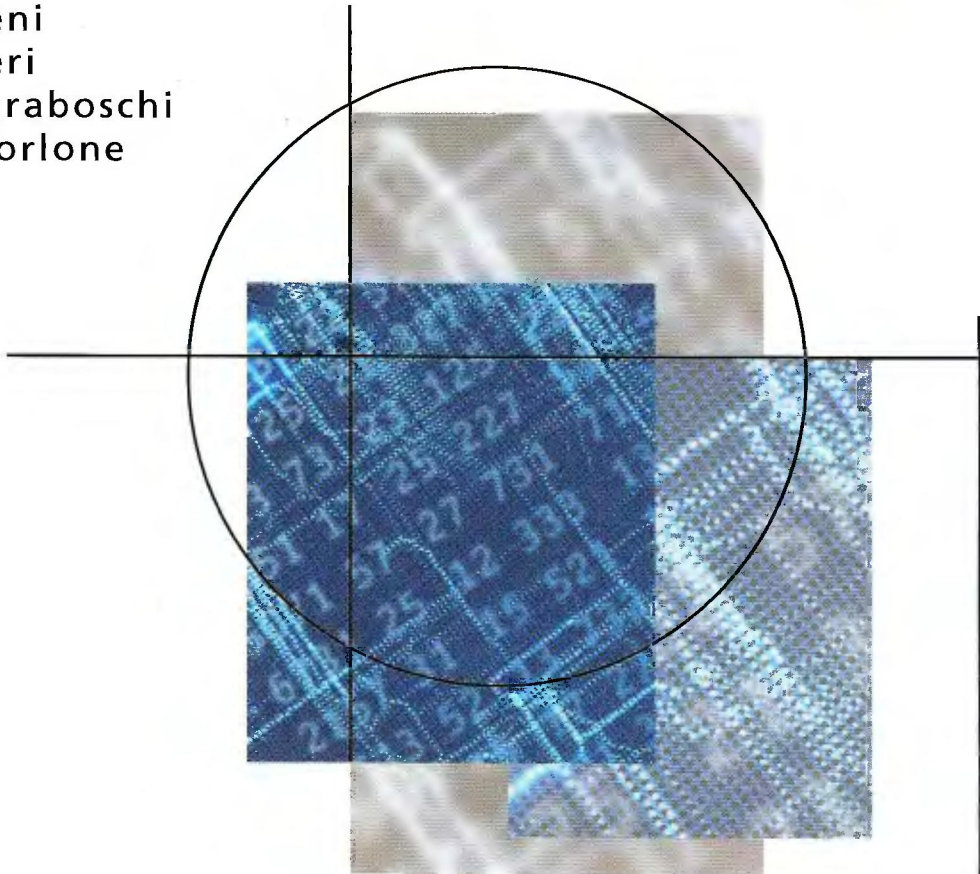
Paolo Atzeni
Stefano Ceri
Stefano Paraboschi
Riccardo Torlone



Basi di dati

Modelli e linguaggi di interrogazione

Seconda edizione

McGraw-Hill

web site

collana di istruzione scientifica
serie di informatica

Al lettore

La realizzazione di un libro comporta costi variabili (carta, stampa e legatura) e costi fissi, cioè indipendenti dal numero di copie stampate (traduzione, preparazione degli originali, redazione, composizione, impaginazione). I fotocopiatori possono contenere il prezzo perché, oltre a non pagare i diritti d'autore, non hanno costi fissi. Ogni fotocopia, d'altra parte, riducendo il numero di copie vendute dall'editore, aumenta l'incidenza dei costi fissi a copia e costringe l'editore ad aumentare il prezzo; questo naturalmente fornisce un ulteriore incentivo a fotocopiare. Se questo circolo vizioso non verrà spezzato, arriveremo al punto in cui gli editori non avranno più convenienza economica a realizzare libri di testo per l'università.

In quel momento non ci saranno più neppure fotocopie.

L'editore

Paolo Atzeni
Stefano Ceri
Stefano Paraboschi
Riccardo Torlone

Basi di dati

Modelli e linguaggi di interrogazione

Seconda edizione

McGraw-Hill

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

Copyright © 2002, 2006 The McGraw-Hill Companies, S.r.l.
Publishing Group Italia
via Ripamonti 89 – 20139 Milano

McGraw-Hill
A Division of The McGraw-Hill Companies

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.

Date le caratteristiche intrinseche di Internet, l'Editore non è responsabile per eventuali variazioni negli indirizzi e nei contenuti dei siti Internet riportati.

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

Editor: Chiara Tartara
Produzione: Donatella Giuliani
Impaginazione: a cura degli Autori
Grafica di copertina: G & G
Stampa: Cromografica Europea, Rho (MI)

ISBN 88-386-6292-4
Printed in Italy
123456789CROLIL09876

Indice

Prefazione xi

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 8
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 12

Parte Prima Basi di dati relazionali: modello e linguaggi 15

2 Il modello relazionale 17
2.1 Il modello relazionale: strutture 17
2.1.1 Modelli logici nei sistemi di basi di dati 17
2.1.2 Relazioni e tabelle 18
2.1.3 Relazioni con attributi 20
2.1.4 Relazioni e basi di dati 23
2.1.5 Informazione incompleta e valori nulli 28
2.2 Vincoli di integrità 31
2.2.1 Vincoli di tupla 33
2.2.2 Chiavi 34
2.2.3 Chiavi e valori nulli 36
2.2.4 Vincoli di integrità referenziale 38
2.3 Conclusioni 41

3 Algebra e calcolo relazionale 45
3.1 Algebra relazionale 46
3.1.1 Unione, intersezione, differenza 46
3.1.2 Ridenominazione 47
3.1.3 Selezione 50

3.1.4 Proiezione 52
3.1.5 Join 53
3.1.6 Interrogazioni in algebra relazionale 61
3.1.7 Equivalenza di espressioni algebriche 65
3.1.8 Algebra con valori nulli 68
3.1.9 Viste 70
3.2 Calcolo relazionale 72
3.2.1 Calcolo relazionale su domini 73
3.2.2 Pregi e difetti del calcolo su domini 78
3.2.3 Calcolo su tuple con dichiarazioni di range 80
3.3 Datalog 83
ESERCIZI 87

4 SQL: Concetti base **91**
4.1 Il linguaggio SQL e gli standard 91
4.2 Definizione dei dati in SQL 94
4.2.1 I domini elementari 94
4.2.2 Definizione di schema 97
4.2.3 Definizione delle tabelle 98
4.2.4 Definizione dei domini 99
4.2.5 Specifica di valori di default 99
4.2.6 Vincoli intrarelazionali 100
4.2.7 Vincoli interrelazionali 102
4.2.8 Modifica degli schemi 105
4.2.9 Cataloghi relazionali 106
4.3 Interrogazioni in SQL 108
4.3.1 Dichiaratività di SQL 108
4.3.2 Interrogazioni semplici 109
4.3.3 Operatori aggregati 123
4.3.4 Interrogazioni con raggruppamento 126
4.3.5 Interrogazioni di tipo insiemistico 130
4.3.6 Interrogazioni nidificate 133
4.4 Manipolazione dei dati in SQL 140
4.4.1 Inserimento 140
4.4.2 Cancellazione 141
4.4.3 Modifica 143
4.5 Esempi riepilogativi 144

5 SQL: Caratteristiche evolute **151**
5.1 Caratteristiche evolute di definizione dei dati 151
5.1.1 Vincoli di integrità generici 151
5.1.2 Asserzioni 152
5.1.3 Viste 154
5.1.4 Le viste per la scrittura di interrogazioni 155
5.1.5 Esempi riepilogativi d'uso delle viste 157
5.1.6 Viste ricorsive in SQL-3 158
5.2 Funzioni scalari 159

5.2.1 Famiglie di funzioni 159
5.2.2 Funzioni condizionali 160
5.3 Controllo dell'accesso 162
5.3.1 Risorse e privilegi 163
5.3.2 Comandi per concedere e revocare privilegi 164
5.3.3 I ruoli in SQL-3 165
5.4 Transazioni 166

6 SQL per le applicazioni **171**
6.1 Procedure 172
6.2 Trigger 174
6.3 SQL Embedded 176
6.3.1 Cursori 179
6.3.2 SQL dinamico 182
6.4 Call Level Interface (CLI) 184
6.4.1 ODBC e soluzioni proprietarie Microsoft 185
6.4.2 Java Database Connectivity (JDBC) 190

Parte Seconda Progettazione di basi di dati **197**

7 Metodologie e modelli per il progetto **199**
7.1 Introduzione alla progettazione 199
7.1.1 Il ciclo di vita dei sistemi informativi 199
7.1.2 Metodologie di progettazione e basi di dati 202
7.2 Il modello Entità-Relazione 206
7.2.1 I costrutti principali del modello 206
7.2.2 Altri costrutti del modello 212
7.2.3 Panoramica finale sul modello E-R 220
7.3 Documentazione di schemi E-R 222
7.3.1 Regole aziendali 223
7.3.2 Tecniche di documentazione 225
7.4 Modellazione dei dati in UML 227
7.4.1 Panoramica su UML 228
7.4.2 Rappresentazione di dati con i diagrammi delle classi . . . 229

8 La progettazione concettuale **243**
8.1 La raccolta e l'analisi dei requisiti 243
8.2 Criteri generali di rappresentazione 249
8.3 Strategie di Progetto 250
8.3.1 Strategia top-down 250
8.3.2 Strategia bottom-up 253
8.3.3 Strategia inside-out 255
8.3.4 Strategia mista 257
8.4 Qualità di uno schema concettuale 258
8.5 Una metodologia generale 260

8.6 Un esempio di progettazione concettuale 261
8.7 Strumenti CASE per la progettazione di basi di dati 265

9 La progettazione logica **273**
9.1 Analisi delle prestazioni su schemi E-R 274
9.2 Ristrutturazione di schemi E-R 278
9.2.1 Analisi delle ridondanze 278
9.2.2 Eliminazione delle generalizzazioni 282
9.2.3 Partizionamento/accorpamento di concetti 286
9.2.4 Scelta degli identificatori principali 289
9.3 Traduzione verso il modello relazionale 291
9.3.1 Entità e associazioni molti a molti 291
9.3.2 Associazioni uno a molti 293
9.3.3 Entità con identificatore esterno 295
9.3.4 Associazioni uno a uno 295
9.3.5 Traduzioni di schemi complessi 297
9.3.6 Tabelle riassuntive 299
9.3.7 Documentazione di schemi logici 300
9.4 Un esempio di progettazione logica 302
9.4.1 Fase di ristrutturazione 303
9.4.2 Traduzione verso il relazionale 308
9.5 Progettazione logica con gli strumenti CASE 308
9.6 Progettazione fisica di una base di dati 310

10 La normalizzazione **317**
10.1 Ridondanze e anomalie 318
10.2 Dipendenze funzionali 319
10.3 Forma normale di Boyce e Codd 321
10.3.1 Definizione di forma normale di Boyce e Codd 321
10.3.2 Decomposizione in forma normale di Boyce e Codd . . . 322
10.4 Proprietà delle decomposizioni 323
10.4.1 Decomposizione senza perdita 324
10.4.2 Conservazione delle dipendenze 326
10.4.3 Qualità delle decomposizioni 327
10.5 Terza forma normale 328
10.5.1 Limitazioni della forma normale di Boyce e Codd 328
10.5.2 Definizione di terza forma normale 329
10.5.3 Decomposizione in terza forma normale 329
10.5.4 Altre forme normali 331
10.5.5 Normalizzazione e scelta degli attributi 332
10.6 Teoria delle dipendenze e normalizzazione 333
10.6.1 Implicazione di dipendenze funzionali 333
10.6.2 Coperture di insiemi di dipendenze funzionali 336
10.6.3 Sintesi di schemi in terza forma normale 337
10.7 Progettazione di basi di dati e normalizzazione 339
10.7.1 Verifiche di normalizzazione su entità 339

10.7.2 Verifiche di normalizzazione su associazioni 341
10.7.3 Ulteriori decomposizioni di associazioni 342
10.7.4 Ulteriori decomposizioni di schemi concettuali 344

A Microsoft Access **349**
A.1 Caratteristiche del sistema 350
A.2 La definizione delle tabelle 351
A.2.1 Specifica dei cammini di join 356
A.2.2 Popolamento delle tabelle 358
A.3 La definizione di query 359
A.3.1 Query By Example 359
A.3.2 L'interprete SQL 365
A.4 Maschere e report 366
A.5 La definizione di macro 369

B DB2 Universal Database **371**
B.1 Caratteristiche generali di DB2 372
B.1.1 Versioni del sistema 372
B.1.2 Istanze e schemi di DB2 373
B.1.3 Interazione con DB2 374
B.2 Gestione di una base di dati con DB2 374
B.2.1 Strumenti per la gestione interattiva 374
B.2.2 Applicazioni 381
B.3 Funzionalità avanzate di DB2 385
B.3.1 Dati complessi 385
B.3.2 Extender 386
B.3.3 Tipi utente 387
B.3.4 Funzioni utente 388

Bibliografia **391**

Indice analitico **395**

Prefazione

Questo testo presenta i concetti fondamentali sulle basi di dati, sui linguaggi di interrogazione e gestione, e sulle tecniche e metodi di progettazione. Esso nasce da una lunga esperienza di insegnamento in corsi riguardanti le basi di dati, sia in ambito universitario sia in ambito industriale e applicativo, e pertanto si rivolge sia al pubblico degli studenti (in particolare di Ingegneria e di Scienze dell'informazione o Informatica) sia a quello dei professionisti (utenti e progettisti di applicazioni). Questa nuova edizione tiene conto anche delle prime esperienze di attuazione del Nuovo Ordinamento didattico universitario, avviato intorno al 2000, con una frequente offerta di un primo modulo di basi di dati al terzo o anche secondo anno di corso.

Contenuti

Il libro si articola in due 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. Vengono illustrati il modello relazionale e i relativi linguaggi, in modo preciso e concreto, con riferimento tanto alle definizioni formali (del modello e dell'algebra e del calcolo) e ai sistemi esistenti (con riferimento soprattutto al linguaggio SQL).

Progettazione di basi di dati. Viene illustrato ed esemplificato il processo di progettazione concettuale e logica delle basi di dati relazionali, che permette, partendo dai requisiti di utente, di arrivare a produrre strutture di basi di dati di buona qualità.

