esito. Questa scelta è motivata dal desiderio di mantenere semplice il protocollo e dalla possibilità di gestire in questo modo richieste concorrenti senza tenere traccia di ciascuna separatamente. Si tratta di una limitazione forte del protocollo, che rende difficile realizzare procedure composte da sequenze di operazioni in ambiente concorrente, come quelle necessarie per effettuare transazioni su basi di dati.
- *HTTP è asimmetrico* (si dice che è *pull-based*): l'iniziativa di stabilire una connessione HTTP può essere presa soltanto dal client, mentre il server svolge un ruolo passivo. In questo modo non è possibile realizzare applicazioni in cui il server notifica al client l'accadimento di eventi.

Accenneremo nel seguito del capitolo alle tecniche che sono state ideate per superare la prima di queste limitazioni.

18.2 Architetture a tre livelli

Le applicazioni più interessanti del Web, quali i WIS, richiedono pagine costruite a partire dai dati del sistema informativo. Questa capacità non è fornita direttamente da HTTP, ma può essere ottenuta tramite opportune estensioni del Web server, che consentano di effettuare la produzione dinamica delle pagine. Si realizza così un'architettura a tre livelli, che comprende il client, realizzato con un browser standard, il server dei dati, realizzato con un DBMS, e il livello intermedio (*middle tier*), che ospita un Web server opportunamente esteso per l'esecuzione di programmi per l'estrazione dei dati e la costruzione dinamica di pagine Web.

Nel seguito di questo paragrafo illustreremo brevemente architetture a tre livelli via via più potenti e sofisticate (se ne può vedere lo schema nelle Figure 18.5, 18.6, 18.8 e 18.11).

18.2.1 Common Gateway Interface

Common Gateway Interface, comunemente noto come CGI, è il primo e più semplice standard architetturale proposto per la creazione dinamica delle pagine.

CGI si basa su un semplice concetto: utilizzare l'URL della richiesta HTTP per invocare un programma presente sul server, che calcolerà la pagina da restituire al client. In altre parole, in CGI la risorsa richiesta è un programma da eseguire, anziché una pagina Web.

Il ciclo di esecuzione di una richiesta CGI (evidenziato nella Figura 18.4) si articola nei passi sotto elencati:

1. Il browser invia una richiesta al Web server. L'URL di tale richiesta identifica un programma (detto *programma* o *script CGI*) e può contenere uno o più parametri

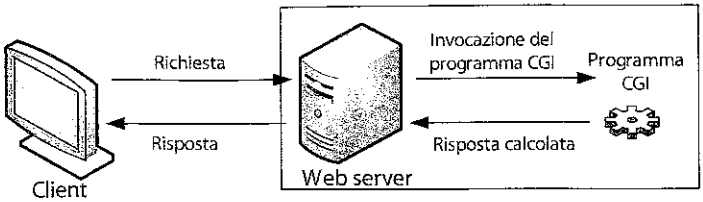

Figura 18.4 Schema della comunicazione tra Web server e programma CGI.

necessari per l'esecuzione dello script. Tali parametri sono inseriti nella parte finale dell'URL, chiamata *query string*:

```
http://www.server.com/nomeScript?par1=val1&par2=val2
```

2. Il Web server, se opportunamente configurato per l'esecuzione di script CGI, riconosce che l'URL fa riferimento a un programma eseguibile, salva in una zona della memoria centrale le cosiddette *variabili CGI*, cioè i parametri della richiesta (tra cui la query string), e lancia in esecuzione il programma, creando un sottoprocesso dedicato.
3. Il programma CGI lanciato in esecuzione legge le variabili CGI, per prelevare gli eventuali parametri della query string, e procede all'elaborazione del codice che costruisce la pagina HTML da restituire al client. Tale elaborazione può richiedere operazioni complesse, per esempio una o più interrogazioni alla base di dati del sistema informativo.
4. Una volta terminato lo script CGI, il Web server usa la pagina HTML prodotta dinamicamente come oggetto della risposta HTTP da inviare al browser, chiudendo il ciclo. Il browser riceve la pagina HTML, che è stata prodotta espressamente a seguito della sua richiesta.

L'esecuzione di uno script CGI in risposta alla richiesta di un client consente la produzione dinamica di pagine Web a partire dal contenuto della base di dati. In questo caso, lo script CGI funge da client nei confronti del DBMS, come illustrato nella Figura 18.5. L'architettura di questa figura è l'esempio più semplice di architettura a tre livelli (*three-tier*), che si presenta come un'evoluzione dell'architettura client-server a due livelli illustrata nel Paragrafo 16.1.

- Il client nell'architettura Web a tre livelli è un'applicazione standard (il browser) che si limita alla resa grafica di pagine HTML; nel caso client-server, invece, il client è un'applicazione dedicata che può svolgere funzioni anche complesse.
- Il server nell'architettura Web a tre livelli si divide in due parti:
  - la base di dati, come nel caso client server, memorizza in modo persistente i dati e permette di effettuare interrogazioni SQL;
  - il middle tier, non presente nell'architettura client-server, contiene il Web server esteso con i programmi CGI e concentra le funzioni per decodificare le ri-

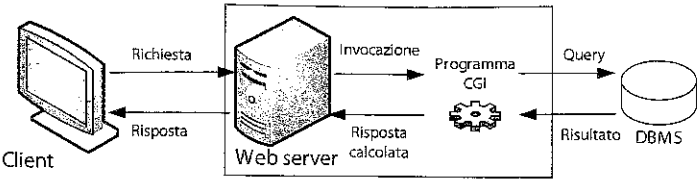

Figura 18.5 Architettura a tre livelli con il middle tier realizzato tramite CGI.

chieste HTTP provenienti dal client e formattare i risultati delle interrogazioni in HTML.

Il termine *gateway*, presente nell'acronimo CGI, si riferisce proprio al ruolo di "tramite" tra le pagine Web e i dati conservati nel sistema informativo svolto dal middle tier.

CGI è il modo più semplice e indipendente dalla piattaforma di collegare il Web e i sistemi informativi; tuttavia due difetti ne diminuiscono l'applicabilità:

- a ogni richiesta CGI il Web server crea un processo, che viene terminato dopo la costruzione della pagina. Questo comportamento genera un elevato sovraccarico delle prestazioni dovuto alla creazione e distruzione di processi;
- la terminazione del processo CGI dopo l'evasione della richiesta che lo ha prodotto non consente di avere a disposizione in memoria centrale strutture dati condivise tra richieste successive, di utenti diversi o dello stesso utente. Quindi, non è possibile mantenere attiva la connessione alla base di dati (tramite memoria condivisa tra utenti diversi) oppure consentire la memorizzazione di risorse temporanee, quali per esempio i contenuti del carrello della spesa in un sito di commercio elettronico (tramite memoria condivisa dalle richieste successive di uno stesso utente).

Queste limitazioni di CGI hanno portato all'ideazione di architetture più potenti, ma anche più complesse.

18.2.2 L'architettura Java Servlet

Le limitazioni di CGI sono frutto di un problema principale: la mancanza di un processo residente posto alle spalle del Web server in grado di eseguire efficientemente i programmi di calcolo delle pagine dinamiche e gestire strutture di memoria condivise.

Le soluzioni architetturali proposte come estensione di HTTP e CGI sono quindi accomunate dalla presenza di un ambiente evoluto di esecuzione dei programmi, realizzato come un processo residente interfacciato direttamente con il Web server.

La Figura 18.6 mostra l'architettura *Java Servlet*, una delle soluzioni più utilizzate per estendere il Web server per l'esecuzione di programmi.

L'aspetto centrale dell'architettura Java Servlet è l'utilizzo, ai fini del calcolo dinamico delle pagine, dell'ambiente di esecuzione di programmi Java chiamato *Java Virtual*

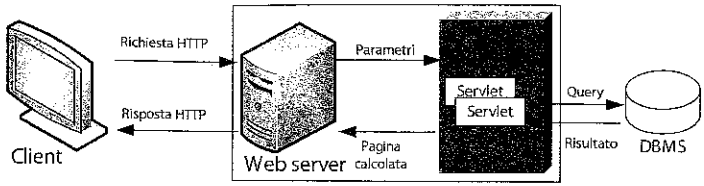

Figura 18.6 L'architettura a tre livelli realizzata con tecnologia Java Servlet.

Machine (JVM). In particolare, la JVM consente l'esecuzione di un programma speciale, detto *servlet container*, che a sua volta mette a disposizione un ambiente ricco di funzionalità sfruttabili per la scrittura di programmi Java di calcolo dinamico di pagine. Tali programmi sono denominati *servlet* e svolgono funzioni simili a quelle degli script CGI. A differenza di quest'ultimi, però, i programmi servlet sono scritti in un linguaggio di programmazione specifico (Java) ed eseguiti all'interno di un ambiente a oggetti (il servlet container) che espone una serie di oggetti di utilità per facilitare l'interazione con il Web server. In particolare, il servlet container offre gli oggetti `request` e `response`, che incapsulano i dati relativi alla richiesta e alla risposta HTTP (tra cui i valori dei parametri della query string), e l'oggetto `session`, che funge da memoria condivisa tra più richieste dello stesso utente. Un programma servlet può quindi interagire con tali oggetti Java, senza doversi preoccupare della comunicazione con il Web server.

Il codice che segue mostra un esempio molto semplice di Java servlet che stampa una pagina HTML contenente il modello di browser usato dall'utente.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class StampaBrowser extends HttpServlet {

 public void doGet(HttpServletRequest request,
                   HttpServletResponse response)
 throws IOException, ServletException {
  response.setContentType("text/html");
  PrintWriter out = response.getWriter();
  out.println("<HTML>");
  out.println("<HEAD>");
  out.println("<TITLE>Stampa del modello di browser
              usato");
  out.println("</TITLE>");
  out.println("</HEAD>");
  out.println("<BODY>");
  out.println("Browser usato:"+request.getHeader
             ("User-Agent"));
```

```
    out.println("</BODY>");
    out.println("</HTML>");
  }
}
```

Il programma servlet consiste di una classe Java (`StampaBrowser`) che estende la classe di libreria `HttpServlet`, ridefinendo il metodo `doGet`. Tale metodo è automaticamente invocato dal servlet container a fronte di richieste HTTP di tipo GET, indirizzate allo specifico programma servlet. Il metodo `doGet` riceve in ingresso i due oggetti `request` e `response`, che fungono da interfaccia verso il Web server. In particolare, l'oggetto `response` viene utilizzato per costruire uno stream di output (`out`) su cui stampare il codice HTML, mentre l'oggetto `request` viene ispezionato tramite il metodo `getHeader` per estrarre il valore del campo `User-Agent` della richiesta HTTP. Il valore trovato viene poi stampato nella pagina HTML, come si vede nella Figura 18.7.

Programma servlet Come esempio di pubblicazione dinamica di contenuti memorizzati in una base di dati, utilizzeremo uno schema relazionale che descrive autori e libri mediante tre relazioni: AUTORE, che descrive gli estremi degli autori, LIBRO, che descrive i dati dei libri, e SCRIVE, che associa libri e autori:

AUTORE(Codice, Nome, Cognome)
SCRIVE(Codice, ISBN)
LIBRO(ISBN, Titolo, Editore, Prezzo)

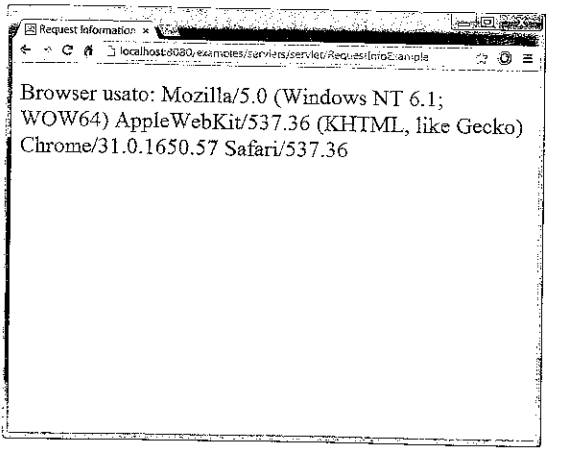

Figura 18.7 Esempio di pagina dinamica calcolata da un programma Java Servlet.

A partire da questo schema, si vuole pubblicare una pagina Web dinamica che contenga l'indice di tutti gli autori, costruita in modo tale che cliccando su un autore compaia una seconda pagina che mostri tutti i suoi scritti. Dalla seconda pagina, un link ipertestuale deve consentire di ritornare all'indice. Per l'accesso alla base di dati, utilizzeremo l'interfaccia JDBC (*Java DataBase Connectivity*), già illustrata nel Capitolo 10, per creare la connessione alla base di dati ed eseguire le interrogazioni.

La pagina dinamica che produce l'elenco dei libri di un autore, identificato da un parametro nella richiesta HTTP, è costruita dal programma servlet mostrato di seguito.

```
import java.io.*; import javax.servlet.*;
import javax.servlet.http.*; import java.sql.*;

public class mostraLibri extends HttpServlet
{ public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
  throws IOException, ServletException
  {
  try {
      // Estrai il parametro "codice" dalla request
      String Codice = request.getParameter("codice");
      // Crea la connessione alla base di dati
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
      Connection conn = DriverManager.getConnection(
                        "jdbc:odbc:DB", "user", "pwd");
      // prepara l'interrogazione
      PreparedStatement pstmt = conn.prepareStatement(
        "select Libro.ISBN, Titolo, Editore, Prezzo "+
        "from Scrive left join Libro on Scrive.ISBN=
         Libro.ISBN" + "where Scrive.Codice = ?");
      // istanzia il parametro dell'interrogazione
      pstmt.setString(1, Codice);
      // esegui l'interrogazione
      ResultSet result = pstmt.executeQuery();
      response.setContentType("text/html"); //il tipo output
      PrintWriter out = response.getWriter();
      out.println("<HTML>");
      out.println("<BODY>");
      out.println("<H1>Elenco dei libri dell'autore</H1>");
      while (result.next()) {
          out.println("<P>");
          out.println(
          "ISBN: " + result.getString("ISBN") + " " +
          "Titolo: " + result.getString("Titolo") + " " +
          "Editore: " + result.getString("Editore") + " " +
          "Prezzo: " + result.getString("Prezzo"));
          out.println("</P>");
      }
      out.println("<A href="indice">Torna all'indice</A>");
      out.println("</BODY>");
```

```
        out.println("</HTML>");
                                // chiudi gli oggetti temporanei
        result.close();
        pstmt.close();
        conn.close();
    }
    catch (ClassNotFoundException e) {// eccezione del driver
       JDBC throw new ServletException(e);
    }
    catch (SQLException e) {          // eccezione del DBMS
        throw new ServletException(e);
    }
  }
}
```

La connessione alla base di dati viene creata in due passi: dapprima si istanzia la classe Java che corrisponde al driver manager JDBC; questo esempio utilizza il driver `jdbc:odbc`, che implementa le funzionalità JDBC sfruttando quelle del driver ODBC dei sistemi Windows. Successivamente si crea un oggetto (`conn`), di classe `Connection`, che rappresenta l'effettiva connessione.

Una volta ottenuto l'oggetto che rappresenta la connessione, si applica a esso la funzione `prepareStatement`, la quale restituisce un oggetto definito di tipo `PreparedStatement`, mediante il quale è possibile eseguire l'interrogazione desiderata. I risultati dell'interrogazione sono memorizzati in un oggetto di tipo `ResultSet`. Il contenuto del result set è scandito mediante il metodo `next` e viene fatto accesso ai campi di ciascun libro mediante l'uso della funzione `getString`.

Il recupero del valore del parametro della query string `codice`, che identifica un autore, sfrutta la funzione Java `getParameter` sull'oggetto che rappresenta la richiesta HTTP. Il valore prelevato dalla richiesta HTTP viene successivamente passato come input, mediante la funzione `setString`, alla query parametrica che estrae dalla base di dati i libri dell'autore specificato.

18.2.3 Template di pagina e server side scripting

Osservando l'esempio di servlet proposto nel paragrafo precedente, balza all'occhio come il codice Java si occupi di stampare non solo la parte variabile della pagina (il modello di browser usato dall'utente o i libri scritti da un certo autore) ma anche il testo e i tag HTML che restano fissi da chiamata a chiamata. Questa considerazione è rilevante, specialmente nello sviluppo di siti di grandi dimensioni, poiché fa sì che un eventuale cambio nella parte fissa delle pagine, per esempio un aggiornamento dell'estetica, richieda la revisione del programma servlet. In altre parole, si è obbligati a utilizzare un programmatore anche laddove servirebbe semplicemente un progettista grafico.

È possibile scrivere programmi per il calcolo dinamico delle pagine con un approccio diverso, basato sull'uso di *template di pagina* e di linguaggi di *server side scripting*; questa soluzione consente una miglior separazione tra le parti fisse e le istruzioni per il calcolo delle porzioni dinamiche di una pagina e rende più semplice aggiornare la parte grafica in modo indipendente dalle istruzioni di programmazione.

Un *template di pagina* può essere definito con un file HTML esteso contenente sia testo con markup HTML, sia *espressioni* e *istruzioni* in un linguaggio di programmazione. Le espressioni corrispondono alle parti dinamiche della pagina, mentre le istruzioni (dette anche *scriptlets*) sono piccoli inserti programmativi il cui scopo è quello di calcolare il valore delle espressioni.

Un esempio molto popolare di linguaggio di server side scripting è costituito da *Java Server Pages* (JSP), una tecnologia Java per l'esecuzione di template programmati in Java all'interno del servlet container.

Il codice JSP che segue illustra, usando la tecnica del server side scripting, la stampa in HTML del modello di browser usato dall'utente, già discussa nel paragrafo precedente:

```
<% String ModBrowser = request.getHeader("User-Agent");%>
<HTML>
 <HEAD>
  <TITLE>Stampa del modello di browser usato dal client
  </TITLE>
 </HEAD>
 <BODY>
   Browser utilizzato: <%= ModBrowser%>
 </BODY>
</HTML>
```

Confrontando i due esempi, emerge con evidenza la maggior chiarezza della soluzione JSP, in cui il template di pagina include per la maggior parte contenuti fissi e solo in piccola percentuale istruzioni di programmazione. In particolare, la riga:

```
<% String ModBrowser = request.getHeader("User-Agent");%>
```

contiene un'istruzione (contenuta tra i delimitatori `<%` e `%>`) che assegna alla variabile `ModBrowser` il valore del campo HTTP `User-Agent`.

Le espressioni hanno una sintassi leggermente diversa: sono precedute dal simbolo `<%=` e chiuse dal delimitatore `%>`. Il significato è quello di stampare al posto dell'espressione il valore ottenuto calcolando la formula specificata. Per esempio, l'espressione `<%= ModBrowser%>` stampa il valore della variabile *ModBrowser* al posto giusto nella pagina.

Programma JSP Riformuliamo l'esempio del paragrafo precedente con un template JSP che costruisce una pagina HTML contenente la lista dei libri scritti da un particolare autore.

```
<%@page language="java" %>
<%@page import="java.sql.*" %>
<%                  // Estrai il parametro dalla request
String Codice = request.getParameter("codice");
                    // Crea la classe Java per il driver JDBC
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

```
                                    // Crea l'oggetto connessione
Connection conn = DriverManager.getConnection(
                        "jdbc:odbc:DB","user", "pwd");
                        // prepara l'interrogazione
PreparedStatement pstmt = conn.prepareStatement(
   "select Libro.ISBN, Titolo, Editore, Prezzo" +
   "from Scrive left join Libro on Scrive.ISBN=Libro.ISBN" +
   "where Scrive.Codice = ?");
                    // istanzia il parametro
dell'interrogazione pstmt.setString(1, Codice);
                                    // esegui l'interrogazione
ResultSet result = pstmt.executeQuery();
%>
<%// produci il markup fisso e il contenuto variabile%>
<HTML>
 <BODY>
  <H1>Elenco dei libri dell'autore selezionato</H1>
  <% while (result.next()) { %>
  <P>
     ISBN: <%= result.getString("ISBN") %>
     Titolo: <%= result.getString("Titolo") %>
     Editore: <%= result.getString("Editore")%>
     Prezzo: <%= result.getString("Prezzo")%>
  </P>
  <% }%>
  <A href="indice.jsp">Torna all'indice</A>
 </BODY>
</HTML>
<%
 result.close();  // chiudi gli oggetti temporanei
 pstmt.close();
 conn.close();
%>
```

La parte iniziale del template crea la connessione alla base di dati ed effettua l'interrogazione, come nel programma servlet del paragrafo precedente. La produzione del markup fisso e del contenuto variabile sfrutta invece la possibilità di inframmezzare al markup HTML istruzioni di scripting ed espressioni.

Dal punto di vista tecnologico, un template JSP viene eseguito esattamente nello stesso modo di un programma servlet; in effetti non è altro che un particolare programma servlet. Infatti, all'atto della prima richiesta di un template JSP, il servlet container abilitato all'esecuzione di file JSP traduce il template in un programma servlet equivalente, compila il sorgente Java così ottenuto ed esegue il codice risultante. La Figura 18.8 mostra l'architettura a tre livelli realizzata con JSP.

Tag eseguibili dal server L'uso di template di pagina non risolve del tutto l'inconveniente di avere nel medesimo file sorgente istruzioni di programmazione, testo e markup.

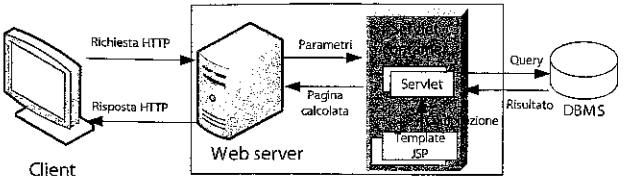

Figura 18.8 Architettura a tre livelli realizzata con JSP.

Una tecnica di progettazione dei template di pagina, proposta a partire dalla versione 1.1 di JSP e ripresa nell'architettura *Microsoft.NET*, propugna la sostituzione del server side scripting con *tag eseguibili dal server*.

Un tag eseguibile dal server è a tutti gli effetti simile a un tag HTML, e come tale può essere inserito nel template di pagina in modo analogo a un paragrafo o a un'immagine HTML.

Tuttavia, un tag eseguibile è associato a un programma opportuno, per esempio una classe Java in ambiente JSP o una classe C#[2] in ambito Microsoft.NET. Il programma associato al tag viene lanciato in esecuzione quando il processore di template (sia esso un servlet container abilitato all'esecuzione di JSP 1.1 o l'esecutore di template ASP.NET) rileva la presenza del tag nella pagina. L'esecuzione solitamente comporta la produzione di contenuti dinamici, quali per esempio porzioni di HTML, che vengono inseriti nella pagina in costruzione al posto del tag eseguibile.

Il codice che segue mostra un esempio di tag eseguito dal server, tratto dalle librerie di tag della piattaforma Microsoft.NET.

```
<HTML>
  <BODY>
    <P>Un tag eseguibile che mostra un calendario</P>
    <asp:Calendar id="MyCal" runat=server/>
  </BODY>
</HTML>
```

Il corpo del documento include il tag `<asp:Calendar>`, contenente un attributo `runat` con valore `server`, che indica che il tag deve essere eseguito dal server. L'esecuzione produce il codice HTML che mostra il calendario visibile nella Figura 18.9.

18.2.4 Model-View-Controller

Il template JSP che mostra i libri scritti da uno specifico autore, mostrato nel paragrafo precedente, evidenzia una certa sovrapposizione tra differenti aspetti della programmazione del middle tier.

[2]C#, pronunciato C Sharp, è un linguaggio proposto da Microsoft per la programmazione dei template e lo sviluppo di applicazioni Web nell'architettura .NET.

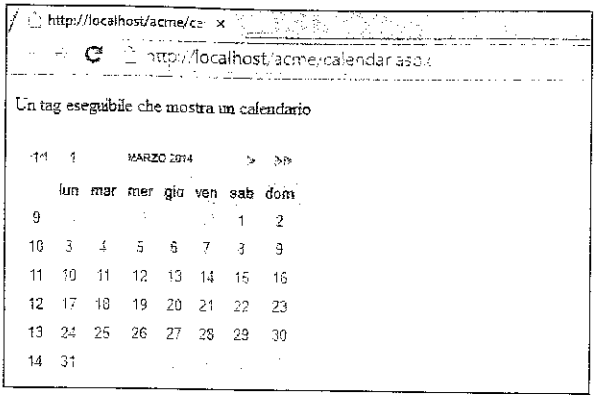

Figura 18.9 Esempio di pagina dinamica contenente un tag eseguibile dal server.

- La logica di accesso ai dati, cioè la parte di codice che si occupa della connessione alla base di dati e della formulazione delle interrogazioni.
- La logica di presentazione, che si occupa di produrre i tag HTML necessari per la visualizzazione.
- La logica di controllo, che determina il comportamento dell'applicazione a fronte dell'interazione dell'utente; nell'esempio di template JSP, tale logica riguarda il link ipertestuale per tornare all'indice, che contiene il nome dello specifico template da eseguire.

La concentrazione di questi tre fattori in un singolo modulo software rende più difficili il riutilizzo e la manutenzione evolutiva, poiché non è possibile apportare modifiche a ciascun aspetto indipendentemente, per esempio cambiare il nome della pagina con l'indice degli autori, senza intervenire sul codice del template che mostra i libri. Queste considerazioni hanno spinto a studiare tecniche di organizzazione dei componenti del middle tier in grado di facilitare la separazione degli aspetti e garantire migliore manutenzione e scalabilità. Tra le soluzioni proposte spiccano i cosiddetti *presentation framework* basati sull'architettura *Model-View-Controller* (MVC). Il termine *framework* denota un insieme di moduli software di supporto all'esecuzione di programmi scritti dallo sviluppatore; *presentation* si riferisce al fatto che tali moduli servono per la costruzione dinamica delle pagine da presentare all'utente; infine, MVC descrive un modo particolare di organizzare il presentation framework, secondo un modello architetturale originariamente concepito per l'ambiente di programmazione a oggetti Smalltalk-80.

In concreto, MVC, schematizzato nella Figura 18.10, prescrive una ripartizione modulare del codice e assegna le responsabilità dei vari moduli:

- un primo modulo, detto *Model*, rappresenta lo stato dell'applicazione, per esempio i principali oggetti del dominio applicativo;

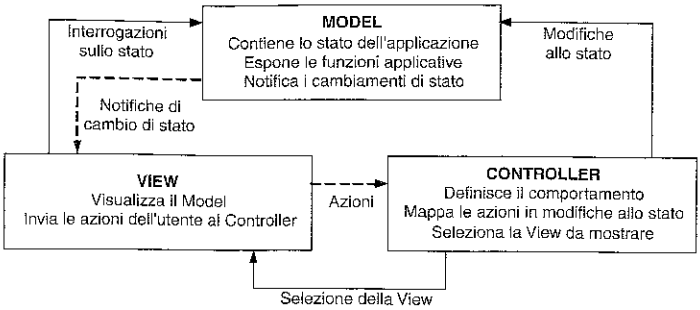

Figura 18.10 Gli elementi dell'architettura Model-View-Controller.

- un ulteriore modulo, detto *View*, presenta all'utente gli oggetti del Model e i comandi per l'interazione;
- un ultimo modulo, detto *Controller*, traduce le azioni effettuate dall'utente sulla View in chiamate alle funzioni di aggiornamento del Model. In base all'interazione dell'utente e all'esito delle azioni invocate sugli oggetti del Model, il Controller seleziona la View da mostrare in risposta all'utente.

Lo schema MVC garantisce un miglior isolamento tra i vari moduli, grazie ai principi di:

- *separazione del Controller*: le azioni dell'utente sono trattate unicamente dal Controller, che è l'unico responsabile del modo con cui formulare la risposta;
- *riutilizzabilità del Model*: il Model non dipende dalla View, per cui può essere riutilizzato con View differenti;
- *indipendenza della View*: la View non dipende dalle modalità di aggiornamento del Model, per cui non risente di cambiamenti nella logica di gestione del Model.

L'architettura MVC originale è stata adattata al Web, e in particolare alla piattaforma Java Servlet, per sopperire alla mancanza in HTTP di meccanismi per mantenere lo stato dell'interazione e per notificare al client cambiamenti insorti negli oggetti del Model residenti sul server.

La realizzazione più famosa dell'architettura MVC per il Web è costituita da Struts, un sistema creato all'interno dell'Apache Software Foundation. Struts può essere definito come un'implementazione avanzata del Controller per la piattaforma Java Servlet, utilizzabile in abbinamento a oggetti del Model realizzati in Java.

L'architettura di Struts, illustrata nella Figura 18.11, è una specializzazione dello schema MVC originario, mostrato nella Figura 18.10.

In Struts, il generatore delle richieste è il browser. Quando l'utente attiva un collegamento ipertestuale nella pagina HTML o usa una form HTML per immettere dati, la richiesta HTTP viene indirizzata al programma Struts Controller, composto

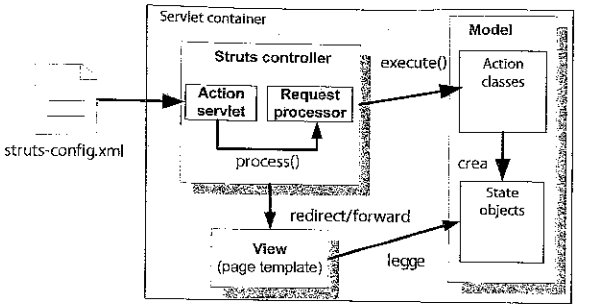

Figura 18.11 L'architettura di Apache Struts.

da un programma servlet (`ActionServlet`) e da una classe di utilità (`RequestProcessor`). Il Controller decide la sequenza di azioni necessarie per soddisfare la richiesta; le azioni ammissibili sono rappresentate sotto forma di componenti object-oriented, chiamati *action class*.

Specificamente, il Controller associa la richiesta HTTP alla corrispondente azione creando o riutilizzando un oggetto di una delle action class e chiamando una sua funzione predefinita (tramite il metodo `execute()`). Nella situazione più semplice, un'action class contiene tutta la logica necessaria a servire la richiesta HTTP; in scenari più complessi l'action class svolge puramente un ruolo di intermediazione nei confronti di altri oggetti, appartenenti al Model, che contengono la logica di business vera e propria. Esempi di azioni eseguite o mediate da un'action class possono essere l'esecuzione di un'interrogazione della base di dati, l'invio di messaggi di posta elettronica, oppure l'autenticazione dell'utente.

Al termine dell'esecuzione, l'azione comunica l'esito al Controller, che decide il passo successivo. Tipicamente, il Controller invoca un template JSP, facente parte della View, il quale accede agli oggetti del Model che contengono lo stato corrente dell'applicazione e costruisce una pagina HTML che visualizza tale stato all'utente.

L'aspetto più importante del Controller Struts è la sua parametricità: le regole che determinano l'azione da invocare a fronte di una richiesta e il template da usare per la visualizzazione di una risposta non sono cablate nel codice, ma espresse in modo dichiarativo con XML e memorizzate nel file `struts-config.xml`. In questo modo è possibile modificare la logica di controllo senza toccare il codice sorgente né interrompere l'esecuzione dell'applicazione, il che rende il comportamento dell'applicazione più facile da cambiare.

Questo modo di strutturare l'applicazione consente una netta separazione dei compiti tra i vari elementi dell'architettura.

- Le decisioni di controllo sono descritte nel file `struts-config.xml`. Il Controller, grazie al file di configurazione, determina l'action class da invocare.
- L'action class e il Controller ignorano l'indirizzo dei template JSP usato per visualizzare la risposta; tali indirizzi sono contenuti nel file `struts-config.xml`.

- Gli oggetti del Model non dipendono dall'architettura Web e quindi possono essere riutilizzati in contesti differenti.
- I template JSP della View operano su strutture dati standard (per esempio, oggetti Java Bean) e ignorano il modo con cui tali oggetti sono costruiti.

Oltre agli aspetti architetturali descritti, la piattaforma Struts offre varie funzioni di utilità, tra cui un meccanismo per la specifica dichiarativa di regole di verifica della correttezza dei dati inseriti dall'utente in form HTML.

18.3 Architetture multi-livello

Le architetture Web di tipo industriale devono essere in grado di soddisfare svariati requisiti non funzionali, come la sicurezza, le prestazioni, la scalabilità e la disponibilità. A questi si aggiunge la necessità di garantire la massima riutilizzabilità del codice, specialmente in ambienti di tipo aziendale in cui diverse architetture, oltre a quella Web, sono impiegate per realizzare applicazioni.

Queste motivazioni spingono a un ulteriore potenziamento dell'architettura a tre livelli descritta nel Paragrafo 18.2, con l'aggiunta di un nuovo elemento.

18.3.1 Il concetto di application server

L'architettura mostrata nella Figura 18.12 interpone tra il middle tier e la base di dati un nuovo elemento, chiamato *application server*.

L'application server è un sistema dedicato all'esecuzione di componenti riutilizzabili che possono essere sfruttati dai template per la costruzione dinamica delle pagine o da servlet per l'aggiornamento dei contenuti del sistema informativo.

L'architettura con application server è anche denominata *multi-livello* (*multi-tier*), in contrapposizione con l'architettura client-server, che prevede due soli livelli. Essa può anche essere vista come una generalizzazione di quella a tre livelli esemplificata nella Figura 18.6.

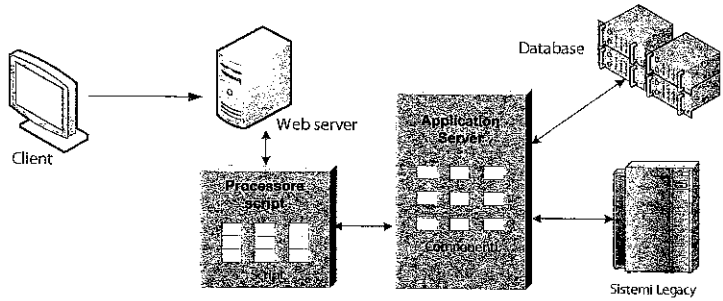

Figura 18.12 Architettura Web con application server.

L'application server offre un ambiente di esecuzione costruito in modo da garantire vari vantaggi, sia per la scalabilità, sia per lo sviluppo software delle applicazioni Web.

- Gestione trasparente della distribuzione, replicazione e bilanciamento del carico di lavoro: l'application server può essere distribuito su più macchine per aumentare le prestazioni e fornisce funzionalità per la replicazione automatica dei componenti e la distribuzione del carico alle varie istanze. L'application server gestisce l'insieme delle istanze di ciascun componente (detto *object pool*); se un componente subisce un picco di richieste, il numero di istanze nell'object pool viene incrementato; quando il carico diminuisce, le istanze in eccesso vengono deallocate in modo da liberare risorse per altre applicazioni. Questo meccanismo, detto *bilanciamento dinamico* (*dynamic load balancing*), garantisce una distribuzione ottimale delle risorse di calcolo, senza che il programmatore debba occuparsene.
- Ripristino da guasti: l'ambiente di esecuzione è in grado di rilevare l'interruzione del funzionamento di un server o di un singolo componente e di indirizzare automaticamente le richieste di servizio inviate al componente guasto verso altre istanze dello stesso componente. In questo modo, il client non si accorge del guasto, se non per un peggioramento delle prestazioni.
- Condivisione delle risorse: l'application server gestisce in modo ottimizzato insiemi di risorse condivise da più componenti, quali per esempio le connessioni verso DBMS.
- Gestione delle transazioni: appoggiandosi su uno o più DBMS, l'application server può consentire all'applicazione client di effettuare sequenze di operazioni sui componenti in modo transazionale, offrendo le classiche funzioni di commit e rollback.
- Interoperabilità con i sistemi legacy: gli application server possono includere librerie per la costruzione di componenti capaci di connettersi in lettura o scrittura a sistemi legacy o proprietari, quali per esempio i sistemi gestionali ERP (Enterprise Resource Planning).
- Ambiente di sviluppo a oggetti: gli application server consentono di programmare i componenti usando linguaggi a oggetti. In tal modo, i componenti risultano meglio strutturati e più facilmente riutilizzabili.
- Gestione di diversi protocolli applicativi: l'application server può essere usato da client che adottano diversi protocolli di comunicazione, quali per esempio il protocollo di distribuzione degli oggetti nativo del linguaggio Java, Java Remote Method Invocation (RMI), oppure il protocollo distribuito Corba Internet InterOrb Protocol (IIOP) definito dall'Object Management Group (OMG).

Come si può notare dal lungo elenco di funzionalità sopra illustrato, l'application server offre alle applicazioni client una serie di servizi che includono alcuni di quelli solitamente garantiti dal DBMS. In qualche misura, si può dire che l'architettura con application server esporta il meglio della tecnologia delle basi di dati verso il mondo delle applicazioni. In molti prodotti commerciali, le funzioni descritte, per esempio il supporto transazionale, sono in realtà costruite appoggiandosi su un DBMS.

18.3.2 Enterprise Java Bean

Una delle realizzazioni più note dell'architettura con application server si fonda sulla piattaforma Java 2 Enterprise Edition (J2EE), proposta originariamente da Sun Microsystems e ora di Oracle.

In particolare, lo sviluppo dei componenti all'interno dell'application server sfrutta lo standard *Enterprise Java Beans* (EJB), un'interfaccia programmativa per la definizione di oggetti Java distribuiti.

Un Enterprise Java Bean è un oggetto Java eseguito all'interno di un particolare ambiente (detto *EJB container*) installato nell'application server. L'EJB container fornisce tutti i servizi caratteristici dell'application server (sicurezza, load balancing dinamico, object pooling, gestione ottimizzata delle risorse condivise); in cambio, il programmatore deve rispettare una serie di convenzioni e vincoli nella scrittura del codice, necessari affinché il componente EJB possa essere adeguatamente gestito nell'ambiente di esecuzione.

La programmazione degli EJB, a partire dalla versione 3.0 dello standard, non richiede grandi sforzi aggiuntivi rispetto alla normale programmazione in Java. In particolare, la realizzazione di un componente EJB richiede lo sviluppo di un'interfaccia, che espone al programma client i metodi utilizzabili, e di una classe, che contiene l'effettiva implementazione dei metodi. Questa modalità di organizzazione del codice segue il principio generale della programmazione modulare, che distingue interfaccia e implementazione per mascherare ai programmi client i dettagli interni della realizzazione di un componente.

La Figura 18.13 illustra le principali modalità d'uso di un componente EJB da parte di un'applicazione client; quest'ultima può essere sia un'applicazione residente nel Web server, per esempio un programma servlet o un template JSP, sia un altro component EJB, sia un'applicazione non-Web scritta in Java, sia un Web Service residente su un diverso host.

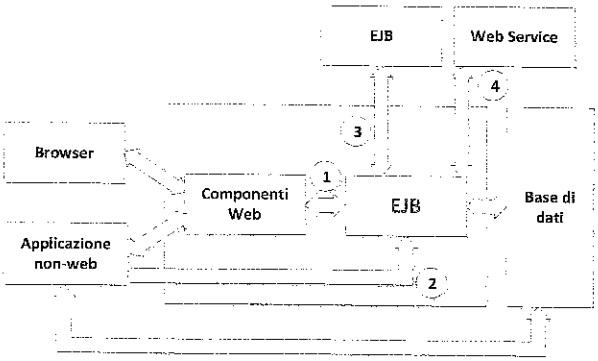

Figura 18.13 Le diverse modalità d'uso di un componente EJB.

I componenti EJB si specializzano in diverse categorie dalle funzionalità differenti, tra cui le principali sono:

- *session bean*: sono componenti destinati a svolgere funzioni di utilità; si distinguono in *stateless*, quando non richiedono la memorizzazione di informazione di stato, e *stateful*, quando richiedono di mantenere alcune informazioni tra una chiamata di un metodo e la successiva. Come esempio di stateless session bean si può citare un oggetto che offre una funzione per la conversione di dati da un sistema di misura a un altro; il tipico esempio di stateful session bean è il carrello della spesa, cioè un oggetto in grado di memorizzare temporaneamente l'insieme degli acquisti di un utente;
- *message bean*: sono componenti utilizzati per lo scambio di messaggi in modo asincrono. Un esempio di message bean è l'oggetto che rappresenta la notifica di pagamento dal sistema bancario al sistema di commercio elettronico;
- *entity bean*: sono componenti che rappresentano l'immagine in memoria centrale di oggetti persistenti memorizzati nella base di dati. Si distinguono dai session bean in quanto racchiudono informazione persistente, tipicamente con durata superiore a quella delle applicazioni client. Un esempio di entity bean è l'oggetto che rappresenta i dati del prodotto acquistato in un sito di commercio elettronico. Gli entity bean sono standardizzati nella specifica *Java Persistence API* (JPA) descritta nel Capitolo 10 e sono utilizzabili anche al di fuori dell'EJB container.

Normalmente, l'applicazione client interagisce solamente con session bean, i quali a loro volta possono utilizzare entity bean per realizzare funzioni di lettura e aggiornamento dei dati persistenti. Nel Paragrafo 10.5 sono stati presentati gli aspetti più significativi relativi alla gestione della persistenza per gli entity bean.

A conclusione di questa breve illustrazione di EJB vale la pena di ricordare come tale tecnologia, data la sua maggiore complessità rispetto a soluzioni quali Java Servlet, JSP e framework MVC, sia conveniente solo nel caso in cui i requisiti di isolamento dei dati e di riutilizzo degli oggetti di business siano estremamente stringenti. Infatti, è comunque possibile ottenere architetture Web affidabili e ad alte prestazioni anche con l'utilizzo delle più semplici architetture a tre livelli.

18.4 Architetture a servizi

L'avvento del Web ha stimolato fortemente l'integrazione organizzativa e tecnologica delle aziende, grazie alla disponibilità di infrastrutture di comunicazione standard e a basso costo. Questo fenomeno ha comportato una crescente necessità di far dialogare sistemi e applicazioni eterogenei e ha spinto alla concezione di una piattaforma tecnologica capace di facilitare l'interoperabilità basandosi sugli standard linguistici e architetturali del Web.

18.4.1 Il concetto di Web Service

La risposta a queste esigenze è il concetto di *Web Service*, definito dal W3C come un sistema per realizzare interazioni tra macchine in rete mediante la descrizione di

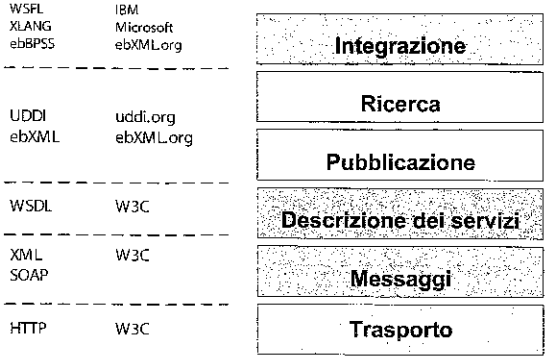

Figura 18.14 I livelli protocollari dell'architettura con Web Service.

interfacce in un formato processabile del calcolatore e lo scambio di messaggi. La definizione poggia su due aspetti salienti: la modalità di rappresentazione dell'interfaccia, che sfrutta un linguaggio denominato *Web Service Description Language* (WSDL), e lo scambio di messaggi, che utilizza il protocollo di trasporto dei messaggi denominato SOAP. Questi due standard saranno descritti in maggior dettaglio nel seguito.

A differenza di altri standard precedenti per la costruzione di sistemi distribuiti, SOAP e WSDL si basano unicamente sull'architettura aperta del Web, e in particolare richiedono come prerequisiti soltanto XML e il protocollo HTTP. Essi si inseriscono in un'architettura complessiva per l'uso dei Web Service, articolata nei sei diversi livelli funzionali illustrati nella Figura 18.14. A ogni livello corrispondono uno o più standard; come si nota, i livelli inferiori della pila protocollare sono caratterizzati da standard univoci e consolidati, mentre per i livelli superiori esistono proposte alternative ancora in corso di sviluppo. Vediamo ora in sintesi le funzionalità dei vari livelli.

- *Livello di trasporto*:[3] copre gli aspetti relativi allo scambio di messaggi tra applicazione e Web Service. Il meccanismo più utilizzato è il protocollo HTTP, anche se è possibile trasportare messaggi con altri protocolli, basati sia su standard aperti per Internet (per esempio il protocollo di email SMTP), sia su prodotti proprietari.
- *Livello dei messaggi*: standardizza le regole di formattazione dell'informazione scambiata tra Web Service e applicazione; sfrutta il formato di interscambio dati di uso generale XML e il protocollo SOAP, specifico dei Web Service.

[3]La nozione di trasporto nel contesto dei Web Service non coincide con l'omonima nozione nel modello di comunicazione ISO/OSI; il livello di trasporto dei Web Service corrisponde al livello di applicazione del modello ISO/OSI.

- *Livello di descrizione dei servizi*: stabilisce un formato XML per la rappresentazione delle interfacce offerte da un Web Service, utilizzabile dal client per determinare le operazioni disponibili e i dati di input e di output di ciascuna operazione.
- *Livello di pubblicazione*: codifica le modalità con cui i produttori di Web Service possono pubblicizzare i propri servizi su Web in modo da renderli reperibili agli utenti ed è strettamente correlato al livello di ricerca, che definisce le modalità con cui un client determina il Web Service da usare. Lo standard principale per la pubblicazione e la ricerca dei servizi è *Uniform Description, Discovery and Integration* (UDDI), che definisce le modalità di pubblicazione dei servizi da parte dei fornitori e le modalità di ricerca da parte degli utenti.
- *Livello di ricerca (discovery)*: la ricerca del servizio può avvenire *staticamente*, nel qual caso lo sviluppatore seleziona il servizio da usare prima di rilasciare in esecuzione l'applicazione, oppure *dinamicamente*, mediante un processo di negoziazione in cui l'applicazione analizza durante l'esecuzione le caratteristiche dei servizi disponibili e sceglie quello più adatto.
- *Livello di integrazione*: è posto alla sommità della pila protocollare e si occupa degli aspetti legati all'uso contemporaneo di più Web Service, per esempio nel contesto di sistemi transazionali o di flussi di lavoro automatizzati in cui diversi passi di una procedura richiedono l'interazione con servizi differenti. In questo settore, la ricerca accademica e industriale è ancora attiva e diverse proposte di linguaggi per esprimere l'integrazione di Web Service sono in via di sperimentazione.

Recentemente, sono stati proposti meccanismi molto sofisticati per la ricerca dinamica e l'integrazione automatica dei Web Service, basati sul cosiddetto *Semantic Web*, di cui si è parlato nel Capitolo 15. L'obiettivo è di utilizzare linguaggi di descrizione dei servizi e tecniche di inferenza logica nel processo di confronto (matching) tra l'obiettivo dell'applicazione e le capacità dichiarate dal servizio. Tra le proposte di ricerca in questo settore citiamo OWL-S [133], realizzato da un consorzio di università e imprese sostenuto dall'agenzia americana DARPA, e Web Service Modeling Ontology (WSMO) (`http://www.wsmo.org/`), definito dai membri della European Semantic Systems Initiative (ESSI).

Infine, vari esempi di Web Service, sia commerciali sia sperimentali, sono disponibili su Web. Tra i servizi più noti possiamo citare il Web Service di Google per effettuare interrogazioni nel motore di ricerca dall'interno di applicazioni Web, i servizi di Amazon per la ricerca di informazioni commerciali sui prodotti in vendita, e il servizio MapPoint di Microsoft, per la costruzione dinamica di mappe e percorsi a partire dai dati di località geografiche.

18.4.2 Il protocollo SOAP

SOAP[4] è un protocollo relativamente semplice, unidirezionale e privo di memoria, per l'invio di messaggi XML da un mittente a un destinatario. SOAP non pone vincoli

[4] Il termine SOAP rappresentava originariamente un acronimo per *Simple Object Access Protocol*; ora è da interpretare come un semplice nome.

sul contenuto e sul significato dei messaggi scambiati, né sul protocollo di comunicazione da usare, né sulle regole di garanzia della qualità del servizio (affidabilità e routing).

SOAP detta la struttura generale del messaggio, che deve essere contenuto in un elemento XML, detto *envelope*, al cui interno si distinguono una sezione per i metadati relativi al messaggio, detta *header*, e una sezione per il contenuto del messaggio, detta *body*. La struttura e gli elementi XML utilizzati all'interno dello header e del body sono dipendenti dal servizio e non prescritti dal protocollo.

Oltre alla struttura dei messaggi, SOAP descrive anche le regole fondamentali per la loro elaborazione da parte dei nodi della rete, denominati *processori SOAP*. Si tratta di regole molto generali che si limitano a prescrivere ai processori SOAP l'obbligo di riconoscere le parti standard di un messaggio, ma che consentono anche di definire ruoli personalizzati e di assegnare comportamenti più stringenti a nodi che dichiarino di svolgere tali ruoli. Il numero e il tipo dei ruoli rilevanti e il comportamento associato ai relativi processori dipendono dalla particolare applicazione.

SOAP prescrive anche le regole per la rappresentazione di tipi di dati complessi in XML, un procedimento denominato *SOAP encoding*. Infine, esso fornisce due modalità alternative di confezione dei messaggi XML. La prima va sotto il nome di modalità *RPC-style* e prevede l'uso di elementi XML convenzionali per esprimere i diversi componenti di una chiamata di funzione e della relativa risposta. Questo stile di formulazione dei messaggi si usa quando il Web Service espone operazioni con pochi parametri. Quando l'applicazione e il servizio devono scambiarsi ed elaborare interi documenti, il formato RPC non è adatto e si preferisce usare la modalità chiamata *document style*, che prevede di allegare un intero documento all'interno del SOAP body.

Come esempio di messaggio con RPC-style, si consideri la chiamata a un Web Service che implementa la funzione `double GetStockQuote([in] string sSymbol)`, con valore del parametro attuale uguale a `IBM`; in SOAP tale chiamata si esprime con il messaggio seguente:

```
POST /InStock HTTP/1.1
Host: www.example.org
Content-Type: application/soap+xml; charset=utf-8
Content-Length: X

<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">

  <soap:Body xmlns:m="http://www.example.org/stock">
    <m:GetStockPrice>
      <m:StockName>IBM</m:StockName>
    </m:GetStockPrice>
  </soap:Body>

</soap:Envelope>
```

L'esempio riporta anche l'intestazione della richiesta HTTP e mostra così il caso frequente in cui il messaggio SOAP è trasportato con tale protocollo.

18.4.3 Il linguaggio WSDL

Il protocollo SOAP si limita a specificare il singolo messaggio e non consente di rappresentare le operazioni offerte da un Web Service. A questo scopo, si usa il linguaggio WSDL che fornisce un insieme di elementi XML con cui descrivere un servizio, le sue operazioni, i messaggi e i tipi di dati necessari per l'interazione.

Una specifica WSDL è un documento XML con una struttura divisa logicamente in due parti (si veda la Figura 18.15).

Una parte astratta che descrive i tipi di dati, i messaggi scambiati e le operazioni offerte, raggruppate in interfacce denominate *portType*; una parte concreta che specifica le modalità di scambio messaggi (*binding*) e la dislocazione fisica del servizio, strutturato come un insieme di porte (*port*) raggruppate in servizi (*service*), che implementano le interfacce astratte.

Le operazioni WSDL di un portTpye possono essere di quattro tipi:

- *one-way*: l'operazione riceve un messaggio e non invia risposta;
- *request-response*: l'operazione riceve un messaggio e restituisce una risposta;

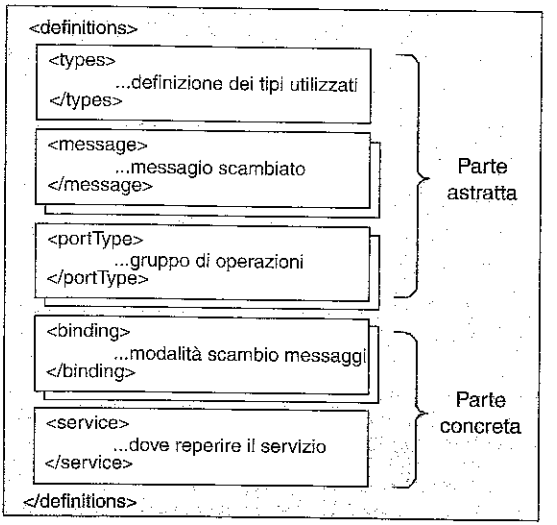

Figura 18.15 Le parti essenziali di un documento WSDL.

- *solicit-response*: l'operazione invia un messaggio e attende una risposta;
- *notification*: l'operazione invia un messaggio senza attendere risposta.

Il documento XML seguente illustra la descrizione WSDL di un Web Service che offre un portType contenente un'operazione di tipo request-response.

Cominciamo con il mostrare la parte astratta, che include la dichiarazione dei messaggi di richiesta e di risposta e la specifica dell'operazione `traduci` all'interno del portType `servizioDizionario`:

```
<message name="richiestaTraduzione">
   <part name="vocabolo" type="xs:string"/>
</message>
<message name="rispostaTraduzione">
   <part name="traduzione" type="xs:string"/>
</message>

<portType name="servizioDizionario">
  <operation name="traduci">
         <input message="richiestaTraduzione"/>
         <output message="rispostaTraduzione"/>
  </operation>
</portType>
```

La parte concreta del servizio richiede di esprimere lo stile di chiamata (con le opzioni `RPC` o `document`) associato al portType, il protocollo di trasporto (tipicamente HTTP) e, per ogni operazione, l'URL corrispondente all'implementazione dell'operazione (`soapAction`) e l'eventuale convenzione di codifica dei messaggi (la parola chiave `literal` significa che non si usa alcuna codifica):

```
<binding type="servizioDizionario" name="bind1">
<soap:binding style="document"
  transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation>
    <soap:operation soapAction="http://www.esempio.org
    /getTr"/>
      <input>  <soap:body use="literal"/> </input>
      <output> <soap:body use="literal"/> </output>
  </operation>
</binding>
```

Il documento WSDL si conclude con la specifica del *service*, cioè dei dati relativi all'implementazione di una coppia portType + binding. In questo modo è possibile descrivere implementazioni multiple dello stesso portType, abbinate a diverse opzioni di binding.

```
<service name="dizionario">
  <port name="servizioDizionario" binding="bind1">
    <soap:address location="http://www.esempio.org"/>
  </port>
</service>
```

18.4.4 I Web Service con REST

In epoca successiva all'introduzione di SOAP e WSDL come strumenti per la realizzazione dei Web Service si è andato affermando uno stile architetturale alternativo per la costruzione di applicazioni distribuite per il Web: *Representational State Transfer* (REST) [115].

REST nasce come una generalizzazione dell'architettura del Web, volta ad applicarne i principi di base non solo all'interazione uomo-macchina ma anche a quella tra applicazioni distribuite. La semplicità di REST e la sua generalità ne hanno fatto in breve tempo lo stile architetturale predominante per le cosiddette *Web Application Programming Interface* (Web API), cioè le interfacce che le applicazioni espongono sul Web per permettere l'interazione da parte di altri sistemi. Oggi sono migliaia le applicazioni che offrono i propri servizi sul Web usando REST e tra queste si annoverano tutte le applicazioni più popolari, quali le principali reti sociali, i motori di ricerca, i sistemi di cartografia digitale e le applicazioni di e-commerce.

REST non è né una nuova tecnologia, né un linguaggio, né un protocollo; si tratta di una serie di regole architetturali per la costruzione di sistemi distribuiti che generalizzano il modo di funzionare del Web. Tali regole si possono riassumere nei principi seguenti.

- *Identificazione delle risorse*: le applicazioni distribuite si scambiano *risorse*. Per esempio, un catalogo di commercio elettronico permette di interrogare i dati dei prodotti, delle offerte speciali, dei punti di vendita e così via. Questo principio richiede che sia assegnato un identificativo univoco a ciascuna delle risorse gestite dall'applicazione. Tale regola generalizza quanto avviene nel Web, in cui le risorse sono le pagine e ciascuna risorsa è identificata univocamente tramite un indirizzo simbolico (l'URL).
- *Uniformità delle interfacce*: questa regola richiede che le applicazioni interagiscano usando un insieme prefissato di comandi. I comandi devono corrispondere ad azioni elementari applicabili alle risorse. Come esempio REST propone di usare in maniera generalizzata i metodi HTTP caratteristici del Web: GET, HEAD, PUT, DELETE e POST. Tali comandi supportano le azioni principali applicabili a una risorsa: GET permette di richiedere una risorsa; HEAD di ottenere informazioni su una risorsa; PUT di modificare una risorsa; DELETE di eliminare una risorsa; infine, POST consente di rappresentare la creazione di una risorsa o una operazione generica il cui significato dipende dall'applicazione. GET e HEAD rappresentano operazioni di sola lettura; PUT e DELETE operazioni *idempotenti*, cioè applicabili ripetutamente alla stessa risorsa senza che l'iterazione cambi il significato dell'operazione; POST, infine, descrive operazioni non ripetibili quali, per esempio, la creazione di una risorsa.
- *Autodescrittività dei messaggi*: questa regola prescrive che le risorse siano scambiate mediante una loro *rappresentazione*. La rappresentazione di una risorsa è una codifica delle sue proprietà secondo un formato noto alle parti che devono interagire e dichiarato all'atto della comunicazione. REST non pone vincoli sul linguaggio di rappresentazione delle risorse, a patto che questo permetta di esprimere i collegamenti logici tra le risorse mediante una qualche forma di *link*. Nella pratica, la rappresentazione delle risorse in molte Web API REST sfrutta principalmente il

formato XML, descritto nel Capitolo 14 e il formato *JavaScript Object Notation* (JSON), illustrato nel seguito. Anche questa regola può essere vista come una generalizzazione del Web, dove si usa per rappresentare le risorse (i documenti) in un formato (HTML) che consente di esprimere i collegamenti ipermediali.

- *Collegamenti ipermediali tra le risorse*: questa regola completa la precendente con il requisito che le relazioni tra le risorse siano esprimibili nella rappresentazione come collegamenti. Per esempio, la rappresentazione di un catalogo di commercio elettronico deve contenere come link i collegamenti ai prodotti in vendita nel catalogo.
- *Interazione semplice (stateless)*: l'ultima regola prescrive la semplificazione dell'interazione tra client e servizio, ottenibile grazie all'eliminazione dello stato dal protocollo di interazione. Questo principio può essere compreso pensando al funzionamento di HTTP e del Web: quando il browser richiede un documento il Web server risponde con la risorsa richiesta e tale interazione non dipende dalle richieste precedenti. Questo genere di protocollo di comunicazione prende il nome di *stateless*, in opposizione ai protocolli di comunicazione che invece richiedono di preservare la storia dei messaggi scambiati (detti *stateful*). Nella pratica, questa regola propone di non codificare lo stato dell'interazione all'interno del protocollo di comunicazione, ma di mantenere lo stato presso il client, oppure di rappresentarlo come parte della risorsa scambiata. Un esempio del primo approccio consiste, per esempio, nel mantenere il carrello della spesa di un'applicazione di commercio elettronico (la tipica informazione che contiene lo stato dell'interazione) a lato client, anziché nella sessione a lato server. Un esempio del secondo approccio consiste nel rappresentare l'indice della pagina di risultati di un motore di ricerca nella pagina stessa, in modo che si possa evincere dalla rappresentazione stessa qual è il blocco di risultati corrente, quello successivo e precedente, cioè lo stato dell'interazione tra client e motore di ricerca.

A titolo di esempio, si pensi a un'applicazione Web che espone un'API per memorizzare, annotare e condividere gli indirizzi di libri recensiti nel sito di una comunità di lettori. La Web API consente di:

- richiedere una lista dei libri associati all'identità dell'utente richiedente, filtrata per annotazione, e con un tetto al numero massimo di risultati restituiti;
- aggiungere, modificare e cancellare libri.

La progettazione delle API REST di un servizio procede dapprima con l'identificazione delle risorse. Nel caso presente, queste sono i libri (book) e le annotazioni (tag). La regola di identificazione impone poi che le risorse abbiano un indirizzo univoco. Ciò richiede di progettare degli schemi di indirizzamento appropriati per tutti i tipi di interrogazione gestiti dal servizio.

Nel nostro esempio, un possibile schema di indirizzamento per l'elenco dei libri e delle annotazioni di un utente identificato da *username* è il seguente, se ipotizziamo che la Web API sia ospitata all'indirizzo `www.server.com/api`:

```
http://www.server.com/api/[username]/books
http://www.server.com/api/[username]/tags
```

Per illustrare il modo in cui REST rappresenta lo stato dell'interazione all'interno della descrizione delle risorse, prevediamo che la richiesta di un elenco possa essere paginata, cioè possa specificare l'indice del primo e dell'ultimo libro che si desidera ottenere. Per far questo, è possibile utilizzare il meccanismo previsto dalla sintassi dell'URL, che permette di appendere parametri in coda all'identificativo della risorsa richiesta. Pertanto, la richiesta per i primi due libri dell'utente può essere codificata come segue:

```
GET http://www.server.com/api/piero/books/?start=1&end=2
```

La rappresentazione delle risorse richiede di progettare un formato per codificare: una lista di libri o di tag; un singolo libro; una singola annotazione.

Per esempio, la rappresentazione di un elenco di libri fornita dal servizio in risposta a una richiesta come quella esemplificata in precedenza potrebbe essere formulata in XML come segue:

```
<?xml version="1.0"?>
<books start="1" end="2"
       next="http://www.server.com/api/piero/books?
       start=3&amp;end=4">
 <book url="http://www.catalogo.mcgraw-hill.it
    /catLibro.asp?item_id=2911"
    tags="basidati,informatica"
    href="http://www.server.com/api/piero/books
    /9788838668005">
    Primo libro</book>
 <book url="http://www.catalogo.mcgraw-hill.it
 /catLibro.asp?item_id=2178"
    tags="basidati,informatica,avanzato"
    href="http://www.server.com/api/piero/books
    /9788838663703">
    Secondo libro</book>
</books>
```

Si noti come la descrizione contenga al contempo la descrizione della risorsa, in questo caso l'elenco dei libri, la rappresentazione dello stato, cioè il minimo e massimo valore dell'indice dell'elenco e i link che rappresentano le relazioni logiche tra risorse, quale il link al singolo libro e al successivo blocco di risultati. In questo modo, l'applicazione client riceve dal servizio tutto ciò che è necessario per proseguire nell'interazione, senza bisogno che lo stato corrente sia rappresentato in strutture memorizzate a lato server, quali per esempio la sessione dell'architettura Java Servlet.

Per continuare nell'esempio, mostriamo una possibile rappresentazione del singolo libro, ottenuta in risposta alla richiesta codificata come segue:

```
GET http://www.server.com/api/piero/books/9788838668005
```

La richiesta identifica il libro mediante il codice ISBN; la risposta può essere codificata mediante un documento XML:

```
<?xml version="1.0"?>
<book url="http://www.catalogo.mcgraw-hill.it
  /catLibro.asp?item_id=2911">
  <description>Basi di dati - Modelli e linguaggi
  di interrogazione 4/ed
  </description>
  <tags count="2">
   <tag name="basidati"
             href="http://www.server.com/api/piero/tags
             /basidati"/>
   <tag name="informatica"
             href="http://www.server.com/api/piero/tags
             /informatica"/>
  </tags>
</book>
```

REST consente anche di trasmettere comandi che creano o modificano risorse. Per esempio, per creare un nuovo libro nella collezione di un certo utente, la Web API può essere chiamata con il metodo HTTP POST, nel modo seguente:

```
POST http://www.server.com/api/piero/books/
<?xml version="1.0"?>
<book url="http://www.catalogo.mcgraw-hill.it
/catLibro.asp?item_id=2911">
  <description>Basi di dati - Modelli e linguaggi
  di interrogazione 4/ed
  </description>
  <tags>
     <tag name="basidati"/>
     <tag name="informatica"/>
  </tags>
</book>
```

Infine la cancellazione di una risorsa, per esempio il libro con codice ISBN 9788838668005, può essere formulata tramite il metodo HTTP DELETE, come nell'esempio seguente:

```
DELETE http://www.server.com/api/piero/books/9788838668005
```

In sintesi, lo stile architetturale REST promuove un modo di organizzare le interfacce dei Web service centrato sulle risorse, al contrario di WSDL che sottolinea il ruolo delle operazioni. REST usa tutti i metodi HTTP in modo consistente e uniforme per tutte le Web API, mentre SOAP e WSDL sfruttano operazioni specifiche con un nome definito dall'utente e usano HTTP solo come protocollo di trasporto per veicolare i messaggi.

18.4.5 Un confronto tra EJB e Web Service

Concludiamo questa carrellata sulle architetture Web con un confronto finale tra i Web Service e l'architettura Enterprise Java Beans. Sebbene entrambe possano essere impiegate a supporto dell'integrazione di applicazioni distribuite, svolgono ruoli diversi. Nello standard EJB prevale l'attenzione ai requisiti non funzionali, quali la scalabilità e la persistenza degli oggetti, a discapito dell'interoperabilità, che è limitata ad applicazioni conformi all'architettura J2EE. I Web Service invece offrono un maggior grado di disaccoppiamento tra l'applicazione client e il fornitore del servizio, grazie all'interposizione di uno strato di messaggi in formati quali XML o JSON, che schermano gli oggetti client dalla conoscenza dei dettagli implementativi delle funzioni server (come illustrato nella Figura 18.16). Tuttavia, questa maggior indipendenza si ottiene a discapito delle prestazioni, a causa del sovraccarico di elaborazione dovuto alla codifica e decodifica dei messaggi.

18.5 Le architetture per Rich Internet Application

Le architetture multilivello per il Web affidano al client un ruolo limitato alla richiesta di risorse e alla visualizzazione delle risposte inviate dal server. Questa forma di interazione ha raggiunto un'enorme diffusione grazie all'adozione pressoché universale di HTML e HTTP per rappresentare le risorse e supportare la comunicazione client-server; tuttavia, ha anche evidenziato delle limitazioni dovute all'incapacità di fornire strumenti adeguati per la creazione di interface utenti usabili e capaci di gestire i dati complessi e anche multimediali tipici delle applicazioni più moderne. In

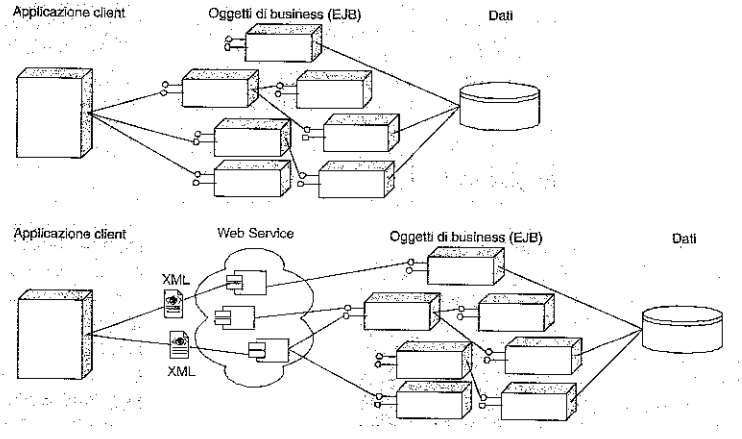

Figura 18.16 Confronto tra architettura EJB e Web Service.

particolare, l'architettura Web classica che fa uso solo di HTML e HTTP presenta le caratteristiche di seguito elencate.

- Supporta solo le funzioni di visualizzazione dei dati permesse da HTML, che sono pensate principalmente per dati di tipo documentale e hanno un supporto pittosto elementare per contenuti di tipo multimediale, quali audio, immagini e soprattutto video. Questa limitazione costringe l'utente all'uso di estensioni del browser o di applicazioni plug-in, non integrate nell'architettura del Web e spesso proprietarie.
- Consente il filtraggio dei dati solo a lato server, poiché il client non supporta la memorizzazione persistente di dati strutturati né l'estrazione delle informazioni di interesse da una base di dati locale al client. Questo limite comporta la necessità di continui cicli di comunicazione client-server, anche al solo scopo di selezionare un sottoinsieme dei dati visibili all'utente.
- Richiede il rinfresco dell'intera interfaccia (cioè della pagina) a ogni interazione dell'utente. Questo fatto produce sia un sovraccarico di comunicazione tra client e server sia la diminuzione dell'utilizzabilità dell'interfaccia, che deve essere ricaricata completamente anche a fronte di un evento i cui effetti sono localizzati a una piccola porzione della pagina.
- Limita gli eventi di interazione alla navigazione dei link ipertestuali e all'invio delle form e non offre la possibilità di sfruttare tutti gli eventi a cui l'utente è abituato nelle applicazioni desktop e mobili, quali il drag & drop, il passaggio del mouse su un oggetto dell'interfaccia e molti altri ancora.

In risposta a queste limitazioni, si è sviluppata una famiglia di tecniche, principi architetturali e applicazioni, comunemente denominati *Rich Internet Applications*, in breve RIA. Una RIA estende l'architettura Web basata su HTTP e HTML in due direzioni principali: una gamma di eventi simili a quelli offerti dalle applicazioni desktop e una forma di comunicazione client-server più flessibile; queste innovazioni sono introdotte nell'architettura senza ledere il principio di universalità caratteristico del Web: anche le RIA sono applicazioni basate su HTML e HTTP, per cui l'utente dotato di un browser può accedere alle funzionalità di qualunque server Web.

18.5.1 Tecnologie per le RIA

L'introduzione di primitive di interazione più potenti nell'architettura Web è un tema all'attenzione sin dagli albori di questa piattaforma. A partire dagli anni Novanta l'uso di HTML come linguaggio per la realizzazione dell'interfaccia utente è stato integrato da una forma di programmazione, detta *client-side scripting*, finalizzata a migliorare l'interazione, grazie un insieme più ricco di eventi ed effetti.

A seguito di un lungo e spesso controverso processo di evoluzione si è giunti alla standardizzazione di JavaScript, che oggi rappresenta il linguaggio più diffuso per la programmazione di interfacce utenti avanzate per il Web. Lo standard, denominato ufficialmente *ECMAScript*, recepisce e consolida un insieme di caratteristiche sviluppate fin dal 1996, anno della concezione della prima versione di JavaScript da parte della Netscape.

JavaScript è un linguaggio interpretato utilizzato prevalemente per arricchire le interfacce Web di funzionalità quali animazioni, eventi di vario tipo e controlli dell'input dell'utente. La programmazione lato client con JavaScript richiede un modello a oggetti della pagina che permetta al programmatore di accedere ai diversi elementi che compongono l'interfaccia per leggerne e modificarne le proprietà. Tale modello è anch'esso uno standard internazionale, promosso dal W3C, noto come Document Object Model (DOM).

18.5.2 DOM

Document Object Model, DOM, è un'interfaccia di programmazione a oggetti per documenti XML, HTML e XHTML. Lo standard in vigore, denominato *DOM Level 3*, specifica una serie di oggetti e metodi per ispezionare e manipolare il contenuto di un documento.

Il concetto principale di DOM è quello di *nodo*: in DOM l'intero documento e ogni sua parte corrispondono a un nodo. Esistono nodi di tipo diverso: elementi, testo, attributi e commenti. La struttura del documento corrisponde a un albero che rappresenta le relazioni di contenimento tra elementi e tra elementi e attributi, testo o commenti.

Per esempio, la Figura 18.17 illustra il modello DOM del frammento di XML utilizzato nel Paragrafo 18.4.4 per rappresentare un libro.

I nodi DOM possono essere ispezionati e manipolati mediante JavaScript o qualsiasi altro linguaggio di programmazione che implementi il modello. I nodi offrono a questo scopo una serie di proprietà e metodi. Tra le proprietà più importanti di un nodo citiamo:

- `nodeName`: rappresenta il nome del nodo, per esempio *book* è il nome di un elemento XML `<book>`;
- `nodeValue`: rappresenta il valore di un nodo a seconda del suo tipo, per esempio il valore di un attributo o il contenuto testuale di un elemento;

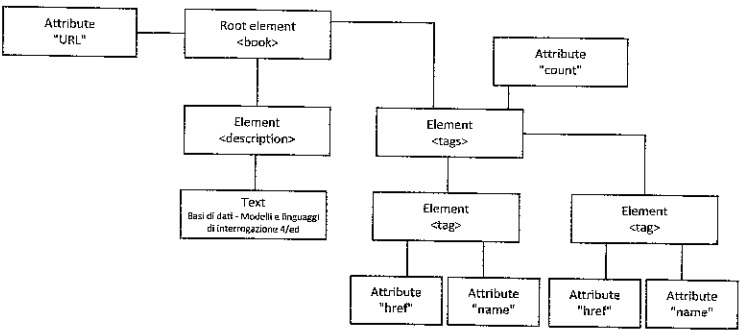

Figura 18.17 Modello DOM del frammento XML che rappresenta un libro.

- `parentNode`: rappresenta il nodo che contiene il nodo corrente;
- `childNodes`: rappresenta i nodi contenuti nel nodo corrente;
- `attributes`: rappresenta gli attributi del nodo corrente.

Tra i metodi più importanti citiamo:

- `xmlDoc=loadXMLDoc(file.xml)`: inizializza l'oggetto `xmlDoc` con l'albero DOM ricavato analizzando il file `file.xml`;
- `nodo1.getElementsByTagName(name)`: restituisce tutti gli elementi con il nome specificato contenuti nel nodo `nodo1`, per esempio tutti gli elementi `<book>`;
- `nodo1.appendChild(nodo2)`: aggiunge il nodo `nodo2` come figlio del nodo `nodo1`;
- `nodo1.removeChild(nodo2)`: rimuove il nodo `nodo2` dai figli del nodo `nodo1`.

A titolo di esempio si consideri il seguente programma, scritto nel linguaggio JavaScript, che utilizza una funzione `loadXMLDoc` per caricare il contenuto di un file XML.

```
<script>
 xmlDoc=loadXMLDoc("books.xml");
 y=xmlDoc.documentElement.childNodes;
 document.write("Nodename: " + y[0].nodeName);
 document.write(" (value: " +
                y[0].childNodes[0].childNodes[0].node-
                Value + ")");
</script>
```

Se il file `book.xml` contiene i dati di un libro, come riportato nel Paragrafo 18.4.4, allora il risultato dell'esecuzione è:

```
Nodename: book (value: Basi di dati - Modelli e linguaggi
di interrogazione 4/ed)
```

cioè il nome del primo elemento del documento seguito dal valore del primo figlio del primo nodo di tale elemento, cioè il valore dell'elemento `<description>`.

18.5.3 JavaScript

JavaScript è un linguaggio di programmazione interpretato, concepito in origine per la programmazione delle interfacce a lato client, ma successivamente esteso anche per la programmazione a lato server.

Il linguaggio ha una sintassi simile a quella del C, poiché comprende strutture di controllo classiche quali le istruzioni condizionali `if` e `switch` e le istruzioni iterative `for` e `while`; tuttavia ne differisce in molti punti importanti.

- *Tipizzazione dinamica*: le variabili non hanno una dichiarazione di tipo statica, ma possono assumere valori di tipi diversi durante l'esecuzione. Un operatore (`typeof`) permette di ispezionare il tipo di un oggetto durante l'esecuzione.

- *Uso dei prototipi*: il linguaggio adotta una forma di programmazione a oggetti basata non sulle classi ma su oggetti prototipali, che possono essere clonati ed estesi con nuove proprietà e metodi per simulare l'ereditarità e promuovere il riutilizzo.
- *Funzioni come oggetti e costruttori di oggetti*: le funzioni sono considerate come veri e propri oggetti, dotati di proprietà e metodi. Le funzioni servono anche per costruire nuovi oggetti a partire dai prototipi predefiniti nel linguaggio o definiti dal programmatore.
- *Supporto delle espressioni regolari*: le espressioni regolari sono oggetti del linguaggio, il che facilita la scrittura di programmi per l'elaborazione del contenuto testuale dei documenti.

Nella programmazione delle interfacce Web, i programmi JavaScript, detti *script a lato client* (*client side script*) sono interpretati da un *motore JavaScript* (*JavaScript engine*) integrato nel browser. Il motore JavaScript offre un ambiente di esecuzione completo, che fornisce accesso agli oggetti DOM che formano il documento e agli eventi prodotti dall'interazione dell'utente.

Gli eventi di interazione associati alle pagine Web sono definiti nello standard HTML DOM, la parte dello standard DOM specifica per i documenti HTML. In particolare, HTML DOM codifica gli eventi che l'utente o il browser possono produrre con una pagina Web; tali eventi comprendono il caricamento della pagina o di un'immagine, il click del mouse, il movimento del mouse su un elemento, la modifica di un campo di input di una form, l'invio del contenuto di una form, la pressione di un tasto.

Come esempio delle capacità di JavaScript di manipolare il contenuto dell'interfaccia, si consideri lo script seguente, annidato all'interno di una semplice pagina HTML:

```
<!DOCTYPE html>
<html>
 <head>
 <script>
 function cambiatesto(id)
 {id.innerHTML="Grazie per aver cliccato!";}
 </script>
 </head>
   <body>
     <h1 onclick="cambiatesto(this)">Clicca sul titolo!
     </h1>
   </body>
</html>
```

Il tag `<script>` contiene la definizione di una funzione JavaScript `cambiatesto()`, che riceve come parametro di input un oggetto HTML DOM e ne cambia il contenuto, assegnando la stringa `Grazie per aver cliccato!` alla proprietà HTML DOM `innerHTML`, che rappresenta appunto il contenuto HTML di un elemento. Per rendere la funzione richiamabile dall'utente con un click sul contenuto dell'elemento `<h1>`, questa viene associata alla proprietà `onClick` dell'elemento desiderato. L'effetto risultante è che quando l'utente clicca sull'elemento

`<h1>`, questo cambia il contenuto mostrando il testo `Grazie per aver cliccato!`.
Grazie a HTML DOM e all'ambiente di esecuzione fornito dal browser, JavaScript può essere usato per arricchire l'interfaccia Web con moltissime funzionalità, quali:

- il controllo dell'input in una form: intercettando gli eventi prodotti dall'utente e attivando opportune funzioni di verifica si può allertare l'utente nel caso di immissione di dati errati. Tale funzione però non impedisce a un utente malintenzionato di disabilitare JavaScript nel browser e inviare comunque dati errati al server, per cui non sostituisce i controlli svolti a lato server per evitare di inserire nella base di dati informazioni scorrette;
- la visualizzazione progressiva di informazioni: agendo sulla proprietà di visibilità degli elementi di HTML DOM, è possibile, per esempio, ottenere dal server una collezione di immagini al caricamento della pagina e poi mostrarle una a una senza formulare ulteriori richieste;
- animazioni e cambio delle proprietà grafiche degli elementi: è possibile associare a eventi interattivi il cambiamento di qualsiasi proprietà CSS degli elementi, ottenendo una galleria pressochè infinita di effetti.

18.5.4 AJAX

L'uso di JavaScript descritto finora non altera sostanzialmente le modalità di comunicazione client-server: i dati vengono richiesti all'inizio dell'interazione dal browser al server e ogni nuova richiesta di informazioni produce il ricalcolo completo della pagina.