Nelle appendici vengono illustrate le caratteristiche salienti di alcuni sistemi relazionali molto diffusi.

Ciascun capitolo è corredato di esempi ed esercizi proposti, 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 primo corso di basi di dati, corrispondente cioè ad un modulo da cinque-sei crediti (circa trenta ore di lezione e venti di

esercitazione). Ad essi è opportuno associare un'ampia attività pratica, in particolare lo svolgimento di un progetto di un piccolo sistema informativo che includa una base di dati. Informazioni utili circa l'organizzazione di alcuni DBMS relazionali sono disponibili in appendice; in particolare l'appendice A, dedicata al sistema Access di Microsoft, ne illustra l'uso in modo assai semplice e diretto, tale da facilitare una prima esperienza di progetto.

Il presente testo ha una naturale continuazione nel testo "Basi di dati: architetture e linee di evoluzione" [5], che può essere utilizzato per un secondo modulo più avanzato oppure, insieme a questo, per un eventuale corso di dieci-dodici crediti. Tale testo, cui nel seguito faremo spesso riferimento come "il secondo volume," copre i seguenti argomenti:

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

Evoluzione delle basi di dati. Vengono illustrate le principali tendenze di sviluppo della tecnologia delle basi di dati, con riferimento a nuovi modelli, nuove funzionalità e integrazione con Internet.

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 ora i corsi di Basi di dati, di Tecnologia delle basi di dati e di Sistemi informativi presso la Facoltà di Ingegneria dell'Università Roma Tre. Stefano Ceri tiene i corsi di Basi di dati 1, Basi di dati 2, e Argomenti avanzati di sistemi informativi presso la Facoltà di Ingegneria dell'informazione del Politecnico di Milano. Ha inoltre tenuto varie edizioni del corso "Principles of Distributed Databases" presso l'Università di Stanford. Stefano Paraboschi tiene corsi di Basi di dati e Sistemi informativi presso le Facoltà di Ingegneria dell'Università di Bergamo e del Politecnico di Milano. Riccardo Torlone tiene corsi di Basi di dati presso la Facoltà 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, Luca Cabibbo, Alessandro Campi, Paolo De Nictolis, Giuseppe Di Battista, Angelo Foglietta, Piero Fraternali, Maurizio Lenzerini, Gianni Mecca, Paolo Merialdo, Barbara Pernici, 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. Siamo ancor più debitori nei confronti dei nostri studenti, che hanno sopportato per anni corsi svolti con dispense parziali e incoerenti, contribuendo spesso al loro miglioramento.

Nota di edizione

Questa nuova edizione esce a dieci anni di distanza dalla pubblicazione del volume: "Basi di dati: concetti, linguaggi e architetture", che ha visto la sua seconda edizione e una edizione internazionale nel 1999. Da allora, abbiamo suddiviso il materiale in due volumi, le cui prime edizioni sono uscite nel 2003 e 2004; presentiamo ora la seconda edizione del primo volume, e prevediamo un aggiornamento del secondo volume nel 2007.

Rispetto al suo predecessore, le modifiche sono così riassunte:

- Il capitolo 4 sull'SQL è stato esteso e di conseguenza diviso in due parti (capitoli 4 e 5). La nuova presentazione tiene conto degli elementi contenuti nelle edizioni recenti SQL:1999 e SQL:2003 dello standard SQL-3. Il nuovo capitolo 4 è dedicato agli aspetti fondamentali e ai dettagli del linguaggio di interrogazione, mentre il capitolo 5 sviluppa aspetti complementari, soprattutto di definizione dei dati.
- Il capitolo 6 è una rielaborazione ed estensione del vecchio capitolo 5 e tratta non solo l'interazione fra SQL e linguaggi di programmazione, aggiornata con riferimento allo sviluppo delle tecnologie, ma introduce anche i trigger, che sono una funzionalità interessante per lo sviluppo delle applicazioni, approfondita comunque nel secondo volume.
- Nel capitolo 6 dell'edizione precedente (che è diventato il capitolo 7) è stata aggiornata ed estesa la sezione che mette in luce il legame tra la progettazione concettuale dei dati, svolta usando il modello Entità-Relazione, e la progettazione a oggetti tramite UML (Unified Modeling Language), che si è diffuso sempre più quale standard nella specifica e nella progettazione dei sistemi software. In particolare è stata introdotta una descrizione accurata del modo in cui i diagrammi delle classi di UML possono essere usati per rappresentare i diagrammi E-R.
- Il capitolo sulla normalizzazione (ora capitolo 10) è stato esteso e significativamente rivisto. In particolare sono discussi alcuni aspetti tecnici sulle proprietà fondamentali delle dipendenze e sulla decomposizione delle relazioni in terza forma normale.
- Le appendici relative ai sistemi sono state aggiornate con riferimento alle nuove versioni dei sistemi presi in considerazione.

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, sono disponibili lucidi che coprono i dieci capitoli della presente edizione in modo completo, ulteriori esercizi, le soluzioni di alcuni esercizi, l'appendice A della edizione del 1996 (relativa al modello reticolare e omessa dalle edizioni successive). Per segnalazioni di errori e altri suggerimenti, gli Autori possono essere contattati attraverso i siti stessi.

1

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 anni Ottanta e ancor più gli anni Novanta, 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 parlata, 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 [66]:

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 prossimo paragrafo.

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, 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 una università, le informazioni relative ai docenti possono essere utilizzate in vario modo e da diverse persone o uffici; ad 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" (ad 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, ad 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 retri-

buzioni, 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. Ad 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 qualche applicazione. Preferiamo però dare una definizione che abbia un contenuto anche tecnologico, attraverso le caratteristiche dei sistemi utilizzati per gestire tali collezioni.

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 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 sul calcolatore 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 (ad 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 (ad 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, riconosciuto in base a un nome d'utente che è specificato all'atto di interagire con il DBMS, viene abilitato a svolgere solo determinate azioni sui dati, attraverso meccanismi di *autorizzazione*.
- Per *efficienza* intendiamo la capacità 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.
- Per *efficacia* intendiamo la capacità della base di dati di rendere produttive, in ogni senso, le attività dei suoi utenti. Questa definizione è chiaramente generica e non corrisponde a un aspetto specifico, dato che un DBMS fornisce vari servizi e funzionalità a utenti differenti. 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, e cioè facoltà, 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 tale base di dati 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 [5]. Esistono varie organizzazioni architetturali per l'accesso a basi di dati: ad 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 (elementari) predefiniti 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 del 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.

DOCENZA

Corso	NomeDocente
Basi di dati	Rossi
Impianti	Neri
Linguaggi	Verdi

MANIFESTO

CdL	Materia	Anno
II	Basi di dati	5
II	Impianti	5
II	Linguaggi	4
IE	Basi di dati	5
IE	Impianti	5

Figura 1.1 Esempio di base di dati relazionale

Il modello relazionale, definito formalmente agli inizi degli anni Settanta ma affermatosi negli anni Ottanta, 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 tre 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 ampiamente 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);
- 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 [5].

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, 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 nei 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; nell'esempio abbiamo le tre coppie:

Basi di dati	Rossi
Impianti	Neri
Linguaggi	Verdi

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 sviluppate ulteriormente 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: relazionale, gerarchico, reticolare o a oggetti).

[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.

- 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. Vedremo l'organizzazione fisica dei dati nel secondo volume [5], dove tratteremo degli aspetti tecnologici.
- 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.

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 (IE) 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.

ELETTRONICA

CdL	Materia	Anno
IE	Basi di dati	5
IE	Impianti	5

Figura 1.2 Una "vista" relazionale

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, per esempio il linguaggio SQL;
- tramite comandi simili a quelli interattivi immersi in linguaggi di programmazione tradizionali, quali C, C++, COBOL, FORTRAN, e così via, come vedremo nel capitolo 6. Questi linguaggi di programmazione si dicono *linguaggi ospite* perché "ospitano" comandi scritti nel linguaggio per basi di dati;
- 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 potremo quindi solo accennare ad alcuni aspetti nelle appendici, 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, accenneremo nelle appendici agli aspetti più semplici e importanti.

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 la 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.
- 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 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.
- 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 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 quelli effettivamente richiesti dagli altri, e ciò può comportare una riduzione di prestazioni.

Concludendo, possiamo dire che si incontrano situazioni nelle quali la 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 [31], Silberschatz, Korth e Sudarshan [57], Ramakhrishnan e Gehrke [53] e Garcia-Molina, Ullman e Widom [35], che coprono in modo equilibrato gli aspetti metodologici e quelli tecnologici. Il testo di Date [28], molto diffuso e ormai

alla ottava edizione, presenta in modo semplice molti aspetti importanti. Quello di Ullman [64] tratta in modo integrato tecnologia esistente e aspetti di natura teorica. Per dettagli relativi ai singoli aspetti citati in questo capitolo, rimandiamo ai capitoli successivi, in cui verranno approfonditi, e alle relative note bibliografiche.

Parte Prima

Basi di dati relazionali: modello e linguaggi

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 [23]), 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 proprio 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 tre capitoli, questo e i due 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 due capitoli successivi, il primo dedicato alla specifica delle operazioni di interrogazione di basi di dati relazionali e il secondo all'introduzione 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 dei modelli 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 realizzati per mezzo di opportune strutture fisiche, ma per accedere ai dati non è necessario conoscere le strutture fisiche stesse.

Infatti, i modelli proposti precedentemente al modello relazionale, quello reticolare e quello gerarchico, includevano invece 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 nella discussione della progettazione concettuale (capitolo 7), 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

[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*.

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

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 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 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.

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 una relazione

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

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 una relazione ternari

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.

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

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:

$$\text{Stringa} \times \text{Stringa} \times \text{Intero} \times \text{Intero}$$

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 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 possono essere presenti due elementi uguali; 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 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 (si veda la 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

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

ordine diverso (secondo lo stile americano in cui la squadra di casa viene indicata dopo quella ospitata).

Per formalizzare i concetti, stabiliamo la corrispondenza fra attributi e domini per mezzo 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:

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

[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.

è una tupla su due attributi.[4]

2.1.4 Relazioni e basi di dati

Come già notato, una relazione può essere utilizzata per organizzare dati rilevanti nell'ambito di una 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;
- 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 gli altri modelli logici, reticolare e gerarchico (definiti prima del modello relazionale ma tuttora in uso), 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 (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;

[4] C'è qui una incoerenza nella notazione: se A è un attributo, allora $t[A]$ è un valore, mentre se X è un insieme di attributi, allora $t[X]$ è una tupla, cioè una funzione. Inoltre, come vedremo, insiemi costituiti da un singolo attributo verranno denotati dal nome dell'attributo stesso; quindi, $t[A]$ denota sia un valore sia una tupla su un attributo. Peraltro, l'ambiguità sarà di solito irrilevante e verrà risolta esplicitamente quando necessario.

STUDENTI

Matricola	Cognome	Nome	Data di nascita
276545	Rossi	Maria	25/11/1971
485745	Neri	Anna	23/04/1972
200768	Verdi	Fabio	12/02/1972
587614	Rossi	Luca	10/10/1971
937653	Bruni	Mario	01/12/1971

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

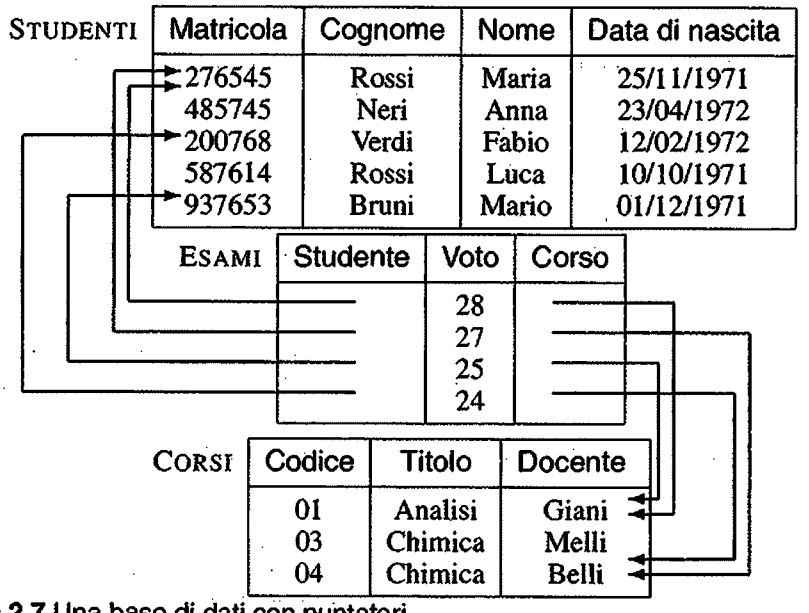


Figura 2.7 Una base di dati con puntatori

SCHEMA: parte invariabile nel tempo
ISTANZA: record

- 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, (discussi nel secondo volume [5]) 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 fissato $R(X)$ è un insieme r di tuple su X.
- Un'*istanza di base di dati* (o semplicemente *base di dati*) su uno schema fissato $\mathbf{R} = \{R_1(X_1), R_2(X_2), \ldots, R_n(X_n)\}$ 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 (si veda la figura 2.8) un'altra relazione sul solo attributo Matricola per indicare gli studenti lavoratori (attraverso i relativi numeri di matricola, che devono apparire nella relazione STUDENTI).

STUDENTI

Matricola	Cognome	Nome	Data di nascita
276545	Rossi	Maria	25/11/1971
485745	Neri	Anna	23/04/1972
200768	Verdi	Fabio	12/02/1972
587614	Rossi	Luca	10/10/1971
937653	Bruni	Mario	01/12/1971

LAVORATORI

Matricola
276545
485745
937653

Figura 2.8 Una relazione su un solo attributo

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 fi-

scali 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 (non sarebbe possibile rappresentare le righe in numero non predeterminato). Possiamo però rappresentare le relative informa-

"DA MARIO"		
Ricevuta n. 1357		
del 5/5/92		
3	coperti	6000
2	antipasti	12000
3	primi	27000
2	bistecche	36000
	Totale	81000

"DA MARIO"		
Ricevuta n. 2334		
del 4/7/92		
2	coperti	4000
1	antipasti	6000
2	primi	15000
2	orate	50000
2	caffè	3000
	Totale	78000

"DA MARIO"		
Ricevuta n. 3002		
del 4/8/92		
3	coperti	6000
2	antipasti	14000
3	primi	20000
1	orate	25000
1	caprese	8000
2	caffè	3000
	Totale	76000

Figura 2.9 Alcune ricevute fiscali

zioni 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 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 a due condizioni:

- 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).