Una delle innovazioni che ha maggiormente determinato il successo delle RIA si ottiene abbinando le capacità di JavaScript con quelle di un ulteriore oggetto di utilità fornito dall'ambiente di esecuzione, che consente di effettuare chiamate asincrone al server senza ricaricare la pagina con cui l'utente sta interagendo.

Questo tipo di effetto si ottiene grazie ad *AJAX* (Asynchronous JavaScript and XML), una tecnica di programmazione che integra le chiamate asincrone con la trasmissione di dati codificati in XML (o JSON) tra client e server.

Il principio alla base di AJAX è l'aggiornamento selettivo e asincrono dell'interfaccia: mentre l'utente sta interagendo con la pagina, se un evento richiede nuovi dati, questo viene processato aprendo un'ulteriore connessione HTTP al server, per richiedere solo la porzione (solitamente piccola) di dati necessaria per aggiornare una regione dell'interfaccia. L'utente ha quindi la sensazione di un'interfaccia stabile, che si aggiorna selettivamente come quella delle applicazioni di tipo desktop.

La chiamata asincrona al server avviene all'interno di un thread parallelo e indipendente rispetto a quello che gestisce l'interazione dell'utente; al sopraggiungere della risposta dal server, un evento di callback permette al thread principale di ricevere i dati e aggiornare l'interfaccia.

La chiave di questo comportamento è l'oggetto `XMLHttpRequest`, che si crea con la semplice istruzione:

```
connessione = new XMLHttpRequest();
```

Tale oggetto è utilizzabile per inviare una richiesta HTTP a un server e riceverne la risposta, come dimostra il semplice esempio che segue.

```
<!DOCTYPE html>
<html>
<head>
<script>
function esempioAjax() {
 var xmlhttp=new XMLHttpRequest();
 xmlhttp.onreadystatechange=function()
  {
   if (xmlhttp.readyState==4)
    {document.getElementById("myDiv").innerHTML=
      "<h1>" + xmlhttp.responseText + "</h1>";}
  }
  xmlhttp.open("GET","nuovotesto.txt",true);
  xmlhttp.send();
}
</script>
</head>
<body>
  <div id="myDiv"><h1>Cambio del titolo con AJAX</h1>
  </div>
  <button type="button"
    onclick="esempioAjax()">Cambia il titolo</button>
</body>
</html>
```

Il documento HTML contiene un elemento `<DIV>` identificato tramite l'attributo HTML `id`, che comprende un'intestazione che si vuole sostituire con un nuovo testo ottenuto mediante una chiamata asincrona al server. L'aggiornamento è prodotto dal click su un bottone, che provoca la chiamata della funzione JavaScript `esempioAjax`. La funzione crea una connessione asincrona al server (lo stesso da cui la pagina è stata scaricata) e invia la richiesta HTTP `GET nuovotesto.txt` che comporta il reperimento del contenuto del file menzionato. L'istruzione `xmlhttp.send()` provoca l'effettivo invio della richiesta al server, in un thread indipendente e separato. Il thread principale riceve la notifica dell'arrivo della risposta sotto forma di un evento asincrono (`onreadystatechange`), che provoca l'esecuzione di una funzione di gestione opportuna. Nell'esempio si tratta di una funzione anonima che rimpiazza il testo del titolo con il testo fornito della risposta HTTP.

L'effetto dello script AJAX è che al click del bottone l'utente vede il solo testo del titolo cambiare e assumere il nuovo contenuto proveniente dal server, senza che la pagina venga ricaricata completamente.

Concludiamo il paragrafo su AJAX mostrando il codice della funzione `loadXMLDoc`, che abbiamo usato nell'esempio del Paragrafo 18.5.2 per caricare un documento XML:

```
function loadXMLDoc(dname)
{
if (window.XMLHttpRequest)
  {
  xhttp=new XMLHttpRequest();
  }
else
  {
  xhttp=new ActiveXObject("Microsoft.XMLHTTP");
  }
xhttp.open("GET",dname,false);
xhttp.send();
return xhttp.responseXML;
}
```

Si noti che la funzione si apre con un'istruzione condizionale, che serve per discriminare tra modalità sintattiche diverse di apertura della connessione, supportate da browser diversi.

18.5.5 JavaScript Object Notation (JSON)

Lo scambio di dati tra client e server con AJAX può avvenire secondo diversi formati, quali un semplice testo ASCII oppure un documento XML. Un formato alternativo molto usato è *JavaScript Object Notation*, che permette di rappresentare oggetti e insiemi di oggetti con una semplice notazione testuale basata su coppie attributo-valore.

Per esempio, il frammento JSON seguente equivale al documento XML che rappresenta una lista di libri discusso nel Paragrafo 18.4.4.

```
{"books": {
  "start": "1",
  "end": "2",
  "next": "http://www.server.com/api/piero/books?start=3
   &end=4",
  "booklist":
   { "book":
     {"url": "http://www.catalogo.mcgraw-hill.it/
      catLibro.asp?item_id=2911",
      "tags": "basidati,informatica",
      "href": "http://www.server.com/api/piero/books/
       9788838668005",
      "text": "Primo libro"
    },
    "book":
     {"url": "http://www.catalogo.mcgraw-hill.it/
      catLibro.asp?item_id=2178",
      "tags": "basidati,informatica,avanzato",
```

```
        "href": "http://www.server.com/api/piero/books/
         9788838663703",
        "text": "Secondo libro"
      }
    ]
}}
```

Un singolo oggetto è rappresentato mediante un record delimitato da parentesi graffe; i campi del record possono a loro volta essere valori semplici o strutturati e quindi contenere altri record; un insieme di oggetti è rappresentato mediante una successione di valori separati da virgola racchiusi tra parentesi quadre.

18.5.6 HTML 5 e la persistenza a lato client

L'evoluzione del Web verso una maggior flessibilità architetturale e utilizzabilità dell'interfaccia utente ha richiesto un profondo aggiornamento del principale standard legato a questa piattaforma: HTML. Tale sforzo ha prodotto la specifica della nuova versione del linguaggio, denominata HTML 5, da parte del W3C e del Web Hypertext Application Technology Working Group (WHATWG).

HTML 5 è molto di più che una semplice revisione del linguaggio di descrizione dei documenti, poichè tratta svariate tematiche, quali la multimedialità, la grafica bidimensionale e il supporto alla persistenza dei dati sul client, che mirano a sistematizzare la transizione del Web verso lo status di piattaforma client-server a pieno titolo avviata con l'avvento delle RIA.

Le principali innovazioni contenute in HTML 5 riguardano:

- grafica bidimensionale: un nuovo elemento (`<canvas>`) permette di inserire, anche dinamicamente tramite JavaScript, elementi di grafica 2D in un documento;
- contenuti audiovisuali: due nuovi elementi (`<video>` e `<audio>`) arricchiscono la gamma di elementi multimediali supportati nativamente dal linguaggio;
- organizzazione dei documenti: svariati nuovi elementi (`<article>`, `<footer>`, `<header>`, `<nav>`, `<section>`) permettono di dare una struttura più significativa ai documenti e semplificano l'applicazione delle regole di presentazione CSS alle diverse parti di una pagina;
- input nelle form: nuovi campi di input, quali gli elementi di tipo colore, data, calendario, email, telefono e intervallo, adeguano la presentazione delle form di inserimento dati agli standard di utilizzabilità abituali nelle applicazioni desktop;
- API per l'interazione con l'ambiente del client: nuove interfacce di programmazione permettono di accedere a servizi forniti dall'ambiente client, quali la posizione geografica rilevata con sensori GPS, la comunicazione client-server mediante TCP-IP e la memoria persistente.

Nel seguito illustriamo brevemente l'estensione di HTML 5 che riguarda maggiormente la gestione dei dati: le tecniche per la persistenza a lato client.

Nelle versioni precedenti di HTML, la possibilità di gestire dati persistenti sul client era limitata all'uso dei cookie, cioè file di dimensioni contenute, fino a un massimo di 4kB, atti a contenere semplici coppie chiave-valore. Tale soluzione, oltre a

impedire la gestione di moli di dati significative, costringeva alla trasmissione del cookie al server a ogni richiesta e poteva essere compromessa dalla disabilitazione esplicita della funzionalità da parte dell'utente.

Lo stato attuale di HTML 5 prevede diversi metodi per la gestione dei dati persistenti a lato client e per la costruzione di applicazioni Web che possono essere usate, almeno parzialmente, anche senza una connessione Internet.

Una prima soluzione consiste nell'uso di `Web Storage API`, un'interfaccia di programmazione che estende il concetto di cookie permettendo la memorizzazione di coppie chiave-valore sul client in modo più potente e flessibile.

Rispetto all'uso dei cookie, WebStorage permette la memorizzazione di quantità maggiori di informazioni, consente all'applicazione Web un controllo più preciso sull'uso delle informazioni memorizzate nel client e fornisce due modalità alternative di gestione della durata dell'informazione locale.

L'interfaccia principale è denominata `Storage` e permette la memorizzazione di liste di coppie chiave-valore, dove la chiave è una stringa qualunque e il valore è un'ulteriore stringa che rappresenta un elemento di uno dei tipi supportati da JavaScript. Un oggetto che implementi `Storage` offre metodi per determinare la lunghezza della lista, accedere all'ennesima chiave o coppia chiave-valore, aggiungere e cancellare elementi dalla lista.

Sono previste due realizzazioni dell'interfaccia: `sessionStorage` e `localStorage`. La prima serve per memorizzare dati con durata pari a quella di una sessione di navigazione dell'utente, la seconda per dati senza una data di scadenza.

Il documento seguente mostra come memorizzare due stringhe (nome e cognome) nella memoria locale e utilizzarne i valori per creare il contenuto di un elemento del documento.

```
<!DOCTYPE html>
<html>
<body>
<h1 id="benvenuto"></h1>
<script>
  localStorage.nome="Mario";
  localStorage.cognome="Bianchi";
  document.getElementById("benvenuto").innerHTML=
                    "Benvenuto" + localStorage.nome + " "
                    + localStorage.cognome;
</script>
</body>
</html>
```

WebStorage fornisce anche un evento apposito (`storage`) per notificare a uno script l'avvenuta modifica del contenuto della memoria persistente. L'evento contiene come parametri la chiave che ha subito la modifica, il valore precendente e l'indirizzo della pagina che ha prodotto l'evento.

Un secondo metodo per la gestione della persistenza a lato client è fornito dall'`Indexed Database API`, un'interfaccia di programmazione che consente

di utilizzare un semplice database a oggetti all'interno di un browser. La API supporta la memorizzazione persistente di oggetti JavaScript, che vengono indicizzati e possono successivamente essere estratti dal database tramite semplici query che fanno uso degli indici.

Continuando l'esempio sui libri del Paragrafo 18.4.4, mostriamo come il client possa creare un nuovo database locale (chiamato `dbLibri`) e popolarlo di oggetti con JavaScript e la Indexed Database API.

```
var dbl;
var success = 0;
var request = indexedDB.open("dbLibri");
request.onupgradeneeded = function(event) {
 dbl = request.result;
 var store = dbl.createObjectStore("libri",
   {keyPath: "isbn"});
 var IndiceTitolo =
   store.createIndex("per_titolo", "titolo",
   {unique: true});
 store.put({titolo: "Basi di dati 4ed",
   isbn: 9788838668005});
 store.put({titolo: "Basi di dati avanzate",
   isbn: 9788838663703});
};

request.onsuccess = function(event) {
   dbl = request.result;
success = 1;
};
```

La creazione di un nuovo database richiede la definizione di una funzione di callback associata all'evento predefinito `onupgradeneeded`; tale funzione crea un nuovo database di oggetti con chiave primaria `isbn`, indicizzati per titolo. Dopo la creazione del database è possibile popolarlo, con l'istruzione `put()`. Al termine della creazione e popolazione, segnalato dall'evento asincrono `onsuccess`, la variabile JavaScript `dbl` memorizza l'oggetto di interfaccia necessario per interagire con il database.

Successivamente, è possibile definire una transazione, anche'essa asincrona rispetto all'interazione dell'utente con la pagina, che estrae un libro dal database locale. Il codice seguente mostra le istruzioni necessarie:

```
var transaction = dbl.transaction(["libri"],
  "readonly");
var store = transaction.objectStore("libri");
var index = store.index("per_titolo");
var getRequest = index.get("Basi di dati 4ed");
getRequest.onsuccess = function() {
var risultato = getRequest.result;
```

```
if (risultato !== undefined)
alert("Trovato:" + risultato.isbn + ",
  " + risultato.titolo);
else
alert("Non trovato.");
};
```

La transazione (`transaction`) permette di creare il database (`store`) e l'indice (`index`) necessari all'interrogazione; l'oggetto indice supporta la chiamata asincrona della funzione `get()` che interroga il database usando l'indice per titolo; tale richiesta produce una chiamata di callback che segnala la fine della transazione: la variabile `risultato` viene assegnata al valore restituito dalla transazione, che corrisponde all'oggetto libro (in caso di ritrovamento) oppure a un valore indefinito.

Per concludere, menzioniamo anche un terzo modo per memorizzare dati a lato client, che, sebbene deprecata a livello di specifica dal W3C, continua a essere in uso in browser quali Chrome, Opera e Safari e in molte librerie di utilità per la realizzazione di RIA. Si tratta della `Web SQL Database` API, che propone un'interfaccia per integrare in JavaScript direttamente un database relazionale. Al momento, l'unico database che supporta la API è SQLite, per cui non è possibile prevedere quale sarà il destino di questa modalità di memorizzazione persistente dei dati a lato client.

18.6 Progettazione delle applicazioni Web

Dopo aver esaminato gli aspetti tecnologici delle applicazioni Web, è utile considerare il loro processo di sviluppo per capire le differenze rispetto a quello delle applicazioni tradizionali. La Figura 18.18 riprende il generico ciclo di vita di un sistema informativo e mette in evidenza le attività di analisi dei requisiti, progettazione e implementazione che sono maggiormente influenzate dal contesto Web.

Nel seguito illustreremo brevemente gli aspetti salienti dell'analisi dei requisiti e della progettazione di applicazioni Web. Estenderemo alla progettazione delle interfacce ipertestuali la tecnica della modellazione concettuale, che permette di affrontare per gradi il progetto, partendo da una specifica di alto livello indipendente dalla piattaforma. Descriveremo le attività di progetto applicandole alla pubblicazione su Web di un catalogo di prodotti.

18.6.1 Analisi dei requisiti

La raccolta e l'analisi dei requisiti hanno lo scopo generale di identificare e definire in maniera semi-formale le funzionalità che l'applicazione deve fornire per soddisfare gli obiettivi di business che ne giustificano la realizzazione. Le tecniche di raccolta dei requisiti e i principi generali di rappresentazione dei dati rimangono validi; tuttavia, in ambito Web, l'attività di analisi pone particolare accento sulla specifica dell'interazione con l'utente, un aspetto estremamente rilevante quando si utilizza il Web come piattaforma di fruizione dell'applicazione.

Il punto di partenza nell'analisi dei requisiti di un'applicazione Web è pertanto l'individuazione dei *gruppi di utenti*; ciascun gruppo è caratterizzato da esigenze

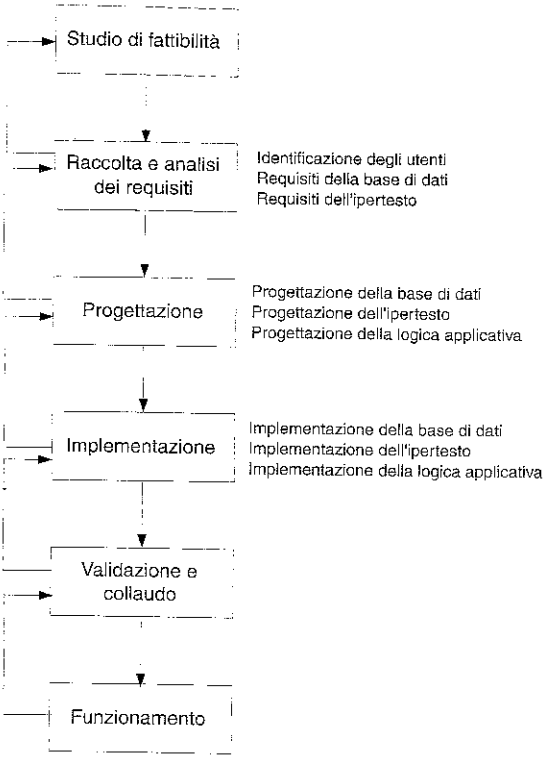

Figura 18.18 Ciclo di vita di un'applicazione Web.

applicative simili e verrà quindi associato a una particolare "versione" dell'applicazione. Per esempio, in un'applicazione per la pubblicazione e gestione di un catalogo prodotti, gli utenti possono essere divisi in *utenti esterni*, cioè quelli che accedono in lettura al catalogo su Internet, e *utenti interni*, cioè il personale dell'azienda.

A loro volta gli utenti esterni possono essere suddivisi in *utenti casuali* e *utenti registrati*, questi ultimi essendo quelli che hanno già effettuato almeno un acquisto e hanno accesso a una zona personalizzata del sito protetta con una password. Gli utenti interni possono invece essere suddivisi in *amministratori*, dotati dei diritti di scrittura su tutti gli oggetti dell'applicazione, e *personale di marketing*, con diritti di scrittura solo sui dati dei prodotti. La Figura 18.19 mostra un possibile modo di rappresentare la tassonomia degli utenti.

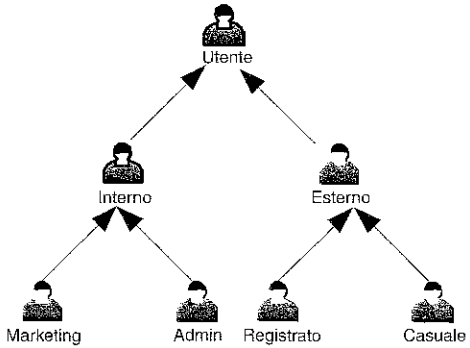

Figura 18.19 Tassonomia degli utenti di un'applicazione Web.

Una volta definite le classi di utenti, si identificano le sotto-applicazioni che verranno realizzate in modo ipertestuale; in genere, i contenuti memorizzati nella base di dati vengono presentati in modo differente a seconda della tipologia dell'utente e del ruolo che quest'ultimo svolge in uno specifico processo applicativo.

Le sotto-applicazioni possono essere considerate come delle *viste di sito* (o *site view*), in analogia con il concetto di *schema esterno* di una base di dati, che abbiamo illustrato come meccanismo per garantire visioni differenti degli stessi contenuti, realizzato nel modello relazionale tramite le view. Una site view contiene le interfacce ipertestuali presentate agli utenti; ogni site view è un insieme omogeneo di pagine Web, cui si accede tipicamente tramite un'unica pagina di ingresso (la *home page*), eventualmente dopo aver superato un controllo di autorizzazione. Per esempio le quattro tipologie di utenza individuate potrebbero dar luogo a quattro site view differenti, di cui solo quella dedicata agli utenti casuali non richiede il controllo dei diritti di accesso; nel seguito svilupperemo questa site view.

Il contesto Web influenza anche l'analisi dei requisiti della base di dati e quindi la formazione del *glossario dei termini*, preliminare al progetto concettuale della base di dati. In particolare, i dati applicativi vanno arricchiti tramite concetti che, anche se non strettamente necessari alla memorizzazione delle informazioni gestite dall'applicazione, sono di aiuto alla pubblicazione dell'informazione in pagine ipertestuali di facile lettura; per esempio, termini che rappresentano categorie di classificazione o raggruppamenti degli oggetti applicativi utili per la definizione di indici e cammini guidati di accesso ai dati. In un'applicazione per la pubblicazione online di un catalogo prodotti, in aggiunta ai concetti indispensabili per la rappresentazione dei dati di prodotto, possono essere previsti concetti ausiliari per facilitare l'accesso al catalogo, quali categorie merceologiche per la classificazione dei prodotti, collezioni speciali di prodotti e così via.

La struttura di una site view può essere specificata compilando un foglio di descrizione come quello riportato di seguito, in cui il sito è suddiviso in aree e per ogni

area si fornisce una specifica di massima delle pagine che lo costituiscono e del loro contenuto.

Nome Vista	Accesso utente casuale
Descrizione	Vista per l'accesso degli utenti casuali
Gruppi	Utente Casuale

Area Prodotti

Nome pagina	Descrizione	Concetti usati
Categorie	Mostra l'indice delle categorie	Categoria
Prodotti	Mostra la lista dei prodotti di una categoria	Prodotto, Tipologia
Prodotto	Mostra i dati salienti di uno specifico prodotto	Prodotto
Immagini	Mostra varie immagini di un prodotto	Immagine, Visualizzazione

Area Notizie

Nome pagina	Descrizione	Oggetti acceduti
Notizie	Mostra l'indice di tutte le notizie	Notizia
Notizia	Mostra il testo di una specifica notizia	Notizia
Prodotto	Mostra i dati del prodotto a cui si riferisce la notizia	Notizia, Riferimento

Altre pagine

Nome pagina	Descrizione	Oggetti acceduti
Home page	Mostra un breve sommario di alcuni prodotti e notizie rilevanti	Prodotto, Notizia
Contatti	Pubblica i dati di contatto dell'azienda e una form per l'invio di messaggi	Nessuno

18.6.2 Progettazione concettuale della base di dati

In questo paragrafo illustriamo alcuni aspetti della progettazione concettuale dei dati che caratterizzano il contesto Web; queste considerazioni integrano le tecniche di progettazione concettuale classiche discusse nei Capitoli 6 e 7. Come già detto nel precedente paragrafo, nelle applicazioni per il Web possono essere utili concetti la cui presenza faciliti l'accesso all'informazione contenuta negli oggetti applicativi. Più in generale, uno schema concettuale progettato per un'applicazione Web spesso include entità e relazioni che assolvono quattro ruoli diversi:

- gli *oggetti applicativi*: sono le entità e relazioni che servono a descrivere i concetti fondamentali gestiti dall'applicazione;

- gli *oggetti d'accesso*: sono entità e relazioni il cui scopo principale è facilitare l'accesso agli oggetti applicativi;
- gli *oggetti d'interconnessione*: sono entità e relazioni che servono a collegare oggetti applicativi e a fornire supporto alla navigazione dell'utente da un oggetto applicativo a un altro a esso collegato logicamente;
- gli *oggetti personalizzati*: sono entità e relazioni che servono a collegare oggetti applicativi al singolo utente o al gruppo di utenti, allo scopo di esprimere dati personali, quali il carrello della spesa di un individuo o le offerte speciali indirizzate a un certo gruppo di utenti.

Questa caratterizzazione di ruoli fa sì che sia possibile suddividere lo schema concettuale di un'applicazione Web in quattro tipi di sottoschemi, evidenziando i concetti che servono a rappresentare le proprietà degli oggetti applicativi oppure a supportarne l'accesso, l'interconnessione e la personalizzazione. Le entità applicative sono comuni a tutte le tipologie e rappresentano il punto di collegamento tra i sottoschemi. Ovviamente, questa organizzazione dello schema concettuale è da prendersi solo come riferimento.

A titolo di esempio, mostriamo nella Figura 18.21 un possibile schema concettuale della base di dati di un'applicazione Web per la pubblicazione di un catalogo di prodotti. Lo schema include due entità applicative principali, PRODOTTO e NOTIZIA. Lo schema di connessione fra queste due entità è rappresentato dalla semplice relazione RIFERIMENTO.

L'entità PRODOTTO è posta in relazione con due ulteriori entità (SCHEDA TECNICA e IMMAGINE), che ne dettagliano il contenuto e costituiscono il sottoschema applicativo; il sottoschema di accesso relativo ai prodotti contiene l'entità CATEGORIA, collegata all'entità PRODOTTO dalla relazione TIPOLOGIA.

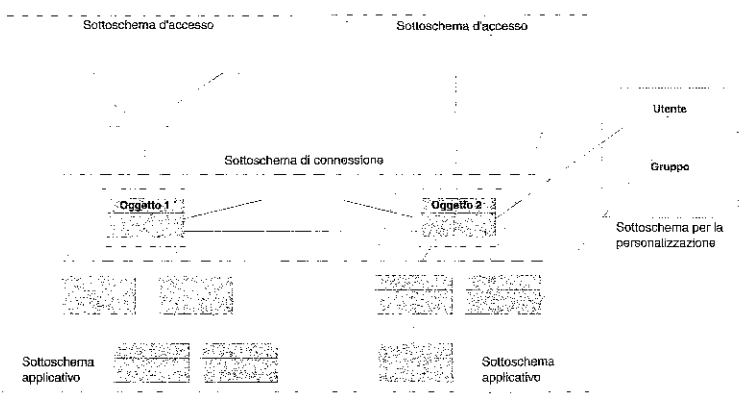

Figura 18.20 I sottoschemi dello schema concettuale della base di dati di una tipica applicazione Web.

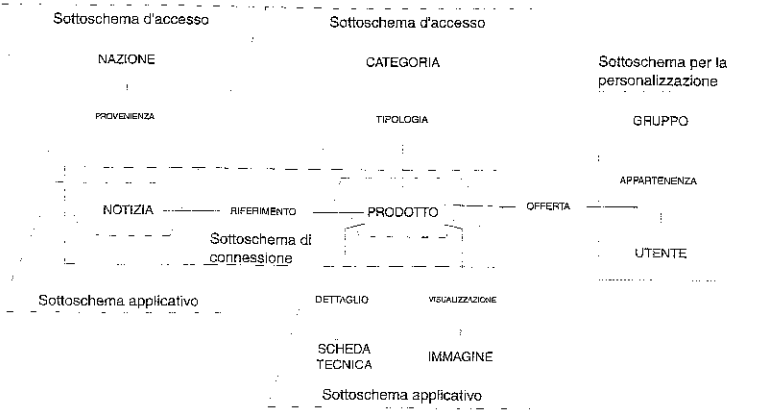

Figura 18.21 Schema scheletro della base di dati di un'applicazione Web per la pubblicazione di un catalogo prodotti.

L'entità NOTIZIA non ha un sottoschema applicativo proprio, ma l'entità NAZIONE descrive l'ambito in cui la notizia è stata diffusa, e assieme alla relazione PROVENIENZA costituisce il relativo sottoschema d'accesso.

Infine, il sottoschema di personalizzazione contiene le due entità UTENTE e GRUPPO, collegate dalla relazione APPARTENENZA che esprime la classificazione degli utenti dell'applicazione in gruppi. Per esprimere la presenza di dati personalizzati, la relazione OFFERTA collega l'entità utente all'entità PRODOTTO. In tal modo è possibile associare in modo personalizzato a ogni utente specifici prodotti a lui consigliati.

Le successive attività di progetto logico e fisico dei dati restano inalterate rispetto a quanto descritto nei Capitoli 8 e 11 per i sistemi informativi tradizionali.

18.6.3 Progettazione dell'ipertesto

La progettazione degli ipertesti che realizzano l'interfaccia dell'applicazione può essere condotta sfruttando modelli concettuali in grado di rappresentare le pagine e i loro contenuti; questo approccio "guidato da un modello" è coerente con la metodologia di progettazione dei dati illustrata in questo volume, perché consente al designer di mantenersi a un livello astratto e indipendente dall'architettura e tecnologia.

In questo paragrafo illustriamo brevemente un modello concettuale chiamato *Interaction Flow Modelling Language* (abbreviato in IFML); si tratta di un nuovo standard basato su Unified Modeling Language (UML), promosso dall'Object Management Group (OMG). IFML è ideato per favorire la progettazione concettuale delle applicazioni interattive di ogni tipo (non solo Web, ma anche mobili e desktop) e

contiene un'estensione apposita per la specifica degli ipertesti Web con un linguaggio semplice e visuale.

Uno schema IFML descrive la struttura di una site view concentrandosi su tre aspetti principali:

- la suddivisione della vista di sito in una gerarchia di contenitori (*ViewContainer*) quali aree e pagine;
- il contenuto di ogni pagina, espresso come un insieme di *componenti* (*ViewComponent*);
- la topologia dei collegamenti navigazionali disponibili all'utente, espresso come un insieme di collegamenti (*InteractionFlow*) che connettono componenti e/o contenitori.

Definiamo ora i costrutti essenziali di IFML, rimandando alla bibliografia del capitolo per ulteriori approfondimenti.

Site view, aree e pagine Abbiamo già detto che un'applicazione Web specificata in IFML consiste in uno o più contenitori di alto livello, chiamati *site view*. A sua volta, una site view può essere divisa in *aree*, ulteriormente suddivisibili in *sotto-aree*. Site view e aree contengono le pagine dell'applicazione, che sono le effettive unità di visualizzazione offerte all'utente. La strutturazione gerarchica delle site view in aree e pagine consente di rappresentare una situazione comune nelle applicazioni Web, in cui un sito è organizzato in sezioni dedicate ad aspetti distinti. Per esempio, un sito istituzionale di un'azienda è spesso diviso in aree standard, quali la sezione prodotti, novità, informazioni sull'azienda, dati di contatto, servizi di supporto ai clienti e così via. Tali sezioni sono rappresentabili in IFML come ViewContainer di tipo area.

La Figura 18.22 mostra la notazione IFML per esprimere la struttura gerarchica di una site view. Site view, aree e pagine sono rappresentate graficamente come rettangoli etichettati con il nome della site view, area o pagina.

In particolare, lo schema nella Figura 18.22 mostra la struttura della site view pubblica dell'applicazione Web utilizzata come esempio, che descrive la pubblicazione di un catalogo di prodotti. La site view contiene due pagine, HOME PAGE e CONTATTI, e tre aree (PRODOTTI, NOTIZIE, SUPPORTO). La figura mostra anche la struttura interna dell'area PRODOTTI, che racchiude quattro pagine (CATEGORIE, PRODOTTI, PRODOTTO, IMMAGINI).

Pagine e aree possono godere di particolari proprietà, che ne rappresentano l'importanza all'interno del modello concettuale.

- Una pagina può essere definita come *home page* della site view cui appartiene, in quanto viene mostrata per default all'utente quando accede alla site view. Tale proprietà è rappresentata da una lettera *H* aggiunta al simbolo della pagina.
- Una pagina o un'area può essere definita come *default* rispetto all'area cui appartiene, in quanto viene mostrata per default quando l'utente accede al modulo che la contiene. Tale proprietà è rappresentata da una lettera *D* aggiunta al simbolo della pagina.

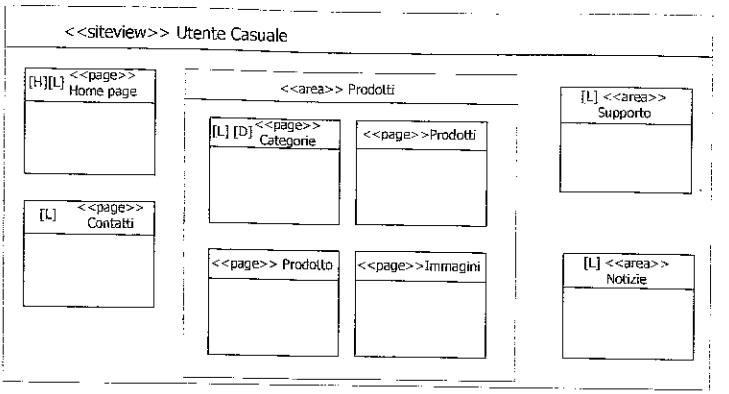

Figura 18.22 Schema di una site view con aree e pagine.

- Una pagina o un'area può essere definita come *landmark* rispetto alla site view o area a cui appartiene, quando questa è globalmente raggiungibile da tutte le altre pagine dell'area o site view che la contiene. Tale proprietà è rappresentata da una lettera L aggiunta al simbolo della pagina o dell'area.

Nella Figura 18.22, una pagina è definita come la HOME PAGE della site view. Le pagine HOME PAGE e CONTATTI e le aree PRODOTTI, NOTIZIE e SUPPORTO sono definite come landmark, cioè sono raggiungibili da qualunque punto della site view. Infine, la pagina CATEGORIE è marcata come default dell'area PRODOTTI, cioè è quella mostrata all'utente quando accede a tale area.

Componenti e interazione in IFML Le pagine che formano le aree e le site view corrispondono alle interfacce Web che verranno utilizzate dagli utenti. Il loro contenuto è specificato utilizzando il concetto di *ViewComponent*. Un componente rappresenta una porzione di pagina finalizzata alla pubblicazione di informazione sugli oggetti della base di dati dell'applicazione o una maschera per l'inserimento di dati da parte dell'utente.

IFML standardizza un certo numero di componenti elementari utili per qualunque tipo di applicazione interattiva (*Detail*, *List* e *Form*). A questi se ne possono aggiungere altri mediante il meccanismo di estensione tipico di UML, che permette di aggiunge ulteriori componenti caratteristici delle interfacce Web. Diversi ViewComponent possono poi essere combinati all'interno di una pagina per ottenere l'effetto applicativo desiderato.

I componenti più importanti sono:

- *Form*, che rappresenta una maschera composta di vari campi per l'inserimento di dati;

- *Detail*, che mostra alcuni attributi di un singolo oggetto;
- *List*, che pubblica un elenco di oggetti e può permettere all'utente di selezionarne uno specifico. Di ciascun oggetto viene mostrato un descrittore, estratto dalla base di dati.

Con l'eccezione della Form, i componenti pubblicano nella pagina informazioni provenienti dalla base di dati, schematizzata nella fase di progettazione concettuale mediante un modello Entità-Relazione.

Pertanto, la specifica di tali componenti include due aspetti necessari a definire il contenuto pubblicato:

- il *DataBinding* indica quale entità della base di dati rappresenta il contenuto pubblicato dal componente. Graficamente il data binding è specificato come un sottoelemento all'interno del componente;
- la *ConditionalExpression* è un predicato utilizzato per determinare quali istanze dell'entità di riferimento contribuiscono a formare il contenuto del componente.

A titolo di esempio, la Figura 18.23 mostra una pagina contenente un List ViewComponent dotato di DataBinding e ConditionalExpression. Il componente pubblica le istanze dell'entità NOTIZIA e in particolare quelle per cui la data di pubblicazione è uguale o successiva al 1/1/2013, come mostrato dalla ConditionalExpression.

Il contenuto di una pagina, e quindi dei componenti in essa compresi, può dipendere dall'interazione dell'utente. IFML consente di esprimere gli eventi prodotti dall'interazione dell'utente, in particolare la navigazione dei link ipertestuali e l'invio di dati con le form, e gli effetti che questa produce sul contenuto delle pagine visualizzate. Questi aspetti sono modellati mediante il concetto di *InteractionFlow* e di evento (*Event*). Un evento, denotato come un cerchio e associato a un ViewContainer o a un ViewComponent, rappresenta il fatto che l'utente può interagire con il ViewContainer o con il ViewComponent. Un InteractionFlow è una connessione orientata tra due componenti o tra due ViewContainer, che può assumere tre significati:

- esprimere la navigazione da un punto all'altro dell'ipertesto, cambiando la pagina correntemente visualizzata dall'utente;

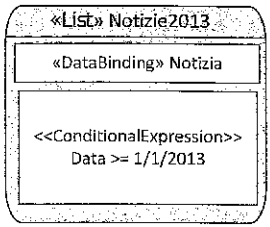

Figura 18.23 Un esempio di List ViewComponent con entità DataBinding e ConditionalExpression.

- modellare il passaggio parametri tra il componente sorgente e quello destinazione del flow, in modo da influenzare il contenuto pubblicato dal componente destinazione in base alla navigazione dell'utente;
- denotare l'attivazione di una computazione a seguito dell'interazione dell'utente.

Dal punto di vista implementativo, un InteractionFlow navigabile è realizzato tipicamente mediante un collegamento ipertestuale o un bottone di invio inserito nella pagina sorgente del flow o che contiene il componente sorgente del flow.

L'utilizzo di InteractionFlow consente di specificare la topologia dei collegamenti ipertestuali tra le pagine di una site view e l'effetto della navigazione dell'utente sul contenuto delle pagine.

L'esempio nella Figura 18.24 dimostra entrambe le caratteristiche dell'InteractionFlow. La prima pagina (CATEGORIE) contiene un List ViewComponent (TUTTE LE CATEGORIE) che pubblica l'elenco di tutte le categorie presenti nella base di dati. Tale componente ha un evento associato e quindi permette l'interazione dell'utente: infatti è l'evento SELEZIONACATEGORIA associato a un InteractionFlow che denota l'effetto della selezione nell'indice e conduce al Detail ViewComponent CATEGORIA posto nella pagina PRODOTTI.

Il Detail ViewComponent è definito sull'entità CATEGORIA e contiene come ConditionalExpression il predicato `OID=OIDCategoria`, dove `OIDCategoria` è un parametro associato all'InteractionFlow entrante del Detail ViewComponent: l'accoppiamento tra il parametro di input della condizione e l'output generato dall'interazione dell'utente con la lista selezionabile di categorie è rappresentato dall'elemento IFML `ParamBindingGroup`.

Il significato della condizione nel Detail ViewComponent è quello di selezionare l'istanza dell'entità CATEGORIA per cui l'attributo OID ha lo stesso valore del parametro `OIDCategoria`. Il significato di tale configurazione è che, a seguito della selezione da parte dell'utente di una categoria nella pagina CATEGORIE, viene mostrata la pagina prodotti e il Detail ViewComponent CATEGORIA evidenzia gli attributi dell'istanza di categoria scelta nella lista. Pertanto, la presenza dell'InteractionFlow consente sia la navigazione dell'utente verso una nuova pagina sia la definizione del contenuto di un componente in base all'interazione dell'utente. In questo caso, l'InteractionFlow si dice *NavigationFlow*.

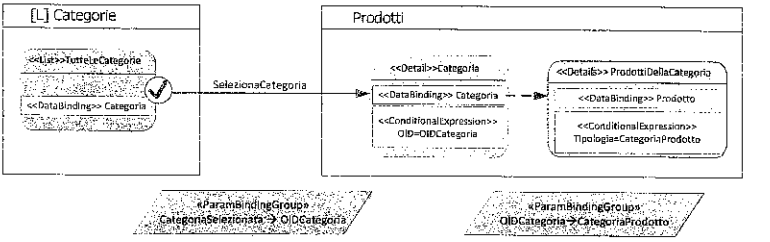

Figura 18.24 Un esempio di componenti collegati da NavigationFlow e DataFlow.

La pagina PRODOTTI contiene anche un secondo componente, il List ViewComponent PRODOTTIDELLACATEGORIA. Tale List ViewComponent è costruito sull'entità PRODOTTO e ha una ConditionalExpression parametrica definita sulla base della relazione TIPOLOGIA. Il significato di tale ConditionalExpression è che l'indice dei prodotti mostra soltanto le istanze di prodotto collegate mediante la relazione TIPOLOGIA alla categoria mostrata nel componente CATEGORIA. Si noti che il componente CATEGORIA, non essendo interattivo, non ha un evento associato e che che l'arco che rappresenta l'InteractionFlow è tratteggiato. Questa notazione indica un *DataFlow*, cioè un InteractionFlow non navigabile. Un Dataflow, a differenza di un NavigationFlow, non viene implementato come un'àncora ipertestuale utilizzabile dall'utente, ma come un semplice passaggio di parametri tra la query che realizza il componente sorgente e quella del componente destinazione. In pratica, quando l'utente sceglie una categoria e apre la pagina PRODOTTI, non solo viene mostrato il contenuto del ViewComponent CATEGORIA, ma anche quello di PRODOTTIDELLACATEGORIA, grazie al passaggio parametri specificato dal DataFlow.

Altri costrutti per la definizione di ipertesti complessi L'illustrazione di IFML fatta nel paragrafo precedente ha omesso vari aspetti del modello, utili per la definizione di applicazioni Web complesse. In particolare, IFML include due ulteriori concetti rilevanti.

- La possibilità di specificare l'esecuzione di operazioni di business a seguito dell'attivazione di InteractionFlow, mediante il concetto di *Action*. Grazie a questa primitiva è possibile rappresentare applicazioni Web per la gestione dei contenuti (creazione, modifica, cancellazione di istanze di entità e relazioni), per l'invio di messaggi, per l'interazione con servizi remoti (per esempio, Web Services, discussi nel Paragrafo 18.4.1) e altro ancora.
- Il controllo degli accessi degli utenti. IFML include la nozione di *Context* e *ViewPoint*; il Context è un oggetto che denota catteristiche del contesto in cui l'utente interagisce, per esempio il suo ruolo aziendale. Il ViewPoint è un insieme di risorse, per esempio un'intera site view, mostrato a un utente che accede in un determinato contesto. Context e ViewPoint permettono di rappresentare una site view protetta, cioè consultabile soltanto da utenti dotati degli opportuni privilegi di accesso basati sul ruolo. Azioni specifiche, quali le operazioni di autenticazione e di cambio di ruolo consentono all'utente di accedere a una site view protetta, di uscire da essa e di cambiare dinamicamente gruppo di appartenenza e quindi diritti di accesso.

18.6.4 Realizzazione dell'applicazione

La realizzazione di un'applicazione Web specificata con IFML richiede la costruzione della base di dati corrispondente allo schema Entità-Relazione e la definizione delle pagine Web atte a pubblicarne i contenuti secondo quanto previsto dallo schema delle site view.

Realizzazione della base di dati La fase di realizzazione della base di dati segue le linee guida esposte per la progettazione delle basi di dati generiche: alla

progettazione concettuale segue la fase di progettazione logica, che trasforma il modello Entità-Relazione in uno schema logico relazionale. Successivamente, la progettazione fisica deriva dallo schema logico l'insieme delle istruzioni DDL necessarie all'installazione dello schema fisico nel DBMS prescelto.

La progettazione fisica e la popolazione della base di dati, tuttavia, possono procedere in modo differente a seconda del contesto in cui l'applicazione Web viene realizzata. Nella pratica, si possono incontrare situazioni diverse.

- Uso di una *base di dati dedicata*: la base di dati dell'applicazione Web non esiste prima dello sviluppo dell'applicazione e viene creata insieme alle restanti parti dell'applicazione. Tipicamente, la popolazione della base di dati avviene via Web, per esempio con applicazioni di gestione dei contenuti specificate in IFML. Un esempio di questa situazione potrebbe essere una intranet aziendale, dove l'applicazione gestisce la distribuzione e l'aggiornamento dei dati relativi ai progetti svolti dal personale.
- Uso di una *base di dati replicata*: i contenuti dell'applicazione Web esistono, totalmente o in parte, prima dello sviluppo dell'applicazione e viene creata una copia di tali contenuti dedicata all'uso da parte dell'applicazione Web. In questo scenario, l'applicazione Web detiene e utilizza in sola lettura una copia dei contenuti aziendali. Questa situazione richiede l'utilizzo di tecnologie per la replicazione automatica della base di dati, quali quelle descritte nel Capitolo 16 oppure la realizzazione di opportuni programmi per la copia periodica dei contenuti dalla fonte dati originale a quella dedicata all'applicazione Web. Un esempio di questa situazione potrebbe essere un sito di commercio elettronico, in cui l'applicazione sfrutta una copia dei dati sui prodotti gestiti all'interno del sistema informativo aziendale.
- *Accesso diretto a fonti dati aziendali*: in questa situazione, l'applicazione Web non ha una propria base di dati, ma accede direttamente, in lettura e scrittura, ai contenuti del sistema informativo aziendale. In questo contesto, la fase di progettazione fisica della base di dati per l'applicazione Web è sostituita dalla realizzazione di opportuni programmi di interfacciamento ai sistemi aziendali, che riproducono i risultati delle query necessari alla pubblicazione delle pagine dinamiche dell'applicazione Web. Ove possibile, l'accesso ai sistemi remoti può essere realizzato sfruttando le tecnologie delle basi di dati distribuite, illustrate nel Capitolo 16, che consentono di effettuare le query di estrazione e aggiornamento dei dati a diversi livelli di trasparenza.

Realizzazione degli ipertesti La realizzazione degli ipertesti specificati a livello concettuale con IFML può sfruttare tutte le tecniche di programmazione Web descritte in questo capitolo.

Per esempio, utilizzando server side scripting in JSP, ogni pagina IFML corrisponde a un template JSP che contiene le istruzioni per il reperimento dei contenuti e la resa grafica dei componenti.

In particolare, i ViewComponent List e Detail corrispondono a query di selezione sulle tabelle relazionali che rappresentano l'entità di riferimento, con una condizione di selezione dettata dalla ConditionalExpression del componente. Invece, una Form

corrisponde a una form HTML, con tanti input field quanti sono i campi della form IFML.

L'aspetto più complesso della realizzazione degli ipertesti è l'implementazione degli InteractionFlow, la cui realizzazione procede in due modi differenti, a seconda che l'InteractionFlow connetta componenti nella stessa pagina o in pagine diverse.

- Se l'InteractionFlow connette due pagine diverse, si procede come descritto nell'esempio di template JSP visto nel Paragrafo 18.2.3. L'InteractionFlow viene reso come un'àncora (se uscente da un Detail) o come un'insieme di àncore (se uscente da un List ViewComponent). Nella costruzione del tag <A> che rappresenta il link è necessario produrre il valore corretto dell'attributo `href`, che specifica l'URL del template da visualizzare a seguito della navigazione dell'InteractionFlow. Tale valore è formato dall'URL del template corrispondente alla pagina destinazione dell'InteractionFlow, cui si aggiunge una query string con tutti i parametri necessari al computo della pagina destinazione (per esempio, l'OID dell'oggetto visualizzato, nel caso di un Detail ViewComponent, oppure l'OID dell'oggetto corrispondente alla riga corrente dell'indice, nel caso di una List ViewComponent).
- Se l'InteractionFlow connette due componenti dentro la stessa pagina, si procede diversamente a seconda che sia un NavigationFlow oppure un DataFlow. Se l'InteractionFlow è un NavigationFlow viene reso come un'àncora usando la stessa tecnica del punto precedente. In questo caso, però, l'URL della pagina destinazione coincide con quello del template che contiene l'àncora. Nel caso invece in cui sia un DataFlow, non è necessario prevedere un'àncora, poiché il passaggio dei parametri avviene senza che l'utente debba interagire. Per ottenere questo effetto, è sufficiente che il template di pagina effettui la query parametrica associata al componente destinazione del flow, sfruttando come valore del parametro in ingresso il risultato della query del componente sorgente del flow. Per esempio, il template che realizza la pagina IFML PRODOTTI mostrata nella Figura 18.24 contiene sia la query per il componente CATEGORIA sia quella per il componente PRODOTTI DELLA CATEGORIA. La seconda query sfrutta come parametro l'OID dell'istanza visualizzata nel Detail ViewComponent.

L'implementazione di ipertesti complessi richiede ulteriori tecniche di programmazione rispetto agli esempi visti in questo capitolo, per esempio al fine di gestire l'autorizzazione all'accesso degli utenti, le operazioni di aggiornamento dei dati, l'insorgenza di errori a runtime, la memorizzazione di informazioni nella sessione e così via. Questi temi esulano dal contenuto di questo volume, per cui si rinvia il lettore interessato alle note bibliografiche per un approfondimento.

Note bibliografiche

Esistono molte fonti di informazioni relative a Internet e al Web, in forma tradizionale oppure elettronica, disponibili attraverso il Web stesso. Citiamo alcuni riferimenti di interesse generale. Una discussione approfondita dei protocolli di rete si trova nel testo di Comer [108]; una presentazione introduttiva dei servizi informativi su Internet (in

particolare, ma non solo, il Web) è offerta da Liu *et al.* [130]. Greenspun [122] considera molti aspetti, fra cui un'articolata discussione dei WIS, commercio elettronico, insieme a una descrizione di HTML e di aspetti architetturali, quali i programmi CGI e i server API.

Un punto di riferimento fondamentale per la storia e le tecnologie del Web è il sito del Consorzio WWW (`http://www.w3.org`), che contiene le definizioni di tutti gli standard rilevanti.

Un'altra preziosa risorsa online è il sito didattico W3 Schools, che offre tutorial su tutti gli aspetti dello sviluppo di applicazioni Web, inclusi HTML, CSS, JavaScript, AJAX, JSON e altro ancora (`http://www.w3schools.com`).

Materiali tecnici sullo sviluppo di programmi per la pubblicazione dinamica di pagine, in particolare su Java Servlet e sul linguaggio di server side scripting JSP, sono reperibili sul sito Web della Oracle (`http://www.oracle.com`).

Il riferimento ufficiale per gli sviluppatori di applicazioni Web per Microsoft .NET è la sezione del sito Microsoft Developer Network dedicata a questo tema, raggiungibile dalla home page `http://msdn.microsoft.com`.

Discussioni sugli aspetti architetturali relativi all'interazione fra Web e basi di dati possono essere trovati nei testi di Ju [126], Cheng e Malaika [106] e Feiler [114].

La formulazione di REST è contenuta nell'articolo [115].

La progettazione concettuale delle applicazioni Web è oggetto di un dibattito che ha portato a numerose proposte di modelli concettuali. Le proposte più recenti derivano da lavori di ricerca antecedenti, sul tema della progettazione delle applicazioni ipermediali. Una sintesi dei principali linguaggi di specifica di applicazioni ipermediali è contenuto nel numero speciale della rivista *Communications of the ACM* [97]. Il tema dello sviluppo delle applicazioni Web, e in particolare dei WIS, è approfondito in una serie di articoli curata da Isakowitz, Bieber e Vitali [125].

La modellazione concettuale con WebML, il linguaggio cha ha dato vita al successivo standard IFML, è trattata ampiamente nel libro di Ceri *et al.* [104], dove si spiegano approfonditamente il linguaggio e il processo di sviluppo di un'applicazione Web, dall'analisi dei requisiti alla realizzazione, con l'ausilio di un caso di studio concreto.

Lo standard IFML è disponibile nella sezione sulle specifiche del sito di OMG (`http://www.omg.org/spec/IFML/`). Materiali didattici, documentazione e un forum per la comunità di sviluppatori in IFML sono ospitati sul sito `http://www.webratio.com`.

Esercizi

Soluzioni sul sito http://www.ateneonline.it/atzeni

18.1 Commentare le caratteristiche di HTTP come protocollo client-server, evidenziando gli aspetti differenti rispetto ai protocolli client-server tradizionali.

18.2 Scrivere una form HTML che permetta all'utente di inviare il proprio nome e cognome a un programma CGI che stampa una pagina di HTML di saluto, contenente il nome e cognome dell'utente e l'ora di sistema.

18.3 Riscrivere il programma CGI dell'esercizio precedente come servlet e come template JSP.

18.4 Dato lo schema concettuale della base di dati contenente autori e libri utilizzato nel Paragrafi 18.2.1, 18.2.2 e 18.2.3, scrivere un template JSP contenente una form di ricerca che permetta di inserire il cognome di un autore. Scrivere un

secondo template che, ricevuto il cognome di un autore, stampi la lista dei libri da lui scritti.

18.5 Scrivere l'interfaccia WSDL di un Web service che riceve in input il CAP di una città e restituisce in output le previsioni metereologiche per i sette giorni successivi alla data corrente. Ogni previsione è formata da una data, i valori delle temperature minima e massima previste e un breve testo che descrive le condizioni meteo.

18.6 Riformulare secondo lo stile REST l'interfaccia e la descrizione delle risorse del web service illustrato nell'esercizio precedente.

18.7 Scrivere un programma a lato client in JavaScript che modifichi lo stato di una pagina HTML a seguito del click dell'utente su un bottone denominato `Evidenzia`, cambiando il colore dello sfondo del testo contenuto in tutti gli elementi HTML di tipo `h1`.

18.8 Scrivere una pagina HTML contenente una form con i campi nome e cognome e un programma JavaScript che risponda al click sul bottone di invio memorizzando in modo persistente i dati inseriti nei campi della form. Formulare una versione del programma che utilizzi la API Storage e una che utilizzi la API Indexed Database.

18.9 Si vuole costruire un sito Web per la società di formazione discussa negli esempi dei Capitoli 7 e 8 di questo libro. Allo scopo:

- specificare i requisiti per il sito, individuando le informazioni di interesse per il pubblico esterno all'azienda;
- progettare lo schema concettuale dell'ipertesto.

18.10 Si vuole costruire un sito Web per la gestione di una bacheca di messaggi. Allo scopo:

- specificare i requisiti per il sito, individuando le informazioni di interesse per gli utenti;
- progettare lo schema concettuale della base di dati e dell'ipertesto;
- realizzare la base di dati e le pagine dell'ipertesto. In particolare, utilizzare JSP per le pagine dinamiche di pubblicazione dei contenuti e Java Servlet per l'operazione di creazione di un messaggio nella bacheca.

19 Architetture per l'analisi dei dati

La tecnologia delle basi di dati è finalizzata prevalentemente alla gestione efficiente e affidabile di dati "in linea" (*On Line Transaction Processing*, OLTP). Tramite questa tecnologia, le imprese accumulano grandi moli di dati relativi alla loro gestione operativa quotidiana. Per esempio, le banche raccolgono dati relativi alle transazioni dei loro clienti, mentre le catene di grandi supermercati dispongono dei dati relativi alle vendite quotidiane dei loro prodotti. Questi dati possono però rivelarsi utili non solo per la gestione dell'impresa, ma anche per la pianificazione e il supporto alle decisioni. Per esempio, è possibile capire quali tra i prodotti finanziari offerti dalle banche hanno maggior successo, oppure osservare le variazioni nelle vendite dei diversi prodotti di un supermercato in funzione delle varie promozioni per stabilirne l'efficacia. In pratica, i dati del presente e del passato possono consentire un'attività di analisi essenziale nella pianificazione e programmazione delle attività future dell'impresa.

Per molti decenni lo sviluppo della tecnologia ha trascurato l'analisi dei dati. Si è pensato che i linguaggi di interrogazione e le architetture dei sistemi di gestione di basi di dati fossero adeguati sia per la gestione operativa sia per l'analisi. In effetti, SQL consente di costruire interrogazioni in modo arbitrario e offre quindi alcune caratteristiche utili per l'analisi dei dati, ma l'analisi viene tipicamente svolta da utenti che non sono necessariamente esperti informatici e il linguaggio SQL non è adatto a loro. Inoltre, è assai difficile ottimizzare un'applicazione in modo che soddisfi contemporaneamente le esigenze di analisi e di gestione in linea. Di conseguenza, le ultime hanno storicamente prevalso sulle prime.

Con l'inizio degli anni Novanta, parallelamente allo sviluppo delle reti e dei prodotti per la distribuzione dei dati, si sono imposte nuove architetture, caratterizzate dalla separazione degli ambienti: a fianco dei sistemi per OLTP si sono sviluppati sistemi dedicati esclusivamente all'elaborazione e analisi dei dati, e cioè a *On Line Analytical Processing* (OLAP). Questa sigla mette in evidenza che l'analisi dei dati avviene tramite strumenti interattivi che forniscono risposte in tempi brevi, anche a fronte di grandi moli di dati da prendere in considerazione.

A causa delle diverse esigenze delle elaborazioni transazionali e analitiche, esistono alcune importanti differenze tra OLTP e OLAP. Innanzitutto, mentre le applicazioni OLTP sono normalmente condivise da un elevatissimo numero di utenti finali, le applicazioni OLAP sono caratterizzati dalla presenza di pochi utenti, che però occupano posizioni di alto livello nell'impresa e svolgono attività decisionali. È possibile che le funzioni di analisi vengano svolte da un gruppo di specialisti (gli "analisti") che realizzano le analisi dei dati commissionate dai dirigenti, ma è sempre più diffusa l'esigenza di dotare gli strumenti OLAP di interfacce di facile uso, in modo da permettere un'attività decisionale più immediata ed efficace, senza la necessità di intermediari. Un'altra differenza è che mentre nei processi OLTP viene descritto di solito solo lo "stato corrente" di un'applicazione, per i motivi sopra citati, i dati presenti nel data warehouse sono tipicamente di tipo *storico-temporale*. Inoltre, mentre l'OLTP è basato prevalentemente su l'esecuzione di transazioni relativamente semplici su pochi dati, l'OLAP richiede interrogazioni complesse su grandi moli di dati. Esiste infine il problema della *qualità dei dati*: spesso, infatti, la semplice raccolta di dati non consente analisi significative, in quanto i dati contengono inesattezze, errori e omissioni. A causa di queste diversità, si sono largamente diffusi strumenti dedicati specificatamente all'analisi e nuovi paradigmi per l'interrogazione e la gestione dei dati.

A partire dagli anni 2000, si è imposta un'architettura per l'analisi dei dati basata sulla costruzione di una particolare base di dati chiamata *data warehouse* (magazzino di dati) a supporto delle decisioni, nella quale vengono raccolte tutte le informazioni di ausilio all'analisi. In questa architettura, i sistemi OLTP svolgono il ruolo di "sorgenti di dati" (*data source*), alimentano cioè l'ambiente OLAP, a seguito di opportune operazioni di integrazione, caricamento e pulizia dei dati. La tecnologia adottata per la costruzione di questi data warehouse è prevalentemente relazionale, anche se si sono diffuse altre tecnologie di memorizzazione dei dati specificatamente pensate per l'OLAP.

Questo scenario si è recentemente evoluto a causa della grandissima disponibilità di dati, di natura diversa da quelli generati dai sistemi OLTP, che oggi, grazie all'evoluzione della tecnologia informatica, possono essere facilmente raccolti e utilizzati per il supporto alle decisioni. Si pensi per esempio alle informazioni che possono essere estratte dai log degli accessi a un sito Web, dai messaggi inviati sui social network relativi a un particolare argomento, dagli interventi degli utenti in blog specializzati e dai questionari di gradimento dei clienti. La raccolta di questi dati, per loro natura altamente destrutturati, produce magazzini di dati talmente grandi che la tecnologia tradizionale per la gestione dei dati risulta spesso inadeguata. È stato coniato allo scopo il termine di *big data*, proprio per indicare quelle collezioni di dati la cui dimensione supera le capacità di memorizzazione, gestione e analisi tipiche dei tradizionali sistemi per basi di dati. Per risolvere questo problema, sono state proposte nuove soluzioni architetturali per l'analisi dei big data che prevedono, per esempio: la distribuzione e la replicazione dei dati su cluster di computer, la loro memorizzazione in formati molto elementari (per esempio basati su semplici coppie chiave–valore), la parallelizzazione delle attività richieste dall'analisi di dati e l'adozione di infrastrutture basate sulla virtualizzazione delle risorse (il cosidetto *cloud computing*). Queste nuove tecniche non intendono necessariamente soppiantare le soluzioni consolidate per l'analisi, ma piuttosto affiancare le tecnologie pensate per contesti classici di OLAP con nuovi strumenti sviluppati appositamente per le caratteristiche dei big data e dei nuovi scenari applicativi.

Questo capitolo è dedicato alla presentazione delle tecniche e delle metodologie per l'analisi dei dati sopra citate. Inizieremo con la descrizione delle architetture consolidate, basate sulla costruzione e l'uso di data warehouse. Ci occuperemo poi di un modello astratto di rappresentazione e gestione dei dati, detto *multidimensionale*, largamente utilizzato in questo contesto. Successivamente, illustreremo le modalità di realizzazione di un data warehouse e presenteremo una metodologia di progetto. Passeremo poi a una descrizione dei principali problemi di *data mining*, ovvero di particolari tecniche algoritmiche utilizzate per la ricerca di informazioni nascoste nei dati e concluderemo il capitolo con una discussione sui big data e sulle soluzioni architetturali per la loro gestione.

19.1 Architettura di un data warehouse

Un data warehouse (DW) è una base di dati utilizzata per il supporto alle decisioni e possiede per questo alcune caratteristiche peculiari che in parte la distinguono rispetto a una base di dati dedicata allo svolgimento di operazioni OLTP.

- *È una base di dati integrata.* I dati di interesse provengono infatti da diverse sorgenti informative preesistenti e questo richiede un'attività propedeutica di riconciliazione delle eterogeneità (diversità nei nomi dei campi, nella struttura, nella codifica dell'informazione). Per esempio, risulta spesso necessario unificare date rappresentate con formati diversi.
- *Contiene informazioni di carattere storico/temporale.* Mentre le basi di dati operazionali mantengono il valore corrente delle informazioni, nel DW, per sua natura, è di interesse l'evoluzione storica delle informazioni. Uno degli obiettivi tipici dell'analisi è infatti quella di individuare in queste evoluzioni andamenti inattesi,

regolarità o irregolarità. Per esempio, mentre a livello operazionale è di interesse il valore corrente di un fondo azionario, in fase di analisi è di maggior interesse il suo andamento temporale.

- *Contiene tipicamente dati in forma aggregata.* Le principali attività di analisi dei dati per il supporto alle decisioni si basano su informazioni ottenute aggregando i dati disponibili rispetto a opportune coordinate. Per esempio, i dati di vendita di una catena di supermercati possono essere analizzati aggregandoli in base al periodo temporale, alla collocazione geografica e alla tipologia di prodotto venduto.
- *Ha un'esistenza autonoma.* Il data warehouse viene sempre mantenuto fisicamente separato dalle sorgenti informative. È infatti difficile fare l'integrazione in linea perché i dati di interesse sono diversi e il mantenimento dei dati storici, l'aggregazione e l'analisi richiedono organizzazioni speciali e metodi di accesso specifici. Inoltre, senza la separazione, assisteremmo a un degrado generale delle prestazioni dei sistemi OLTP.
- *È una base di dati fuori linea.* I meccanismi di importazione dei dati sono normalmente di tipo asincrono e periodico, in modo da non penalizzare le prestazioni delle sorgenti di dati, specie se si tratta di sistemi OLTP con requisiti di prestazioni particolarmente critici. In tal caso, il DW non contiene dati perfettamente aggiornati rispetto al flusso di transazioni che operano nei sistemi OLTP. Un disallineamento controllato dei dati, però, è in genere ritenuto accettabile per molte applicazioni di analisi.

Come illustrato in Figura 19.1, un'architettura adatta a soddisfare questi requisiti comprende generalmente i seguenti componenti.

- Le *sorgenti* dei dati o *data source*. I dati vengono estratti da uno o più sistemi preesistenti, nel sistema informativo aziendale, per la gestione operativa quotidiana (spesso detti sistemi operazionali) oppure esterni a esso ma comunque accessibili con regolarità (per esempio rilevazioni ISTAT). I data source includono una vasta

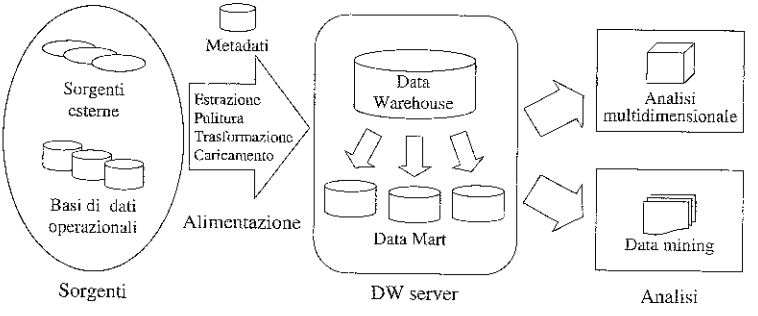

Figura 19.1 Architettura di riferimento per un sistema di data warehouse.

tipologia di sistemi, comprendenti anche raccolte di dati non gestite tramite DBMS oppure gestite da DBMS di vecchia concezione (i cosiddetti sistemi *legacy*).

- Il *data warehouse server*. È il sistema dedicato alla gestione del data warehouse e può basarsi su diverse tecnologie (ROLAP o MOLAP) come verrà illustrato nel Paragrafo 19.3. Questa componente memorizza i dati mediante opportune strutture fisiche e realizza in modo efficiente interrogazioni complesse, caratterizzate da join fra tabelle, ordinamenti e aggregazioni. Esso consente anche speciali operazioni, quali *roll-up*, *drill-down* e *data cube*, che verranno illustrate nel Paragrafo 19.2.2. Molto spesso a partire dal data warehouse si costruiscono dei *data mart*, ovvero dei sottoinsiemi logici del DW in grado di soddisfare specifiche esigenze di analisi. Per esempio, quelle relative a un particolare settore dell'azienda.
- Un *sistema di alimentazione*. Esso consiste di una serie di strumenti detti di ETL (*Extract, Transform, Load*) che svolgono le seguenti attività di base.

  - L'*estrazione* dei dati dalle sorgenti. Dopo il popolamento iniziale, il processo di estrazione è tipicamente incrementale: lo strumento per l'esportazione dei dati colleziona le sole modifiche (inserimenti o cancellazioni) dei data source secondo modalità illustrate più avanti e le comunica ai moduli successivi.
  - La *pulizia* dei dati (*data cleaning*). Lo scopo è quello di analizzare la correttezza dei dati prima dell'inserimento nel data warehouse. Questo avviene applicando dei filtri in grado di eliminare dati palesemente scorretti sulla base di controlli che si applicano a singoli data source, oppure rilevando e talvolta correggendo errori e inconsistenze nei dati estratti. Tale attività è essenziale per assicurare un sufficiente livello di qualità dei dati.
  - La *trasformazione* dei dati. Questo processo predispone i dati all'uso operativo, svolgendo prevalentemente un'opera di riconciliazione delle eterogeneità presenti nelle varie sorgenti informative. In questa fase vengono svolte, per esempio, conversioni, trasformazioni di formato e associazioni tra campi equivalenti di sorgenti diverse. Vengono inoltre eseguite operazioni di denormalizzazione (come infatti vedremo nel Paragrafo 19.3.1, un data warehouse è spesso non normalizzato), di ordinamento e di aggregazione.
  - Il *caricamento* dei dati nel data warehouse. Tipicamente le operazioni di acquisizione vengono svolte a lotti (in *batch*) quando il DW non è utilizzato per l'analisi (in genere di notte) e si può procedere in due maniere. In alcune applicazioni, caratterizzate da un numero limitato di dati, si acquisisce periodicamente l'intero contenuto del DW. Più spesso i dati vengono allineati in modo incrementale (*refresh*), propagando al DW le modifiche fatte sulle sorgenti dei dati. Nel processo di *refresh* si possono usare due tecniche: *invio dei dati* (*data shipping*) e *invio delle transazioni* (*transaction shipping*). La prima tecnica può utilizzare dei trigger collocati nel data source che, in modo trasparente alle applicazioni, registrano gli aggiornamenti in opportuni *archivi variazionali*. La seconda tecnica costruisce gli archivi variazionali utilizzando i log di transazione (si veda il Paragrafo 2.2.2). In entrambi i casi, gli archivi variazionali vengono prodotti dal modulo di estrazione e utilizzati dal modulo di caricamento per aggiornare il DW, aggiungendo i dati relativi agli inserimenti e, in genere, marcando i dati corrispondenti alle cancellazioni come dati storici.

- Alcuni *strumenti di analisi*. Questi strumenti consentono di effettuare analisi dei dati usufruendo dei servizi offerti dal DW server e offrono interfacce amichevoli in grado di presentare, in forma adeguata e facilmente comprensibile, i risultati delle analisi. Si distingue in particolare tra due principali tipologie di analisi (e quindi di strumenti):
  - l'analisi multidimensionale, che consiste sostanzialmente in operazioni interattive di aggregazione/disaggregazione dei dati da analizzare lungo opportune coordinate o "dimensioni" (temporali, geografiche ecc.) secondo modalità che verranno discusse nel Paragrafo 19.2;
  - il *data mining*, che consente di svolgere ricerche sofisticate sulle informazioni "nascoste" nei dati, secondo le tecniche discusse nel Paragrafo 19.5. Come vedremo, in questo caso i dati sono tipicamente destrutturati e per questo motivo le operazioni necessarie al popolamento del data warehouse sono spesso semplificate.

In aggiunta, un DW è spesso dotato di moduli di ausilio alla sua progettazione e gestione. In particolare:

- un archivio contenente una descrizione completa del data warehouse (i cosiddetti *metadati*), utile per comprendere da un lato come è stato costruito il DW e dall'altro quali analisi dei dati possono essere eseguite su di esso;
- un componente per l'*assistenza allo sviluppo* del data warehouse, che consente di definire lo schema dei dati e i meccanismi per l'alimentazione del DW; si tratta di uno strumento di ausilio alla progettazione, del tipo di quelli illustrati alla fine dei Capitoli 7 e 8.

Citiamo inoltre il fatto che può essere presente un livello fisico intermedio tra sorgenti e DW (detto *staging area*) nel quale vengono memorizzati i dati dopo la fase di estrazione, pulizia e integrazione ma prima della fase di caricamento nel DW. Questa scelta consente di gestire separatamente le attività propedeutiche alla costruzione del DW.

Va detto che nelle architetture reali non tutte le componenti illustrate sono presenti e sono possibili alcune varianti. In particolare, la costruzione di un *data warehouse aziendale*, che descriva tutti i dati utili all'analisi presenti in un'organizzazione, è un obiettivo ambizioso e assai difficile da realizzare. Per questo motivo, è oggi prevalente l'approccio di costruire il DW in maniera incrementale, concentrandosi separatamente su sottoinsiemi più semplici dei dati aziendali (dati dipartimentali o settoriali), per i quali sia ben chiaro l'obiettivo dell'analisi. Come detto in precedenza, ciascun archivio semplificato dei dati da analizzare prende il nome di *data mart*. Secondo questo approccio il data warehouse è di fatto costituito da un insieme di data mart. Ne consegue che, con riferimento all'architettura riportata in Figura 19.1, non esiste un archivio "globale" centralizzato. Questa soluzione richiede però uno sviluppo coordinato dei vari data mart, nel quale si deve cercare di mantenere omogenee le dimensioni comuni di data mart diversi. In questa maniera, è possibile confrontare e, se necessario o utile, integrare data mart sviluppati separatamente. Il risultato del coordinamento di dimensioni omogenee appartenenti a data mart diversi viene detto *data warehouse bus*. L'architettura complessiva di un sistema di data warehouse con le varianti appena discusse è illustrata in Figura 19.2.

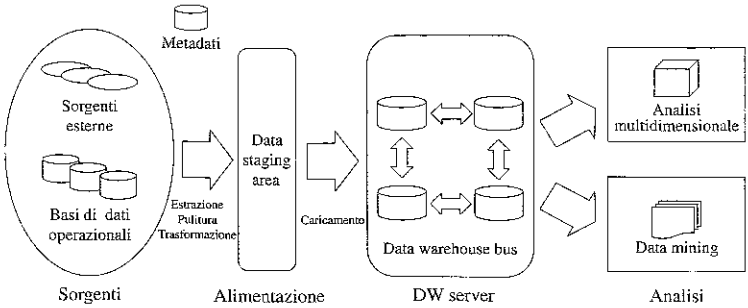

Figura 19.2 Architettura di un DW con staging area e data warehouse bus.

Concludiamo questa presentazione dell'architettura con alcune considerazioni relative alla qualità dei dati. La qualità dei dati è un elemento essenziale per il successo di un data warehouse. Infatti, se i dati memorizzati contengono imprecisioni o errori, l'analisi risultante sarà necessariamente fuorviante, e l'uso del DW potrà risultare addirittura controproducente. Purtroppo, vari fattori pregiudicano la qualità dei dati.

- In basi di dati prive di vincoli di integrità, magari perché gestite con tecnologie pre-relazionali, il tasso di errori (*dirty data*) è assai elevato; alcune stime indicano che i dati errati in applicazioni commerciali oscillano tra il 5 e il 30% del totale.
- In DW costruiti assemblando dati estratti da fonti plurime si aggiungono problemi di disallineamento delle informazioni presenti in basi di dati diverse; questo problema è ovviamente accentuato dall'eterogeneità dei sistemi utilizzati.

Per ottenere elevati livelli di qualità è necessario usare in maniera opportuna gli strumenti di pulizia, esprimendo un gran numero di regole di integrità ed eliminando o correggendo i dati che non soddisfano tali regole. Più in generale, il problema della qualità di una data source si dovrebbe affrontare a monte, osservando con attenzione il processo di produzione dei dati, e associando a esso azioni di verifica e correzione di eventuali dati errati.

19.2 Rappresentazione multidimensionale dei dati

I dati presenti in un data warehouse vengono presentati all'utente finale (che, ricordiamo, è generalmente un utente esperto del dominio applicativo ma spesso non ha competenze tecniche) mediante una rappresentazione di alto livello che prescinde dai criteri di memorizzazione dei dati e ne favorisce l'analisi. Questa rappresentazione fa riferimento a un modello di rappresentazione noto come *modello multidimensionale*. Tenendo conto del fatto che non esiste una standardizzazione, nel seguito daremo una descrizione generale, ma non rigorosa, di questo modello.

19.2.1 Il modello multidimensionale

Il modello dei dati multidimensionale è basato su tre concetti di base: il fatto, la misura e la dimensione. Un *fatto* è un concetto del sistema informativo aziendale (o più precisamente, della relativa realtà di interesse) sul quale ha senso svolgere un processo di analisi orientato al supporto alle decisioni. Una *misura* è invece una proprietà atomica di un fatto che intendiamo analizzare (tipicamente un attributo numerico o un conteggio delle sue istanze). Infine, una *dimensione* è una particolare prospettiva lungo la quale l'analisi di un fatto può essere effettuata. I valori possibili per una dimensione vengono generalmente detti *membri*.

Per esempio, in un'azienda commerciale di vendita al dettaglio che dispone di una catena di supermercati un fatto può essere il concetto di *Vendita* e possibili misure per questo fatto possono essere la *quantità* venduta di un prodotto in un certo periodo di tempo e l'*incasso* relativo. Possibili dimensioni di analisi per una vendita possono essere l'*articolo* venduto, il *periodo di tempo* nel quale la vendita è stata effettuata e il *luogo* in cui si è svolta.

Per un'assicurazione invece un fatto può essere il *sinistro* di cui si chiede un rimborso assicurativo e possibili misure possono essere il *numero* di sinistri in un certo periodo di tempo e il *costo* relativo. Possibili dimensioni di analisi per un sinistro possono essere il *cliente* che richiede il rimborso, la *tipologia del sinistro*, la *polizza* e il *periodo di tempo* nel quale è avvenuto il sinistro.

Le dimensioni vengono tipicamente organizzate in gerarchie di *livelli di aggregazione*. Per esempio, la Figura 19.3 presenta una possibile strutturazione a livelli delle dimensioni relative all'applicazione commerciale appena descritta.

In questo esempio la dimensione LUOGO è stata organizzata nei livelli negozio, città, provincia e regione, la dimensione ARTICOLO nei livelli prodotto, categoria e marca e infine la dimensione TEMPO nei livelli giorno, mese, trimestre e anno. Le varie frecce stanno a indicare possibili aggregazioni lungo una dimensione. Per esempio, nella dimensione LUOGO i negozi possono essere raggruppati per città, le città per provincia e le province per regione. Si osservi che è in generale possibile avere diverse aggregazioni per lo stesso livello. Per esempio, nella dimensione ARTICOLO i prodotti possono essere raggruppati sia per categoria sia per marca. Non è invece possibile raggruppare le categorie di prodotto per marca.

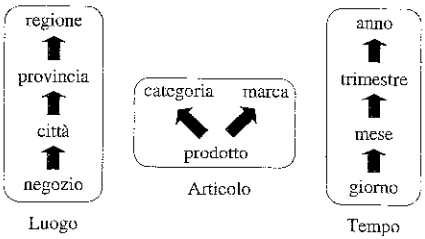

Figura 19.3 Organizzazione di dimensioni in livelli.

I valori possibili per un livello di una dimensione sono detti *membri* della dimensione. Possibili membri per il livello città della dimensione LUOGO possono essere, per esempio, tutti i comuni italiani nei quali esistono negozi dei quali vogliamo analizzare le vendite.

Una volta fissate le dimensioni di un fatto e un livello di aggregazione di interesse per ogni dimensione, un'*istanza* del fatto assegna a ogni combinazione valida di membri dei livelli selezionati un valore per ciascuna misura del fatto. Per esempio, una possibile istanza di un fatto Vendita sulle dimensioni di Figura 19.3 potrebbe associare il valore 2578 della misura *quantità venduta* ai membri: *Roma* del livello città della dimensione LUOGO, *Televisori* del livello prodotto della dimensione ARTICOLO e *maggio 2014* del livello mese della dimensione TEMPO.

Esiste una naturale rappresentazione grafica, peraltro largamente utilizzata dagli strumenti di analisi, nella quale le istanze di un fatto sono rappresentate da *cubi* multidimensionali costituiti da elementi atomici detti *celle*. Un esempio di cubo che rappresenta un'istanza del fatto Vendita per la catena di supermercati sui livelli negozio della dimensione LUOGO, trimestre della dimensione TEMPO e prodotto della dimensione ARTICOLO sopra citata è riportato in Figura 19.4.

In questi cubi ogni dimensione "fisica" corrisponde a una dimensione "concettuale" del fatto a un certo livello di aggregazione e le celle del cubo contengono le istanze del fatto. Fissando uno specifico membro per ogni dimensione, tra quelli ammissibili per il livello specificato, si determina la coordinata di una particolare cella del cubo. In questa cella viene riportata l'istanza relativa, ovvero la misura del fatto associata ai membri dati. Per esempio, nella cella di coordinate *2 trim. 2014*, *Roma-1* e *Tablet* viene riportata la quantità di tablet venduta nel negozio Roma-1 nel secondo trimestre del 2014.

In questo modello di rappresentazione, un data warehouse è costituito da un insieme di cubi dimensionali definiti su una collezione uniforme di dimensioni (questo garantisce la possibilità di combinare il contenuto di cubi diversi sulla base di membri di una dimensione condivisa). In una organizzazione basata su data warehouse bus, si può assumere che ogni cubo corrisponde a un data mart.

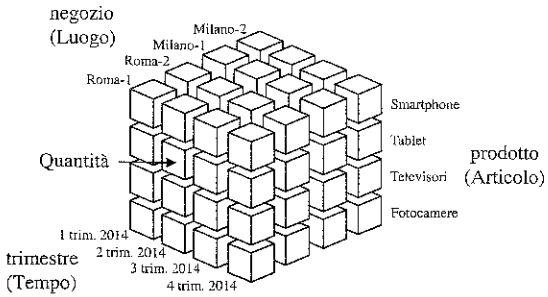

Figura 19.4 Un cubo multidimensionale rappresentante un fatto di analisi.

19.2.2 Operazioni nel modello multidimensionale

L'analogia con i cubi non si limita alla presentazione dei dati. Vengono infatti definite per questo modello alcune operazioni di analisi che si esprimono come operazioni sui cubi. Si tratta cioè di operazioni che si applicano a cubi multidimensionali e restituiscono nuovi cubi, non necessariamente con lo stesso numero di dimensioni. Le più note sono lo *slice-and-dice*, il *roll-up* e il *drill-down*.