In entrambi i casi, si può risolvere il problema aggiungendo un attributo, che indica la posizione della riga sulla ricevuta (si veda la 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

RICEVUTE

Num	Data	Totale
1357	5/5/92	81000
2334	4/7/92	78000
3002	4/8/92	76000

DETTAGLIO

Num	Q.tà	Descr	Importo
1357	3	coperti	6000
1357	2	antipasti	12000
1357	3	primi	27000
1357	2	bistecche	36000
2334	2	coperti	4000
2334	1	antipasti	6000
2334	2	primi	15000
2334	2	orate	50000
2334	2	caffé	3000
3002	3	coperti	6000
3002	2	antipasti	14000
3002	3	primi	20000
3002	1	orate	25000
3002	1	caprese	8000
3002	2	caffé	3000

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

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 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

RICEVUTE

Num	Data	Totale
1357	5/5/92	81000
2334	4/7/92	78000
3002	4/8/92	76000

DETTAGLIO

Num	Riga	Q.tà	Descr	Importo
1357	1	3	coperti	6000
1357	2	2	antipasti	12000
1357	3	3	primi	27000
1357	4	2	bistecche	36000
2334	1	2	coperti	4000
2334	2	1	antipasti	6000
2334	3	2	primi	15000
2334	4	2	orate	50000
2334	5	2	caffè	3000
3002	1	3	coperti	6000
3002	2	2	antipasti	14000
3002	3	3	primi	20000
3002	4	1	orate	25000
3002	5	1	caprese	8000
3002	6	2	caffè	3000

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

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 mo-

Città	IndirizzoPrefettura
Roma	Via Quattro Novembre
Firenze	NULL
Tivoli	NULL
Prato	NULL

Figura 2.12 Una relazione con valori nulli

mento, 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 Prato, nel momento in cui scriviamo, è da poco stata istituita e non sappiamo se la prefettura sia già 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.

Per una 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 controllare 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 prossimo paragrafo vedremo

STUDENTI

Matricola	Cognome	Nome	Data di nascita
276545	Rossi	Maria	NULL
NULL	Neri	Anna	23/04/1972
NULL	Verdi	Fabio	12/02/1972

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

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.

- 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

STUDENTI

Matricola	Cognome	Nome	Data di nascita
200768	Verdi	Fabio	12/02/1792
937653	Rossi	Luca	10/10/1971
937653	Bruni	Mario	01/12/1971

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

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* 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 le tre categorie più 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 precedente 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 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:

$$\text{Netto} = \text{Importo} - \text{Ritenute}$$

Matricola	Cognome	Nome	Nascita	Corso
4328	Rossi	Luigi	29/04/59	Ing. Informatica
6328	Rossi	Dario	29/04/59	Ing. Informatica
4766	Rossi	Luca	01/05/61	Ing. Civile
4856	Neri	Luca	01/05/61	Ing. Meccanica
5536	Neri	Luca	05/03/58	Ing. Meccanica

Figura 2.15 Una relazione per la discussione delle chiavi

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 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.[5] 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:

- 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 secon-

[5]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.

da) 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; peraltro esso 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.

Matricola	Cognome	Nome	Nascita	Corso
6328	Rossi	Dario	29/04/59	Ing. Informatica
4766	Rossi	Luca	01/05/61	Ing. Civile
4856	Neri	Luca	01/05/61	Ing. Meccanica
5536	Neri	Luca	05/03/58	Ing. Civile

Figura 2.16 Un'altra relazione per la discussione delle chiavi

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 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, 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 la 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 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.

Matricola	Cognome	Nome	Nascita	Corso
NULL	Rossi	Mario	NULL	Ing. Informatica
4766	Rossi	Luca	01/05/61	Ing. Civile
4856	Neri	Luca	NULL	NULL
NULL	Neri	Luca	05/03/58	Ing. Civile

Figura 2.17 Una relazione con valori nulli su tutte le chiavi

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 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 alla base delle chiavi.

Matricola	Cognome	Nome	Nascita	Corso
3976	Rossi	Mario	NULL	Ing. Informatica
4766	Rossi	Luca	01/05/61	Ing. Civile
4856	Neri	Luca	NULL	NULL
5591	Neri	Luca	05/03/58	Ing. Civile

Figura 2.18 Una relazione con chiave primaria

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.[6] 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.

INFRAZIONI

Codice	Data	Agente	Articolo	Prov	Numero
143256	25-10-92	567	44	RM	4E5432
987554	26-10-92	456	34	RM	4E5432
987557	26-10-92	456	34	RM	2F7643
630876	15-10-92	456	53	MI	2F7643
539856	12-10-92	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

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 tu-

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

pla 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:

- 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.[7] Ovviamente, dovremo avere due coppie di attributi, che non potranno avere gli stessi nomi. Per esempio, lo schema potrebbe essere:

INCIDENTI(<u>Codice</u>, 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.

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

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 relativo 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: al-

INFRAZIONI

Codice	Data	Agente	Articolo	Prov	Numero
987554	26-10-92	456	34	RM	2F7643
630876	15-10-92	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

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

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

cuni 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 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 [31], Silberschatz, Korth e Sudarshan [57] e Ramakhrishnan e Gehrke [53], nonché quello di Garcia-Molina, Ullman, e Widom [35]. 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 [4] e in inglese quelli di Ullman [64], Maier [43], Atzeni e De Antonellis [6], Abiteboul, Hull e Vianu [1]. Può essere interessante consultare l'articolo di Codd [23] che contiene la proposta originaria del modello, per constatarne tuttora l'attualità.

Esercizi

Esercizio 2.1 Considerare le informazioni per la gestione dei prestiti di una biblioteca personale. Il proprietario presta libri ai suoi 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.

Esercizio 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.

Esercizio 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 soddisfa i vincoli.

Esercizio 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.

Esercizio 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.

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

Esercizio 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.

Esercizio 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.

PAZIENTI

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

RICOVERI

Paziente	Inizio	Fine	Reparto
A102	2/05/94	9/05/94	A
A102	2/12/94	2/01/95	A
S555	5/10/94	3/12/94	B
B444	1/12/94	2/01/95	B
S555	5/10/94	1/11/94	A

MEDICI

Matr	Cognome	Nome	Reparto
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

3

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 base di dati, sullo stesso schema. Un'interrogazione, invece, è essenzialmente una funzione che, data una 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 prossimo capitolo, 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 operatori, 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 potendo, 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 sugli altri. 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;

- 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;
- 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.

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

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 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

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

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

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 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 $i = 1, \ldots, n$. 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

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

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

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

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.

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

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."

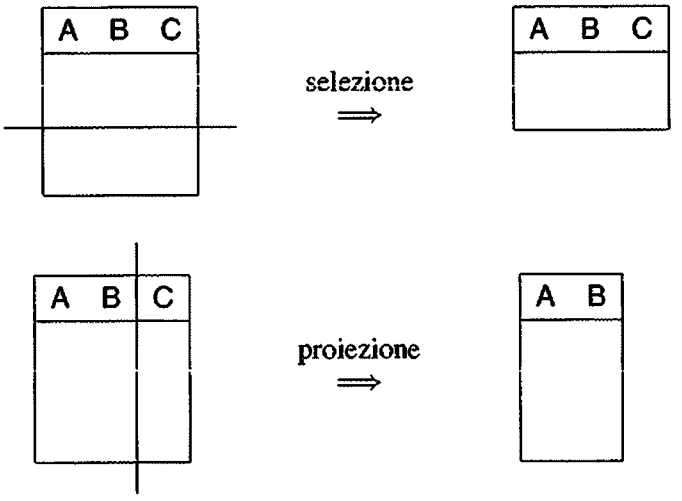


Figura 3.6 Una schematizzazione degli operatori di selezione e proiezione

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

- 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).

IMPIEGATI

Cognome	Nome	Età	Stipendio
Rossi	Mario	25	2.000.000
Neri	Luca	40	3.000.000
Verdi	Nico	36	4.500.000
Rossi	Marco	40	3.900.000

$\sigma_{\text{Età}>30 \wedge \text{Stipendio}>4.000.000}(\text{IMPIEGATI})$

Cognome	Nome	Età	Stipendio
Verdi	Nico	36	4.500.000

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

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

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

Figura 3.8 Un'altra selezione

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

Possiamo a questo punto completare la definizione:

- la selezione $\sigma_F(r)$ 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.

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

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 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:

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

- 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 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.

[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.

Join naturale – Il *join naturale* è un operatore (che 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 faremo 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.

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

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_1X_2 (cioè sull'unione degli insiemi X_1 e X_2), come segue:

$$r_1 \bowtie r_2 = \{\, t \text{ su } X_1X_2 \;\mid\; \text{esistono } t_1 \in r_1 \text{ e } t_2 \in r_2 \text{ con } t[X_1] = t_1 \text{ e } t[X_2] = t_2 \}$$

Più sinteticamente, ma in modo equivalente, possiamo scrivere:

$$r_1 \bowtie r_2 = \{\, t \text{ su } X_1X_2 \;\mid\; t[X_1] \in r_1 \text{ e } t[X_2] \in r_2 \}$$

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.

INFRAZIONI

Codice	Data	Ag	Art	Prov	Numero
143256	25-10-92	567	44	RM	4E5432
987554	26-10-92	456	34	RM	4E5432
987557	26-10-92	456	34	RM	2F7643
630876	15-10-92	456	53	MI	2F7643
539856	12-10-92	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	25-10-92	567	44	RM	4E5432	Bini Luca	Via Aceri
987554	26-10-92	456	34	RM	4E5432	Bini Luca	Via Aceri
987557	26-10-92	456	34	RM	2F7643	Verdi Piero	Via Tigli
630876	15-10-92	456	53	MI	2F7643	Luci Gino	Via Aceri
539856	12-10-92	567	44	MI	2F7643	Luci Gino	Via Aceri

Figura 3.12 Le relazioni INFRAZIONI e AUTO e il loro join

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 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*):

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

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").

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*

Come caso limite, è ovviamente possibile che nessuna delle tuple degli operandi sia combinabile, e allora il risultato del join è la relazione vuota (si veda un esempio nella 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 $|r_1| \times |r_2|$ tuple (dove $|r|$ indica la cardinalità della relazione r).

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

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

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 $X_1 \cap X_2$ contiene una chiave per r_2 e sussiste il vincolo di riferimento fra $X_1 \cap X_2$ (o un suo sottoinsieme) 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.

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 quindi nel capi-

tolo 4) 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.

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

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 uguale a $(r_1 \bowtie r_2) \bowtie r_3$. Pertanto, potremo scrivere, ove necessa-

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

rio, 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 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_1X_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).

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

IMPIEGATI

Impiegato	Progetto
Rossi	A
Neri	A
Neri	B

PROGETTI

Codice	Nome
A	Venere
B	Marte

$\sigma_{\text{Progetto}=\text{Codice}}$(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

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 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 chiamato *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 tra l'altro 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 ed 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.

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

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:

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

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*:

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

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

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

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):

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

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

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*

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

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):

$$\pi_{\mathsf{Matr,Nome,Stip,MatrC,NomeC,StipC}}$$
$$(\sigma_{\mathsf{Stip>StipC}}(\rho_{\mathsf{MatrC,NomeC,StipC,EtàC \leftarrow Matr,Nome,Stip,Età}}(\textsc{Impiegati})$$
$$\bowtie_{\mathsf{MatrC=Capo}}(\textsc{Supervisione} \bowtie_{\mathsf{Imp=Matr}} (\textsc{Impiegati})))) \qquad (3.4)$$

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

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:

$$\pi_{\mathsf{Matr,Nome}}(\textsc{Impiegati} \bowtie_{\mathsf{Matr=Capo}}$$
$$(\pi_{\mathsf{Capo}}(\textsc{Supervisione}) -$$
$$\pi_{\mathsf{Capo}}(\textsc{Supervisione} \bowtie_{\mathsf{Imp=Matr}} (\sigma_{\mathsf{Stip} \leq 40}(\textsc{Impiegati}))))) \qquad (3.5)$$

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 essere assoluta, oppure dipendere dallo schema:

- $E_1 \equiv_{\mathbf{R}} E_2$ 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. 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 diverse:

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

   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:

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

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 utile allo scopo potrebbe specificare il prodotto cartesiano delle due relazioni seguito da una selezione congiuntiva e poi da una proiezione:

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

Con le regole precedenti, possiamo pensare di migliorare significativamente la qualità dell'espressione, che 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). Con la regola 1, spezziamo la selezione:

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

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:

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

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

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

Altre trasformazioni possono risultare utili, 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.

Altre trasformazioni interessanti sono quelle che si basano sulla corrispondenza 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:

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

PERSONE

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

Figura 3.26 Una relazione con valori nulli

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, o 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: 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 il risultato presenta comunque 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 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. Si introducono due nuove forme di condizioni atomiche di selezione, con lo scopo di verificare 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:

[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.

- *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 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 10) 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:

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

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. La definizione precisa del concetto di *valore di verità* di una formula va oltre gli obiettivi di questo testo, ma esso può essere illustrato 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 di R;
- 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:

$$\sigma_{\text{Stipendio}>40}(\text{IMPIEGATI}) \tag{3.6}$$

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

$$\{\mathsf{Matr} : m,\ \mathsf{Nome} : n,\ \mathsf{Età} : e,\ \mathsf{Stipendio} : s \mid \\ \textsc{Impiegati}(\mathsf{Matr} : m,\ \mathsf{Nome} : n,\ \mathsf{Età} : e,\ \mathsf{Stipendio} : s) \wedge \\ s > 40\} \tag{3.7}$$

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):

$$\pi_{\mathsf{Matr,Nome,Età}}(\sigma_{\mathsf{Stipendio}>40}(\textsc{Impiegati}))$$

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:

$$\{\mathsf{Matr} : m,\ \mathsf{Nome} : n,\ \mathsf{Età} : e \mid \\ \exists s(\textsc{Impiegati}(\mathsf{Matr} : m,\ \mathsf{Nome} : n,\ \mathsf{Età} : e,\ \mathsf{Stipendio} : s) \wedge \\ s > 40)\} \tag{3.8}$$

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

$$\{\mathsf{Matr} : m,\ \mathsf{Nome} : n,\ \mathsf{Età} : e \mid \\ \textsc{Impiegati}(\mathsf{Matr} : m,\ \mathsf{Nome} : n,\ \mathsf{Età} : e,\ \mathsf{Stipendio} : s) \wedge s > 40\} \tag{3.9}$$

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

utitizzare variabili ripetute per indicare te condizioni di join. Per esempio, l'interrogazione che vuote *trovare le matricole dei capi degli impiegati che guadagnano più di 40 mila euro*, formulata in algebra con l'espressione 3.2:

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

può essere formulata nel calcolo con:

$$\{\mathsf{Capo} : c \mid \text{IMPIEGATI}(\mathsf{Matr} : m,\ \mathsf{Nome} : n,\ \mathsf{Età} : e,\ \mathsf{Stipendio} : s) \wedge \text{SUPERVISIONE}(\mathsf{Impiegato} : m,\ \mathsf{Capo} : c) \wedge s > 40\} \quad (3.10)$$

dove ta variabite m, comune alte due condizioni atomiche, realizza ta stessa corretazione fra tupte specificata nel join. Anche qui potremmo utitizzare quantificatori esistenziati per tutte le variabili che non compaiono nelta target list, ma, come net caso precedente, ciò non è necessario, e appesantirebbe ta formutazione.