L'operazione di *slice-and-dice* consiste nella semplice selezione di un sottoinsieme delle celle di un cubo e viene chiamata in questa maniera proprio perché si può ottenere "affettando e tagliando a cubetti" il cubo stesso. In Figura 19.5 viene riportato per esempio il cubo bidimensionale che si ottiene applicando un'operazione di slice-and-dice sul cubo vendita di Figura 19.4. Questa operazione seleziona le vendite per trimestre e negozio dei soli lettori di DVD.

L'operazione di *roll-up* consiste in un'aggregazione dei dati di un cubo seguita dall'applicazione di una funzione aggregativa (in genere la somma). Ci sono due maniere per procedere. La prima consiste nell'aggregare lungo i livelli di una o più dimensioni del cubo, salendo nelle rispettive gerarchie. Un esempio di questo tipo viene riportato in Figura 19.6 che mostra il risultato di un'operazione di roll-up sul cubo in Figura 19.5, ottenuto passando dal livello negozio al livello città lungo la dimensione LUOGO e applicando la funzione di somma.

Il secondo tipo di roll-up consiste nella completa eliminazione di una dimensione del cubo seguita dall'applicazione di una funzione aggregativa. Per esempio, se eliminiamo la dimensione LUOGO dal cubo in Figura 19.4 e applichiamo ancora una somma otteniamo un cubo bidimensionale come quello di Figura 19.7. Questo cubo contiene le vendite complessive di tutti i negozi organizzate per tipologia di prodotto e per trimestre.

TABLET	1 trim. 14	2 trim. 14	3 trim. 14	4 trim. 14
Roma-1	38	91	66	198
Roma-2	155	219	248	265
Milano-1	121	273	266	326
Milano-2	222	122	155	200

Figura 19.5 Il risultato di un'operazione di slice-and-dice sul cubo in Figura 19.4.

TABLET	1 trim. 14	2 trim. 14	3 trim. 14	4 trim. 14
Roma	193	310	314	463
Milano	343	395	421	526

Figura 19.6 Il risultato di un'operazione di roll-up sul cubo in Figura 19.5.

VENDITE TRIM.	1 trim. 14	2 trim. 14	3 trim. 14	4 trim. 14
Cellulari	536	705	735	989
Tablet	567	716	606	717
Televisori	187	155	186	226
Fotocamere	175	191	202	319

Figura 19.7 Il risultato di un'operazione di roll-up sul cubo in Figura 19.4.

Si osservi che in realtà non esiste una differenza sostanziale tra i due tipi di roll-up. Infatti l'eliminazione di una dimensione da un cubo consiste di fatto nell'aggregazione lungo la dimensione eliminata dal livello originale a una sorta di livello massimo (tipicamente denominato *all*) che raggruppa in un solo elemento tutti i possibili membri di una dimensione.

Un concetto importante legato al roll-up è quello dell'*additività*. Si dice che la misura di un fatto è additiva lungo una dimensione se ha senso sommarla rispetto a ogni possibile aggregazione ottenuta sui livelli della dimensione. In altri termini, la misura è additiva se nell'ambito della dimensione, una qualunque operazione di roll-up basata sulla somma produce un risultato corretto. Le misure *quantità* e *incasso* del fatto Vendita sono entrambe additive per tutte le dimensioni. La scorta di magazzino di un certo prodotto è invece non additiva lungo la dimensione temporale, perché non ha ovviamente senso effettuare un roll-up basato su una somma per tale dimensione, mentre è additiva per la categoria di prodotto. Si dice che una misura è *semiadditiva* se, come in quest'ultimo esempio, è possibile sommarla aggregando rispetto ad alcune (ma non a tutte) dimensioni. Naturalmente, può avere senso lo stesso aggregare misure non completamente additive mediante funzioni diverse dalla somma (per esempio, medie pesate). L'additività è una proprietà importante perché le applicazioni di data warehousing devono solitamente combinare molti fatti e il modo più comune di combinare un insieme di fatti è di sommarli. In questi casi, l'additività garantisce la significatività della relativa operazione di roll-up. Purtroppo, però, gli strumenti non sempre offrono un supporto per la verifica di questa proprietà. Tale verifica risulta in questo caso a carico dell'analista, che si deve basare sulla sua conoscenza del dominio applicativo.

Il *drill-down* è l'operazione inversa del roll-up. Consente cioè di aggiungere dettaglio a un cubo disaggregandolo lungo una o più dimensioni. Questo si ottiene passando dal livello originale di una dimensione del cubo a un altro che lo precede nella rispettiva gerarchia. Per esempio, il cubo in Figura 19.8 è ciò che si ottiene applicando al cubo in Figura 19.7 un'operazione di drill-down che commuta il livello della dimensione TEMPO da trimestre a mese.

Si osservi come un'operazione di drill-down richiede dati di dettaglio che sono tipicamente esterni alla tabella su cui opera. Con riferimento all'esempio appena visto, risulta evidente che le vendite mensili non possono essere ottenute dalle vendite trimestrali, ma bisogna avere a disposizione, per esempio, le vendite giornaliere.

VENDITE MENS.	gen 14	feb 14	mar 14	apr 14	mag 14	giu 14	..
Cellulari	165	178	193	205	244	256	..
Tablet	154	201	212	245	255	216	..
Televisori	54	88	45	24	65	66	..
Fotocamere	56	64	55	52	64	75	..

Figura 19.8 Il risultato di un'operazione di drill-down sul cubo in Figura 19.7.

Ne consegue che mentre un'operazione di roll-up è sempre possibile, il drill-down richiede che siano disponibili dati a un adeguato livello di dettaglio.

In analogia all'operazione di roll-up, il drill-down si può effettuare aggiungendo semplicemente una dimensione a un cubo. Per esempio, un drill-down che aggiunge al cubo in Figura 19.7 la dimensione LUOGO al livello città restituisce un cubo tridimensionale che riporta il totale delle vendite mensili dei prodotti nelle varie città. Anche in questo caso, i dati vanno reperiti esternamente al cubo sul quale si opera.

Per concludere, c'è da dire che generalmente i risultati di analisi multidimensionali vengono visualizzati in opportune vesti grafiche per semplificare la loro interpretazione. Gli strumenti offrono a tale riguardo diverse tipologie di rappresentazione quali linee, istogrammi, torte, bolle, ecc. Un esempio viene riportato in Figura 19.9 nella quale viene presentato il contenuto del cubo bidimensionale di Figura 19.7 sotto forma di un istogramma.

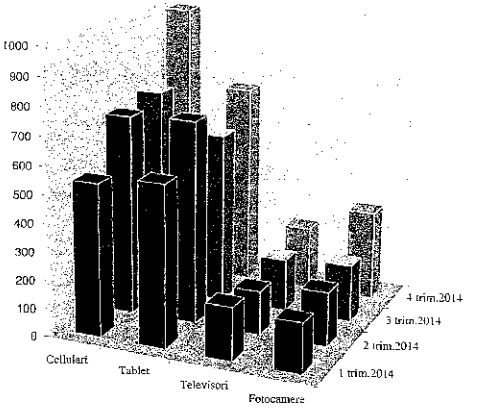

Figura 19.9 La visualizzazione grafica del cubo in Figura 19.7.

19.3 Realizzazione di un data warehouse

Per la realizzazione di un data warehouse si contrappongono due soluzioni alternative, significativamente diverse dal punto di vista tecnologico.

- La prima soluzione consiste nell'uso della tecnologia relazionale, opportunamente adattata ed estesa; i dati vengono memorizzati tramite tabelle e le operazioni di analisi vengono tradotte in opportune istruzioni SQL. Queste istruzioni vengono eseguite efficientemente tramite l'ausilio di strutture fisiche particolari, sviluppate appositamente per queste applicazioni (si veda il Paragrafo 19.3.3). Sistemi di questo tipo si dicono ROLAP (*Relational OLAP*).
- La seconda soluzione, più radicale, memorizza i dati direttamente in forma multidimensionale, tramite speciali strutture dati tipicamente proprietarie. Sistemi di questo tipo si dicono MOLAP (*Multidimensional OLAP*).

La soluzione MOLAP è praticata da piccole ditte informatiche, spesso poi acquisite dalle grandi, che hanno realizzato prodotti specializzati nella gestione di data warehouse. La soluzione ROLAP è praticata dai grandi costruttori relazionali e utilizza tutta l'esperienza tecnologica dei DBMS relazionali. Esistono anche soluzioni ibride, ovvero sistemi basati su tecnologia ROLAP che offrono anche la possibilità di memorizzare in speciali strutture MOLAP alcuni data mart.

In ogni caso, le tecnologie ROLAP e MOLAP utilizzano soluzioni *ad hoc* per l'accesso ai dati, in particolare per quanto concerne l'uso di indici e la materializzazione delle viste. Queste soluzioni tengono conto del fatto che il DW viene usato essenzialmente per operazioni di lettura e di caricamento iniziale dei dati, mentre modifiche e cancellazioni sono assai rare. I DW di grandi dimensioni utilizzano anche il parallelismo, con opportune frammentazioni e allocazioni dei dati, per rendere più efficienti le interrogazioni (si veda il Capitolo 16). Nel seguito ci concentreremo sulla tecnologia ROLAP.

19.3.1 Rappresentazione relazionale di un data warehouse

Come accennato nel Paragrafo 19.1, la costruzione di un data warehouse aziendale è un obiettivo difficile da realizzare ed è prevalente l'approccio di costruire il DW in maniera incrementale, per collezione di data mart settoriali. Ogni data mart può essere rappresentato nel modello presentato nel Paragrafo 19.2.1 mediante un fatto, ovvero, con un cubo multidimensionale come quello di Figura 19.4.

Schema a stella In una realizzazione ROLAP i dati di un fatto multidimensionale sono organizzati secondo una semplice struttura relazionale, detta *schema dimensionale* oppure più semplicemente *schema a stella* o *star schema*. Il primo termine mette in luce la presenza delle dimensioni di analisi, il secondo la struttura "stellare" dello schema. Lo schema a stella ha una struttura molo semplice ed è composta da:

- una relazione principale, detta *tabella dei fatti* o *fact table*, che memorizza le istanze di un fatto;

- varie relazioni ausiliarie chiamate *tabelle dimensione* o *dimension tables* che memorizzano i membri delle dimensioni associate al fatto;
- un insieme di vincoli di integrità referenziale ognuno dei quali collega un attributo della tabella dei fatti a una tabella dimensione.

Disponendo la tabella dei fatti al centro e le tabelle dimensionali "a raggiera" intorno a essa si ottiene una speciale configurazione che dà il nome allo schema. Un esempio di schema a stella viene riportato in Figura 19.10 nel quale viene utilizzata la notazione grafica per schemi relazionali presentata nel Paragrafo 9.3.7. Questo schema a stella corrisponde all'implementazione relazionale del fatto rappresentato dal cubo multidimensionale di Figura 19.4 con l'aggiunta di un'ulteriore dimensione (il cliente della vendita).

Lo schema a stella possiede alcune caratteristiche generali che si possono osservare nella sua struttura. Distinguiamo le caratteristiche della tabella dei fatti da quelle delle tabelle dimensione.

- Tabella dei fatti:
  - ha una chiave composta da attributi che sono riferimenti alle chiavi di tabelle dimensione;
  - gli altri attributi rappresentano le misure del fatto e sono solitamente numerici;
  - soddisfa la forma normale di Boyce-Codd (si veda il Capitolo 9).
- Tabelle dimensione:
  - hanno una chiave semplice (un solo attributo);

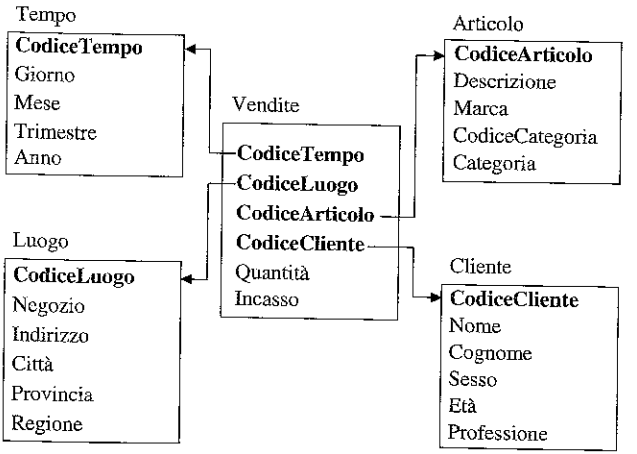

Figura 19.10 Uno schema a stella.

- gli altri attributi rappresentano i livelli della dimensione oppure qualche loro proprietà (per esempio, l'indirizzo di un negozio) e sono tipicamente testuali e descrittivi;
- sono generalmente *denormalizzate* (non normalizzate); per esempio, nella tabella LUOGO l'attributo Regione dipende funzionalmente da Provincia che non è una chiave e quindi lo schema non soddisfa la forma normale di Boyce-Codd.

Le tabelle dimensione si mantengono generalmente denormalizzate per motivi di efficienza. In questa maniera infatti, pur generando una certa ridondanza, si evitano nelle interrogazioni onerose operazioni di join tra tabelle, come vedremo nel prossimo paragrafo.

Un esempio di possibile istanza per lo schema a stella in Figura 19.10 è riportato in Figura 19.11. In questa istanza, oltre alla tabella dei fatti VENDITE, che riporta le vendite al livello di granularità più basso (vendite giornaliere nei negozi per singolo cliente), viene anche riportata la vista VENDITETRIMESTRALI che si ottiene da questa per aggregazione sui clienti e sui giorni di un trimestre. Questa tabella è in grado di alimentare il cubo in Figura 19.4. Si osservi che per poter effettuare aggregazioni corrette su una tabella dei fatti, è in genere necessario associare un codice univoco a tutti i membri delle dimensioni ai vari livelli di aggregazione.

Schema a fiocco di neve Nel caso in cui si decide di normalizzare (sia pur parzialmente) uno schema a stella per ridurre la ridondanza degli schemi dimensionali si ottiene uno schema come quello in Figura 19.12 che, per la sua conformazione,

ARTICOLO

CodA	Prod	Marca	CodCat	Categoria
...	...	...	...	...
P43	KV21	Sony	TEL	Televisore
...	...	...	...	...
P377	NULL	NULL	CEL	Cellulare
...	...	...	...	...

LUOGO

CodL	Negozio	Indirizzo	Città	Prov	Regione
L1	Roma1	Via Po, 3	Roma	RM	Lazio
L2	Roma2	P. Navona, 8	Roma	RM	Lazio
L3	Milano1	P. Duomo, 7	Milano	MI	Lombardia
...	...	...	...	...	...
L100	NULL	NULL	Roma	RM	Lazio
L101	NULL	NULL	Milano	MI	Lombardia
...	...	...	...	...	...
L1000	NULL	NULL	NULL	RM	Lazio
...	...	...	...	...	...

VENDITE

CodA	CodT	CodL	CodC	Quantità	Incasso
...	...	...	...	...	...
P43	T99	L3	C79	1	2K
...	...	...	...	...	...

VENDITETRIMESTRALI

CodP	CodT	CodL	Quantità
...	...	...	...
P377	T504	L2	219
...	...	...	...

TEMPO

CodT	Giorno	Mese	Trimestre	Anno
...	...	...	...	...
T99	1 Apr 14	Apr-14	2 trim. 14	2014
...	...	...	...	...
T504	NULL	NULL	2 trim. 14	2014
...	...	...	...	...

CLIENTE

CodC	Nome	Cognome	Sesso	Età	Professione
...	...	...	...	...	...
C79	Maria	Neri	F	30	Impiegata
...	...	...	...	...	...

Figura 19.11 Una possibile istanza per lo schema in Figura 19.10.

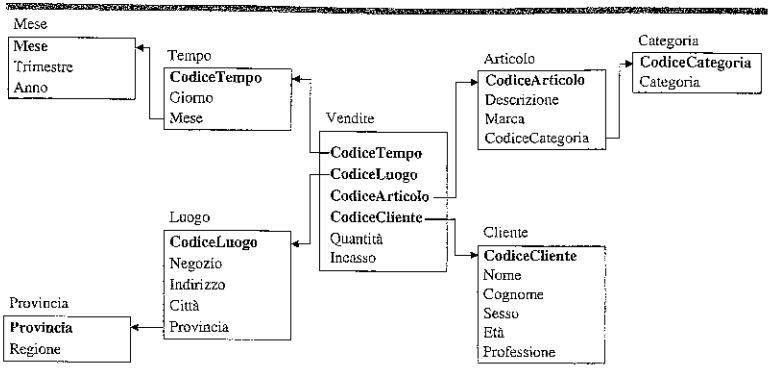

Figura 19.12 Uno schema a fiocco di neve.

viene detto *schema a fiocco di neve* o *snowflake schema*. C'è però da dire a questo proposito che è in genere sconsigliato procedere a normalizzazioni troppo spinte degli schemi a stella perché generalmente il beneficio che si ottiene in termini di riduzione di spazio non compensa il degrado delle prestazioni che le operazioni di join necessarie per ricostruire le dimensioni possono generare. C'è infatti da tenere presente che una tabella dei fatti è spesso molto più grande di una tabella dimensione. Ne consegue che anche riducendo fortemente l'occupazione di memoria di una tabella dimensione, l'occupazione di memoria complessiva si riduce, in percentuale, piuttosto limitatamente. Un altro motivo è dovuto al fatto che gli aggiornamenti sulle dimensioni sono piuttosto rari. Non si deve perciò affrontare, in molti casi, il problema delle anomalie di aggiornamento delle relazioni non normalizzate, descritto nel Capitolo 9.

Dimensioni che cambiano lentamente Il problema dell'aggiornamento delle dimensioni di uno schema dimensionale merita un approfondimento. In genere, in uno schema dimensionale ogni dimensione dovrebbe essere logicamente indipendente da tutte le altre dimensioni in quanto ognuna di esse descrive un punto di vista sostanzialmente differente dalle altre sui fatti. In realtà, molte dimensioni dipendono dal tempo, non solo perché l'insieme dei membri della dimensione cambia nel tempo ma anche perché possono cambiare le descrizioni dei membri. Per esempio, può cambiare la descrizione di un prodotto o un dato demografico di un cliente. Molte dimensioni soggette a cambiamenti sono in realtà quasi costanti nel tempo e possono essere considerate sostanzialmente indipendenti dalla dimensione tempo. Per poterle gestire, oltre allo stato "corrente" della dimensione, è necessario rappresentare in qualche maniera i cambiamenti nel tempo. Le dimensioni "quasi costanti" sono chiamate dimensioni che *cambiano lentamente* (*slowly changing dimensions*).

Si consideri il seguente esempio: la cliente Anna Sili non è stata sposata fino al 15 maggio 2014. Questa informazione è descritta dall'attributo StatoCivile della dimensione CLIENTE. Anna Sili si sposa il 15 maggio 2014. Come può essere gestito

questo cambiamento? Sono possibili tre scelte: (1) sovrascrivere il valore precedente (nel nostro esempio, viene modificato il valore di StatoCivile della riga relativa a Anna Sili); (2) aggiungere una nuova riga nella tabella dimensione con i nuovi valori per gli attributi cambiati (due righe per Anna Sili: una con il valore corrente di StatoCivile e una con il valore passato); (3) aggiungere un attributo alla dimensione (o più se si vogliono memorizzare diversi stati passati) per memorizzare il valore precedente (nel nostro caso, potremmo avere due attributi: StatoCivileCorrente e StatoCivilePassato).

La prima soluzione è la modalità più semplice ma non tiene traccia della storia passata dei membri della dimensione. Infatti, dopo il 15 maggio 2014, risulterà che Anna Sili è sposata "da sempre". Questa modalità di gestione è comunque utile nella correzione degli errori; per esempio, va bene se solo il 15 maggio 2014 si scopre che Anna Sili è sposata. La seconda soluzione consente di gestire "versioni di oggetti" in quanto la tabella dimensione non contiene più una riga per ciascun membro della dimensione ma una riga per ciascuna "versione" di membro della dimensione. Se nel nostro esempio ogni fatto successivo al 15 maggio 2014 verrà correttamente associato alla versione di Anna Sili in cui il valore di StatoCivile è "sposata", nelle interrogazioni non sarà necessario specificare la versione alla quale si fa riferimento e le analisi saranno corrette. L'associazione può essere gestita mediante una chiave "generalizzata" della dimensione con valore univoco per ogni versione di un membro. Bisogna poi però tenere traccia della relazione tra chiave "di base" e chiavi generalizzate dei membri soggetti a versione. Nel nostro esempio, va memorizzato (tipicamente tra i metadati) il legame tra la chiave del membro "Anna Sili" e le chiavi generalizzate che identificano le due versioni di "Anna Sili". La terza soluzione richiede più attributi per le varie versioni ed è la modalità di gestione più complessa da realizzare in quanto in ogni singola interrogazione bisognerà specificare la versione dei membri alla quale ci si vuole riferire. Viene per questo usata solo in casi specifici e viene spesso preferita la modalità di gestione di tipo 2.

Minidimensioni, dimensioni degeneri e fatti senza misure Chiudiamo la discussione sulla rappresentazione relazionale di cubi dimensionali con delle considerazioni su alcuni casi particolari.

È spesso necessario dover gestire dimensioni con moltissimi attributi. Per esempio la dimensione CLIENTE può essere caratterizzata da tanti attributi (nell'ordine delle centinaia) e da diverse gerarchie. Per semplificare la gestione di queste dimensioni, è spesso utile memorizzare separatamente, in tabella distinta, un gruppo di attributi correlati della dimensione in oggetto. Per esempio, potremmo definire una tabella per gli attributi demografici del cliente (come sesso, età, stato civile) con chiave CodiceDemografico e lasciare in una tabella dimensionale "principale" con chiave CodiceCliente gli altri attributi di interesse per il cliente (come cap, città e regione di residenza). La nuova tabella è generalmente chiamata *minidimensione* e viene solitamente referenziata sia dalla tabella fatti sia dalla tabella dimensione. Si osservi che, nel nostro caso, una riga della minidimensione demografica non è più associata a un solo cliente, ma può descrivere possibili combinazioni significative degli attributi demografici. L'introduzione di una minidimensione può avere un effetto positivo nella gestione dei cambiamenti della dimensione stessa. Per esempio, i cambiamenti di un

cliente di cui si vuole tenere traccia avvengono solitamente sui suoi dati demografici. In questo caso, la dimensione cliente può essere gestita cambiando semplicemente il valore dell'attributo CodiceDemografico nella tabella dimensionale principale Cliente (modalità di tipo 1) ottenendo benefici simili a quelli delle altre modalità: dal cliente sarà possibile accedere alle informazioni demografiche correnti, mentre dalla tabella fatti sarà possibile accedere sia (direttamente) alle informazioni demografiche del cliente al momento della registrazione del fatto sia (indirettamente) alle informazioni demografiche correnti del cliente.

Possono inoltre esistere dimensioni che non richiedono l'esistenza di una tabella dimensionale. Si pensi per esempio a una tabella fatti che memorizza i singoli acquisti presenti negli scontrini di un supermercato (in pratica, le sue righe). È assolutamente sensato effettuare aggregazioni per scontrino, ma la dimensione scontrino non ha attributi significativi. Si parla in questo caso di dimensione *degenere*: la loro utilità si limita al poter raggruppare direttamente i fatti su questa dimensione. Nel caso specifico, avremmo un attributo CodiceScontrino nella tabella fatti senza avere una tabella dimensione SCONTRINO.

Possono infine esistere tabelle fatto *senza misure*. Si consideri l'analisi di eventi realizzata con una tabella fatti nella quale ogni riga rappresenta la singola occorrenza di un evento, per esempio la partecipazione di un singolo studente di un'università a una specifica lezione tenuta in un'aula in una certa ora di un certo giorno. Se si vogliono conoscere i corsi più frequentati, le aule più utilizzate o l'occupazione media delle aule in funzione dell'ora è sufficiente contare il numero di righe distinte della tabella fatti rispetto a un certo raggruppamento. In questi casi è utile introdurre una misura numerica fittizia a cui viene assegnato, in ciascuna riga, il valore 1. Nel nostro caso introdurremmo la misura Presenza. Le interrogazioni verrano poi realizzate sommando questa nuova misura e risulteranno certamente più comprensibili.

19.3.2 Operazioni su data mart relazionali

Come tutti i data warehouse server, i sistemi ROLAP offrono strumenti di analisi evoluti che presentano un data mart all'utente finale (l'analista) secondo un modello di rappresentazione di alto livello come quello presentato nel Paragrafo 19.2.1. Questi strumenti trasformano poi le interrogazioni multidimensionali descritte nel Paragrafo 19.2.2 in istruzioni SQL sullo schema a stella del data mart in questione.

In particolare, l'operazione multidimensionale più interessante, quella di roll-up, si traduce in un'interrogazione SQL con una struttura semplice e regolare:

```
select D1.L1,.., Dn.Ln, Aggr1(F.M1),.., Aggrk(F.Mk)
from Fatti as F,
     Dimensione1 as D1, ..., DimensioneN as Dn
where join-predicate(F,D1) and ...
    and join-predicate(F,Dn)
    and selection-predicate(F)
group by D1.L1, ..., Dn.Ln
order by D1.L1, ..., Dn.Ln
```

In questa interrogazione, FATTI è il nome della tabella dei fatti, `Mj` è il nome della j-esima misura della tabella FATTI, `Di` è il nome della i-esima tabella dimensione, `Li` è il nome del livello della i-esima tabella dimensione rispetto al quale si vuole fare il roll-up, `Aggrx` indica una funzione aggregativa (per esempio `sum`), `join-predicate(F,Di)` indica la condizione di join che lega la tabella dei fatti e la i-esima tabella dimensione, infine `selection-predicate(F)` indica una eventuale condizione di selezione sulle tabelle dimensione.

Per esempio, la seguente interrogazione SQL restituisce il risultato di un roll-up che, applicato alla tabella dei fatti VENDITE, elimina le dimensioni LUOGO e CLIENTE e restituisce le vendite complessive nel 2014 per categoria di articolo e trimestre.

```
select A.Categoria, T.trimestre, sum(V.Quantita)
from Vendite as V, Articolo as A, Tempo as T
where V.CodiceArticolo = A.CodiceArticolo and
      V.CodiceTempo = T.CodiceTempo and T.Anno = 2014
group by A.Categoria, T.trimestre
order by A.Categoria, T.trimestre
```

Si osservi che la stessa interrogazione fatta sullo schema normalizzato in Figura 19.12 richiede due join in più (con le tabelle CATEGORIA e MESE) con un possibile degrado delle prestazioni.

La frequenza dell'uso delle aggregazioni nel contesto di operazioni OLAP ha suggerito l'introduzione nello standard SQL di un operatore molto potente, detto *data cube*, che effettua *tutte* le possibili aggregazioni su una tabella basate sugli attributi di raggruppamento specificati. Descriviamo l'operatore tramite un esempio.

```
select Citta, Categoria,
       count(Quantita) as VenditeCC
from  Vendite as V, Articolo as A, Luogo as L
where V.CodiceArticolo = A.CodiceArticolo and
      V.CodiceLuogo = L.CodiceLuogo
group by cube(Citta, Categoria)
```

Un possibile risultato di questa istruzione è riportato in Figura 19.13. Si osservi che sono state calcolate aggregazioni su tutte le possibili combinazioni di attributi presenti clausola `group by cube`. Si osservi inoltre che, per rappresentare l'aggregazione, viene utilizzato il valore polimorfo ALL, che (come NULL) è presente in tutti i domini e corrisponde all'insieme di tutti i possibili valori presenti nel dominio.

La complessità della valutazione del data cube cresce in modo combinatorio con il crescere del numero degli attributi di raggruppamento. Per questo motivo è stata introdotta una diversa estensione di SQL in cui le aggregazioni sono progressive rispetto all'ordine degli attributi di raggruppamento. In questo modo le aggregazioni da calcolare crescono solo linearmente con il crescere di tali attributi. Tale estensione richiede la clausola `rollup`, che sostituisce la clausola `cube`, come illustrato dal seguente esempio.

Città	Categoria	VenditeCC
Roma	Televisori	361
Roma	Cellulari	1280
Roma	Tablet	1187
Roma	Fotocamere	458
Milano	Televisori	393
Milano	Cellulari	1685
Milano	Tablet	1419
Milano	Fotocamere	429
Roma	ALL	3286
Milano	ALL	3926
ALL	Televisori	754
ALL	Cellulari	2965
ALL	Tablet	2606
ALL	Fotocamere	887
ALL	ALL	7212

Figura 19.13 Risultato di un'operazione di cube.

```
select Citta, Categoria,
       count(Quantita) as VenditeCC
from  Vendite as V, Articolo as A, Luogo as L
where V.CodiceArticolo = A.CodiceArticolo and
      V.CodiceLuogo = L.CodiceLuogo
group by rollup(Citta, Categoria)
```

Il risultato di questa istruzione è riportato in Figura 19.14. Si osservi che in questo caso le vendite per città non vengono calcolate.

Le clausole `cube` e `rollup` sono presenti in molti DBMS commerciali anche se in forme diverse da quanto suggerito nello standard. Questo significa che con i sistemi relazionali si possono effettuare semplici operazioni di OLAP anche se non si dispone di strumenti dedicati all'analisi dei dati. Del resto, un'interpretazione secondo lo schema a stella è sempre possibile: basta considerare gli attributi nella clausola `group by` come livelli di dimensioni e i rimanenti attributi della clausola `select` aggregazioni di misure.

19.3.3 Indici bitmap e indici di join

Gli *indici bitmap* consentono una realizzazione efficiente di congiunzioni o disgiunzioni di predicati di selezione, oppure operazioni insiemistiche di unione e intersezione. Essi si basano sull'idea di rappresentare ciascuna tupla di una tabella come un elemento di un vettore di bit. Quindi la lunghezza del vettore coincide con la cardinalità della tabella. Mentre la radice e i nodi intermedi di un indice bitmap sono simili

Città	Categoria	VenditeCC
Roma	Televisori	361
Roma	Cellulari	1280
Roma	Tablet	1187
Roma	Fotocamere	458
Milano	Televisori	393
Milano	Cellulari	1685
Milano	Tablet	1419
Milano	Fotocamere	429
ALL	Televisori	754
ALL	Cellulari	2965
ALL	Tablet	2606
ALL	Fotocamere	887
ALL	ALL	7212

Figura 19.14 Risultato di un'operazione di roll-up.

a indici tradizionali (per esempio, alberi B o B+ descritti nel Capitolo 11), le foglie degli indici contengono, per ciascun valore dell'indice, un vettore di bit i cui elementi sono posti a `1` in corrispondenza delle tuple che contengono quel valore e a `0` per le altre.

Supponiamo per esempio di disporre di un indice bitmap sugli attributi Marca e Categoria della tabella PRODOTTO nello schema in Figura 19.10. Per individuare le tuple corrispondenti al predicato `Marca = 'Sony' and Categoria = 'Televisore'` basta accedere separatamente, tramite gli indici, ai due vettori corrispondenti alle costanti 'Sony' e 'Televisore', estrarli e farne l'*and* bit a bit. Il vettore risultante conterrà un `1` in corrispondenza delle tuple che soddisfano la condizione, che vengono così individuate. Operazioni analoghe sui bit consentono di gestire le disgiunzioni. Ovviamente, un indice bitmap è difficile da gestire se la tabella subisce modifiche frequenti, ma è conveniente in un ambiente come quello OLAP nel quale, come abbiamo già detto, gli aggiornamenti sono piuttosto rari e i valori possibili per i membri delle dimensioni (sui quali si definiscono predicati di selezione) sono in numero limitato.

Gli *indici di join* consentono invece una realizzazione efficiente delle operazioni di join fra le tabelle dimensione e la tabella dei fatti. Tali operazioni di join, come evidenziato dalla struttura dell'interrogazione descritta nel Paragrafo 19.3.2, estraggono tuple della tabella dei fatti che soddisfano condizioni poste sulle tabelle dimensione. Gli indici di join vengono costruiti sulle chiavi delle tabelle dimensione. Essi contengono nelle foglie, al posto dei puntatori alle tuple delle dimensioni, puntatori agli insiemi di tuple delle tabelle dei fatti che contengono quei valori di chiave.

Sempre facendo riferimento al data mart descritto nel Paragrafo 19.3.1, un indice di join sull'attributo CodiceArticolo conterrà perciò nelle sue foglie riferimenti alle tuple dei fatti corrispondenti a un articolo. Una volta estratta una categoria di prodotto

corrispondente alla condizione di selezione (per esempio tramite un indice bitmap), l'indice di join consente di estrarre tutte le vendite di articoli appartenenti a quella categoria. È anche possibile costruire indici di join su insiemi di chiavi di dimensioni diverse, per esempio su CodiceArticolo e CodiceLuogo.

Come sempre nel caso della progettazione fisica (si veda il Paragrafo 1.6), l'utilizzo di indici bitmap e di join è soggetto a un'analisi costi-benefici, ove i costi sono essenzialmente dovuti alla necessità di costruire e memorizzare persistentemente gli indici, e i benefici sono legati al loro uso effettivo da parte del DW server nel risolvere le interrogazioni e al fatto che non ci sono costi dovuti ad aggiornamenti continui.

19.3.4 Materializzazione delle viste

Le interrogazioni di un data warehouse richiedono ripetutamente aggregazioni e sintesi assai laboriose. In tal caso può essere conveniente valutare viste che esprimano i dati aggregati una volta per tutte, e memorizzarle. Questa tecnica, introdotta nel Capitolo 3, prende il nome di *materializzazione delle viste*. Per esempio, nel data mart relativo alla gestione delle vendite, una vista materializzata potrebbe contenere i dati di vendita aggregati per prodotto, oppure le vendite mensili di ciascun negozio. Tutte le interrogazioni interessate a queste aggregazioni, o che possono essere ottenute da esse per ulteriori aggregazioni, verrebbero eseguite direttamente sulle viste, invece che sul data mart originario.

La scelta delle viste da materializzare è un problema abbastanza complesso, che richiede la conoscenza delle tipiche interrogazioni usate in un data mart e della loro frequenza di esecuzione. In genere, una vista è conveniente quando può sensibilmente ridurre il tempo di esecuzione di una o più interrogazioni svolte frequentemente. Gli strumenti offrono in genere un supporto nella scelta.

Come osservato nel Capitolo 3, ciascuna vista dipende da un insieme di tabelle di base. Quando le tabelle di base subiscono modifiche, è necessario aggiornare le viste, propagando gli effetti delle modifiche sulle tabelle di base (come osservato nel Capitolo 5, questo compito è una tipica applicazione dei trigger). Ne consegue che la materializzazione è assai conveniente in un ambiente, quale quello dei sistemi OLAP, nel quale le tabelle di base non subiscono modifiche frequenti.

19.4 Progettazione di un data warehouse

Negli ultimi anni, si è largamente affermata una metodologia di sviluppo di data warehouse proposta da Kimball basata sulla *progettazione dimensionale* che ha dimostrato di essere particolarmente efficace nella realizzazione di DW di buona qualità. Questa metodologia copre l'intero ciclo di vita di un data warehouse, prevedendo attività iniziali dedicate alla pianificazione e alla raccolta dei requisiti, una fase centrale dedicata alla progettazione e alcune attività finali dedicate alla validazione, alla manutenzione e allo sviluppo incrementale. In questo paragrafo illustreremo brevemente questa metodologia concentrando l'attenzione sulla fase cruciale di progettazione dei dati.

19.4.1 Una metodologia generale

Un inquadramento metodologico generale per la costruzione e l'uso di data warehouse è illustrato in Figura 19.15 e si articola nelle seguenti fasi principali.

- **Pianificazione**. Serve a definire, in maniera per quanto possibile precisa, gli obiettivi, i costi e i possibili benefici della realizzazione di un DW. In base a queste informazioni, si stabilisce la fattibilità del progetto e vengono pianificate le attività e le risorse necessarie alla realizzazione e all'uso del prodotto finale.
- **Raccolta e analisi dei requisiti**. Come avviene nel tradizionale ciclo di vita dei sistemi informativi, in questa fase vengono definiti i requisiti di analisi che il data warehouse dovrà soddisfare sulla base delle sorgenti di dati a disposizione. Costituisce anche il momento in cui vengono definiti i requisiti software e hardware del sistema.
- **Progettazione**. Questa fase richiede tipicamente tre attività che, sebbene possano in parte essere svolte in parallelo, richiedono un forte coordinamento a causa delle forti dipendenze che esistono tra di esse:
  - la progettazione tecnologica, dedicata alla definizione dell'architettura complessiva e dei prodotti che verranno utilizzati per realizzare il DW e implementare i processi di estrazione, caricamento e pulizia dei dati;
  - la progettazione dei dati, che consiste nella definizione della struttura del DW e delle relative procedure di alimentazione;
  - la progettazione delle applicazioni, che si occupa della definizione dei meccanismi di accesso ai dati mediante strumenti di interrogazione diretta e/o di produzione di report.
- **Validazione e avviamento**. Consiste nella realizzazione del DW secondo la struttura e le caratteristiche definite nella fase di progettazione e nella verifica del corretto funzionamento del sistema. In questa fase il DW diventa operativo e possono essere eseguite le analisi per i quali il DW era stato originariamente progettato.

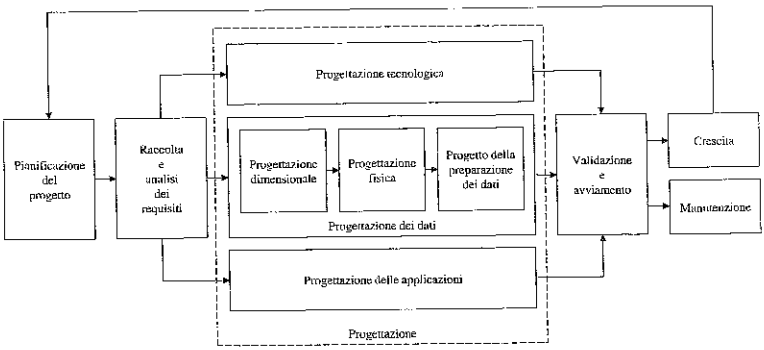

Figura 19.15 Una metodologia di sviluppo di data warehouse.

- **Manutenzione e crescita.** Il DW viene manutenuto effettuando periodicamente le operazioni necessarie alla sua alimentazione con nuovi dati provenienti dalle sorgenti. Si procede inoltre a un eventuale arricchimento del DW con nuovi cubi dimensionali costruiti per soddisfare nuove necessità di analisi dei dati.

Come avviene in tutti i processi di sviluppo del software, è anche presente un'attività di gestione, non mostrata in Figura 19.15, che si svolge nell'intero ciclo di vita del DW ed è finalizzata a monitorare lo sviluppo del progetto, curare la comunicazione tra i partecipanti e verificare il rispetto delle tempistiche e il raggiungimento degli obiettivi prefissati.

Con riferimento a quanto discusso nel Paragrafo 19.1 sulle architetture, va ricordato che la realizzazione di un data warehouse si fonda tipicamente sullo sviluppo coordinato di alcuni data mart, ciascuno dei quali è costruito per una determinata attività di analisi. Questo obiettivo si raggiunge mediante un processo iterativo di sviluppo nel quale le varie fasi delineate in Figura 19.15 vengono ripetute più volte, con un'iterazione per ciascuno dei vari data mart che costituiscono il data warehouse finale. Naturalmente, ogni iterazione utilizza i risultati delle iterazioni precedenti e questo consente lo sviluppo coordinato dei vari data mart, indispensabile per garantire l'uniformità delle dimensioni comuni. Procedendo in questa maniera, il data warehouse viene realizzato attraverso il classico modello di sviluppo incrementale dei prodotti software. In quest'ottica, nel resto di questo paragrafo discuteremo con maggior dettaglio le fasi di analisi dei requisiti e di progettazione dei dati assumendo che l'obiettivo di ogni iterazione consiste nella realizzazione di un nuovo data mart.

19.4.2 Raccolta e analisi dei requisiti

La raccolta dei requisiti avviene tipicamente attraverso interviste e produce una descrizione (solitamente in linguaggio naturale o semi-strutturato) delle esigenze aziendali di analisi. Viene inoltre censito il patrimonio informativo aziendale. Questa attività produce una collezione di schemi di tutte le basi di dati che possono alimentare il DW e la relativa documentazione di supporto (per esempio, il glossario aziendale dei termini). Inoltre, spesso accade che l'analisi a supporto delle decisioni richieda la correlazione dei dati aziendali con altri non di proprietà dell'azienda, ma comunque accessibili da essa (per esempio, statistiche fornite dall'ISTAT, dati sull'andamento della borsa); è allora necessaria una descrizione anche degli schemi di tali sorgenti.

La fase di analisi dei requisiti è basata sulla correlazione tra i requisiti e le sorgenti informative disponibili. Si tratta di un'attività strategica, con benefici che vanno al di là dalla mera realizzazione del DW, perché si tratta di un'occasione di analisi del patrimonio informativo aziendale che può evidenziare anomalie e criticità. L'analisi inizia con la selezione delle sorgenti informative: alcune risulteranno irrilevanti e verranno per questo motivo trascurate. Dopo la selezione, è opportuno rappresentare le sorgenti selezionate secondo un medesimo modello dei dati, per favorire il confronto dei contenuti e la loro integrazione. In particolare, risulta utile far uso di un modello astratto, indipendente dall'implementazione, in quanto il confronto degli schemi risulta più semplice ed efficace. Nel caso di sorgenti legacy, questo può richiedere una complessa attività di *reverse-engineering*, ma d'altro canto esistono oggi diversi

strumenti che offrono un grosso supporto a questo processo. Successivamente, è utile avviare un'attività preliminare all'integrazione di schemi che ha lo scopo di individuare concetti irrilevanti (che possono essere rimossi dai rispettivi schemi) oppure concetti che risultano essere rappresentati in più schemi. Nel secondo caso, è possibile assegnare delle preferenze; per esempio, sarà preferibile una sorgente nella quale un concetto di interesse è gestito in modo più accurato e aggiornato. Il prodotto finale è costituito da uno schema per ciascuna sorgente di dati di interesse secondo un medesimo modello di rappresentazione, corredato dalla opportuna documentazione di supporto.

19.4.3 Progettazione dei dati

Come illustrato in Figura 19.15, la progettazione dei dati si articola in tre fasi principali:

- la progettazione dimensionale, dedicata alla definizione di fatti, dimensioni e misure dei vari data mart che costituiscono il DW, secondo un modello dimensionale logico, ancora indipendente dalle implementazioni;
- la progettazione fisica, che consiste nella definizione delle modalità con cui i vari data mart definiti nel passo precedente vengono realizzati e acceduti;
- il progetto della preparazione dei dati, che si occupa della definizione dei meccanismi di estrazione, caricamento e pulizia dei dati necessari all'alimentazione del DW.

Approfondiamo, nell'ordine, ciascuna di queste fasi.

Progettazione dimensionale La fase di progettazione dimensionale ha come obiettivo la definizione dello schema logico del data warehouse e costituisce il momento cruciale dell'intero processo. Come abbiamo detto, il DW finale sarà costituito da un insieme coerente di data mart, ognuno dei quali è la restrizione del DW a un singolo processo dell'organizzazione. Ricordiamo che un insieme di data mart è coerente se le loro dimensioni sono conformi, se hanno cioè un significato uniforme in tutto il DW. Secondo gli esperti, un tipico DW ha da 10 a 30 data mart.

La progettazione dimensionale inizia quindi con la selezione e l'elencazione dei data mart che costituiranno il DW. Si inizia poi a realizzare un data mart significativo che permette analisi interessanti ed è semplice da realizzare, per esempio perché ha un'unica sorgente. Successivamente, possono essere realizzati altri data mart, più complessi, per esempio a sorgente multipla. Nel seguito faremo riferimento alla rappresentazione relazionale di un data mart secondo lo schema "a stella" illustrato nel Paragrafo 19.3.1 con una singola tabella dei fatti e un insieme di tabelle dimensione. Questo schema verrà chiamato nel seguito semplicemente *schema dimensionale*.

La progettazione di ciascun data mart richiede lo svolgimento (in sequenza o quasi) dei seguenti quattro passi:

- scelta del processo da modellare (il "fatto" nella terminologia introdotta nel Paragrafo 19.2.1), supportato da uno o più basi di dati operazionali, i cui dati possono essere utilizzati per popolare lo schema dimensionale;

- scelta della grana del processo, ovvero del livello di dettaglio atomico che deve essere rappresentato nella tabella fatti per il processo; livelli tipici per la grana sono le transazioni individuali, l'istantanea (detta anche *snapshot*) giornaliera, l'istantanea mensile; in un data mart è spesso necessario memorizzare dati a una grana sufficientemente piccola, per permettere alle interrogazioni di selezionare e raggruppare i dati in modo sufficientemente preciso e mirato;
- scelta delle dimensioni da cui dipende ciascun record della tabella fatti con l'elenco di tutti i suoi attributi (solitamente testuali, discreti e descrittivi) necessari nelle selezioni e nelle aggregazioni (cioè i livelli della dimensione); si distingue in genere tra dimensioni "primarie", ovvero indipendenti dalle altre, e dimensioni "supplementari", nel senso che dipendono funzionalmente dalle dimensioni primarie (fissati i valori per le dimensioni primarie è univoca la scelta dei valori per quelle supplementari); se una dimensione supplementare non è conforme alla grana della tabella fatti perché richiede maggior dettaglio nei dati, allora la scelta della grana deve essere corretta;
- scelta delle grandezze di interesse (solitamente numeriche, continue e additive) del processo selezionato, ovvero delle misure da memorizzare in ogni record della tabella fatti.

Tutte queste scelte sono guidate dai requisiti e dalle sorgenti disponibili.

Durante la scelta delle dimensioni, bisogna garantire l'uniformità delle dimensioni dei vari data mart. In pratica, se due data mart hanno una dimensione in comune, si deve fare in modo che la dimensione sia la stessa. In effetti dimensioni molto usate (per esempio quella temporale) diventano spesso degli standard aziendali. Per garantire l'uniformità delle dimensioni, risulta molto utile correlare data mart e dimensioni con una matrice che descrive l'architettura a bus del data warehouse. Ciascuna riga di questa matrice rappresenta un data mart, ciascuna colonna rappresenta una dimensione e ciascun elemento, all'intersezione di un data mart e una dimensione, viene marcato se la dimensione è di interesse per il data mart. Per esempio, la Figura 19.16 mostra una possibile matrice del bus di una DW per una catena di negozi. La definizione della matrice che descrive l'architettura a bus del data warehouse viene considerata una"pietra miliare" del processo di progettazione dell'intero data warehouse in quanto costituisce il luogo dove viene fissato l'insieme delle dimensioni conformi.

I nomi degli attributi delle dimensioni devono essere il più possibile descrittivi e non ambigui. Per esempio, nella matrice in Figura 19.16 potrebbero essere presenti

	Tempo	Fornitore	Prodotto	Cliente	Promozione	Negozio	Magazzino
Vendite	×		×	×	×	×	
Inventario	×	×	×				×
Ordini	×		×	×	×		
Acquisti	×	×	×			×	
Distribuzione	×	×	×			×	×

Figura 19.16 Matrice del bus di un data warehouse per una catena di negozi.

più dimensioni geografiche che fanno riferimento al luogo del negozio, del magazzino e all'indirizzo del cliente. In questo caso è opportuno diversificare i nomi. Per esempio, l'attributo relativo alla città va distinto, nelle tre dimensioni, in CittàNegozio, CittàMagazzino e CittàCliente. Inoltre, per ciascuna dimensione, è opportuno definire la strategia di gestione dei cambiamenti lenti e le eventuali gerarchie di aggregazione significative per l'utente. Infine, per ciascun fatto, va definita la durata storica (quanti dati storici devono essere rappresentati?), va pianificato il caricamento incrementale (con che periodicità deve essere aggiornato il data mart? Con che urgenza?) ed è utile specificare, per ciascuna misura, la modalità di aggregazione di default (somma, minimo, massimo, ultimo valore).

Progettazione fisica A questo punto, possiamo procedere con la realizzazione dei data mart individuati nella fase di progettazione dimensionale. Qui bisogna ragionare in termini del sistema che si decide di adottare, ovvero, come precisato nel Paragrafo 19.3, uno strumento MOLAP o ROLAP.

Nel primo caso è difficile fare considerazioni generali perché i sistemi sono piuttosto diversi gli uni dagli altri e offrono soluzioni proprietarie ma è in genere facile specificare lo schema del DW partendo dallo schema dimensionale ottenuto.

Nel caso di scelta relazionale invece va fatta una scelta sul livello di normalizzazione che si vuole avere nel DW. Nel caso di schema completamente normalizzato si può procedere con le tecniche di normalizzazione classiche descritte nel Capitolo 9, ottenendo così uno schema a fiocco di neve. Nel caso invece di schema completamente denormalizzato è sufficiente definire, oltre a una tabella per l'entità fatto, una sola tabella per ogni dimensione. Otteniamo in questa maniera la configurazione a stella classica. A tale riguardo, va detto che la soluzione denormalizzata è solitamente preferibile perché riduce il costo delle operazioni di join necessario a correlare la tabella fatti con le tabelle dimensioni, al prezzo di un aumento di occupazione di memoria (dovuto alle ridondanze sicuramente presenti) piuttosto contenuto. Naturalmente sono possibile soluzioni intermedie tra i due estremi nelle quali le tabelle sono solo parzialmente normalizzate.

Sullo schema ottenuto si può decidere di definire delle aggregazioni se non si desidera avere uno schema finale a un livello di dettaglio troppo spinto. Inoltre, eventuali viste ottenibili per aggregazione di questo schema possono essere materializzate, secondo le modalità descritte nel Paragrafo 19.3.4.

Questa fase si completa con la definizione delle strutture fisiche di accesso. Nel caso di implementazione relazionale, si può procedere come descritto nella progettazione fisica di basi di dati illustrata nel Capitolo 11, tenendo conto che esistono in questo contesto speciali tecniche di ottimizzazione, tipicamente basate su indicizzazioni *ad hoc* (si vedano gli indici di bitmap, descritti nel Paragrafo 19.3.3) e meccanismi di materializzazione di viste (Paragrafo 19.3.4).

Progetto della preparazione dei dati Questa fase ha come obiettivo la definizione delle procedure che consentono di alimentare il data warehouse progettato nelle fasi precedenti con le sorgenti di dati a disposizione. L'attività principale consiste nella specifica dei processi di estrazione, trasformazione ed eventualmente di

integrazione che, applicati a dati nelle sorgenti, producono il contenuto del DW. Il processo è in genere descritto mediante una rappresentazione di alto livello, tipicamente grafica, con nodi che rappresentano: (a) le sorgenti dei dati (per esempio un file o una tabella di una base di dati operazionale); (b) le destinazioni; (c) le varie operazioni da applicare ai dati (estrazione, trasformazione, pulizia e integrazione). Gli archi del grafo rappresentano invece il percorso dei dati dalle sorgenti al DW. Un esempio concreto è mostrato più avanti, in Figura 19.18.

Il problema più critico in questa fase è quello dell'integrazione di dati provenienti da diverse basi di dati nel caso di un data mart multi-sorgente. In effetti, si tratta di un problema che ha molte caratteristiche in comune con quello dell'integrazione di basi di dati a fini transazionali, tranne che per le finalità, decisamente rivolte in questo caso alla gestione di dati per l'analisi. L'obiettivo principale dell'integrazione è l'identificazione di tutte le porzioni delle diverse sorgenti informative che si riferiscono a uno stesso attributo dello schema dimensionale di un data mart del DW e all'*unificazione* della loro rappresentazione. L'approccio è orientato alla risoluzione di conflitti tra rappresentazioni diverse degli stessi concetti nei vari schemi. Tali conflitti possono essere terminologici, strutturali e di codifica. I primi corrispondono all'uso di termini diversi per indicare lo stesso concetto (per esempio, "impiegato" in una base di dati e "dipendente" in un'altra). I secondi alla rappresentazione dello stesso concetto in forme diverse (per esempio, l'uso di un singolo attributo per rappresentare un'informazione che è rappresentata da più attributi in un'altro schema). Gli ultimi infine corrispondono all'uso di criteri diversi per codificare la medesima informazione (per esempio, le lettere M e F o le cifre 0 e 1 per rappresentare il sesso). L'integrazione può essere utilmente guidata dai requisiti del DW, nonché dai criteri di priorità identificati nella fase di analisi dei requisiti. Per esempio, la risoluzione di eventuali conflitti strutturali può essere guidata dal criterio che privilegia le sorgenti nelle quali la rappresentazione dei dati è più accurata. Sulla base delle ristrutturazioni effettuate sugli schemi per ottenere l'integrazione, vanno inoltre definite le procedure necessarie per l'integrazione dei dati.

19.4.4 Un esempio di progettazione

In questo paragrafo illustriamo, con un esempio pratico nel settore commerciale, le fasi centrali della progettazione di un DW.

Raccolta e analisi dei requisiti Supponiamo di lavorare nella direzione di una grande catena di negozi alimentari chiamata A4Y (All 4 You) che comprende 460 grandi negozi, distribuiti in diverse regioni italiane. Ogni negozio di A4Y è un supermercato che dispone di diversi reparti (pane, carne, frutta e verdura ecc.) e ha circa 60 000 prodotti individuali nei suoi scaffali chiamati unità di vendita (o SKU, *Stock Keeping Unit*); per esempio, una SKU è la lattina di birra di una certa marca. Ogni variante di confezionamento dei prodotti costituisce una diversa SKU; per esempio, la confezione da sei lattine di birre è una SKU diversa dalla singola lattina della stessa birra. Circa 40 000 delle SKU vengono da fornitori esterni e su di esse è stampato un codice a barre chiamato codice universale del prodotto (o UPC, *Univer-*

sal Product Code). La grana degli UPC è la stessa delle SKU. Le altre 20 000 SKU corrispondono a prodotti come frutta e carne che non sono confezionati o che sono confezionati localmente e non hanno UPC. Anche a questi prodotti è associato un codice SKU. Questo codice viene assegnato da A4Y ed è condiviso da tutti i negozi della catena.

La direzione di A4Y si occupa della logistica delle ordinazioni, della disposizione delle merci sugli scaffali, della vendita dei prodotti e ha come obiettivo la massimizzazione del profitto. Per aumentare il profitto si può decidere, per esempio, di aumentare i prezzi, di ridurre i costi di acquisizione dei prodotti e delle spese generali, di cercare di attrarre quanti più clienti è possibile e di aumentare le vendite mediante promozioni mirate. In particolare le promozioni possono consistere in: riduzioni temporanee di prezzo, pubblicità (su diversi media), esposizione di prodotti in spazi speciali del negozio. Si vuole quindi svolgere un'analisi dei dati a disposizione a supporto di queste decisioni.

Le sorgenti informative disponibili sono: (a) i dati relativi alle vendite, raccolti in ciascuna cassa mediante dei sistemi POS (point of sale) e memorizzati in una base di dati relazionale; (b) i dati relativi ai negozi, memorizzati in un foglio elettronico; (c) i dati relativi ai prodotti, memorizzati in un file gestito dalla direzione, il cui contenuto viene trasferito periodicamente ai POS. È responsabilità della direzione recepire i nuovi UPC e creare dei nuovi record nel file dei prodotti: a ogni nuovo UPC deve essere assegnato un numero di SKU univoco. La direzione assegna anche i numeri di SKU ai prodotti "locali".

Progettazione dei dati Affrontiamo ora la fase più importante svolgendo i passi nei quali è articolata.

Scelta del processo Supponiamo che uno degli obiettivi principali della direzione di A4Y sia la comprensione dell'impatto delle promozioni sulle vendite e, quindi, sui profitti. In pratica, si cerca di pianificare e progettare le promozioni future sulla base della comprensione dell'impatto delle promozioni passate. Si stabilisce quindi di costruire un data warehouse a supporto delle decisioni iniziando con il progetto del data mart delle vendite giornaliere dei prodotti. Questo data mart sarà alimentato dai sistemi di POS, dal foglio elettronico dei negozi e dal file dei prodotti e dovrà permettere l'analisi degli effetti delle promozioni passate su: aumento della vendita dei prodotti in promozione; diminuzione della vendita al termine della promozione; zone in cui la promozione ha avuto più successo.

Scelta della grana La grana scelta per il data mart potrebbe essere, per esempio, l'unità di vendita (la SKU) analizzata per giorno, per negozio e per promozione applicata. Scelte alternative per la grana potrebbero essere: (a) il prodotto analizzato per giorno, per negozio e per promozione applicata, ma non consente di distinguere l'importanza del confezionamento; (b) la SKU per mese, per negozio e per promozione, ma non consente di distinguere le vendite nei fine settimana da quelle degli altri giorni; (c) la SKU per transazione individuale (scontrino) e per cliente (se è nota l'identità del cliente si possono effettuare interessanti analisi di market basket), però non disponiamo al momento dei dati del cliente. La scelta di grana fatta (unità di

vendita per giorno, per negozio e per promozione) permette per esempio di trovare una risposta alle seguenti domande: è utile vendere più varianti di confezionamento di uno stesso prodotto? Di quanto aumenta il profitto a fronte della promozione di un certo prodotto? Quali sono i dieci prodotti più venduti dai miei concorrenti che invece la catena non vende? (Questa ultima domanda richiede ulteriori dati, magari forniti da società di analisi specializzate). Si osservi che nessuna delle analisi proposte è interessata esplicitamente alle singole SKU. Tuttavia, come detto nel Paragrafo 19.4.3, è preferibile scegliere una grana piuttosto bassa per avere maggiori possibilità di analisi.

Scelta delle dimensioni Fissati il processo (vendite giornaliere dei prodotti) e la grana (SKU per giorno, per negozio e per promozione) bisogna scegliere le dimensioni. In questo caso, la scelta delle dimensioni tempo, prodotto e negozio è immediata. Queste dimensioni sono primarie, nel senso che i movimenti giornalieri dei prodotti dipendono funzionalmente dal tempo, dal prodotto e dal negozio. Un'altra dimensione è la promozione: ogni membro di questa dimensione rappresenta una promozione (o più) applicata alle vendite di un'unità di vendita, in un giorno, in un negozio. Si tratta in questo caso di una dimensione supplementare perché la promozione dipende funzionalmente dalla data, dal prodotto e dal negozio. Altre possibili dimensioni supplementari potrebbero essere il fornitore che ha fornito il prodotto al negozio e il responsabile delle vendite nel negozio nel giorno. Assumiamo però che queste dimensioni non vengono selezionate perché non tracciate e quindi non ricostruibili. Esaminiamo ora nel dettaglio le dimensioni scelte.

La dimensione tempo è presente nella maggior parte degli schemi dimensionali e praticamente in tutti i data warehouse. Nel caso in esame, questa dimensione descrive i giorni di un intervallo temporale di interesse, quindi i relativi membri rappresentano i giorni di questo intervallo. Possibili attributi per questa dimensione sono: CodiceTempo (la chiave, un numero intero), Data (la data del giorno), Anno, Mese, Trimestre, GiornoSettimana (per esempio, mercoledì), Periodo (per esempio, Natale). Per ciascun membro della dimensione è importante scegliere valori concreti della dimensione (come "nessun periodo" per l'attributo Periodo) per evitare la presenza di valori nulli.

La dimensione prodotto descrive le unità di vendita (SKU) della catena di negozi e deve essere aggiornata in seguito a modifiche nel file dei prodotti. Vanno quindi definite le modalità di aggiornamento della dimensione, come illustrato nel Paragrafo 19.3.1. Il file dei prodotti contiene molti attributi descrittivi per ciascuna SKU, alcuni dei quali possono diventare attributi della relativa dimensione. Per esempio: CodiceSKU (la chiave), Confezione (per esempio, "pacco da sei"), Marca (per esempio, "Brawny"), Categoria (per esempio, "Alcolici"), SottoCategoria (per esempio, "Birra"), Settore (per esempio, "Bevande"). Possono naturalmente essere di interesse altri attributi generalmente disponibili nel file dei prodotti come il peso, il colore, la data di scadenza ecc.

La dimensione negozio descrive i negozi della catena A4Y ed è una dimensione essenzialmente geografica: i negozi occupano un punto nello spazio e possono essere aggregati rispetto a ogni possibile raggruppamento geografico. Ne consegue che possibili attributi di questa dimensione sono: CodiceNegozio (la chiave),

NomeNegozio, Indirizzo, CAP, Città, Regione e Zona (nozione relativa alla struttura organizzativa della catena).

La dimensione promozione descrive infine ogni possibile promozione che si applica alla vendita dei prodotti. Le diverse modalità di promozione (riduzioni temporanee di prezzo, pubblicità, esposizione in spazi speciali) possono essere applicate contemporaneamente e ogni membro della tabella dimensione delle promozioni descrive una possibile combinazione delle modalità di promozione. Possibili attributi di questa dimensione sono: CodicePromozione (la chiave), NomePromozione, RiduzionePrezzo (per esempio, "10%"), Pubblicità (per esempio, "giornale"), Esposizione (per esempio, "zona promozioni"), DataInizio, DataFine.

Scelta delle misure Tenendo conto degli obiettivi di analisi previsti, possibili misure relative alle vendite giornaliere dei prodotti e ricavabili dai dati registrati dai POS sono: l'incasso totale in euro (attributo IncassoTotale), il numero totale di unità vendute (VenditaTotale), il costo totale in euro (CostoTotale) relativo al prezzo del fornitore, il numero di clienti che hanno acquistato il prodotto calcolato contando il numero di scontrini in cui è presente il prodotto (NumeroClienti). Le prime tre misure sono additive rispetto a tutte le dimensioni. Il numero di clienti è invece una misura semi-additiva perché non additiva rispetto alla dimensione prodotto: se un prodotto A è stato acquistato da 20 clienti e un prodotto B da 30 clienti, quanti clienti hanno comprato A o B? Tuttavia, è additiva rispetto alle altre dimensioni. In effetti, i conteggi sono solitamente misure semi-additive: possono essere sommate correttamente restringendo le aggregazioni alle dimensioni in cui sono additive. Otteniamo così per il data mart sulle vendite lo schema dimensionale in Figura 19.17.

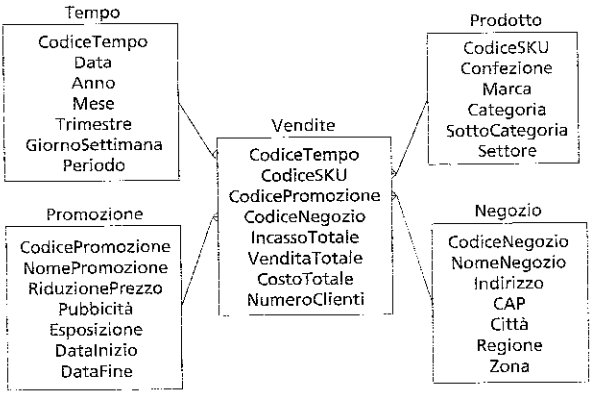

Figura 19.17 Schema dimensionale del data mart sulle vendite.

Progettazione fisica Supponiamo che si decida di procedere a un'implementazione relazionale. Per definire il livello di normalizzazione delle tabelle dimensionali, è necessario fare alcune stime relative alla quantità di dati che saranno contenuti nel data mart. Il numero complessivo di SKU venduti può essere calcolato conoscendo l'incasso complessivo della catena, per esempio 4×10^9 euro all'anno, e il costo medio della voce di vendita, per esempio 2 euro. Nel nostro caso abbiamo circa 2×10^9 vendite di singoli prodotti. Ne consegue che il numero di SKU venduti al giorno in un negozio della catena è pari a $2 \times 10^9/(365 \times 460)$, ovvero circa 12 000. Supponendo che lo stesso SKU venga venduto mediamente 4 volte in una giornata, possiamo assumere che ogni negozio vende giornalmente 3000 SKU diversi. Si osservi come sia necessario elaborare i dati provenienti dalle sorgenti (i POS) per produrre un nuovo record della tabella fatti. A questo punto possiamo fare una stima dell'occupazione di memoria della tabella fatti. Se le chiavi delle tabelle dimensione sono un intero di 4 byte per tempo, prodotto e promozione e di 2 byte per negozio, i quattro campi chiave della tabella fatti occupano complessivamente 14 byte. Se ogni misura è rappresentata da un intero di 4 byte avremo che ogni riga della tabella fatti occupa 30 byte. Ne consegue che la tabella fatti contiene circa $500 \times 3000 \times 365 = 547\,500\,000$ record per ogni anno. Se vengono mantenuti dati storici relativi a due anni, l'occupazione di memoria della tabella fatti è di circa 30 GB.

Vediamo ora le dimensioni. Una tabella dimensionale per il tempo è semplice e non richiede molto spazio. Se la grana è il giorno, i giorni per dieci anni sono poco più di 3650. Anche la tabella dimensionale per i negozi è piuttosto contenuta (non più di 500 righe) e non conviene normalizzarla. La dimensione della tabella dimensionale per le promozioni è paragonabile a quella del tempo. Se infatti ci sono anche 1000 pubblicità sui giornali, 1000 riduzioni temporanee dei prezzi e 200 esposizioni, le combinazioni effettive sono limitate (5000). La dimensione più grande in termini quantitativi è quella dei prodotti per la quale abbiamo stimato che sia necessario memorizzare i dati di circa 60 000 unità. Supponendo che ogni record richieda circa 2000 byte, avremo un'occupazione per la tabella relativa di 120 MB circa. Questa dimensione può essere ridotta procedendo a una normalizzazione della tabella. Va però osservato che l'occupazione complessiva del data mart è ampiamente dominata dalla dimensione della tabella dei fatti (30 GB): anche se riduciamo l'occupazione di memoria della dimensione prodotto del 50%, ridurremmo l'occupazione complessiva di meno dell'1% al prezzo di un sicuro peggioramento delle prestazioni nel caso di selezioni e/o aggregazioni sui prodotti venduti. Anche in questo caso, non è quindi consigliabile procedere a una normalizzazione.

A questo punto, per migliorare le prestazioni, si può procedere alla definizione di opportuni indici e di viste materializzate. Per esempio, potrebbe essere utile definire una vista ottenuta aggregando a livello dei mesi (per avere una visione più sintetica dell'andamento delle misure) ed eliminando la dimensione dei negozi (per avere una visione globale delle vendite nella catena).

Progetto della preparazione dei dati Per alimentare il data mart definito nella fase di progettazione dei dati sono necessarie tre sorgenti di dati: (1) le vendite registrate nei sistemi di POS; (2) il file con la descrizione dei prodotti; (3) il foglio elettronico con la descrizione dei negozi. I dati estratti dalla prima sorgente devono

essere puliti, per eliminare inesattezze e dati inconsistenti. I dati ottenuti vanno poi combinati con quelli provenienti dalla seconda sorgente (opportunamente formattati per consentire il confronto) aggregando il risultato per negozio. Si deve poi procedere anche a un raggruppamento per data (opportunamente codificata). Infine, il risultato ottenuto va integrato con i dati provenienti dalla terza sorgente (anche in questo caso, opportunamente formattati).

Il processo descritto è rappresentato graficamente in Figura 19.18. Questi tipi di rappresentazione sono proprio quelli adottati dagli strumenti di ETL, che mettono a disposizione interfacce amichevoli, una serie di operazioni standard da applicare sui dati e un ambiente per la definizione di procedure definite dall'utente.

19.5 Data mining

Il termine *data mining* viene utilizzato per caratterizzare tecniche di analisi diverse da quelle viste finora ma con il medesimo obiettivo: estrarre informazioni utili da un data warehouse. In particolare, il data mining serve a cercare "informazioni nascoste" nel contenuto di un data warehouse. Per esempio, il data mining è utilizzato per analisi di mercato, tese a individuare oggetti acquisiti assieme o in sequenza; oppure analisi di comportamento, quali per esempio le frodi e gli usi illeciti delle carte di credito; o ancora per analisi di previsione, per esempio del costo futuro delle cure mediche. Il data mining è una materia interdisciplinare, che utilizza, oltre alle tecnologie per la gestione dei dati, anche la statistica – per definire la qualità delle osservazioni – e l'intelligenza artificiale – nella costruzione di basi di conoscenza. Recentemente, il data mining ha acquisito un'enorme popolarità e ha garantito un vantaggio competitivo a molte imprese di tipo commerciale, che hanno potuto definire migliori politiche di gestione e marketing.

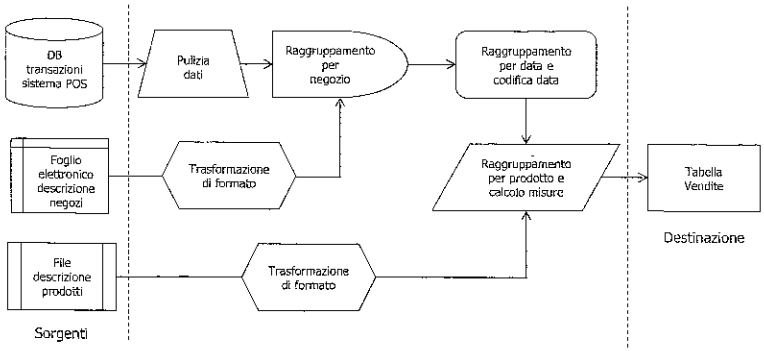

Figura 19.18 Rappresentazione grafica di un processo di ETL.

19.5.1 Il processo di data mining

Il processo di data mining viene anche chiamato *knowledge discovery on database* o KDD e ha come obiettivo l'estrazione, da una collezione di dati, di informazioni che siano valide in base a un ragionevole grado di certezza, precedentemente sconosciute, potenzialmente utili al supporto alle decisioni e comprensibili per i possibili fruitori. Il processo opera tipicamente su dati che possono essere anche semi-strutturati (per esempio log di transazioni) e, applicando opportuni algoritmi, è in grado di restituire espressioni di un certo linguaggio che descrivono, in maniera sintetica e facile da comprendere, le informazioni estratte. Queste espressioni sono dette solitamente *pattern*, perché descrivono regolarità presenti nei dati.

Si consideri, per esempio, il grafico riportato in Figura 19.19 nel quale sono stati indicati, con una croce, i clienti di una compagnia di assicurazione auto che hanno avuto almeno un sinistro negli ultimi cinque anni e, con un punto, quelli che non ne hanno avuti, con riferimento all'età e alla classe di rischio di appartenenza.

Un processo di data mining applicato a questi dati, teso a individuare polizze a rischio, potrebbe restituire un pattern del tipo:

IF *Età* $< k$ AND *Classe* $> n$ THEN rischio alto

Si osservi come la validità del pattern è associata a un livello di probabilità che cambia se modifichiamo i valori di k e n.

Il processo di data mining procede in maniera iterativa e adattativa, innescando un meccanismo di costruzione progressiva della conoscenza che richiede molteplici interazioni e si articola in cinque fasi principali.

1. *Comprensione del dominio.* È impossibile estrarre informazioni utili se non si sviluppa a priori una buona comprensione del dominio applicativo in cui si opera.

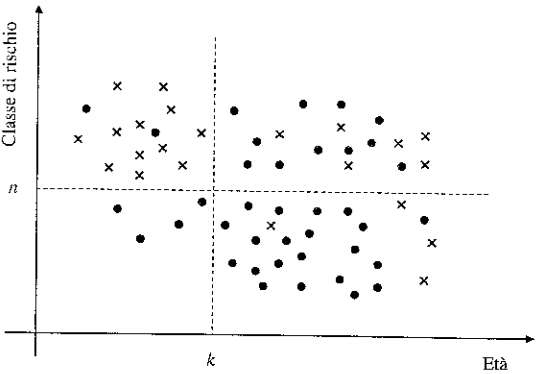

Figura 19.19 Distribuzione sinistri per età e classe di merito.

2. *Preparazione del set di dati.* Questo passo richiede l'individuazione di un sottoinsieme dei dati del DW su cui effettuare il data mining e la loro codifica in modo da poter essere dati in input a un algoritmo di data mining. Per esempio, valori di pressione possono essere *discretizzati* in semplici quantità quali "alta", "media" e "bassa", e questa operazione può consentire successivamente di correlare valori discreti di pressione con l'assunzione di un farmaco.
3. *Scoperta dei pattern.* Consiste nell'applicazione del processo di data mining al set di dati estratto precedentemente, in modo da estrarre pattern dai dati. Nel seguito del paragrafo concentreremo l'attenzione specialmente sulle varie tecniche utilizzate in questo passo.
4. *Valutazione dei pattern.* Consiste nel trarre implicazioni applicative dai pattern scoperti, valutando quali esperimenti svolgere successivamente e quali ipotesi formulare o quali conseguenze trarre nel processo di scoperta della conoscenza.
5. *Utilizzo dei risultati.* Consiste nell'interpretare i pattern scoperti e nel prendere decisioni operative basate su questa interpretazione, per esempio modificando le politiche di allocazione delle merci nei grandi magazzini oppure le politiche di concessione di credito.

19.5.2 Problemi di data mining

Anche se ciascun problema applicativo ha qualche caratteristica specifica, negli ultimi anni sono stati riconosciuti vari problemi di data mining di struttura ricorrente, per i quali sono stati definiti e realizzati degli specifici algoritmi. In genere, gli algoritmi di data mining sono caratterizzati da buona scalabilità, garantiscono cioè buone caratteristiche di efficienza quando vengono applicati a grandi moli di dati. Nel seguito vediamo due problemi classici: la determinazione di regole di associazione e la classificazione.

Regole di associazione Le regole di associazione si applicano a dati descritti mediante una tabella relazionale partizionata tramite una clausola di raggruppamento. L'esempio classico è una tabella che descrive gli acquisti in un grande magazzino, raggruppati per transazione di acquisto. Questo tipo di analisi prende tradizionalmente il nome di *basket analysis*. Una tabella che descrive transazioni di acquisto in un grande magazzino è rappresentata in Figura 19.20: ciascuna tupla rappresenta l'acquisizione di una specifica merce. Il codice della transazione d'acquisto è presente in tutte le tuple e viene utilizzato per raggruppare assieme tutte le tuple relative allo stesso acquisto. In questo contesto, le regole di associazione descrivono situazioni in cui la presenza di una merce in una transazione è associata alla presenza di un'altra merce con elevata probabilità.

Più propriamente, una regola di associazione consta di una premessa e una conseguenza e ha la forma *premessa* → *conseguenza*. Premesse e conseguenze possono essere valori o insiemi di valori presenti nella tabella. Per esempio, la regola: *sci* → *bastoncini* indica che l'acquisto di sci (premessa) è accompagnato "spesso" da un acquisto di bastoncini (conseguenza). Una regola famosa e meno scontata, scoperta tramite data mining delle vendite nei supermercati, indica *pannolini* → *birra*; la

CodTrans	Data	Oggetto	Qta	Prezzo
1	17/12/14	pantaloni-sci	1	140
1	17/12/14	scarponi	1	180
2	18/12/14	maglietta	1	25
2	18/12/14	giacca	1	300
2	18/12/14	stivali	1	70
3	18/12/14	giacca	1	300
4	19/12/14	giacca	1	300
4	19/12/14	maglietta	3	25

Figura 19.20 Base di dati per basket analysis.

regola si spiega considerando il fatto che i pannolini vengono spesso acquistati dai padri (si tratta di un acquisto semplice e nel contempo voluminoso), i quali si assumono anche tipicamente la delega all'acquisto della birra. Si dice che questa regola abbia fatto aumentare i profitti della catena di supermercati semplicemente spostando le birre nel reparto vicino a quello dei pannolini, favorendo in questo modo una associazione inaspettata, osservata nel passato.

È possibile definire in modo preciso le probabilità relative alle regole di associazione. Si usano allo scopo due misure: il supporto e la confidenza. Date un insieme di tuple rappresentanti transazioni, come quelle in Figura 19.20, e una regola $X \rightarrow Y$, dove X e Y sono insiemi di oggetti presenti nelle tuple:

- il *supporto* è la probabilità che in una transazione siano presenti sia la premessa sia la conseguenza e si ottiene facendo il rapporto tra il numero di transazioni che contengono gli oggetti in $X \cup Y$ e il numero totale di transazioni;
- la *confidenza* è la probabilità che in una transazione in cui è presente la conseguenza di una regola sia presente anche la premessa e si ottiene facendo il rapporto tra il numero di transazioni che contengono gli oggetti in $X \cup Y$ e il numero totale di transazioni che contengono gli oggetti in X.

Intuitivamente, il supporto misura l'importanza di una regola (quanto spesso premessa e conseguenza sono presenti insieme in tuple della tabella) mentre la confidenza ne misura l'affidabilità (tra tutte le tuple della tabella in cui è presente la premessa, quanto spesso è presente anche la conseguenza). Il problema di data mining relativo alla scoperta delle regole di associazione viene quindi enunciato come segue: *trovare tutte le regole di associazione con supporto e confidenza superiori a valori prefissati.*

Per esempio, la Figura 19.21 mostra le regole di associazione con supporto e confidenza maggiori o uguali a 0.25; se fossimo invece interessati alle sole regole che hanno supporto superiore a 0.4 e confidenza superiore a 0.7, estrarremmo la sola regola *maglietta* $\rightarrow$ *giacca*.

Variazioni di questo problema, ottenute con diverse estrazioni dei dati ma essenzialmente con lo stesso algoritmo di ricerca, consentono per esempio le merci vendute a seguito di una promozione, oppure merci vendute assieme d'estate ma non d'inverno, oppure merci vendute assieme solo quando sono disposte vicine. Altre varianti

Premessa	Conseguenza	Supporto	Confidenza
{pantaloni-sci}	{scarponi}	0.25	1
{scarponi}	{pantaloni-sci}	0.25	1
{maglietta}	{stivali}	0.25	0.5
{maglietta}	{giacca}	0.5	1
{stivali}	{maglietta}	0.25	1
{stivali}	{giacca}	0.25	1
{giacca}	{maglietta}	0.5	0.66
{giacca}	{stivali}	0.25	0.33
{maglietta, stivali}	{giacca}	0.25	1
{maglietta, giacca}	{stivali}	0.25	0.5
{stivali, giacca}	{maglietta}	0.25	1

Figura 19.21 Regole di associazione per il problema di gestione delle vendite in un grande magazzino.

del problema che richiedono un diverso algoritmo di ricerca consentono invece di studiare andamenti temporali nelle vendite, per esempio le merci vendute in sequenza allo stesso cliente; un esempio tipico è l'acquisto del lettore di DVD poco dopo aver acquistato il televisore. Si parla in questo caso di *pattern sequenziali* e si tratta di regole di evidente utilità, per esempio nell'impostare campagne promozionali mirate nel contesto di vendite on-line.

Le regole di associazione e i pattern sequenziali consentono di studiare problemi diversi dalla basket analysis; per esempio, in medicina, è possibile dare indicazioni su quali resistenze antibiotiche sono contemporaneamente presenti in un antibiogramma, oppure scoprire che il diabete può portare a una forte diminuzione della vista dopo dieci anni dalla sua insorgenza.