Se in un'espressione è richiesto il coinvotgimento di tuple diverse di una stessa relazione (in algebra, il join di una relazione con se stessa), è sufficiente includere netta formuta più condizioni sulto stesso predicato, con variabiti diverse. L'interrogazione che vuote *trovare nome e stipendio dei capi degli impiegati che guadagnano più di 40 mila euro*, realizzata in atgebra con l'espressione 3.3:

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

viene formulata net calcolo richiedendo, per ciascuna tupla det risuttato, t'esistenza di tre tuple, una relativa a un impiegato che guadagna più di 40 mila euro, una seconda che indica chi è it suo capo e t'uttima (di nuovo netta retazione IMPIEGATI) che fornisce le informazioni di dettaglio sut capo:

$$\{\mathsf{NomeC} : nc,\ \mathsf{StipC} : sc \mid \text{IMPIEGATI}(\mathsf{Matr} : m,\ \mathsf{Nome} : n,\ \mathsf{Età} : e,\ \mathsf{Stipendio} : s) \wedge s > 40 \text{ SUPERVISIONE}(\mathsf{Impiegato} : m,\ \mathsf{Capo} : c) \wedge \text{IMPIEGATI}(\mathsf{Matr} : c,\ \mathsf{Nome} : nc,\ \mathsf{Età} : ec,\ \mathsf{Stipendio} : sc)\} \quad (3.11)$$

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 datla precedente solo per ta necessità di confrontare valori detlo stesso attributo provenienti da tuple diverse (espressione 3.4 in algebra), il che non causa problemi particolari:

$$\{\mathsf{Matr} : m,\ \mathsf{Nome} : n,\ \mathsf{Stip} : s,\ \mathsf{MatrC} : c,\ \mathsf{NomeC} : nc,\ \mathsf{StipC} : sc \mid \text{IMPIEGATI}(\mathsf{Matr} : m,\ \mathsf{Nome} : n,\ \mathsf{Età} : e,\ \mathsf{Stip} : s) \wedge \text{SUPERVISIONE}(\mathsf{Impiegato} : m,\ \mathsf{Capo} : c) \wedge \text{IMPIEGATI}(\mathsf{Matr} : c,\ \mathsf{Nome} : nc,\ \mathsf{Età} : ec,\ \mathsf{Stip} : sc) \wedge s > sc\} \quad (3.12)$$

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):

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

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:

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

In alternativa, possiamo utilizzare quantificatori universali:

$$\begin{gathered}\{\mathsf{Matr} : c,\ \mathsf{Nome} : n \mid \\ \textsc{Impiegati}(\mathsf{Matr} : c,\ \mathsf{Nome} : n,\ \mathsf{Età} : e,\ \mathsf{Stip} : s) \wedge \\ \textsc{Supervisione}(\mathsf{Impiegato} : m,\ \mathsf{Capo} : c) \wedge \\ \forall m'(\forall n'(\forall e'(\forall s'(\\ \neg(\textsc{Impiegati}(\mathsf{Matr} : m',\ \mathsf{Nome} : n',\ \mathsf{Età} : e',\ \mathsf{Stip} : s') \wedge \\ \textsc{Supervisione}(\mathsf{Impiegato} : m',\ \mathsf{Capo} : c)) \vee s' > 40))))\}\end{gathered} \tag{3.14}$$

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. Vedremo nell'appendice dedicata al sistema Access 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 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 campi di variabilità: 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. Perciò, 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 3.7):

$$\{i.* \mid i(\text{IMPIEGATI}) \mid i.\text{Stipendio} > 40\} \tag{3.15}$$

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 3.9 in calcolo su domini), è sufficiente modificare la target list:

$$\{i.(\mathsf{Matr}, \mathsf{Nome}, \mathsf{Età}) \mid i(\textsc{Impiegati}) \mid i.\mathsf{Stipendio} > 40\} \qquad (3.16)$$

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 3.10 in calcolo su domini), può essere formulata con:

$$\{s.\mathsf{Capo} \mid i(\textsc{Impiegati}),\ s(\textsc{Supervisione}) \mid i.\mathsf{Matr} = s.\mathsf{Impiegato} \wedge i.\mathsf{Stipendio} > 40\} \qquad (3.17)$$

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

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 3.11), può essere realizzata con l'espressione:

$$\{\mathsf{NomeC}, \mathsf{StipC} : i'.(\mathsf{Nome}, \mathsf{Stip}) \mid i'(\textsc{Impiegati}),\ s(\textsc{Supervisione}),\ i(\textsc{Impiegati}) \mid i'.\mathsf{Matr} = s.\mathsf{Capo} \wedge s.\mathsf{Impiegato} = i.\mathsf{Matr} \wedge i.\mathsf{Stipendio} > 40\} \qquad (3.18)$$

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 3.12 in calcolo su domini):

$$\{i.(\mathsf{Nome}, \mathsf{Matr}, \mathsf{Stip}),\ \mathsf{NomeC}, \mathsf{MatrC}, \mathsf{StipC} : i'.(\mathsf{Nome}, \mathsf{Matr}, \mathsf{Stip}) \mid i(\textsc{Impiegati}),\ s(\textsc{Supervisione}),\ i'(\textsc{Impiegati}) \mid i.\mathsf{Matr} = s.\mathsf{Impiegato} \wedge s.\mathsf{Capo} = i'.\mathsf{Matr} \wedge i.\mathsf{Stipendio} > i'.\mathsf{Stipendio}\} \qquad (3.19)$$

Le interrogazioni con i quantificatori mostrano appieno la maggiore sinteticità e praticità del calcolo su tuple con dicharazioni 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 3.13 o 3.14 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:

$$\{i.(\mathsf{Matr}, \mathsf{Nome}) \mid i(\textsc{Impiegati}),\ s(\textsc{Supervisione}) \mid i.\mathsf{Matr} = s.\mathsf{Capo} \wedge \forall i'(\textsc{Impiegati})(\forall s'(\textsc{Supervisione})(\neg(s.\mathsf{Capo} = s'.\mathsf{Capo} \wedge s'.\mathsf{Impiegato} = i'.\mathsf{Matr}) \vee i'.\mathsf{Stipendio} > 40))\} \qquad (3.20)$$

Con quantificatori esistenziali negati:

$$\{i.(\mathsf{Matr}, \mathsf{Nome}) \mid i(\textsc{Impiegati}),\ s(\textsc{Supervisione}) \mid$$
$$i.\mathsf{Matr} = s.\mathsf{Capo} \wedge \neg(\exists i'(\textsc{Impiegati})(\exists s'(\textsc{Supervisione})$$
$$(s.\mathsf{Capo} - s'.\mathsf{Capo} \wedge s'.\mathsf{Impiegato} = i'.\mathsf{Matr} \wedge$$
$$i'.\mathsf{Stipendio} \leq 40)))\} \tag{3.21}$$

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, richie-

dendo 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 ancora 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:

- 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:

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

- 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:

$$?\textsc{Impiegati}(\mathsf{Matr} : m, \mathsf{Nome} : n, \mathsf{Età} : 30, \mathsf{Stipendio} : s)$$

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 3.10, definiamo un predicato intensionale CapiDeiRicchi, con la regola:

$$\begin{aligned}&\textsc{CapiDeiRicchi}(\mathsf{Capo} : c) \leftarrow \\ &\quad \textsc{Impiegati}(\mathsf{Matr} : m,\ \mathsf{Nome} : n,\ \mathsf{Età} : e,\ \mathsf{Stipendio} : s), \\ &\quad \textsc{Supervisione}(\mathsf{Impiegato} : m,\ \mathsf{Capo} : c), s > 40 \qquad (3.22)\end{aligned}$$

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 3.22 definisce la relazione intensionale CAPIDEIRICCHI come costituita dalle stesse tuple che compaiono nel risultato dell'espressione 3.10 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 3.13:

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

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

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

[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".

quindi utilizziamo questo predicato in forma negata:

CAPISOLODIRICCHI(Matr : c, Nome : n) ←
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) ←
SUPERVISIONE(Impiegato : i, Capo : c)

SUPERIORE(Impiegato : i, SuperCapo : c) ←
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 [31] e Silberchatz, Korth e Sudarshan [57] per trattazioni generali e quelli di Atzeni, Batini e De Antonellis [4], Ullman [64], Maier [43], Atzeni e De Antonellis [6], Abiteboul, Hull e Vianu [1] per approfondimenti più formali e teorici. Per il Datalog, si può consultare il testo di Ceri, Gottlob e Tanca [17].

Esercizi

Esercizio 3.1 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.

Esercizio 3.2 Con riferimento allo schema nell'esercizio 3.1, 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.

Esercizio 3.3 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.

Esercizio 3.4 Mostrare come le interrogazioni nell'esercizio 3.3 possano trarre vantaggio, nella specifica, dalla definizione di viste.

Esercizio 3.5 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.

Esercizio 3.6 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 50000 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.

Esercizio 3.7 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.

Esercizio 3.8 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.

Esercizio 3.9 Considerare uno schema relazionale contenente le relazioni:

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

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

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

Esercizio 3.10 Con riferimento allo stesso schema dell'esercizio 3.9, formulare in algebra relazionale le interrogazioni realizzate in calcolo su domini dalle seguenti espressioni:

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

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

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

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

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

Esercizio 3.11 Trasformare la seguente espressione dell'algebra, che fa riferimento allo schema

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

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

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

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 S. 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) e descrivendo l'integrazione tra SQL e i tradizionali linguaggi di programmazione (capitolo 6).

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

[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.

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 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 due nuove versioni dello standard, cui si fa riferimento con il nome informale SQL-3, aventi nome ufficiale SQL:1999 e SQL:2003. 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. Ad esempio, la parte 13 di SQL:2003, 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.

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, ...

Figura 4.1 Evoluzione dello standard SQL

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 riferimento 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 della evoluzione più recente della tecnologia delle basi di dati, tra cui i trigger, le viste ricorsive e il supporto per il paradigma ad 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. 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 [5]).

Pur senza le estensioni introdotte in SQL-3, SQL-2 è un linguaggio ricco e complesso, tanto che, ad alcuni 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 nella 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 un personal computer monoutente 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 un'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, ovvero 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` che 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 consistono nell'offerta di un insieme di costruttori, 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 [5], 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 stata 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.). Una 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.

[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.

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`:

`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 dei costruttori di dominio 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, l'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 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 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 assumere solo 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 ad 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, che 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, senza che il sistema provi a 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`), 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, ovvero 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

[3] Si suggerisce quindi di sfruttare a proprio vantaggio il supporto transazionale offerto dai sistemi (si veda il paragrafo 5.4), 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.

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.

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, ed altre, per un totale di ventitré viste che descrivono la struttura della base di dati.

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

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.

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 (si veda il 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 una 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 una 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'
```

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

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.

Salario
45
80

Figura 4.5 Risultato della interrogazione 1

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.

```
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.

Nome	Cognome	Dipart	Ufficio	Stipendio	Città
Mario	Rossi	Amministrazione	10	45	Milano
Carlo	Rossi	Direzione	14	80	Milano

Figura 4.6 Risultato della interrogazione 2

```
select Stipendio/12 as StipendioMensile
from Impiegato
where Cognome = 'Bianchi'
```

StipendioMensile
3,00

Figura 4.7 Risultato della 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.

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

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.

```
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 una 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 una 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 una interrogazione che richieda l'uso sia di `and` che di `or`, conviene esplicitare l'ordine di valutazione mediante parentesi.

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 della interrogazione 4

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.

Nome	Cognome
Giovanni	Verdi

Figura 4.10 Risultato della interrogazione 6

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'
```

Applicando l'interrogazione alla tabella in figura 4.4 si ottiene il risultato in figura 4.11.

Nome	Cognome
Mario	Rossi
Carlo	Bianchi
Giovanni	Verdi
Paola	Rosati
Marco	Franco

Figura 4.11 Risultato della interrogazione 7

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')
```

Nome
Mario

Figura 4.12 Risultato della interrogazione 8

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`).

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'
```

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 della interrogazione 9

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 mila euro. 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 (le vedremo nel 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 una interrogazione SQL nella sua forma più semplice:

select T_1.Attributo$_{11}$, ..., T_h.Attributo$_{hm}$
from TABELLA$_1$ T_1, ..., TABELLA$_n$ T_n
where *Condizione*

si può costruire una 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):

$$\pi_{T_1.\text{Attributo}_{11},\ldots,T_h.\text{Attributo}_{hm}}(\sigma_{Condizione}(\text{TABELLA}_1 \times \ldots \times \text{TABELLA}_n))$$

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 $T_1, T_2, \ldots, T_h$ siano presenti nella clausola `select` e che $T_{h+1}, T_{h+2}, \ldots, T_n$ non lo siano[4], la generica istruzione `select`

[4]Si noti che è sempre possibile riordinare le variabili in modo tale che questa condizione sia soddisfatta.

ha una semantica che è uguale a quella della seguente espressione del calcolo relazionale su tuple con dichiarazioni di range:

$$\{t_1.\text{Attributo}_{11}, \ldots, t_h.\text{Attributo}_{hm} \mid i_1(\text{TABELLA}_1), \ldots, i_h(\text{TABELLA}_h) \mid \exists t_{h+1}(\text{TABELLA}_{h+1}), (\ldots, (\exists t_n(\text{TABELLA}_n) (\textit{Condizione}'))\ldots)\}$$

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.

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) (vedi un esempio nella 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.

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

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.

```
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.

Città
Verona
Verona
Milano

Città
Verona
Milano

Figura 4.15 Il risultato delle interrogazioni 10 e 11

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*]

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

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 quale 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, alla 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.

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 della interrogazione 13

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)
```

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.

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 della interrogazione 14

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 I3 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 (si veda il 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 una 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 della 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 (vedi l'interrogazione 5). Vi sono però altre ragioni per usare gli alias.

Per prima cosa, utilizzando gli alias è possibile far riferimento a più esemplari della stessa tabella, in modo analogo 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 una 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 contenente tutte le righe di IMPIEGATO. 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`.

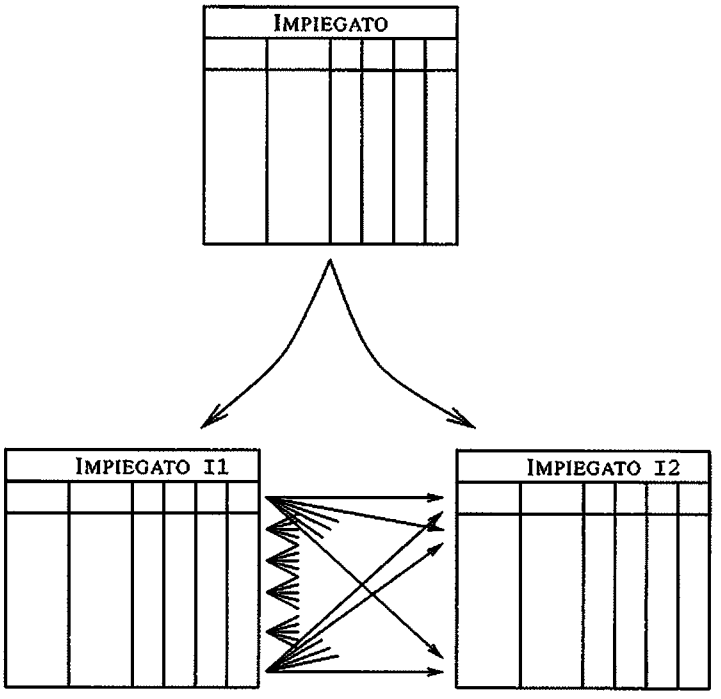


Figura 4.19 Descrizione della esecuzione della interrogazione 17

La definizione di alias risulta anche molto importante per la realizzazione di sofisticate interrogazioni nidificate, come vedremo nel paragrafo 4.3.6.

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 mila euro.

```
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.

SQL permette di specificare un ordinamento delle righe del risultato di una 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.

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 della interrogazione 19

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 (ovvero, 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
where Dipart = NomeDip and
      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
where Dipart = NomeDip and
      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 una 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
```

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

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:

```
select Dipart, Stipendio
from Impiegato
```

Il risultato ottenuto a questo punto è mostrato in figura 4.22.

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

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.

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

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 della interrogazione, ovvero l'ammontare totale degli stipendi elargiti agli impiegati di ogni dipartimento, divisi per dipartimento.

La sintassi SQL impone che, in una 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

Dipart	sum(Stipendio)
Amministrazione	125
Produzione	82
Distribuzione	45
Direzione	153

Figura 4.24 Risultato della interrogazione 27

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 della 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; si veda il paragrafo 10.2 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 una interrogazione che fa uso della clausola `group by`. Ogni sottoinsieme di righe costruito dalla `group by` fa parte del risultato della interrogazione solo se il predicato argomento della `having` risulta soddisfatto.

Interrogazione 31: Estrarre i dipartimenti che spendono più di 100 mila euro in stipendi.

```
select Dipart, sum(Stipendio) as SommaStipendi
from Impiegati
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 argomento della clausola `having`, che seleziona i dipartimenti per cui la somma degli stipendi, per tutti gli elementi del sottoinsieme, è superiore a 100 mila euro. Il risultato dell'interrogazione è rappresentato dalla tabella in figura 4.25.

Dipart	SommaStipendi
Amministrazione	125
Direzione	153

Figura 4.25 Risultato della interrogazione 31

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 una 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 una 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 mila euro.

```
select Dipart
from Impiegato
where Ufficio = 20
group by Dipart
having avg(Stipendio) > 25
```

La forma sintetica 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 della 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 i dati di partenza siano quelli contenuti nella tabella in figura 4.21, il risultato della valutazione della 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.

Nome
Mario
Carlo
Giovanni
Franco
Lorenzo
Paola
Marco
Rossi
Bianchi
Verdi
Neri
Gialli
Rosati

Figura 4.26 Risultato della interrogazione 33

Nome
Carlo
Franco
Carlo
Lorenzo
Marco
Bianchi
Neri
Rossi
Gialli
Franco

Figura 4.27 Risultato della interrogazione 34

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.

Nome
Franco

Figura 4.28 Risultato della interrogazione 35

Interrogazione 36: Estrarre i nomi degli impiegati che non sono cognomi di qualche impiegato.

```
select Nome
from Impiegato
   except
select Cognome
from Impiegato
```

Il risultato di questa interrogazione è mostrato in figura 4.29.

Nome
Mario
Carlo
Giovanni
Lorenzo
Paola
Marco

Figura 4.29 Risultato della interrogazione 36

4.3.6 Interrogazioni nidificate

Fino ad 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 predicati 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 una 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 una interrogazione, sorge il problema di disomogeneità dei termini del confronto. Infatti, da una parte abbiamo il risultato dell'esecuzione di una 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 dalla 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 una 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 una 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 (π_{Nome}(DIPARTIMENTO) – π_{Dipart} ($\sigma_{\text{Cognome='Rossi'}}$(IMPIEGATO))), e 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.

"in" equivale a "= any"

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 una 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 una 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 Una interpretazione molto semplice e intuitiva delle interrogazioni nidificate consiste nell'assumere che l'interrogazione nidificata venga eseguita prima di analizzare le righe della interrogazione esterna. Il risultato della interrogazione può essere salvato in una tabella temporanea e il controllo sulle righe della 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 della 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

any => verifica che ci sia
qualche valore che
soddisfi la relazione (TRUE)

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 una 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 una 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 della interrogazione 45, con l'unica differenza che il predicato è soddisfatto nel caso che il risultato della 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 della 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 Manipolazione 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(` *ListaDiValori*) |
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 una 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; oppure,
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 sotto i 30 mila euro, 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 mila euro, questo soddisferà la condizione del primo comando di aggiornamento, per cui l'attributo Stipendio verrà posto pari a 33 mila euro. 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` (verrà presentato nel 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, ad esempio tramite l'uso di cursori, gli aggiornamenti desiderati (descriveremo il funzionamento dei cursori nel paragrafo 6.3.1).

4.5 Esempi riepilogativi

1. Dato il seguente schema relazionale che descrive il calendario di una manifestazione sportiva a squadre nazionali:

   STADIO(Nome,Città,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:

$$\pi_{\text{Nome}}(\text{STADIO}) - \pi_{\text{NomeStadio}}((\pi_{\text{Paese}}(\sigma_{\text{Continente}='Europa'}\ \text{NAZIONALE})) \bowtie_{\text{Squadra1=Paese}\vee\text{Squadra2=Paese}} (\pi_{\text{NomeStadio,Squadra1,Squadra2}}(\text{INCONTRO})))$$

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, Città : 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:

$$\pi_{\text{Nome}}(\ (\text{PROPRIETARIO} \bowtie \text{MOTO})$$
$$\bowtie_{\text{Marca}\neq\text{Marca2}\wedge\text{Nome}=\text{Nome2}}$$
$$(\rho_{\text{Nome2}\leftarrow\text{Nome}}(\text{PROPRIETARIO}) \bowtie \rho_{\text{Marca2}\leftarrow\text{Marca}}\ (\text{MOTO}))) -$$
$$\pi_{\text{Nome}}\ (\text{PROPRIETARIO} \bowtie \sigma_{\text{Nazione}\neq\text{'}Giappone\text{'}}\ \text{MOTO})$$

Note bibliografiche

SQL è stato inizialmente descritto da Chamberlin *et al.* in [19] e [20]. Studi sistematici del linguaggio SQL, relativamente alle tecniche di ottimizzazione e al significato delle query, sono contenuti in [16] e [40].

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 [12] (di cui esiste la traduzione italiana) e il libro di Melton e Simon [47]. Eisemberg e Melton [30] discutono il processo di standardizzazione in generale e con riferimenti specifici all'area delle basi di dati. Melton e Simon [48] 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 sapere quali funzionalità di SQL sono state effettivamente implementate nel particolare sistema.

Esercizi

Esercizio 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)`.

Esercizio 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".

Esercizio 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.

Esercizio 4.4 Dare le definizioni SQL delle tabelle AUTORE(Nome,Cognome, DataNascita, Nazionalità), LIBRO(TitoloLibro, NomeAutore, CognomeAu-

tore, Lingua). Per il vincolo di *foreign key* specificare una politica di `cascade` sulle cancellazione e di `set null` sulle modifiche.

Esercizio 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'
```

Esercizio 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
```

Esercizio 4.7 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.

Esercizio 4.8 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.

Esercizio 4.9 Dare una sequenza di comandi di aggiornamento che modifichi l'attributo Stipendio della tabella IMPIEGATO, aumentando del 10% gli stipendi sotto i 30 mila euro e diminuendo del 5% gli stipendi sopra i 30 mila euro.

5

SQL: Caratteristiche evolute

Il capitolo continua la presentazione delle caratteristiche del linguaggio SQL, mostrando alcune caratteristiche evolute del linguaggio. Completeremo la descrizione dei servizi per la definizione dei dati, mostrando l'uso del linguaggio di interrogazione in questo contesto (paragrafo 5.1). Illustreremo poi l'uso di funzioni scalari (paragrafo 5.2) e i comandi per il controllo dell'accesso ai dati (paragrafo 5.3). Infine, descriveremo le caratteristiche di base dei comandi SQL per la gestione delle transazioni (paragrafo 5.4).

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 più importanti esigenze, ma senza esaurire i bisogni delle applicazioni. Per specificare ulteriori vincoli, SQL-2 ha introdotto la clausola `check`, con la seguente sintassi:

`check` (*Condizione*)

Le condizioni che si possono usare sono quelle 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 di tupla descritti nel paragrafo 4.2.6, e anche di più, poiché la condizione può far riferimento ad altri attributi.

Una efficace dimostrazione della potenza del costrutto consiste nel far vedere come i vincoli predefiniti possano tutti essere descritti con la clausola `check`. Per questo possiamo prendere 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 definiti 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 di questo costrutto si può per esempio descrivere un vincolo che obbliga l'impiegato ad avere un manager del proprio dipartimento, a meno che la matricola non inizi con la cifra 1. Si estende la precedente definizione di tabella con le seguenti dichiarazioni:

```
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 però associati a nessun attributo o tabella in particolare, bensì appartengono direttamente allo schema.

Mediante le asserzioni è possibile esprimere tutti i vincoli che abbiamo prima specificato nell'ambito della definizione delle tabelle. Le asserzioni permettono poi di esprimere vincoli che non sarebbero altrimenti definibili, come vincoli su

più tabelle o vincoli 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 in 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, vedi paragrafo 5.4).

Il controllo differito viene tipicamente introdotto per gestire situazioni in cui non è possibile costruire una situazione consistente con una singola modifica della base di dati. 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 di 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 sempe 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 modificheranno la modalità di controllo dei vincoli nominati, o di tutti i vincoli se si usa l'opzione `all`, per il contesto della transazione all'interno della quale è stato 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 ed 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 mila euro:

```
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 i 10 e i 50 mila euro:

```
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 grossi 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 chiede pure che l'insieme di attributi della vista contenga almeno una chiave primaria della tabella. La clausola `check option` può essere utilizzata solo nel contesto di queste viste. La `check option` specifica che possono essere ammessi aggiornamenti solo sulle righe della vista, e dopo gli aggiornamenti le righe devono continuare ad appartenere alla vista. Ciò può per esempio capitare se si assegna ad 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 rimosse 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 delle viste coinvolte); l'opzione di default è quella di `cascaded`.

Dato che la vista 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 mila euro 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 mila euro 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.

Si vuole determinare qual è il dipartimento che spende il massimo in stipendi. Per questo fine, 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 caratterizzato dal 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,Città,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 nuova 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 alla 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 del linguaggio, 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 PostgreSQL, un DBMS open-source particolarmente interessante.

- 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 isola una qualsiasi componente 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 ad 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 strategie di conversione accettabili; 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, controllando l'indentazione, la dimensione di ogni campo, e altre caratteristiche del formato di rappresentazione.
- Funzioni matematiche: si applicano su 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 espliciti.

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 complementare 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 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 ed 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 di identificazione, per cui a un utente del sistema possono corrispondere più utenti della base di dati e viceversa.

5.3.1 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 di comunque 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; e, infine,
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 ed 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`

prio 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 diventa 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.

5.4 Transazioni

Introduciamo ora la nozione di *transazione*, un concetto che riveste un ruolo fondamentale nell'uso delle basi di dati. Una *transazione* identifica una unità elementare di lavoro svolta da una applicazione, cui si vogliono associare particolari caratteristiche di correttezza, robustezza e isolamento. Un sistema che mette a disposizione un meccanismo per la definizione e l'esecuzione di transazioni viene detto *sistema transazionale*.

Una transazione può essere definita sintatticamente: in SQL, l'inizio di una transazione è rappresentato dal comando: `start transaction`. Il termine della transazione è rappresentato da due istruzioni particolari, `commit work` e `rollback work`, cui facciamo riferimento usando i due termini *commit* e *abort*, che indicano l'azione associata alla rispettiva istruzione (in effetti, la sintassi SQL ammette anche i comandi nella forma senza il termine `work`). L'effetto di questi due comandi è decisivo per l'esito della transazione, la quale "va a buon fine" solo a seguito di un commit, mentre non ha alcun effetto tangibile sulla base di dati quando viene eseguito l'abort. Dal punto di vista del potere espressivo, si noti che l'istruzione `rollback work` è molto potente, in quanto tramite essa l'utente della base di dati può annullare gli effetti del lavoro svolto dalla transazione, indipendentemente dalla sua complessità.