Classificazione La classificazione mira alla catalogazione di un fenomeno in un insieme di classi predefinite. Il fenomeno viene generalmente presentato sotto forma di un record di osservazioni elementari (tupla). Il *classificatore* è un algoritmo che svolge la classificazione; esso viene costruito automaticamente partendo da un insieme di dati di prova (detto *training set*) costituito da un insieme di fenomeni già classificati; viene poi applicato per la classificazione di fenomeni generici. Tipicamente i classificatori si presentano come alberi di decisione. Ai nodi di questi alberi si associano dei predicati che consentono di prendere decisioni sulla base dei valori presenti nella tupla che rappresenta il fenomeno. Quando i fenomeni sono descritti da un gran numero di attributi, i classificatori si occupano anche di determinare gli attributi significativi, separandoli da quelli irrilevanti.

Supponiamo per esempio di voler classificare le polizze di una compagnia assicurativa, attribuendo loro un rischio elevato oppure basso. Partendo da una collezione di osservazioni che descrivono polizze, il classificatore determina inizialmente che i soli attributi significativi per definire il rischio di una polizza sono l'età del guidatore

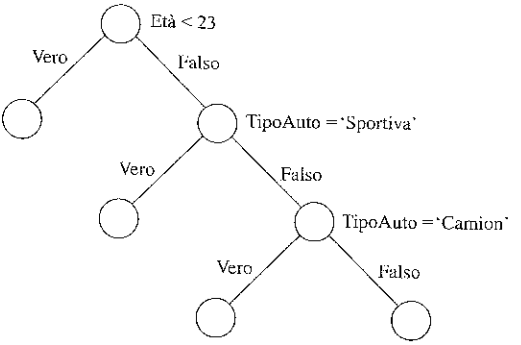

Figura 19.22 Classificatore per polizze a rischio in base a età e tipo di auto.

e il tipo di auto, e costruisce poi un albero di decisione come quello illustrato in Figura 19.22. In questo albero un rischio elevato viene attribuito a tutti i guidatori di età inferiore ai 23 anni, o ai guidatori di auto sportive o ai guidatori di camion.

Clustering L'obiettivo del clustering è quello di suddividere un insieme di fenomeni osservati in gruppi omogenei, detti *cluster*, in maniera che i fenomeni dello stesso gruppo presentino un comportamento simile rispetto ad alcuni attributi di riferimento. Rispetto al precedente problema di classificazione, in questo caso si assume di non avere nessuna conoscenza pregressa sulle possibili aggregazioni. I cluster ottenuti possono essere sovrapposti e organizzati gerarchicamente.

Consideriamo per esempio la distribuzione di sinistri riportata in Figura 19.19 e supponiamo di voler individuare dei cluster omogenei di clienti in base ai sinistri. Un possibile risultato dell'applicazione di un algoritmo di cluster è riportato in Figura 19.23. Si osservi come i cluster individuati possono essere sovrapposti o avere degli elementi in comune.

La tecnica del clustering ha diverse applicazioni. Può essere utile quando si vogliono individuare, come nell'esempio visto, popolazioni omogenee di clienti in basi di dati di marketing, oppure nel monitoraggio delle attività di aziende concorrenti o nella valutazione dei risultati di esperimenti clinici. Il clustering viene anche usato come fase preliminare di una successiva attività di classificazione, per avere un supporto all'individuazione delle classi di riferimento.

19.6 Big data

La quantità di dati disponibile nel mondo cresce a una velocità esponenziale. È stato stimato che dagli 800 Terabytes (circa 10^{12} bytes) di dati complessivamente disponibili nel 2000, si è passati a 160 Exabytes (10^{18} bytes) nel 2006, a 500 Exabytes nel

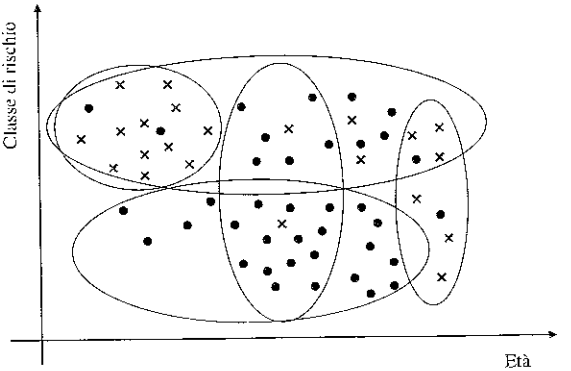

Figura 19.23 Clustering di clienti di una compagnia di assicurazione in base ai sinistri.

2009, a 2,7 Zettabytes (10^{21} bytes) nel 2012 e si prevede che si raggiungeranno i 35 Zettabytes nel 2020 con una crescita del 40% all'anno. Questa crescita vertiginosa è principalmente dovuta a dati provenienti da "nuovi" scenari applicativi, la cui raccolta e memorizzazione è resa sempre più economica dall'evoluzione tecnologica (si stima che nel 2013, con 600 dollari circa sia possibile comprare un disco nel quale memorizzare tutta la musica del mondo). Si pensi per esempio al volume di dati generato dai social network (solo Facebook produce ogni giorno circa 10 TB di dati), dal calcolo scientifico (che tipicamente deve elaborare grandissime quantità di dati), dalle reti di sensori (che possono raccogliere ogni secondo milioni di dati) e dalle transazioni finanziarie (si parla di miliardi di transazioni al giorno). In questi contesti, le soluzioni hardware e software adottate per le applicazioni "classiche" risultano spesso inefficaci a causa della dimensione dei dati, decisamente superiore alla norma e in continua crescita, e della loro natura tipicamente destrutturata.

Il termine *big data* è stato coniato proprio per indicare quelle collezioni di dati la cui dimensione supera le capacità di memorizzazione, gestione e analisi tipiche dei tradizionali sistemi per basi di dati. Le caratteristiche generali di questi grossi magazzini di dati vengono spesso descritte secondo tre dimensioni, dette le tre "v" dei big data.

- Volume: dimensione della base di dati espressa non solo in termini assoluti, ma anche in termini di andamento di crescita e di requisiti di prestazioni per la loro elaborazione.
- Varietà: tipologia dei dati e delle sorgenti; nella maggior parte dei casi si tratta di dati semi-strutturati (file in formato XML, Jason, CVS o in formati proprietari) o destrutturati (pagine Web, file di testo, documenti) per i quali tipicamente non esiste uno schema.

- Velocità: rapidità con la quale i dati arrivano e devono essere elaborati; alcune applicazioni consentono l'elaborazione batch, ma in molti casi bisogna operare il *real-time* o quantomeno in *near-time* e non sono rari i casi in cui i dati viaggiano in *stream*, ovvero in flussi, che vanno elaborati alla velocità nella quale arrivano.

Accanto a questi aspetti, viene spesso citata un'altra "v": il *valore*, ovvero le potenzialità dei dati raccolti in termini di vantaggi competitivi che possono essere raggiunti dalla loro analisi. Il problema principale dei big data è quindi quello di riuscire a derivare dai dati nuova conoscenza del dominio applicativo di interesse. In questo senso, gli obiettivi dei big data sono gli stessi del data warehousing, con approcci che si basano sostanzialmente su tecniche di data mining, opportunamente ripensate nel contesto dei big data. Per esempio, con riferimento al caso di studio relativo a una catena di negozi discusso nei paragrafi precedenti, si tende oggi a raccogliere e analizzare dati provenienti dai social network e dai questionari di gradimento dei clienti per comprendere gusti e preferenze degli utenti (analisi esplorativa) ma anche predisposizioni all'acquisto (analisi predittiva). Per questi motivi, il ciclo di vita dei big data ricalca quello tipico dei processi di data warehousing e data mining, con una prima fase dedicata alla selezione dei dati, seguita dalla loro estrazione e integrazione nei casi di sorgenti diverse. Una volta memorizzati, nella maggior parte dei casi in forma poco e per niente strutturata, si applicano ai dati tecniche di analisi (spesso denominate di *data analytics*) che evolvono ed estendono le tecniche di data mining presentate nel Paragrafo 19.5 con l'obiettivo di interpretare i dati e ottenere da essi nuove informazioni. Queste tecniche si basano prevalentemente su combinazioni di metodi statistici con tecniche di machine learning.

Una marcata differenza rispetto ad applicazioni di knowledge discovery "tradizionali" è che la grandissima disponibilità di dati consente di applicare modelli più sofisticati e di ottenere risultati molto più precisi che nel passato. Per fare un esempio, i grandi miglioramenti che ci sono stati negli ultimi anni nella meteorologia sono proprio dovuti alla possibilità di applicare modelli complessi a moltitudini di dati di granularità estremamente fine provenienti dai satelliti e da grandi reti di stazioni meteorologiche. Si possono fare molti altri esempi di successo di analisi dei big data in campi molto diversi tra loro, come l'automazione degli investimenti nel mercato finanziario, la generazione di pubblicità personalizzata basata sul contesto e il ruolo decisivo svolto dall'analisi delle reti sociali nella campagna elettorale di Obama, nel 2012. Le potenzialità future sono praticamente infinite. Grosse speranze sono riposte nella diagnosi e cura delle malattie genetiche basata sull'analisi dei dati genomici, nelle scoperte astronomiche basate sull'analisi dei dati raccolti dai nuovi telescopi ad alta definizione, nell'avanzamento dell'epidemiologia basata sull'analisi della distribuzione e della frequenza di malattie nella popolazione.

A causa delle caratteristiche appena discusse, le soluzioni tecnologiche per la memorizzazione e l'elaborazione dei big data sono piuttosto diverse da quelle illustrate nei paragrafi precedenti. È difficile a tale riguardo definire architetture di riferimento precise come è stato fatto nel caso dei data warehouse classici, a causa delle innumerevoli soluzioni che sono state proposte. Peraltro, in molti casi queste soluzioni sono pensate per domini applicativi specifici e quindi non sono generalizzabili. È però possibile individuare alcune caratteristiche dei sistemi per la gestione dei big

data che, anche se non costituiscono soluzioni imprescindibili, sono molto comuni nelle applicazioni reali.

- I dati vengono distribuiti e replicati su cluster di computer per garantire la scalabilità delle applicazioni e per aumentare la tolleranza ai guasti e la disponibilità dei dati. L'infrastruttura di base utilizzata nella quasi totalità dei casi è quella di *Apache Hadoop*, un framework open-source per l'elaborazione distribuita basato sul file-system HDFS, originariamente proposto da Google. Approfondimenti su Hadoop e sulle architetture parallele sono presenti nel Capitolo 17.
- I dati vengono memorizzati in formati semi o de-strutturati di cui non è in genere disponibile lo schema. Si usano spesso sistemi NoSQL (o in generale non-relazionali) che poggiano su Hadoop e offrono alle applicazioni interfacce di più alto livello nei quali però i dati vengono gestiti in formati elementari, per esempio mediante semplici coppie chiave-valore. Questi sistemi di gestione dei dati sono stati discussi nel Capitolo 13, analizzando i loro vantaggi e svantaggi.
- Le operazioni richieste dall'analisi dei dati vengono seguite in parallelo. In genere, si usa allo scopo il paradigma di programmazione MapReduce, che è nativo in Hadoop e che consente l'esecuzione di algoritmi paralleli e distribuiti su cluster di computer. Hadoop mette a disposizione anche alcuni strumenti per il supporto allo sviluppo di programmi MapRduce come *Apache Pig*, che offre un linguaggio di alto livello nello stile di SQLMapReduce. I modelli MapReduce e Apache Pig sono stati illustrati nel Capitolo 17.
- Le risorse hardware e software vengono virtualizzate adottando il paradigma del cloud-computing. Sul mercato esiste un'ampia disponibilità di offerte in questo ambito provenienti da aziende leader nel settore, come Amazon e Microsoft. I modelli di servizio offerti possono essere diversi, andando dall'*Infrastructure as a Service* (IaaS), nella quale solo l'infrastruttura è virtualizzata, al *Software as a Service* (SaaS), nel quale si virtualizza anche il software, che risiede e opera sul cloud. A questi modelli si aggiunge il *Data as a Service* (DaaS), nel quale si virtualizza la base di dati e il servizio di cloud offre le sole funzionalità di accesso ai dati. Le architetture per il cloud sono state ampiamente illustrate nel Capitolo 17.

Accanto a queste tecnologie, molte altre sono utilizzate nella gestione dei big data. Tra queste, vale la pena di citare: i sistemi di basi di dati *Main Memory* (MMDBMS), che per migliorare le prestazioni gestiscono i dati direttamente in memoria principale; il *croudsourcing*, un modello di elaborazione nel quale alcuni attività come la raccolta o la pulizia dei dati vengono svolte da grandi comunità di persone; strumenti come *Apache Hive* e *Hadapt*, che offrono interfacce relazionali verso Hadoop consentendo l'esecuzione distribuita di interrogazioni SQL su cluster di computer. Esistono poi diversi strumenti che consentono di definire ed eseguire analisi sofisticate sui dati, tra cui citiamo *Apache Mahout*, il quale offre una libreria di algoritmi di machine learning che possono essere eseguiti su Hadoop (ma non solo), e *R*, un linguaggio e un ambiente di programmazione di pubblico dominio per lo sviluppo di elaborazioni statistiche per il quale esistono molte librerie di procedure predefinite. Naturalmente, anche tutti i maggiori leader nel mercato dei DBMS, come Oracle, Microsoft e IBM,

offrono soluzioni complete per la gestione dei big data adottando anche soluzioni non relazionali.

Si tratta di un mondo ancora in forte evoluzione ma è evidente il forte interesse che tutte le organizzazioni, pubbliche e private, di piccole, medie e grandi dimensioni, hanno in questo settore e i forti investimenti che prevedibilmente ci saranno nei prossimi anni porteranno certamente a una rapida evoluzione dei metodi e delle tecniche per la gestione dei big data.

Note bibliografiche

Nonostante il modello multidimensionale dei dati sia stato definito verso la fine degli anni Settanta, i primi sistemi OLAP sono emersi solo agli inizi degli anni Novanta. Una definizione delle caratteristiche di OLAP e un elenco di regole che i sistemi OLAP devono soddisfare è data da E.F. Codd, inventore del modello relazionale, in un articolo del 1995 [107]. Ci sono oggi molti testi che descrivono i data warehouse e le tecniche di progettazione. Tra di essi citiamo quello di Kimball [128] e il testo in italiano di Golfarelli e Rizzi [118]. L'operatore `data cube` è introdotto da Gray e altri nell'articolo [120]. La metodologia di progettazione presentata in questo capitolo si basa sulla modellazione dimensionale, proposta da Kimball [128] e largamente utilizzata nel mondo aziendale.

La letteratura sul data mining è ampia; una presentazione sistematica delle problematiche del settore è nel libro di Han, Kamber e Pei [123]. Il testo di Rajaraman e Ullman [138] offre una descrizione dettagliata delle principali tecniche di data mining e con esempi di applicazione nel campo dei big data.

Infine, anche sui big data esiste già una vasta letteratura, ma dato che il tema è ancora in forte evoluzione, molti testi sono prevalentemente divulgativi per non rischiare una rapida obsolescenza sul fronte delle tecnologie. Una buona introduzione all'argomento viene fornita da un rapporto di McKinsey del 2011 [132].

Esercizi

Soluzioni sul sito http://www.ateneonline.it/atzeni

19.1 Progettare un cubo multidimensionale relativo all'analisi dei sinistri per una compagnia assicurativa, basandosi sulle specifiche accennate nel Paragrafo 19.2.1.

19.2 Descrivere alcune operazioni slice-and-dice, roll-up e drill-down per il cubo multidimensionale definito nell'esercizio precedente.

19.3 Indicare cosa si ottiene applicando un roll-up che elimina la dimensione ARTICOLO dal cubo in Figura 19.7.

19.4 Scrivere un'interrogazione SQL in grado di eseguire un roll-up che, a partire dallo schema stella in Figura 19.10, calcola l'incasso totale per marca di prodotto e città.

19.5 Scrivere un'interrogazione SQL in grado di eseguire un roll-up che, a partire dallo schema a fiocco di neve in Figura 19.12, calcola il numero di articoli venduti per categoria di prodotto, mese ed età del cliente.

19.6 Scrivere un'interrogazione SQL che, mediante la clausola CUBE, calcola le vendite complessive per trimestre e marca di prodotto a partire dallo schema stella in Figura 19.10 e indicare un possibile risultato.

19.7 Mostrare il risultato che si ottiene sostituendo la clausola CUBE con la clausola ROLLUP nell'interrogazione dell'esercizio precedente.

19.8 Indicare una scelta motivata di indici bitmap, indici di join e viste materializzate per lo schema a stella in Figura 19.10.

19.9 Progettare un cubo dimensionale relativo alla gestione degli esami universitari, considerando come fatti gli esiti degli esami sostenuti dagli studenti e come dimensioni di analisi il tempo, la sede dell'esame (supponendo una facoltà distribuita su più sedi), il docente coinvolto e le caratteristiche degli allievi (per esempio, i dati relativi all'andamento scolastico pre-universitario, il punteggio nell'esame di ammissione e il corso di laurea prescelto).

19.10 Realizzare in un sistema ROLAP il cubo dell'esercizio precedente definendo sia uno schema a stella sia uno schema a fiocco di neve. Indicare infine una scelta motivata di indici bitmap, indici di join e viste materializzate per gli schemi ottenuti.

19.11 Si consideri la seguente base di dati relazionale, relativa alle prescrizioni di farmaci acquisite da un insieme di farmacie:

- RICETTE(Numero, CodFarmacia, CFPaziente, Data)
- FARMACIA(CodFarmacia, Nome)
- ELEMENTIRICETTA(NumeroRicetta, NumeroLinea, CodFarmaco)
- FARMACI(Codice, Descrizione, CodMolecola, CodCasa, Prezzo, Fascia)
- MOLECOLE(CodMolecola, Descrizione)
- PAZIENTI(CF, Cognome, Nome, DataNascita, Via, NumeroCivico, Città)
- CASEFARMACEUTICHE(CodCasa, Nome)
- ASL(Codice, Nome)
- TERRITORIO(Via, Città, NumeroCivico, ASL)

Si noti che ci sono dati che cambiano nel tempo fra cui prezzi e fasce ('A', 'B' o 'C') dei farmaci e indirizzi dei pazienti. Costruire, in tale contesto, uno schema a stella che permetta di analizzare le prescrizioni (quantità e prezzi complessivi) rispetto a: (1) data (dimensione standard i cui dettagli possono essere omessi); (2) farmaci, con le loro proprietà (molecola e casa farmaceutica); (3) ASL di residenza e fascia d'età (per esempio, 0-3,4-17, 18-30, ...) dei pazienti; (4) farmacia. Supporre che, per ovvie ragioni di privacy, non possano essere riportati dati che permettano di risalire alle identità dei pazienti.

19.12 Progettare uno o più data mart relativi alla gestione delle linee ferroviarie, considerando come fatti il numero complessivo di passeggeri giornaliero per ciascuna tariffa su ciascun treno e su ciascuna tratta della rete e come dimensioni le tariffe, la distribuzione geografica delle città attraversate, la composizione del treno, il tempo, i guasti e i ritardi. Realizzare uno o più cubi dimensionali e darne la traduzione in forma relazionale.

19.13 Si consideri la seguente base di dati relazionale, relativa alle vendite presso una catena di supermercati:

- VENDITE(NumeroScontrino, CodNegozio, CFCliente, Data, Ora)
- ELEMENTIVENDITA(NumeroScontrino, NumeroLinea, CodArticolo, Quantità)
- ARTICOLI(CodArticolo, Descrizione, CodMarca, Prezzo)

- CLIENTI(CFCliente, Nome, DataNascita, CodCategoria, CittàResidenza)
- NEGOZI(CodNegozio, Nome, Indirizzo, Città)
- CITTÀ(Nome, Provincia)
- PROVINCE(Sigla, Regione)
- MARCHE(CodMarca, Nome, CodNazione)
- NAZIONI(CodNazione, Nazione)
- CATEGORIE(CodCategoria, Descrizione)

Si noti che, ovviamente, il prezzo di un prodotto cambia molto rapidamente nel tempo, mentre la residenza di un cliente può cambiare, ma molto di rado. Costruire, in tale contesto, uno schema dimensionale che permetta di analizzare le vendite giornaliere (quantità e incassi) rispetto a: (1) data (dimensione standard aziendale, i cui dettagli possono essere omessi); (2) prodotti, con tutte le loro proprietà; (3) negozi, con tutte le loro proprietà; (4) clienti, con le relative fasce d'età (per esempio 18-24 anni, 25-35 anni ecc.), categorie e città, provincia e regione di residenza. Indicare sinteticamente come si ottengono le relazioni dello schema dimensionale da quelle della base di dati.

19.14 L'ufficio statistico dell'ateneo riceve spesso, dai presidi di facoltà e da altri docenti, richieste volte a conoscere le questioni elencate di seguito.

- Il numero di immatricolati per la prima volta nel sistema universitario nazionale in un corso di laurea in un certo anno accademico (nel seguito si usa il termine *coorte* per fare riferimento a tale insieme di studenti);
- Per una specifica coorte e uno specifico anno accademico, il numero degli studenti che appartengono a ciascuna delle seguenti categorie: (a) iscritti a ciascuno degli anno di corso; (b) iscritti fuori corso; (c) laureati; (d) non più iscritti al corso di laurea, non laureati, ma iscritti ad altro corso dello stesso ateneo; (e) non più iscritti al corso di laurea, non laureati e non iscritti ad altro corso dello stesso ateneo.
- Per una specifica coorte, con riferimento a una certa data con riferimento agli studenti ancora iscritti (in quella data), il numero degli studenti che hanno conseguito crediti in numero compreso in un certo intervallo (supponendo di interesse gli intervalli multipli di 10 e quelli multipli di 15).
- Per una specifica coorte e per uno specifico corso, il numero di studenti che hanno, a una certa data, superato il relativo esame.
- Per una specifica coorte e per uno specifico corso, il voto medio riportato dagli studenti che hanno, a una certa data, superato il relativo esame.

Progettare uno o più data mart che permettano di rispondere alle esigenze sopra formulate, supponendo che le informazioni necessarie allo scopo siano nella base di dati delle segreterie studenti. In particolare: (1) mostrare i frammenti di schema (ER e relazionale) della base di dati delle segreterie che si suppone di utilizzare come sorgente dei dati; (2) mostrare gli schemi a stella dei data mart; (3) mostrare (anche in modo schematico) le trasformazioni necessarie per passare dalla sorgente ai data mart.

19.15 Si consideri la tabella descritta in Figura 19.24. Estrarre le regole di associazione con supporto e confidenza maggiori o uguali al 20%. Indicare poi quali regole sono estratte se si richiede invece un supporto superiore al 50%.

19.16 Riferendosi alla tabella di Figura 19.24, discretizzare i prezzi in tre valori (basso, medio e alto); trasformare i dati in modo che per ciascuna transazione si indichi con una sola tupla la presenza di una vendita di una particolare classe. Costruire poi regole di associazione che indicano la compresenza nella stessa transazione di oggetti appartenenti a differenti classi di prezzo. Infine, interpretare i risultati.

CodTrans	Data	Oggetto	Qta	Prezzo
1	17/12/14	pantaloni-sci	1	140
1	17/12/14	scarponi	1	180
1	17/12/14	bastoncini	1	20
2	18/12/14	maglietta	1	25
2	18/12/14	giacca	1	200
2	18/12/14	stivali	1	70
3	18/12/14	giacca	1	200
4	19/12/14	giacca	1	200
4	19/12/14	maglietta	3	25
5	20/12/14	maglietta	1	25
5	20/12/14	giacca	1	200
5	20/12/14	cravatta	1	25

Figura 19.24 Tabella che contiene dati di vendita.

19.17 Descrivere una base di dati delle vendite di automobili in cui compaiono descrizioni delle auto (auto sportive, compact, station wagon ecc.), costo e cilindrate delle auto (discretizzati in classi), età e reddito degli acquirenti (discretizzati in classi). Ipotizzare poi la struttura di un classificatore relativo alle propensioni all'acquisto di automobili da parte di categorie differenti di persone.

19.18 Si considerino i data mart dell'Esercizio 8.11. Quali sono i problemi di data mining che vi vengono in mente?

Bibliografia

Capitoli 1-12

[1] S. Abiteboul, R. Hull, V. Vianu. *Foundations of Databases*. Addison-Wesley, Reading, Massachussetts, 1995.

[2] A. Albano. *Basi di dati – strutture e algoritmi*. Addison-Wesley Italia, Milano, 1992.

[3] A. Albano. *Costruire sistemi per basi di dati*. Addison-Wesley Italia, Milano, 2001.

[4] A. Albano, V. De Antonellis, A. Di Leva (editors). *Computer-Aieded Database Design: The DATAID Project*. North-Holland, Amsterdam, 1985.

[5] P. Atzeni, C. Batini, V. De Antonellis. *Teoria relazionale dei dati*. Boringhieri, Torino, 1985.

[6] P. Atzeni, S. Ceri, P. Fraternali, S. Paraboschi, R. Torlone. *Basi di dati: architetture e linee di evoluzione*. McGraw-Hill, Milano, I edizione 2003, II edizione 2007.

[7] P. Atzeni, V. De Antonellis. *Relational Database Theory*. Benjamin-Cummings, Menlo Park, California, 1993.

[8] C. Batini, S. Ceri, S. B. Navathe. *Conceptual Database Design, an Entity-Relationship Approach*. Benjamin-Cummings, Menlo Park, California, 1992.

[9] C. Batini, G. De Petra, M. Lenzerini, G. Santucci. *La progettazione concettuale dei dati*. Franco Angeli, Milano, 1991.

[10] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, P. O'Neil. A Critique of ANSI SQL Isolation Levels, *Proc. ACM SIGMOD 95*, pp.1–10, San Jose, California, 1995.

[11] P. Bernstein, V. Hadzilacos, N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Reading, Massachusetts, 1987.

[12] P.A. Bernstein, E. Newcomer. *Principles of Transaction Processing for the Systems Professional*. Morgan Kaufmann, San Francisco, California, 1997.

[13] P. Bonnet, D. Shasha. *Database Tuning: Principles, Experiments and Troubleshooting Techniques*. Morgan Kaufmann, San Francisco, California, 2002.

[14] G. Booch, I. Jacobson, J. Rumbaugh. Second edition. *The Unified Modeling Language User Guide*. Addison-Wesley, Reading, Massachussetts, 2005.

[15] M.L. Brodie and M. Stonebraker. *Migrating Legacy Systems: Gateways, Interfaces & the Incremental Approach*. Morgan-Kaufmann, Los Altos, California, 1995.

[16] L. Cabibbo, R. Torlone, C. Batini. *Basi di dati: progetti ed esercizi svolti*. Pitagora editrice, Bologna, 1995.

[17] S.J. Cannan, G.A.M. Otten. *SQL - The Standard Handbook*. McGraw-Hill, New York, 1992 (edizione italiana: *Il manuale SQL*, McGraw-Hill Italia, Milano).

[18] S. Castano, M.G. Fugini, G. Martella, P. Samarati. *Database Security*. Addison Wesley, New York, 1994.

[19] S. Ceri (editor). *Methodology and Tools for Database Design*. North-Holland, Amsterdam, 1983.

[20] S. Ceri, P. Fraternali. *Designing Database Applications with Objects and Rules: The IDEA Methodology*. Addison-Wesley, Reading, Massachussetts, 1997.

[21] S. Ceri, G. Gottlob. Translating SQL into relational algebra: Optimization, semantics, and equivalence of SQL queries. *IEEE Transactions on Software Engineering*, vol. 11, n. 4, pagg. 324–345, 1985.

[22] S. Ceri, G. Gottlob, L. Tanca. *Logic Programming and Data Bases*. Springer-Verlag, Berlino, 1989.

[23] S. Ceri, G. Pelagatti. *Distributed Databases: Principles and Systems*. McGraw-Hill, New York, 1984.

[24] D.D. Chamberlin. *A Complete Guide to DB2 Universal Database*. Morgan-Kaufmann, San Mateo, California, 1998.

[25] D.D. Chamberlin, R.F. Boyce. SEQUEL: A structured English query language. *Proceedings of ACM SIGMOD Workshop*, vol. 1, pagg. 249–264, 1974.

[26] D.D. Chamberlin, M.M. Astrahan, K.P. Eswaran, P.P. Griffiths, R.A. Lorie, J.W. Mehl, P. Reisner, B.W. Wade. SEQUEL 2: A unified approach to data definition, manipulation, and control. *IBM Journal of Research and Development*, vol. 20, n. 6, pagg. 97–137, 1976.

[27] S. Chaudhuri, V.R. Narasayya. Self-Tuning Database Systems: A Decade of Progress. *VLDB*, pp.3-14, 2007.

[28] P.P. Chen. The Entity-Relationship model: Toward a unified view of data. *ACM Transaction on Database System*, vol. 1, n. 1, pagg. 9–36, 1976.

[29] P. Ciaccia, D. Maio. *Lezioni di basi di dati*. Leonardo Editore, Bologna, 1995.

[30] E.F. Codd. A relational model for large shared data banks. *Communications of the ACM*, vol. 13, n. 6, pagg. 377–387, 1970.

[31] E.F. Codd. Further normalization of the data base relational model. In R. Rustin, *Database Systems*, pagg. 33–64, Prentice Hall, Englewood Cliffs, New Jersey, 1972.

[32] E.F. Codd. Relational completeness of database sublanguages. In R. Rustin, *Database Systems*, pagg. 65–98, Prentice Hall, Englewood Cliffs, New Jersey, 1972.

[33] E.F. Codd. Extending the database relational model to capture more meaning. *ACM Transaction on Database System*, vol. 4, n. 4, pagg. 397–434, 1979.

[34] E.F. Codd. Relational database: A practical foundation for productivity. *Communications of the ACM*, vol. 25, n. 2, pagg. 109–117, 1982.

[35] R. Cochrane, H. Pirahesh, N. Mattos. Integrating triggers and declarative constraints in SQL database systems. *Proceedings of the 22nd International Conference on Very Large Data Bases*, Bombay, pagg. 567–578, 1996.

[36] S. Ceri, J. Widom. Deriving production rules for constraint maintenance. In *Proceedings of the 16th International Conference on Very Large Data Bases*, Brisbane, pagg. 566–577, 1990.

[37] C.J. Date. *An Introduction to Database Systems*. Eighth edition. Addison-Wesley, Reading, Massachussetts, 2003.

[38] C.J. Date, H. Darwen. *A Guide to the SQL Standard*. Third edition. Addison-Wesley, Reading, Massachussetts, 1993.

[39] A. Eisenberg, J. Melton. Standards in practice. *ACM SIGMOD Record*, vol. 27, n. 3, pagg. 53–58, 1998.

[40] A.K. Elmagarmid (editor). *Database Transaction Models for Advanced Applications* Morgan Kaufmann, San Mateo, California, 1992.

[41] R.A. ElMasri, S.B. Navathe. *Fundamentals of Database Systems*. Fourth edition. Benjamin-Cummings, Menlo Park, California, 2003. (edizione italiana: *Sistemi di basi di dati*, in due volumi, Pearson Education Italia, Milano).

[42] C.C. Fleming, B. von Halle. *Handbook of Relational Database Design*. Addison-Wesley, Reading, Massachussetts, 1989.

[43] M. Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Third Edition. Addison-Wesley, Reading, Massachussetts, 2003. (edizione italiana: *UML distilled. Guida rapida al linguaggio di modellazione standard, 3/ed*, Pearson Education Italia, Milano).

[44] C. Francalanci, F. Schreiber, L. Tanca. *Progetto di dati e funzioni*. Società Editrice Esculapio, Bologna, 2003.

[45] H. Garcia Molina, J.D. Ullman, J. Widom. *Database System Implementation*. Prentice-Hall, Upper Saddle River, New Jersey, 2000.

[46] H. Garcia-Molina, J.D. Ullman, J. Widom. *Database Systems: The Complete Book*. Prentice Hall, 2002.

[47] C. Ghezzi, M. Jazayeri, D. Mandrioli. *Ingegneria del software Fondamenti e principi*. Seconda edizione. Pearson Education Italia, Milano).

[48] J. Gray, A. Reuter. *Transaction Processing Concepts and Techniques*. Morgan-Kaufmann, San Mateo, California, 1994.

[49] R. Hull, R. King. Semantic database modelling: survey, applications and research issues. *ACM Computing Surveys*, vol. 19, n. 3, pagg. 201–260, 1987.

[50] IBM Corporation. *IBM DATABASE 2 SQL Guide for Common Servers, Version 2*. 1995.

[51] W. Kim. On optimizing an SQL-like nested query. *ACM Transactions on Database Systems*, vol. 7, n. 3, pagg. 443–469, 1982.

[52] D.E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison Wesley, Reading, Massachusetts, 1973.

[53] K. Kulkarni, N. Mattos, R. Cochrane. Active Database Features in SQL-3, in N. Paton (ed.) *Active Rules in Database Systems*, Springer-Verlag, 1999.

[54] V.Y. Lum, S.P. Ghosh, M. Schkolnik, R.W. Taylor, D. Jefferson, S. Su, J.P. Fry, T.J. Teorey, B. Yao, D.S. Rund, B. Kahn, S.B. Navathe, D. Smith, L. Aguilar, W.J. Barr, P.E. Jones. 1978 New Orleans Data Base Design Workshop Report.

Proceedings of the 5th International Conference on Very Large Data Bases, Rio de Janeiro, pagg. 328–339, 1979.

[55] K. Larman. *Applying UML and Patterns*. Third edition. Prentice-Hall, Englewood Cliffs, New Jersey, 2004. (edizione italiana: *Applicare UML e i pattern 3/ed*, Pearson Education Italia, Milano).

[56] D. Maier. *The Theory of Relational Databases*. Computer Science Press, Potomac, Maryland, 1983.

[57] D. Maio, S. Rizzi. *Esercizi di progettazione di basi di dati*. Pitagora editrice, Bologna, 1995.

[58] H. Mannila, K.J. Raiha. *The Design of Relational Databases*. Addison-Wesley, Reading, Massachussetts, 1992.

[59] J. Melton. SQL-3 update. *Proceedings of the IEEE International Conference on Data Engineering 1996*, New Orleans, pagg. 666-672, 1996.

[60] J. Melton, A.R. Simon. *Understanding the New SQL*. Morgan-Kaufmann, San Mateo, California, 1993.

[61] J. Melton, A.R. Simon. *SQL:1999 - Understanding Relational Language Components*. Morgan-Kaufmann, San Mateo, California, 2001.

[62] Oracle Corporation. *Oracle 8 Server: Concepts Manual*. Redwood City, California, 1998.

[63] Oracle Corporation. *Oracle 8 Server: SQL Language Reference Manual*. Redwood City, California, 1998.

[64] J. Paredaens, P. De Bra, M. Gyssens, D. Van Gucht. *The Structure of the Relational Database Model*. Springer-Verlag, Berlino, 1989.

[65] R.S. Pressman. *Software Engineering, a Practitioners Approach*. Sixth edition. McGraw-Hill, New York, 2005. (edizione italiana: *Principi di Ingegneria del software 4/ed*, McGraw-Hill Italia, Milano).

[66] R. Ramakrishnan, J. Gehrke. *Database Management Systems*. Third edition. McGraw-Hill, New York, 2002. (edizione italiana: *Sistemi di basi di dati*, McGraw-Hill Italia, Milano).

[67] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen. *Object-Oriented Modelling and Design*. Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[68] A. Salvaggio. *Access: Programmazione VBA*. Edizioni FAG, Milano, 2005.

[69] D. Shasha. *Database Tuning: A Principled Approach*. Morgan-Kaufmann, San Mateo, California, 1992.

[70] D. Shasha, P. Bonnet. *Database Tuning: Principles, Experiments, and Troubleshooting Techniques* Morgan-Kaufmann, San Mateo, California, 2002.

[71] A. Silberschatz, H.F. Korth, S. Sudarshan. *Database System Concepts*. McGraw-Hill, New York, 1997.

[72] J.M. Smith, D.C.P. Smith. Database Abstractions: Aggregation and Generalization. *ACM Transactions on Database Systems*. vol. 2, n. 1, pagg. 105–133, 1977.

[73] I. Sommerville. *Software Engineering*. Seventh Edition. Addison-Wesley, Reading, Massachussetts, 2004. (edizione italiana: *Ingegneria del software, 7/ed.*, Pearson Education Italia, Milano).

[74] M. Stonebraker (editor). *Readings in Database Systems*. Second edition. Morgan-Kaufmann, San Mateo, California, 1994.
[75] T. Teorey. *Database Modeling and Design: the E-R Approach*. Morgan-Kaufmann, San Mateo, California, 1990.
[76] T. Teorey, J.P. Fry. *Design of Database Structures*. Prentice-Hall, Englewood Cliffs, New Jersey, 1982.
[77] T. Teorey, D. Yang, J.P. Fry. A logical design methodology for relational databases using the extended Entity-Relational approach. *ACM Computing Surveys*, vol. 18, n. 2, pagg. 201–260, 1986.
[78] D. Tsichritzis, F.H. Lochovsky. *Data Models*. Prentice-Hall, Englewood Cliffs, New Jersey, 1982.
[79] J.D. Ullman. *Principles of Database and Knowledge Base Systems, vol. 1*. Computer Science Press, Potomac, Maryland, 1988 (edizione italiana: *Basi di dati e basi di conoscenza*, Gruppo Editoriale Jackson, Milano, 1991).
[80] J.D. Ullman, J. Widom. *A First Course in Database Systems*. Prentice-Hall, Upper Saddle River, New Jersey, 1997.
[81] *Vocabolario della lingua italiana*. Istituto della Enciclopedia Italiana, 1987.
[82] G. Vossen. *Data Models, Database Languages, and Database Management Systems*. Addison-Wesley, Reading, Massachussetts, 1990.
[83] G. Weikum, G. Vossen. *Fundamentals of Transactional Information Systems: Theory, Algorithms, and Practice of Concurrency Control and Recovery*. Morgan Kaufmann, San Francisco, California, 2001.
[84] J. Widom, S. Ceri. *Active Database Systems*. Morgan Kaufmann, San Mateo, California, 1996.
[85] G. Wiederhold. *Database Design*. McGraw-Hill, New York, 1983.
[86] C. Zaniolo. Database relations with null values. *Journal of Computer and System Science*, vol. 28, n. 1, pagg. 142–166, 1984.
[87] C. Zaniolo, S. Ceri, C. Faloutsos, R.T. Snodgrass, V.S. Subrahmanian, R. Zicari. *Introduction to Advanced Database Systems*. Morgan Kaufmann, San Mateo, California, 1997.

Capitoli 13-19

[88] S. Abiteboul, D. Suciu, P. Buneman. *Data on the Web : From Relations to Semistructured Data and XML*. Morgan Kaufmann, San Francisco, California, 1999.
[89] J-R. Abrial. Data Semantics. IFIP Working Conference Data Base Management, pp. 1-60, 1974.
[90] G. Alonso, F. Casati, H. Kuno, V. Machiraju. *Web Services*. Springer Verlag, Berlino, 2003.
[91] P. Atzeni, S. Ceri, P. Fraternali, S. Paraboschi, R. Torlone. *Basi di dati: modelli e linguaggi di interrogazione*. McGraw-Hill, Milano, IV edizione 2013.
[92] P. Atzeni, C.S. Jensen, G. Orsi, S. Ram, L Tanca, R. Torlone The relational model is dead, SQL is dead, and I don't feel so good myself. *SIGMOD Record*, 42(2):64V68, 2013.