Un esempio di transazione è dato dal seguente codice, che trasferisce dal conto 42177 al conto 12202 l'ammontare 10.

```
start transaction;
update ContoCorrente
  set Ammontare = Ammontare + 10
  where NumConto = 12202;
update ContoCorrente
  set Ammontare = Ammontare - 10
  where NumConto = 42177;
commit work;
```

Viene detta *ben formata* una transazione iniziata da `start transaction` nel corso della cui esecuzione viene invocato uno solo dei due comandi `commit work` o `rollback work`. In alcune interfacce transazionali, viene immediatamente e implicitamente eseguito dopo ogni commit o abort un comando `start transaction`, in modo da rendere tutte le computazioni transazioni ben formate.

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").

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 la 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 il valore di un conto corrente bancario.

Concludendo, si noti 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.

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 il controllo dell'accesso nelle basi di dati, il testo [13] è dedicato totalmente all'argomento. Per quanto riguarda il concetto di transazione, si fa riferimento al secondo volume [5], che costituisce il naturale proseguimento di questo testo.

Esercizi

Esercizio 5.1 Dato il seguente schema:

AEROPORTO(Città,Nazione,NumPiste)
VOLO(IdVolo,GiornoSett,CittaPart,OraPart,CittaArr,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).

Esercizio 5.2 Dare una sequenza di comandi di aggiornamento che modifichi l'attributo Stipendio della tabella IMPIEGATO, aumentando del 10% gli stipendi sotto i 30 mila euro e diminuendo del 5% gli stipendi sopra i 30 mila euro.

Esercizio 5.3 Definire sulla tabella IMPIEGATO il vincolo che il dipartimento Amministrazione abbia meno di 100 dipendenti, con uno stipendio medio superiore ai 40 mila euro.

Esercizio 5.4 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.

Esercizio 5.5 Definire una vista che mostra per ogni dipartimento il valore medio degli stipendi superiori alla media.

Esercizio 5.6 Tramite la definizione di una vista, permettere all'utente "Carlo" di accedere al contenuto di IMPIEGATO, escludendo l'attributo Stipendio.

Esercizio 5.7 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
```

6

SQL per le applicazioni

L'accesso di gran lunga più tipico a una base di dati avviene attraverso applicazioni integrate nel sistema informativo. Il dialogo diretto con l'interprete SQL è riservato a pochi utenti esperti. L'uso di apposite applicazioni per accedere alle informazioni è giustificato da una serie di considerazioni. Molto spesso chi deve accedere alle informazioni non è direttamente un utente, ma un'applicazione non interattiva (detta anche *batch*). Anche nel caso di accessi interattivi, spesso le modalità di accesso sono semplici e prevedibili; è quindi utile ridurre la complessità nell'accesso alla base di dati costruendo un'applicazione che fornisca un'interfaccia semplificata per lo svolgimento del compito. SQL viene incontro alle esigenze di costruzione delle applicazioni in due direzioni.

La prima direzione è legata a un incremento delle funzionalità di SQL e in generale del DBMS. Il linguaggio SQL non si limita infatti 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 6.1) e trigger (paragrafo 6.2). Una evoluzione ulteriore in questa direzione è rappresentata da strumenti di sviluppo che permettono di costruire applicazioni complete per la gestione di basi di dati in modo guidato. Buona parte dei sistemi di basi di dati commerciali offrono, oltre al gestore di base di dati vero e proprio, anche un insieme di strumenti specializzati (e proprietari) per lo sviluppo di applicazioni (si veda ad esempio la descrizione di Microsoft Access nell'appendice A). Esiste poi una offerta di prodotti non legati a una base di dati in particolare, ciascuno dei quali in grado di gestire il dialogo con il sistema relazionale grazie all'uso dello standard SQL. Questi strumenti permettono di definire agevolmente gli schemi della base di dati e di costruire interfacce sofisticate. Non tratteremo specificamente queste soluzioni, sia perché è la modalità relativamente meno diffusa, sia per la scarsa uniformità che rende difficile una trattazione sistematica.

La seconda direzione invece assume che le applicazioni siano realizzate facendo uso dei tradizionali linguaggi di programmazione di alto livello. In questo

caso, il problema da risolvere è relativo all'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. La seconda parte del capitolo analizzerà il supporto richiesto per lo sviluppo di applicazioni nei linguaggi di programmazione di alto livello, di tipo procedurale e a oggetti; descriveremo prima SQL Embedded, in cui SQL viene integrato direttamente all'interno del normale linguaggio di programmazione (paragrafo 6.3); illustreremo infine gli approcci in cui l'integrazione con SQL avviene tramite l'invocazione di un'opportuna libreria di funzioni (approccio a *Call Level Interface*, paragrafo 6.4).

Vogliamo infine osservare che la trattazione di questi argomenti in questo testo è volutamente sintetica. Maggiori approfondimenti sui trigger si trovano nel secondo volume [5], in cui viene dedicato un intero capitolo all'argomento. Nel secondo volume si mostrano ulteriori esempi di integrazione tra SQL e linguaggi di programmazione, presentando strumenti che, nel contesto dell'integrazione tra il mondo del Web e le basi di dati, permettono di inserire codice SQL all'interno di moduli software responsabili di generare pagine HTML.

6.1 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 `:NuovaCitta`, 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(select *
      from Dipartimento
      where Nome = :NomeDip) = NULL
     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 > Scoperto 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*.

6.2 Trigger

I *trigger*, detti anche *regole attive*, rappresentano una funzionalità particolarmente significativa dei moderni sistemi relazionali. I trigger seguono il paradigma *Evento-Condizione-Azione* (ECA): ogni trigger si attiva quando occorre uno specifico evento all'interno della base di dati; se è soddisfatta una data condizione,

allora il trigger esegue un'azione stabilita. I trigger realizzano quindi nell'ambito delle basi di dati relazionali un paradigma di programmazione procedurale a regole. Lo scopo è normalmente quello di specificare le reazioni che il sistema deve mettere in atto in presenza di situazioni prestabilite.

Senza presentare in dettaglio le caratteristiche della sintassi, mostriamo un esempio di trigger SQL-3.

```
create trigger ImpiegatiSenzaDip
after insert into Impiegati
for each row
when (new.Dipart is null)
update Impiegato
   set Dipart = 'NuoviArrivati'
   where Matr = new.Matr
```

Il trigger viene attivato tutte le volte che il sistema rileva l'inserimento di tuple all'interno della tabella IMPIEGATI. Se la condizione, rappresentata dal predicato SQL che segue la parola chiave when, è soddisfatta, ovvero se l'impiegato inserito presenta il valore nullo per l'attributo Dipart, l'azione del trigger viene eseguita. In questo trigger l'azione è rappresentata da un comando di update che assegna all'attributo Dipart del nuovo impiegato il valore "NuoviArrivati". Il trigger garantisce che la base di dati reagirà a ogni inserimento di tuple in IMPIEGATO aventi valore nullo per l'attributo Dipart, andando a sostituire il valore nullo con il valore sopra definito.

Analizziamo brevemente in modo informale la struttura sintattica dei trigger SQL-3. La parte *evento* può solo presentare operazioni SQL di aggiornamento dello stato della base di dati, ovvero insert e delete su tabelle, e update su tabelle o singoli attributi. Ogni trigger è sensibile a un solo evento. La parte *condizione* è rappresentata da un generico predicato SQL. Nella valutazione della condizione, così come nell'ambito dell'azione, è possibile utilizzare una coppia di variabili predefinite new e old, che permettono di far riferimento rispettivamente alla nuova e vecchia versione della tupla che è stata oggetto del comando di insert, update o delete (naturalmente, la variabile new non sarà utilizzabile nell'ambito di trigger che reagiscono all'evento di delete, così come la variabile old non sarà definita per trigger che reagiscono a eventi di insert). Infine, l'*azione* del trigger è rappresentata da un generico singolo comando o da una procedura SQL.

I trigger sono presenti nei sistemi relazionali fin dagli anni Ottanta, quindi ben prima della loro standardizzazione avvenuta in SQL-3; presentano per questo motivo delle caratteristiche sintattiche e funzionali eterogenee. La presenza in SQL-3 di una sintassi e un modello di esecuzione ufficiale definisce un obiettivo verso il quale i sistemi tenderanno in futuro, ma ancora oggi la limitata portabilità dei trigger costituisce un ostacolo al loro uso.

I trigger possono quindi servire alla realizzazione di molteplici servizi. Una delle applicazioni più classiche dei trigger è rappresentata dalla gestione di vincoli di integrità generici. L'approccio si basa sulla scrittura di trigger che: (1) sono

sensibili a tutti gli eventi che possono introdurre violazioni nei vincoli, (2) verificano nella condizione se le violazioni sono effettivamente presenti e in tal caso (3) eseguono le azioni che hanno la responsabilità di intervenire sulla base di dati per eliminare le violazioni. Questo approccio è applicabile a una grande varietà di vincoli, andando ben al di là delle tipologie di vincoli predefiniti riconosciuti dalla sintassi SQL. Inoltre, mediante i trigger è possibile gestire la presenza di eventuali violazioni dei vincoli intervenendo sulla base di dati per "riparare" l'inconsistenza, adottando quindi un approccio che può essere più vicino a quelli che sono i requisiti dell'applicazione, in contrapposizione alla gestione dei vincoli di sistema per i quali il modo tipico di reagire alle inconsistenze consiste nel rifiutare l'aggiornamento che aveva introdotto la violazione del vincolo.

I trigger sono dei costrutti molto ricchi, che presentano un gran numero di varietà sintattiche e di comportamento e vi sono in generale diversi aspetti significativi che caratterizzano la loro definizione. Come già segnalato, all'argomento è dedicato un intero capitolo del secondo volume [5]. Volendo rimanere nell'ambito di questo testo al livello di una trattazione introduttiva, si possono mettere in evidenza le principali problematiche che rendono difficile sfruttare appieno questo strumento così potente e flessibile.

Il principale problema che si presenta a chi vuole sfruttare il potenziale dei trigger è dato dalla complessità del comportamento di un sistema composto da molteplici trigger. Uno degli aspetti caratterizzanti dei trigger è il fatto che i trigger devono reagire a eventi di modifica della base di dati, qualunque sia la loro origine, e che l'azione del trigger può a sua volta modificare lo stato della base di dati. Una conseguenza di ciò è che l'azione di un trigger può a sua volta produrre eventi che attivano altri trigger, creando delle catene di esecuzione che risultano di norma assai difficili da analizzare. Le catene di attivazione possono inoltre degenerare, dando luogo ad attivazioni infinite, che si possono verificare quando vi sono trigger che si attivano mutuamente in modo ciclico. La gestione di queste situazioni è particolarmente difficile e richiede di far uso di strumenti sofisticati, che non sono comunque risolutivi. In pratica, lo sviluppo di sistemi di trigger richiede un notevole impegno da parte del progettista della base di dati.

Per concludere, si può mettere in evidenza come il potenziale dei trigger sia comunque estremamente elevato e quindi è bene che ogni utente avanzato di un sistema relazionale sia a conoscenza delle caratteristiche di questo strumento, per poter gestire in modo efficace diversi requisiti delle applicazioni, nella situazioni nelle quali si ha la garanzia che il comportamento del sistema non possa dare luogo a comportamenti anomali.

6.3 SQL Embedded

L'integrazione del linguaggio SQL con i normali linguaggi di programmazione di alto livello (C, C++, COBOL, Java, ...) 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.

L'incastonamento 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, bisognerà far precedere la compilazione del linguaggio di alto livello dall'esecuzione di un apposito preprocessore che riconoscerà le istruzioni SQL e sostituirà a esse un insieme opportuno di chiamate ai servizi del DBMS, tramite una libreria specifica per ogni sistema. Il preprocessore riconoscerà il significato dei singoli comandi SQL e sarà in grado di predisporre l'insieme opportuno di strutture ausiliarie richieste per la loro esecuzione. Il preprocessore sarà inoltre in grado di segnalare al momento della compilazione eventuali anomalie 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 la soluzione sia applicabile, è necessario che sia disponibile un preprocessore per la particolare combinazione di DBMS-piattaforma-linguaggio-compilatore usato per lo sviluppo (ad esempio, *Oracle Server, Solaris, C, gcc*). I sistemi commerciali di maggiore diffusione mettono a disposizione soluzioni di questo tipo per le configurazioni d'uso più frequente.

Un esempio d'uso di SQL Embedded appare in figura 6.1, dove si usa concretamente la soluzione ECPG, un preprocessore di SQL Embedded per il linguaggio C e PostgreSQL.

L'esempio in figura 6.1 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 verificata un'anomalia 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 come parametri per i comandi SQL. Si osserva infatti come le variabili del linguaggio vengano usate nel comando di inserimento che compare alla linea (13) del programma. Il meccanismo che viene offerto per l'uso delle variabili è molto agevole e richiede solamente di far precedere il nome della variabile dal carattere di due punti. 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

```
(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 6.1 Un programma C con SQL Embedded

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 problema del *conflitto d'impedenza* (*impedance mismatch*[1]). 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 una interrogazione un'intera tabella.

Questo problema ammette due diverse soluzioni. La prima soluzione si basa sull'uso dei *cursori*. I cursori fanno parte dello standard SQL fin dalla versione SQL-2 e non pongono particolari restrizioni sul linguaggio di programmazione.

Una seconda soluzione consiste nell'utilizzare un linguaggio di programmazione che abbia a disposizione dei costruttori di dati più potenti e in particolare riesca a gestire in modo naturale 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 6.4) seguono questa impostazione per la risoluzione del problema.

[1] 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.

6.3.1 Cursori

Un cursore è uno strumento 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 muoversi liberamente sul risultato dell'interrogazione. L'opzione finale `for update` specifica se il cursore deve essere utilizzato nell'ambito di un comando di modifica, permettendo di specificare eventualmente gli attributi che saranno 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` prende una riga dal cursore e la ripone nelle variabili del programma che compaiono in *ListaDiFetch*. In *ListaDiFetch* vi sarà una variabile per ogni elemento della target list della interrogazione, con una corrispondenza dettata dall'ordinamento nella lista e in cui ogni elemento è compatibile con i domini degli elementi della target list. Esiste l'importante concetto di riga corrente, che rappresenta l'ultima riga letta. Il parametro *Posizione* permette di specificare quale riga dovrà 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* (come `absolute`, solo che viene preso come punto di riferimento la posizione della riga corrente).

Queste opzioni sono utilizzabili a condizione che sia stata specificata al momento della definizione del cursore l'opzione `scroll`, la quale appunto garantisce che sia possibile muoversi liberamente all'interno del risultato dell'interrogazione. Se l'opzione `scroll` non è specificata, l'unico valore accettabile per il parametro *Posizione* è `next`. In questo caso, l'implementazione può diventare

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 sintassi 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 permetta di accedere a una riga effettiva di una tabella, non essendo 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 vengono liberate le risorse dedicate al cursore, in particolare rilasciando lo spazio di memoria utilizzato per conservare il risultato.

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
```

Viene così associato il cursore `CursoreImpiegati` alla interrogazione che permette di ottenere i dati relativi ai dipendenti che guadagnano tra 40 e 100 mila Euro.

Nella figura 6.2 vediamo un semplice esempio di procedura C che fa uso dei cursori. Le variabili del programma vengano rappresentate nei comandi SQL con il nome preceduto dal carattere ":" (due punti). Le variabili devono essere dichiarate di tipo compatibile ai 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`.

```
(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 6.2 Una procedura C che fa uso di cursori

Le interrogazioni per cui è garantito che venga restituita una sola riga (ad 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 della 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 copiati nelle variabili `nomeDip` e `cognomeDip`.

6.3.2 SQL dinamico

In diverse situazioni sorge la necessità di permettere all'applicazione di definire al momento dell'esecuzione le interrogazioni da effettuare sulla base di dati.

Se le interrogazioni hanno una struttura predefinita e ciò che varia è solamente il valore dei parametri usati nell'interrogazione, allora diventa possibile costruire una applicazione che gestisce le tipologie di interrogazione richieste dall'utente sfruttando il meccanismo di passaggio dei parametri dall'ambiente del programma al sistema di gestione di basi di dati, come illustrato negli esempi del paragrafo precedente. In altri casi però l'utente ha bisogno di effettuare delle 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 della interrogazione sia prefissata (si parla infatti di SQL *statico*). Una famiglia alternativa di comandi, che rientra sempre tra le soluzioni con SQL Embedded, permette l'uso di SQL *dinamico*. Con questi comandi è possibile costruire un programma che esegue delle istruzioni SQL costruite al momento dell'esecuzione del programma. Questo meccanismo richiede però un supporto speciale da parte del sistema.

Il problema principale che deve essere affrontato è costituito dal 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 sono i parametri richiesti in ingresso e quali sono 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 la modalità di interazione con il sistema. Nel caso di SQL statico, i comandi SQL vengono gestiti da un preprocessore che analizza la struttura del comando e ne può costruire una traduzione nel linguaggio interno del sistema. In questo modo il comando non deve essere analizzato e ottimizzato ogni volta che viene richiesta la sua esecuzione. Ciò porta dei considerevoli vantaggi in termini di prestazioni. SQL dinamico cerca di offrire quando possibile questi vantaggi, mettendo a disposizione due diverse modalità di interazione: si può eseguire direttamente l'interrogazione, per cui alla analisi segue immediatamente l'esecuzione della interrogazione, o la gestione della 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 una 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 ad esempio 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 viene eseguito più volte, o quando il programma deve gestire uno scambio di parametri di ingresso o di uscita con l'istruzione, diventa necessario distinguere le due fasi di preparazione e di esecuzione.

Fase di preparazione Il comando `prepare` analizza un'istruzione SQL e la traduce nel linguaggio procedurale interno del sistema. 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. Ad esempio:

```
prepare :comando
  from "select Città from Dipartimento where Nome = ?"
```

In questo modo alla variabile `comando` del prógramma corrisponde la traduzione della istruzione, con un parametro di ingresso che rappresenta il nome del dipartimento che deve essere selezionato dalla interrogazione.

Quando una istruzione SQL che era stata tradotta non serve più, è possibile rilasciare la memoria occupata dalla traduzione dell'istruzione utilizzando il comando di `deallocate prepare`, con la seguente sintassi:

`deallocate prepare` *NomeComando*

Ad esempio, per deallocare il comando precedente, si potrà utilizzare il seguente comando:

```
deallocate prepare :comando
```

Fase di esecuzione Per eseguire un comando che è stato preelaborato da una `prepare` si usa il comando di `execute`, con la seguente sintassi:

`execute` *NomeComando* [`into` *ListaTarget*] [`using` *ListaParametri*]

La lista dei target contiene l'elenco dei parametri in cui deve essere scritto il risultato dell'esecuzione del comando (questa parte è opzionale qualora il comando SQL non restituisca dei valori). La lista dei parametri specifica invece quali sono i valori che devono essere assunti dai parametri variabili della lista (anche questa parte può non essere presente qualora il comando SQL sia privo di parametri).

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.

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 della interrogazione invece che l'interrogazione stessa, e che i comandi d'uso del cursore ammettono la specifica delle 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
```

6.4 Call Level Interface (CLI)

Le soluzioni che sono state descritte fino a ora ricadono nella famiglia delle soluzioni SQL Embedded, dove si prevede di disporre di un preprocessore 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 che sono presenti nella libreria offerta dal DBMS e che realizzano il dialogo con la base di dati.

Una soluzione alternativa consiste nel mettere direttamente a disposizione del programmatore un insieme di funzioni 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 sulla 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 e descrivere 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 6.3 viene mostrato un quadro riassuntivo delle diverse soluzioni.

6.4.1 ODBC e soluzioni proprietarie Microsoft

La piattaforma software Microsft 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 una 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 della 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. 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 una applicazione che faccia accesso a un DBMS.

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 6.3 Caratteristiche delle soluzioni CLI

L'obiettivo di ODBC è di permettere la costruzione di applicazioni che facciano accesso a basi di dati relazionali di tipo eterogeneo. 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 una interfaccia ODBC, le applicazioni possono accedere a dati presenti su sistemi relazionali generici, eventualmente presenti su sistemi remoti raggiungibili tramite una rete; il linguaggio supportato da ODBC è un SQL "ristretto", caratterizzato cioè da un insieme limitato di istruzioni, definito nell'ambito del comitato *SQL Access Group* (SAG).

Nella architettura ODBC, il collegamento tra una applicazione e un server richiede 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 garantire dei driver che prevedano l'uso del DBMS nell'ambito di una specifica rete e con uno specifico sistema operativo. Per esempio, la tripla (*Sybase, Windows/NT, Novell*) identifica uno specifico driver. Il driver così maschera tutti i problemi di interoperabilità (non solo quelli posti dal DBMS), e facilita la scrittura delle applicazioni.

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. L'applicazione è trasparente rispetto alla scelta del protocollo di comunicazione, del server DBMS e del sistema operativo del nodo ove il DBMS è installato, tutti mascherati dal driver.
- Il *driver manager* è responsabile di caricare i driver a richiesta dell'applicazione. Questo software è fornito dalla Microsoft e garantisce anche alcune funzioni per gestire corrispondenze fra nomi e funzioni di inizializzazione che assicurano il buon funzionamento dei driver.
- I *driver* sono responsabili di eseguire funzioni ODBC; pertanto, sono in grado di eseguire interrogazioni in SQL, possibilmente traducendole in modo da adattarsi alla sintassi e alla semantica degli specifici prodotti cui viene fatto accesso. I driver sono anche responsabili di restituire i risultati alle applicazioni, tramite meccanismi di buffering.
- La fonte di informazione (in inglese: *data source*) è il sistema che esegue le funzioni trasmesse dal *client*.

I codici di errore sono standardizzati, così da consentire il controllo delle condizioni di errore a tempo di esecuzione. Le interrogazioni SQL possono essere specificate in modo statico oppure essere incluse in stringhe che vengono generate ed eseguite dinamicamente. L'esecuzione può quindi generare degli errori, quando il codice SQL contenuto nella stringa è scorretto.

ODBC è quindi un meccanismo standard che risolve un problema significativo. Data la sua complessità non è però adatto a essere utilizzato direttamente dal programmatore per lo sviluppo di applicazioni. A questo scopo Microsoft ha sviluppato delle soluzioni proprietarie, che semplificano la realizzazione di applicazioni che fanno accesso ai servizi di una base di dati. Queste soluzioni innalzano il livello di astrazione, sfruttando i vantaggi del paradigma di programmazione a oggetti, estendendo inoltre 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 ad 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. Ad 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) costituisce un'interfaccia di alto livello ai servizi offerti da OLE DB. Il modello di ADO si basa sulla definizione di quattro concetti 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 si assume che diverse anomalie possono essere attivate contemporaneamente. Il comando (*Command*) rappresenta la stringa di caratteri che contiene l'istruzione SQL che si vuole far eseguire sulla 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 che verrà usata per conservare i risultati dei comandi inviati alla sorgente dati; il RecordSet offre meccanismi per scandire l'insieme di tuple e accedere a una tupla alla volta, in modo analogo ai cursori.

ADO è utilizzabile nei diversi linguaggi di programmazione utilizzati per lo sviluppo di applicazioni sulla piattaforma Microsoft Windows (ad es., Visual C++, Visual Basic, VBscript, Jscript). La figura 6.4 presenta un esempio di una procedura Visual Basic che fa uso di ADO.

Anche senza conoscere il linguaggio, dovrebbe essere facile comprendere la struttura della procedura `InterrogaBD` in figura 6.4. 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 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 della Microsoft o a testi specifici sull'argomento per avere il livello di dettaglio che consenta di sviluppare direttamente 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. In ADO.NET invece i dati vengono gestiti tramite i *DataSet*, i quali costituiscono

```
(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 6.4 Una procedura Visual Basic che accede a una sorgente dati ODBC tramite ADO

dei contenitori di oggetti (tra cui i *DataTable*, che rappresentano le normali tabelle). A differenza di ADO, l'accesso interno agli elementi dei *DataSet* e *DataTable* avviene utilizzando i normali meccanismi di accesso a collezioni di oggetti.

Un altro aspetto che distingue ADO.NET è che un *DataSet* permette la gestione di relazioni e vincoli di integrità tra gli oggetti al suo interno. Questa flessibilità è resa possibile dal fatto che i *DataSet* rappresentano strutture che risiedono pienamente nell'ambiente del programma, mentre in ADO il risultato di una query viene mantenuto nel contesto della sessione con la base di dati e diviene disponibile al programma tupla per tupla tramite l'invocazione dei metodi di accesso al *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 (ad 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 invece aggiorna lo stato della sorgente dati riportando in essa le variazioni introdotte sul *DataSet*. Vi sono molteplici 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. Oltre a ciò, ADO.NET è coerente all'impostazione comune di .NET e permette di scambiare facilmente oggetti all'interno dell'ambiente .NET, utilizzando XML come formato di rappresentazione.

6.4.2 Java Database Connectivity (JDBC)

Java è un moderno linguaggio di programmazione, proposto dalla Sun Microsystems, che sta riscuotendo un notevole successo nello sviluppo di applicazioni, soprattutto dove esistono problemi significativi di portabilità 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 del linguaggio che hanno contribuito al suo successo. Java è un moderno linguaggio di programmazione a oggetti, ma il linguaggio di programmazione è solo il componente centrale di una proposta molto articolata. L'architettura prevede che i programmi Java vengano tradotti in un formato compatto (bytecode), e vengano quindi eseguiti da una Java Virtual Machine, normalmente realizzata da un sistema software. Per eseguire un'applicazione Java in un sistema di calcolo è quindi sufficiente disporre di una Java Virtual Machine per il particolare ambiente. Visto che esistono diverse implementazioni di Java Virtual Machine per molti diversi ambienti, un'applicazione Java è in grado di offrire un livello di portabilità molto elevato.

Il linguaggio, oltre a offrire tutte le caratteristiche di un moderno linguaggio di programmazione (modello a oggetti, eccezioni, meccanismi per l'esecuzione concorrente e la sincronizzazione, invocazione remota, sicurezza ecc.), è arricchito da una estesa libreria di moduli che offrono delle soluzioni immediate e robuste per costruire applicazioni in tanti diversi contesti. Ad 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. Tra questi moduli, è presente il modulo Java Database Connectivity (JDBC). JDBC permette a programmi Java di accedere in modo uniforme a basi di dati relazionali, in modo simile a quanto offerto dal protocollo ODBC. L'architettura 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). Vi sono molteplici architetture che possono essere utilizzate per implementare le funzioni di accesso al DBMS (vedi figura 6.5):

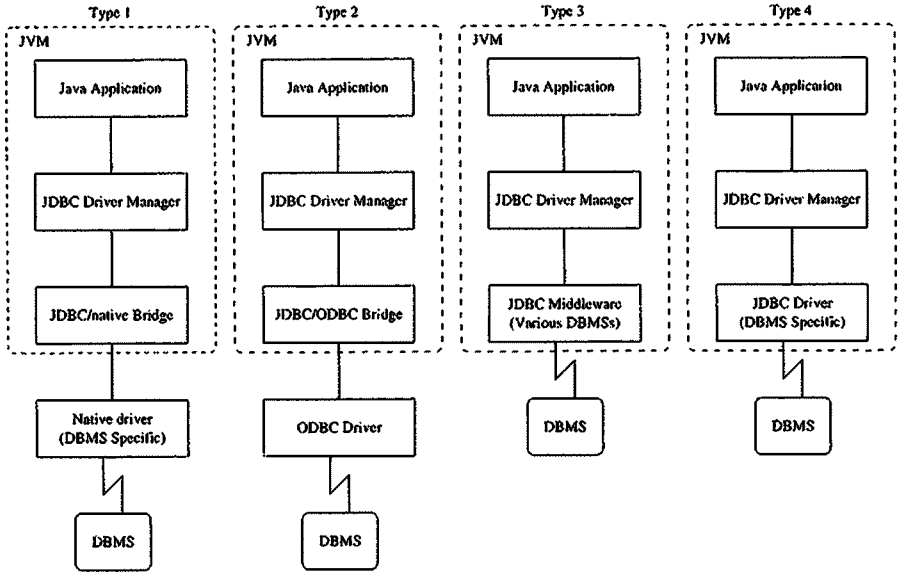


Figura 6.5 Le architetture JDBC

1. *Driver nativo*: questa soluzione prevede la realizzazione di funzioni Java che convertono le richieste JDBC in richieste a un driver in codice nativo della macchina che esegue il programma Java. Il driver è specifico per il DBMS cui si vuole accedere ed è normalmente un driver realizzato per permettere l'acces-

so al DBMS da parte di applicazioni scritte nei tradizionali linguaggi di alto livello.