[93] F. Bancilhon, C. Delobel, P. Kanellakis. *Building an Object-Oriented Database System: The Story of O_2*. Morgan Kaufmann, San Mateo, California, 1992.

[94] T. Berners-Lee, J. Hendler, O. Lassila. The Semantic Web. *Scientific American Magazine*, 2001.

[95] P. A. Bernstein. Middleware: a model for distributed system services. *Communications of the ACM*, vol. 39, n. 2, pagg. 86–98, 1996.

[96] E. Bertino, L. Martino. *Sistemi di basi di dati orientati agli oggetti: concetti e architetture*. Addison-Wesley Italia, Milano, 1992.

[97] M. Bieber, T. Isakowitz (A cura di). *Special Issue on Designing Hypermedia Applications*. Communications of the ACM, Vol. 38, num. 8, 1995.

[98] C. Bizer, T. Heath, T. Berners-Lee. Linked Data - The Story So Far. *Int. Journal of Semantic Web Information Systems*, 5(3), pagg. 1-22, 2009.

[99] N. Bradley. *The XML Companion*. Addison Wesley Professional, Boston, Massachusetts, 2003.

[100] C. Buil-Aranda, M. Arenas, O. Corcho, A. Polleres. Federating queries in SPARQL 1.1: Syntax, semantics and evaluation. *Web Semantics*, 18(1), pagg. 1-17, 2013.

[101] A. Calì, G. Gottlob, T. Lukasiewicz. Datalog+-, an Integrated Approach to Ontologies and Integrity Constraints. *Proceedings of the 12th International Conference on Database Theory*, pagg. 14–30, 2009

[102] R. G. G. Cattel. *Object Data Management - Object-Oriented and Extended Relational Database Systems*. Revised Edition. Addison Wesley, Reading, Massachussetts, 1994.

[103] R. G. G. Cattel. Scalable SQL and NoSQL data stores. *SIGMOD Record*, 39(4):12–27, 2010.

[104] S. Ceri, P. Fraternali, A.Bongio, M. Brambilla, S. Comai, M. Matera. *Designing data-intensive Web applications*. Morgan Kaufmann, San Francisco, California, 2003 (edizione italiana: *Progettazione di dati e applicazioni per il Web*, McGraw-Hill Italia, Milano, 2003).

[105] A. B. Chaudhri, A. Rashid, R. Zicari. *XML Data Management: Native XML and XML-Enabled Database Systems*. Addison Wesley Professional, Boston, Massachusetts, 2003.

[106] J. Cheng, S. Malaika. *Web Gateway Tools: Connecting IBM and Lotus Applications to the Web*. John Wiley & Sons, New York, 1997.

[107] E. F. Codd. Twelve rules for On-Line Analytical Processing. *Computerworld*, aprile 1995.

[108] D.E. Comer *Internetworking with TCP/IP, Volume I: Principles, Protocols, and Architecture*. Fifth edition. Prentice-Hall, Englewood Cliffs, New Jersey, 2006.

[109] J. Dean, S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Proceedings of OSDI'04*, pagg. 137–150, 2004.

[110] E. Della Valle, I. Celino, D. Cerizza. *Semantic Web. Dai fondamenti alla realizzazione di un'applicazione* Pearson, 2009.

[111] E. Della Valle, S. Ceri, F. van Harmelen, D. Fensel. It's a Streaming World! Reasoning upon Rapidly Changing Information. *IEEE Intelligent Systems*, 24:6, 2009.

[112] E. Della Valle, S. Ceri. Querying the Semantic Web: SPARQL. In J. Domingue, D. Fensel and J. A. Hendler (ed), *Handboook of Semantic Web technologies*, Springer-Verlag, 2011.

[113] D. J. DeWitt, J. Gray. Parallel Database Systems: The Future of High Performance Database Systems. *Communications of the ACM*, vol. 35, n. 6, pagg. 85–98, 1992.

[114] J. Feiler. *Database-Driven Web Sites*. Morgan Kaufmann, San Mateo, California, 1999.

[115] R. T. Fielding, R. N. Taylor. Principled Design of the Modern Web Architecture. *Proceedings of ICSE'00*, pagg. 407–416, 2000.

[116] A. F. Gates. *Programming Pig: Dataflow Scripting with Hadoop*. O'Reilly Media, Sebastopol, California, 2011.

[117] A. F. Gates, J. Dai, T. Nair. Apache Pig's Optimizer. *IEEE Data Engineering*, Vol. 36, num. 1, pagg. 34–45, 2013.

[118] M. Golfarelli, S. Rizzi. *Data Warehouse. Teoria e pratica della progettazione*. McGraw-Hill Italia, Milano, 2002.

[119] J. Gray, M. Anderton. Distributed computer systems: Four case studies. *IEEE Proceedings*, vol. 75, n. 5, 1987.

[120] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, H. Pirahesh. Data-cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Mining and Knowledge Discovery*, vol. 1, n. 1, pagg. 29–53, 1997.

[121] J. Gray, L. Lamport. Consensus on Transaction Commit. *ACM Transactions on Database System*, Vol.31, No. 1, pagg. 133-160, 2006.

[122] P. Greenspun. *Philip & Alex's Guide to Web Publishing*. Morgan Kaufmann, San Mateo, California, 1999.

[123] J. Han, M. Kamber. *Data Mining: Concepts and Techniques*. Second Edition. Morgan-Kaufmann, San Francisco, California, 2006.

[124] E. R. Harold, W. S. Means. *XML in a Nutshell*. O'Reilly UK, Londra, 2002.

[125] T. Isakovitz, M. Bieber, F. Vitali. Web Information Systems. *Communications of the ACM*, vol. 41, n. 7, pagg. 78–117, 1998.

[126] P. Ju. *Databases on the Web: Designing and Programming for Network Access*. IDG Books Worldwide, Foster City, California, 1997.

[127] W. Kim. *Modern Database Systems: The Object Model, Interoperability, and Beyond*. ACM Press/Addison-Wesley, New York, 1995.

[128] R. Kimball. *The Data Warehouse Toolkit*. Second Edition. John Wiley & Sons, New York, 2002.

[129] L. Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, vol. 21, n. 7, pagg. 558–565, 1978.

[130] C. Liu, J. Peek, R. Jones, B. Buus, A. Nye *Managing Internet Information Services*. O'Reilly & Associates, Sebastopol, California, 1994.

[131] M. E. S. Loomis. *Object Databases: The Essentials*. Addison Wesley, Reading, Massachusetts, 1995.

[132] J. Manyika, M. Chui, B. Brown, J. Bughin, R. Dobbs, C. Roxburgh, A. Hung Byers. *Big Data: The Next Frontier for Innovation, Competition, and Productivity*. McKinsey Report, McKinsey Global Institute, 2011.

[133] D. Martin (editor). *OWL-S: Semantic Markup for Web Services*. World Wide Web Consortium – W3C, http://www.w3.org/Submissions/OWL-S, 2004.
[134] P. Mell, T. Grance. *The NIST Definition of Cloud Computing*. National Institute of Standards and Technology, Special Publication 800-145, 2011.
[135] J. Melton. *Advanced SQL:1999*. Morgan Kaufmann, San Francisco, California, 2003.
[136] J. Melton, S. Buxton. *Querying XML: XQuery, XPath, and SQL/XML in context*. Morgan Kaufmann, San Francisco, California, 2006.
[137] R. Obermark. Distributed deadlock detection algorithm. *ACM Transactions on Database Systems*, vol. 7, n. 2, 1982.
[138] A. Rajaraman, J. D. Ullman. *Mining of Massive Datasets*. Cambridge University Press, Cambridge, 2011.
[139] G. Samaras, K. Britton, A. Citton, C. Mohan. Two-phase commit optimizations in a commercial distributed environment. *Journal of Distributed and Parallel Databases*, vol. 3, n. 4, pagg. 325–360, 1995.
[140] H. Samet, *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, Reading, Massachusetts, 1989.
[141] N. Shadbolt, W. Hall, T. Berners-Lee. The Semantic Web Revisited. *IEEE Intelligent Systems*, 2006.
[142] A. P. Sheth, J. A. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Computing Surveys*, vol. 22, n. 3, pagg. 183–236, 1990.
[143] S. St.Laurent. *XML: A Primer*. John Wiley & Sons, New York, 2001.
[144] M. Stonebraker. *Object-Relational DBMSs - The Next Great Wave*. Morgan Kaufmann, San Mateo, California, 1994.
[145] M. Stonebraker and R. Cattell. 10 rules for scalable performance in 'simple operation' datastores. *Commun. ACM*, 54(6):72–80, 2011.
[146] M. Stonebraker and U. Çetintemel. "One Size Fits All": An Idea Whose Time Has Come and Gone (Abstract) *ICDE*, 2V11, 2005.
[147] M. Stonebraker, L. A. Rowe, B. G. Lindsay, J. Gray, M. J. Carey, M. L. Brodie, P. A. Bernstein, D. Beech. Third-Generation Database System Manifesto - The Committee for Advanced DBMS Function. *ACM SIGMOD Record*, vol. 19, n. 3, pagg. 31–44, 1990.
[148] V. S. Subrahmanian. *Principles of Multimedia Database Systems*. Morgan Kaufmann, San Mateo, California, 1998.
[149] P. Walmsley *XQuery*. O'Reilly & Associates, Sebastopol, California, 2007.
[150] T. White. *Hadoop: The Definitive Guide*. O'Reilly Media, Sebastopol, California, 2009.

Indice analitico

2-d tree, 513
3-tier, 645

A

abort
    global (nel commit a due fasi), 598
    presunto, 603
accesso
    ad intervallo, 431
    controllo di, 179
    calcolato, 411–415
    diretto, 431
    puntuale, 415, 431
accessore, metodo, 489, 503
accorpamento
    di associazioni, 282, 292, 293
    di entità, 282, 292
aciclicità, 492
acidità delle transazioni, 183
additività, 706
ADO, ActiveX Data Object, 367
ADO.NET, 369
affidabilità dei dati, 4
    gestione della, 445–455
aggiornamento fantasma, 458
aggregazioni
    in UML, 222
AJAX, 676
albero
    B e B+, 423–425
    struttura ad, 415–425
algebra relazionale, 43–69
    espressione in, 58
    interrogazione in, 58–61
    operatori dell', 44
`alias`, 107, 113
alias in SQL, 118
`all`, 114, 121, 128, 131
allocazione, trasparenza di, 587
`alter`, 102
amministratore della base di dati, 11
analisi
    dei dati, 697–737
    dei requisiti, 191, 235–239, 257, 263
    delle ridondanze, 282–286
    di prestazioni su schemi E-R, 278–282
    multidimensionale, 701
àncora, 525
annotazione, 510
anomalie
    aggiornamento fantasma, 458
    di aggiornamento, 324
    di transazioni concorrenti, 456–459
    inserimento fantasma, 459, 467
    lettura inconsistente, 458
    lettura sporca, 457
    perdita di aggiornamento, 456
ANSI, 89
`any`, 131
Apache
    Hive, 637
    Pig, 628
application server, 658
applicazioni distribuite, 583
`array`, 501
    struttura ad, 410
`asc`, 120
asserzioni in SQL, 150
associazioni
    accorpamento di, 282, 292, 293
    in UML, 219, 221–223
    nel modello E-R, 199–202
    partizionamento di, 282, 292, 293
    verifica di normalizzazione, 346, 347
atomicità, 184
    delle transazioni distribuite, 596–597
    globale, 509
attributi
    cardinalità di, 206, 207
    composti, 202
    di relazione, 19
    in UML, 220
    multivalore, eliminazione di, 291
    nel modello E-R, 202, 203

audio, 510
autocommit, 183, 375
    JDBC, 378
autorizzazioni, 4, 179
avg, 121

B
B e B+, albero, 423–425
backup, 4
bag-of, 483
base di dati, 2–6
    amministratore della, 11
    condivisione di, 4
    cooperazione, 611–615
    dimensione di una, 4
    distribuita, 582–609
    estensione della, 8
    intensione della, 8
    istanza di, 8
    multidimensionale, 702–707
    multimediale, 509
    parallela, 619–623
    progettazione di, 191–321
    replicata, 609–611
    schema della, 7
    utenti della, 12, 13
    XML nativa, 549–550
batch, 355
Bean Java, 660
big data, 698, 733–737
bigint, 95
blob, 95
blob, 506
blocco, 403
    critico, 475–477
    distribuito, 593
    fattore di, 408
    select-from-where, 107
    tempo di accesso a un, 403
block header e block trailer, 407
boolean, 95
bottom-up
    primitive di trasformazione, 252
    strategia, 252–253, 257
Boyce e Codd, forma normale di, 326–328
buffer, 404–406
    gestore del, 401, 453
business rules, 214–216

C
C++, 488, 490
C#, 654
calcolo relazionale, 69–79
    su domini, 70–76
    interrogazione nel, 71–74
    su tuple, 76–79, 113
caldo, ripresa a, 453
Call Level Interface, 364
cammino
    indice, 507
    di join, 305
caratteristica delle operazioni, 277, 279
cardinalità
    di attributi, 206, 207
    di relazioni, 204–206, 223
cascade, 101–104, 138, 139
cascaded, 152, 153
Cascading Style Sheet (CSS), 526
case, 159
CASE, 262–265, 312, 313, 509
cast, 158
cataloghi relazionali, 104
char_length, 158
character, 92
check, 149, 150
check-in, 508
check-out, 508
checkpoint, 449, 454
chiave
    nel modello E-R, 207
    nel modello relazionale, 31–34, 98, 99, 114, 126, 326
ciclo di vita dei sistemi informativi, 191–193
classi, 486
    in UML, 219, 220
classificazione
    di schedule, 470
    nel data mining, 732–733
cleaning dei dati, 700
CLI, 364
client side script, 675
client-server, 507, 580–582
client-side scripting, 672
clob, 95
CLOB, 550
close, 359
cloud e cloud computing, 623
Cloud Infrastructure as a Service, 625
Cloud Platform as a Service, 625
Cloud Software as a Service, 625
clustering, 733
coalesce, 158
CODASYL, 6
collaudo di sistemi informativi, 193
commit, 182, 450
    a due fasi, 597–609
    a quattro fasi, 603
    a tre fasi, 604
    global (nel commit a due fasi), 598

Paxos, 605
-precedenza, 450
Common Gateway Interface, 645–647
compensazione, transazione di, 509
complessità
semantica, 492
strutturale, 483
complete (nel commit a due fasi), 598
completezza
di uno schema E-R, 256
composizioni in UML, 223
Computer-Aided-Design (CAD), 481
concorrenza
anomalie della, 456–459
gestione della, 455–460
condivisione dei dati, 4
conflict-equivalenza, 462
conflict-serializzabilità, 462
conflitto
di nome, 494
d'impedenza, 357, 491
confluenza, 173
consistenza, 184
container EJB, 660
controllo
di accesso, 179
di concorrenza
basato su locking a due fasi (2PL), 463–468
basato su timestamp, 468–470
distribuito, 592–596
multiversioni, 469
contro-varianza, 498
cookie, 679
cooperazione di basi di dati, 611–615
correttezza di uno schema E-R, 256
costrutti del modello E-R, 196
costruttore
di tupla in SQL, 135
metodo, 489, 503
`count`, 121, 126
co-varianza, 498
`create`
`assertion`, 151
`domain`, 96
`role`, 181
`schema`, 95
`table`, 96
`view`, 152
criteri di rappresentazione, 241, 242
`cube`, 714
cubo multidimensionale, 704
`current of`, 359
`current_date`, 157
`current_time`, 157
`current_timestamp`, 157
cursori, 358, 359, 363

D

data cleaning, 700
data cube, 714
data mart, 701, 704
data mining, 701, 728–733
data source, 697
data warehouse, 697–702, 704
bus, 701
progettazione di un, 717–728
server, 700
Datalog, 79–83
interrogazioni in, 80–82
negazione in, 83
`date`, 94
dati, 2
affidabilità dei, 4
indipendenza dei, 9
privatezza dei, 4, 179
progettazione dei, 192
DB2, trigger in, 167, 168
DBA, 11
DBMS, 3–6, 13, 196
DDL, Data Definition Language, 10, 89
deadlock, 475–477
distribuito, 593
`deallocate`, 363
`decimal`, 93
decisioni, sistemi di supporto alle, 697–737
`declare`
`cursor`, 358
`section`, 356
decomposizione
qualità, 328–333
`default`, 97, 101, 105, 139
`deferred`, 151
definizione
di domini in SQL, 96
di schema in SQL, 95
di tabelle
in SQL, 96
`delete`, 102, 138, 179
denominazione di oggetti, 495
denormalizzazione, 700, 710–711
dereferenziamento, 504
`desc`, 120
design pattern, 242–249
determinismo delle osservazioni, 173
diagramma
degli oggetti, 219
degli stati, 219
dei casi d'uso, 219

dei componenti, 219
delle attività, 219
delle classi, 218–220, 226
di collaborazione, 219
di comunicazione, 219
di distribuzione dei componenti, 220
di sequenza, 219
differenza (operatore), 44
dimension table, 709
dipendenza funzionale, 325–326, 338–343
copertura, 341–342
implicazione di, 338–340
`distinct`, 114, 121
distribuita, base di dati, 582–609
distruttore, metodo, 489
dizionario
dei dati
in SQL, 104
nel modello E-R, 216
di pagina, 407
DML, Data Manipulation Language, 10, 89
documentazione di schemi E-R, 213–216
documenti, ricerca di, 511
DOM, 550, 673
level 3, 673
domini
definizione di – in SQL, 96
elementari di SQL, 92–94
introdotti in SQL-3, 94
dot notation, 484, 504
`double precision`, 93
drill-down, 706–707
`drop`, 103
dump, 450, 455
duplicati in SQL, 114
DTD, 527, 547

E

ECMAScript, 672
ECPG, 356
elemento XML, 526
eliminazione
delle generalizzazioni, 286–289
delle gerarchie, 282
di attributi multivalore, 291
embedded, SQL, 356
Enterprise Java Beans (EJB), 660, 671
entità, 198–199
accorpamento di, 282, 292
identificatori di, 207–209
partizionamento di, 282, 290–292
-relazione, 482, 492, 506
verifica di normalizzazione, 344–346
entity
bean, 661
class, 382, 385
entry SQL, 91
equi-join, 58
equivalenza
di espressioni algebriche, 62–64
di linguaggi di interrogazione, 75
ereditarietà, 492
multipla, 493
espressione
del calcolo relazionale su domini, 71–74
dell'algebra relazionale, 58
equivalenza di, 62–64
estensione, 487
della base di dati, 8
evento-condizione-azione, 164
`except`, 128, 129, 132, 136
`exec sql`, 356, 360
`execute`, 362, 363
ETL, 700

F

fact table, 708
fail-stop, modello, 452
fattore
di blocco, 408
di riempimento, 413
`fetch`, 358
fisica
progettazione – di una base di dati, 436–440
fix, 405
fixpoint, 82, 83
`float`, 93
flush, 406
FLWOR, 535, 539
`for`, 535
force, 406
foreign key, 35, 100
forma normale, 323
di Boyce e Codd, 326–328, 333, 334, 336
seconda, 336
terza, 333–337, 342, 343
frammentazione
dei dati, 584
orizzontale, 584
trasparenza di, 587
verticale, 584
freddo, ripresa a, 455
`from`, 107, 109
`full join`, 115–117
full SQL, 91
funzionamento di sistemi informativi, 193

funzione hash, 412
funzioni
in SQL-3, 503
progettazione delle, 192
scalari in SQL, 157–160

G
garbage collection, 495
generalizzazioni, 488, 493
eliminazione delle, 286–289
esclusive
in UML, 226
nel modello E-R, 210
gerarchie di, 211, 488, 492–495
in UML, 225, 226
nel modello E-R, 209–212
parziali
in UML, 226
nel modello E-R, 210
sovrapposte
in UML, 226
nel modello E-R, 210
totali
in UML, 226
nel modello E-R, 210
gerarchia
in SQL-3, 502
gestione
dei buffer, 404–406
dell'affidabilità, 445–455
della concorrenza, 455–460
in SQL, 459, 460
delle tuple nelle pagine, 407, 408
di dati spaziali, 513
gestore
dei metodi di accesso, 401
del buffer, 401
delle interrogazioni, 401
GIS, 481, 513
glossario dei termini, 238, 257
grafo
di attivazione, 173
dei conflitti, 462
`grant`, 180
granularità dei trigger, 164
`group by`, 123–127
`cube`, 714
`rollup`, 714
guasti in un DBMS, 452

H
Hadoop, 625, 627, 736
Hadoop Distributed File Sytem (HDFS), 625
handle, 495
hash, 506
funzione, 412
join, 432
struttura, 411–415
`having`, 126, 127
heap, 409
hosting, 623
HTML, 524
HTML5, 679
HTTP, 643

I
IaaS, 625
idempotenza di undo e redo, 449
identificatori
di entità, 207, 209
di oggetto, 482, 484
esterni e interni, 208
in UML, 224–225
scelta degli – principali, 282
scelta degli – primari, 294
scelta degli – primari, 293
identità di oggetti, 485
IFML, 687–694
immagini, 509
`immediate`, 151
impedenza, conflitto di, 491
implementazione
di sistemi informativi, 192
di una classe, 486
di un metodo, 488
incapsulamento, 486
incastonamento, 356
Indexed Database API, 680
indice, 194, 196, 282, 415–425
2-d tree, 513
bitmap, 715–717
cammino, 507
complesso, 507
definizione di – in SQL, 426
di join, 715–717
di prestazione, 278, 285
invertito, 512
multilivello, 419
nidificato, 507
primario, 416
quad-tree, 514
secondario, 294, 416
indipendenza
dal dominio, 75
dalla conoscenza, 163
dei dati, 9, 401
information retrieval, 511

informazione, 2
    incompleta, 25–28
`inner join`, 115
inserimento fantasma, 459, 467
`insert`, 137, 179
inside-out, strategia, 253, 254
`instead of`, 172
`integer`, 93
integrazione di schemi, 252, 253, 255, 257
integrità referenziale
    con trigger, 175
    di oggetti, 495
intensione della base di dati, 8
interfaccia di una classe, 486
Intermediate SQL, 91
interoperabilità, 579
interpretazione algebrica di SQL, 113
interrogazioni
    gestione delle, 427–436
    in algebra relazionale, 58–61
    in Datalog, 80–82
    in SQL, 106–137
    insiemistiche in SQL, 128–130
    nel calcolo relazionale su domini, 71–74
    nidificate, 130–137
    ottimizzazione delle, 427–436
`intersect`, 128
intersezione (operatore), 44, 56
`interval`, 94
`into`, 162, 357, 358, 361, 363
invertito, indice, 512
ipermedia, 642
ipertesto, 641
ISO, 89
isolamento, 184
istanza
    di base di dati, 8
    di relazione, 23
    di una classe, 493
    nel modello E-R, 196

J

Java, 488
    Bean, 660
    Persistence Query Language, 393
    Server Pages (JSP), 652
    servlet, 647–651
JavaScript, 674
    engine, 675
    Object Notation (JSON), 668, 678
JDBC, 370–374, 579
join, 51–58, 63, 64, 430
    equi-, 58
    esterno, 55
    interni ed esterni in SQL, 115–118
    naturale, 51–53
    metodi di, 431
    (operatore) cammino di, 305
    theta-, 56
    tTheta-, 58
JOINED, 386
    subclass, 385
JPA, 381, 661
JPQL, 393
    Named Query, 394
    Native Query, 394
    path expression, 397
    Query, 394
JSON, 678

K

Knowledge discovery, 729

L

Lamport, metodo di, 593
late binding, 496
`left join`, 115, 116
leggibilità di uno schema E-R, 256
`let`, 535, 537
lettura
    inconsistente, 458
    sporca, 457, 459
`like`, 111
linguaggio
    dichiarativo, 43, 69
    di definizione dei dati, 10
    di manipolazione dei dati, 10
    equivalenza di, 75
    indipendenza dal dominio del, 75
    per basi di dati, 10, 11
    procedurale, 43, 69
    trasparenza di, 588
list-of, 483
lock, 463
    conflitti di, 464
    gerarchico, 472
    gestione dei, 472
    upgrade, 464, 476
locking
    a due fasi, 463–468
        distribuito, 593
        fase crescente e fase decrescente, 465
        prevenzione delle anomalie, 466
        stretto, 467
    ottimistico, 375
        JPA, 392
    pessimistico, 375
log e logging, 448

logica, progettazione, 194, 277–313
long transaction, 509
lower, 158

M
manipolazione dei dati in SQL, 137–140
MapReduce, 625, 627, 636, 736
markup, 524
materializzazione di viste, 717
max, 121, 132
membro di una classe, 493
memoria
secondaria, 402
stabile, 447
merge, 423
scan (metodo di join), 432
message bean, 661
metadati, 701
metamodello, 219
metodo (OO), 482, 488
accessore, 503
costruttore, 503
di accesso, 409
gestore dei, 401
di join
hash join, 432
merge scan, 432
nested loop, 432
in SQL-3, 503
in UML, 220
trasformatore, 503
metodologia di progettazione, 193–196
generale, 257, 258
migrazione di oggetti, 493, 496
min, 121, 132
minimalità di uno schema E-R, 256
minus, 128
mista, strategia, 254, 255
Model-View-Controller, 654
modellazione dei dati
in UML, 218–227
nel modello E-R, 196–213
modello
a oggetti, 7
basato su record e puntatori, 21
basato su valori, 21
concettuale, 7, 194
traduzione da – a modello logico, 194, 277, 294–301
dell'applicazione, 219
dei dati, 6–7
entità-relazione, 7, 196–213
attributi nel, 202–203
chiave nel, 207
costrutti del, 196
dizionario dei dati nel, 216
entità, 198, 199
generalizzazioni nel, 209–212
identificatori nel, 207, 209
istanza nel, 196
relazioni nel, 199–202
vincoli di integrità nel, 204, 214, 215
E-R, 482, 492, 506
fail-stop, 452
fisico, 194
gerarchico, 6
logico, 8, 194
traduzione da modello concettuale a, 194, 277, 294–301
multidimensionale, 702–707
relazionale, 6, 15–42
traduzione verso il, 294–301
reticolare, 6
modifica degli schemi in SQL, 102
MOLAP, 700, 708
molteplicità di associazioni, 223, 224
motore
di ricerca, 511
JavaScript, 675
multi-indice, 507
multi-insieme, 483
multi-tier, 658
multidatabase, 613
multidimensionale
analisi, 701
base di dati, 702–707
cubo, 704
modello dei dati, 702–707
multimediale, base di dati, 509

N
negazione in Datalog, 83
nested loop (metodo di join), 432
nested transaction, 509
.NET, 654
nidificato, indice, 507
nil, 482
no action, 101
normalizzazione, 194, 323–354
not final, 500
not null, 98
notazione punto, 484
nullif, 159
nullo, valore, 25–28, 33, 34, 65–67
numeric, 93
n-upla, 17

O

object
- identifier, 482, 484
- object server, 505, 508

occorrenza nel modello E-R, 196
ODBC, 364–367, 579
ODMG, 490
OLAP, 697–698
OLE DB, 367
OLTP, 697–698
On Line Analytical Processing, 697
On Line Transaction Processing, 697
`open`, 358
Open Geospatial Consortium, 514
operatori
- aggregati in SQL, 120–123
- dell'algebra relazionale, 44

operazioni batch e interattive, 279
Oracle, trigger in, 169–171
ORDBMS, 499–505
`order by`, 119, 535
ordinamento, 430
- in SQL, 119

ordinata, struttura sequenziale, 411
orizzontale, approccio, 505
ORM, 379
ottimizzazione delle interrogazioni, 427–436
- distribuite, 588, 591

`outer join`, 115–117
overloading, 496
overriding, 495

P

PaaS, 625
page header e page trailer, 407
pagina
- dizionario di, 407
- gestione delle tuple nella, 407, 408

partecipazione obbligatoria e facoltativa
- a relazione, 205

partizionamento
- di associazioni, 282, 292, 293
- di entità, 282, 290–292

path
- expression, 397, 507
- indice, 507

pattern, 532, 729
- di progetto concettuale, 242–249
- sequenziali, 732

Paxos, commit, 605
PCDATA, 528
perdita di aggiornamento, 456
persistenza, 185
- di oggetti, 495

Pig Latin, 628
PL/SQL, 162
pointer swizzling, 508
polimorfismo dei metodi, 490
portabilità, 579
Postgres, 157, 356
precisione, 512
predicato
- estensionale, 80
- intensionale, 80

`prepare`, 362
prestazioni su schemi E-R
- analisi di, 278–282

primario, indice, 416
`primary key`, 99
primitive di trasformazione
- bottom-up, 252
- top-down, 251, 252

privatezza dei dati, 4, 179
privato, metodo, 489
privilegi in SQL, 179
procedure in SQL, 160–163
prodotto cartesiano (operatore), 56
profili delle relazioni, 428
profonda, uguaglianza, 485
progettazione
- concettuale, 194, 235, 265, 343, 344
  - pattern di, 242–249
- dei dati, 192
- delle funzioni, 192
- di basi di dati, 191–321
- dimensionale, 720–722
- di sistemi informativi, 191
- di un data warehouse, 717–728
- fisica, 194, 436–440
- logica, 194, 277–313, 343, 344
- metodologia di, 193
  - generale di, 257, 258

proiezione (operatore), 49–51, 63
prolog, 79
prototipizzazione di sistemi informativi, 193
pubblico, metodo, 489
puntatore, 506
punto fisso, 82, 83
puntuale, accesso, 415

Q

quad-tree, 514
qualità
- dei dati, 702
- di uno schema E-R, 255–257

quantificatore, 70, 71, 74, 77

R
raccolta dei requisiti, 191, 235–239
raffinamento
di attributi e metodi, 492
di tipo, 497
raggiungibilità di oggetti, 495
raggruppamento in SQL, 123–127
range list, 76
rappresentazione
concettuale di dati, 240–249
criteri di, 241, 242
schemi relazionali, 304
`read`
`committed`, 459
`uncommitted`, 459
ready (nel commit a due fasi), 598
`real`, 93
recall, 512
record-of, 482, 483
recovery, 4
redo, 449, 454
idempotenza di, 449
`ref`, 500
`references`, 100–102, 179
referenziale, vincoli di integrità, 35–38, 100–102
regole
attive, 163
aziendali, 163, 175, 177, 214–216
Datalog, 80
ricorsive, 80, 82
di associazione, 730–732
XSL, 534
reificazione, 222, 242
relazione, 16–25
attributo di, 19
derivata, 67
di base, 67
istanza di, 23
nel modello E-R, 199–202
cardinalità di, 204–206
molti a molti, 205
ricorsive, 201
uno a molti, 205
uno a uno, 205
schema di, 22
virtuale, 67
`repeatable read`, 459
replicazione, trasparenza di, 587
Representational State Transfer (REST), 667
requisiti
analisi dei, 191, 235–239
di una base di dati, 194
`return`, 535, 536
`returns`, 504
`revoke`, 180
RIA, 671, 672
ricerca di documenti, 511
Rich Internet Applications, 672
richiamo, 512
richieste
distribuite, 589
remote, 589
ricorsione, 484
ricorsive
interrogazioni – in SQL, 156, 157
relazioni, 201
ridefinizione, 494
dei metodi, 492, 495–499
ridenominazione (operatore), 44–47
ridondanze, 3, 324
analisi delle, 282–286
in uno schema E-R, 256
riempimento, fattore di, 413
riferimenti
a oggetti, 484
`right join`, 115, 116
ripresa a caldo e a freddo, 453–455
ristrutturazione di schemi E-R, 277, 282–294
ROLAP, 700, 708
rollback, 182
roll-up, 705–706, 713
ruoli, 181

S
SaaS, 625
scansione (operazione), 430
scelta degli identificatori
primari, 293, 294
principali, 282
schedule, 460
conflict-serializzabile, 462
seriale, 461
serializzabile, 461
view-serializzabile, 461
scheduler, 460
schema
a fiocco di neve, 711
a stella, 708
concettuale, 194, 196
proprietà di uno, 255–257
definizione di – in SQL, 95
del modello E-R, 196
di base di dati, 7
di operazione, 281
di relazione, 22
rappresentazione di uno, 304
E-R
analisi di prestazioni su, 278–282

completezza di, 256
correttezza di, 256
documentazione di, 213–216
leggibilità di, 256
minimalità di, 256
proprietà di uno, 255
ristrutturazione di, 277, 282, 294
esterno, 8
fisico, 16, 194, 196
interno, 8
logico, 8, 16, 194, 196
scheletro, 255, 258
script a lato client, 675
`scroll`, 358
secondario, indice, 416
segnatura, 488, 490, 503
di un documento, 513
`select`, 106–108, 179
selezione (operatore), 47–49, 62, 63
`self`, 490
Semantic Web, 663
sequenziale ordinata
struttura, 411
seriale
struttura, 409
`serializable`, 459
serializzabilità, 461
globale, 592
server side scripting, 651
session bean, 661
`set`
`constraints`, 151
`default`, 101
`dirty`, 405
`null`, 101
`role`, 181
set-of, 483
single table per hierarchy, 385
sistema
di gestione di basi di dati, 3–6
di supporto alle decisioni, 697–737
informatico, 1
informativo, 1
analisi dei requisiti nel, 191
ciclo di vita del, 191–193
collaudo di, 193
funzionamento di, 193
geografico, 481, 513
implementazione di, 192
progettazione di, 192
prototipizzazione, 193
raccolta dei requisiti nel, 191
validazione di, 193
site view, 684
slice-and-dice, 705
`smallint`, 93
snowflake schema, 711
SOAP, 663
sorgente di dati, 697, 699
sotto-tipazione, 497
spaziali, gestione di dati, 513
specializzazione, 210, 493
split, 422
SQL
alias in, 118
asserzioni in, 150
costruttore di tupla in, 135
definizione
di domini in, 96
di indici in, 426
di schema in, 95
di tabella in, 96
dizionario dei dati in, 104
domini elementari di, 92–94
duplicati in, 114
embedded, 356
funzioni scalari in, 157–160
gestione della concorrenza in, 459, 460
interpretazione algebrica di, 113
interrogazioni, 106–137
insiemistiche in, 128–130
ricorsive in, 156, 157
join interni ed esterni in, 115–118
manipolazione dei dati in, 137–140
modifica degli schemi in, 102
operatori aggregati in, 120–123
ordinamento in, 119
privilegi in, 179
procedure, 160–163
raggruppamento in, 123–127
standardizzazione di, 89–91
trigger in, 163
valori
di default in, 97, 98
nulli in, 112, 113
variabili in, 118, 119, 131, 133–135
vincoli di integrità in, 98–102
viste in, 152–157
SQL-2, 90, 94, 97, 105, 112, 115, 117, 149, 156–158, 160, 161, 357
SQL-3, 90, 143, 156, 161, 163, 499–505, 548
domini introdotti in, 94
ruoli in, 181
trigger in, 164–167
SQL-89, 89, 91, 112
SQL:1999, 90, 501
SQL:2003, 90, 95
SQL:2006, 90
SQL:2008, 90
SQL/XML, 548

sqlca, 356
sqlcode, 356
staging area, 701
stallo, 475–477
standardizzazione di SQL, 89–91
star schema, 708
strategia di progetto, 250–255, 257
    bottom-up, 252–253
    inside-out, 253–254
    mista, 254–255
    top-down, 250
struts, 656
struttura
    ad albero, 415–425
    con accesso calcolato, 411–415
    hash, 411–415
    sequenziale, 409
        ad array, 410
        ordinata, 411
    seriale, 409
studio di fattibilità, 191
substring, 158
sum, 121
superchiave, 31, 297
superficiale, uguaglianza, 485
surrogato, 506
sviluppo basato sui dati, 193
swizzling, pointer, 508

T
tabella, 16–17
    definizione di – in SQL, 96
    dei fatti, 708
    dimensione, 709
table per concrete class, 385
tag, 524
    eseguibile, 653
target list, 70
    in SQL, 107
tavola
        degli accessi, 281, 285
        dei volumi, 280, 285
        delle operazioni, 280, 285
TCP/IP, 640
template, 490
    di pagina, 651
tempo
    di accesso a un blocco, 403
    di latenza, 403
    di posizionamento della testina, 403
temporaneo, oggetto, 495
terminazione dei trigger, 172
theta-join, 56–58
time, 94
timestamp, 94
    controllo di concorrenza su, 468–470
    in contesto distribuito, 593
tipo
    di dato astratto, 486
    complesso, 482
    in SQL-3, 500
    nei linguaggi OO, 482–486
    record, 482
top-down
    primitive di trasformazione, 251
    strategia, 250–257
traduzione
    da modello concettuale a logico, 194, 277, 294–301
    equivalente, 294
transazione, 182, 185, 445
    ben formata, 183
    complessa, 509
    di compensazione, 509
    distribuita, 589
    lunga, 509
    negli OODB, 508
    remota, 589
trasformatore, metodo, 489, 491, 503
trasparenza
    di allocazione, 587
    di frammentazione, 587
    di linguaggio, 588
    di replicazione, 587
trigger, 163
    terminazione dei, 172
tupla, 20
    costruttore di – in SQL, 135
    dangling, 53
    gestione della – nelle pagine, 407–408

U
UDDI, 663
uguaglianza di oggetti, 485
UML, 219–227, 265
    aggregazioni in, 222
    associazioni in, 219, 221–223
    attributi in, 220
    classi in, 219, 220
    composizioni in, 223
    generalizzazioni in, 225, 226
    identificatori in, 224, 225
    metodi in, 220
undo, 449, 454
    idempotenza di, 449
unfix, 406
union, 128

unione (operatore), 44
`unique`, 98
unlock, 464
`update`, 102, 139, 140, 179
`upper`, 158
URL, 644
`usage`, 180
utenti, 12–13

V

validità di documenti XML, 527
validazione di sistemi informativi, 193
valore
    nullo, 25–28, 33–34, 65–67
        in SQL, 112, 113
    di default in SQL, 97, 98
variabili in SQL, 118, 119, 131, 133–135
versione, 509
verticale, approccio, 506
video, 510
view-equivalenza, 461
view-serializzabilità, 461
vincoli di integrità, 28–31
    di tupla, 30, 31
    in SQL, 98–102
    intrarelazionali, 98, 99
    nel modello E-R, 204
    referenziale, 35–38, 100–102
virtuale, relazione, 67
vista, 9, 67–69
    in SQL, 152–157
    materializzata, 67
    ricorsiva, 156, 157
volume dei dati, 279, 280

W

Web Application Programming Interface (Web API), 667
Web service, 661, 671
Web SQL Database API, 682
Web Storage API, 680
`where`, 110
    in XQuery, 535, 538
WIS, 639
Write Ahead Log (WAL), 450
WSDL, 662, 665–666

X

X-Open DTP, 579, 606
XML, 523–550
    in DBMS relazionali, 546
    Schema Definition, 529
XPath, 531, 538
XQuery, 530, 535–546
    3.0, 545
XSD, 529
XSL, 532–535

Questo volume, sprovvisto del talloncino a fronte, è da considerarsi copia saggio-campione gratuito fuori commercio. Fuori campo applicazione IVA ed esente da bolla di accompagnamento (art. 22 L.67/1987, art. 2, lett. I D.P.R. 633/1972 e art. 4 n. 6 D.P.R. 627/1978).

P. Atzeni - S. Ceri - P. Fraternali - S. Paraboschi - R. Torlone
BASI DI DATI
Quarta edizione
McGraw-Hill Education (Italy), S.r.l.
978 88 386 6587-5