2. *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.

Queste due soluzioni non sono realmente portabili, perché richiedono l'utilizzo di componenti *nativi*, cioè specifici dell'ambiente in cui vengono eseguiti. Esse sono diffuse in architetture a più livelli al fine di sfruttare la disponibilità di strumenti esistenti.

Vi sono poi due soluzioni che si appoggiano su un ambiente completamente Java:

3. *Middleware-server*: questa soluzione prevede l'uso di un server scritto in Java 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.
4. *Driver Java*: questa soluzione prevede l'uso di un driver specifico per il particolare sistema relazionale, in modo analogo ai driver in codice nativo usati in ODBC. I driver vengono normalmente offerti come opzioni 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.

Mostriamo l'uso di JDBC con un semplice esempio, che appare in figura 6.6. La prima istruzione specifica la posizione nell'albero delle classi Java delle classi che realizzano i 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 `conn` della classe `Connection`, che viene inizializzato a `null`. L'istruzione alla linea (7) specifica il driver che deve essere caricato (trascuriamo i dettagli); in questo caso il driver è di tipo 2 e si appoggia a una soluzione ODBC sottostante. Nell'istruzione alle linee (9-10) viene effettivamente stabilita la connessione con la base di dati, 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 in una di queste istruzioni, il sistema genererà un'eccezione che

terminerà l'esecuzione del programma (istruzione `System.exit(1)` alla linea (12)). Una volta che la connessione è stata stabilita, è possibile inviare comandi SQL alla base di dati. Nell'istruzione alle linee (15-16) viene creato l'oggetto `interrogazione` appartenente alla classe `Statement`, associato alla connessione `conn`. Il metodo `executeQuery` offerto dalla classe `Statement` viene invocato su `interrogazione`, fornendo come parametro il testo della query SQL di cui si richiede l'esecuzione immediata. 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 considerarle una alla volta. In ogni istante esiste in `ResultSet` una tupla corrente (all'inizio sarà la prima) e il metodo `next` permette di rendere tupla corrente la tupla successiva. Il metodo `getString` estrae dalla tupla corrente l'attributo il cui nome viene passato

```
(1)  import java.sql.*;
(2)  public class PrimoJdbc {
(3)    public static void main(String[] arg) {
(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)     }
(12)     catch (Exception e) { System.exit(1); }
(13)     try {
(14)       // Esecuzione dell'interrogazione
(15)       Statement interrogazione =
(16)                   conn.createStatement();
(17)       ResultSet risultato =
(18)         interrogazione.executeQuery(
(19)                         "select * from Corsi");
(20)       while (risultato.next()) {
(21)         String nomeCorso =
(22)                  risultato.getString("NomeCorso");
(23)         System.out.println(nomeCorso);
(24)       }
(25)     }
(26)     catch (Exception e) { System.exit(1); }
(27)   }
(28) }
```

Figura 6.6 Una classe Java che accede a una sorgente dati JDBC tramite un bridge JDBC-ODBC

come parametro al metodo (in questo caso `NomeCorso`); `getString` assume che l'attributo abbia un valore rappresentabile come stringa di caratteri; vi sono diversi altri metodi specifici per ciascun dominio (ad es., `getInt`, `getFloat`, `getDate` ecc.). Il ciclo `while` scandirà quindi il contenuto di `risultato` fino a che il metodo `next()` non restituirà il valore booleano falso.

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 [12] (di cui esiste la traduzione italiana) e i libri di Melton e Simon [47, 48].

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 della Sun 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.

Esercizi

Esercizio 6.1 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 quantita' 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 sì, viene inserito un nuovo elemento nella tabella ORDINEINCORSO, con i valori di *Prod* e del corrispondente attributo QtaRiordino.

Esercizio 6.2 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.

Esercizio 6.3 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 diartimento, ordinati per cognome.

Esercizio 6.4 Realizzare l'esercizio precedente usando ADO.

Esercizio 6.5 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.

Esercizio 6.6 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 INDICE(Numero,Titolo,NumPagine) in cui si presenta il numero di pagina nel quale inizia ogni capitolo, supponendo che il capitolo 1 inizia sulla prima pagina e che i capitoli devono iniziare su pagine dispari (eventualmente introducendo una pagina bianca alla fine del capitolo).

Parte Seconda

Progettazione di basi di dati

7

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 in qualche maniera 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 7.1, 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 7.2 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*. Le prime due fasi verranno presentate in dettaglio nei due capitoli successivi a questo. La terza fase si fonda su aspetti tecnologici che non sono affrontati nel dettaglio in questo testo e verrà quindi trattata brevemente, e a un livello piuttosto astratto, alla fine del capitolo sulla progettazione logica. Questa 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.

7.1 Introduzione alla progettazione

7.1.1 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 7.1, 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 nella individuazione e nello studio delle proprietà e delle funzionalità che il sistema informativo dovrà avere. Questa fase richiede una interazione con gli utenti del sistema e produce una descrizione completa, ma generalmente informale, dei dati coinvolti (anche in termini di 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 7.1, il processo non è quasi mai strettamente sequenziale in quanto spesso, durante l'esecuzione di una 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

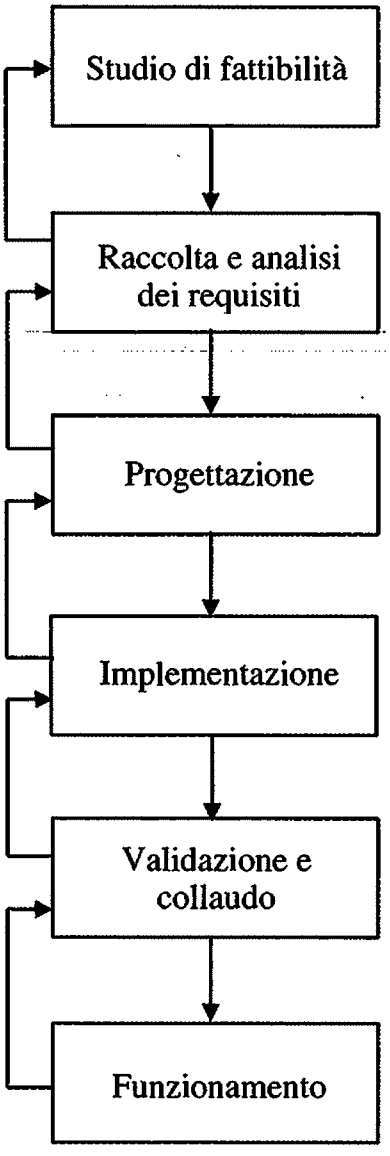


Figura 7.1 Ciclo di vita di un sistema informativo

fase del ciclo di vita riportato in figura 7.1, 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.

7.1.2 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* sia delle strategie che 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 (vedi la figura 7.2) 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.

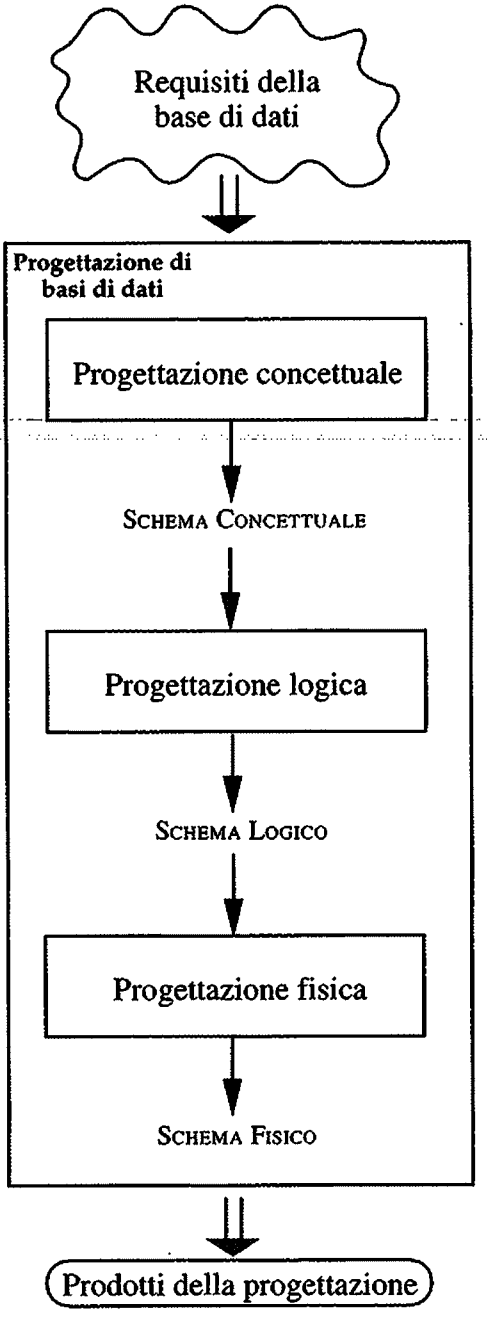


Figura 7.2 Le fasi della progettazione di una base di dati

- **Progettazione logica.** Consiste nella traduzione dello schema concettuale definito nella fase precedente, nel 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*, 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, può essere utile come riferimento per le operazioni di interrogazione e aggiornamento.

In figura 7.3 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

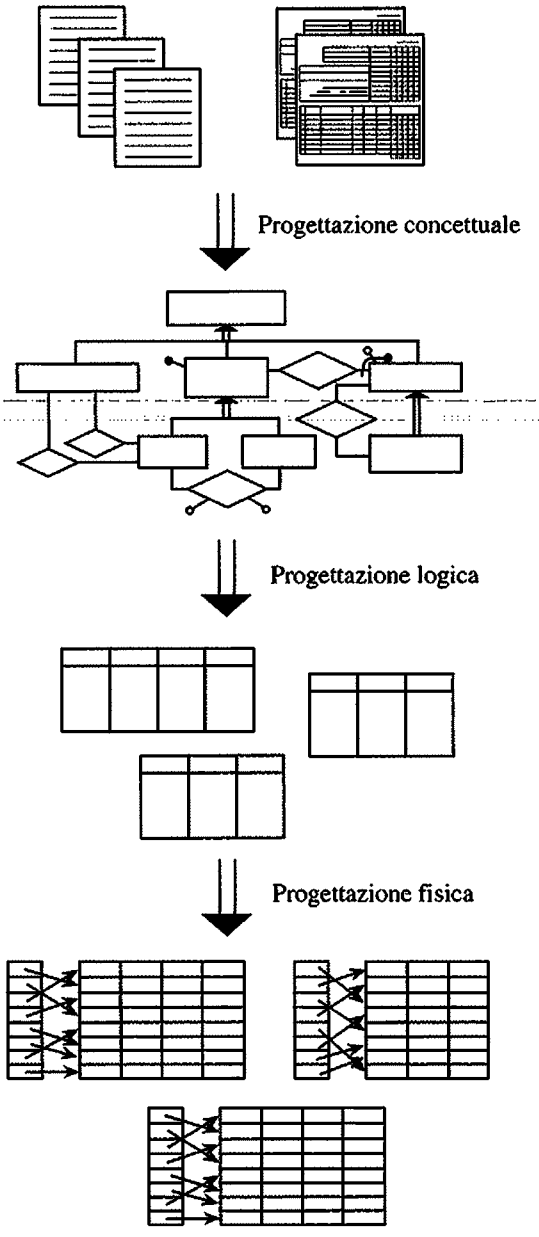


Figura 7.3 I prodotti delle varie fasi del progetto di una base di dati relazionale con il modello Entità-Relazione

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 7.2 e con riferimento ai modelli usati nella figura 7.3. 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, come abbiamo visto nei capitoli precedenti, è attualmente il modello dei dati più diffuso nei sistemi di gestione di basi di dati.

7.2 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 del tempo. Nella tabella in figura 7.4 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 ne semplifica l'interpretazione.

7.2.1 I costrutti principali del modello

Cominciamo ad analizzare i i costrutti principali di questo modello: le entità, le relazioni e gli attributi.

Entità Rappresentano classi di oggetti (fatti, cose, persone, per esempio) che hanno proprietà comuni ed esistenza "autonoma" ai fini dell'applicazione di interesse: CITTÀ, DIPARTIMENTO, IMPIEGATO, ACQUISTO e VENDITA sono esempi di entità di un'applicazione aziendale. Una occorrenza di una entità è un oggetto della classe che l'entità rappresenta. Le città di Roma, Milano e Palermo

[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.

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 7.4 I costrutti del modello E-R e la loro rappresentazione grafica

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 una esistenza (e una 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 7.5 riporta alcuni esempi di entità.

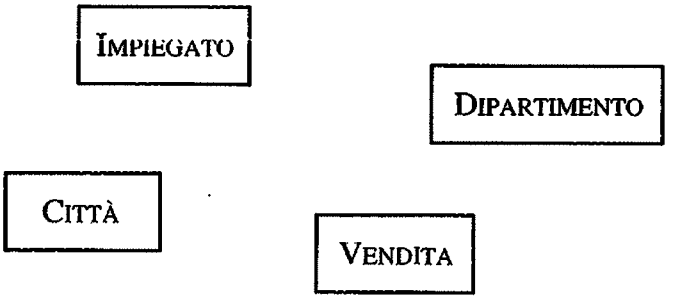


Figura 7.5 Esempi di entità nel modello E-R

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 una 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 entità STUDENTE e CORSO sono le coppie e_1, e_2, e_3, e_4, e_5 ed e_6 riportate in figura 7.6, nella quale vengono raffigurate anche le occorrenze delle entità coinvolte.

[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).

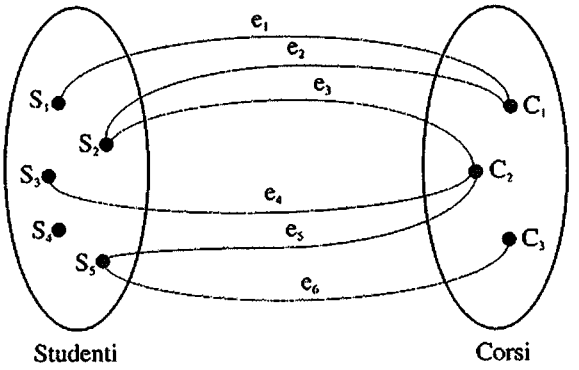


Figura 7.6 Esempio di occorrenze della relazione ESAME

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 7.7 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 a individuare un "verso" alla relazione. Per esempio, SEDE DI LAVORO è da preferire a LAVORA IN.

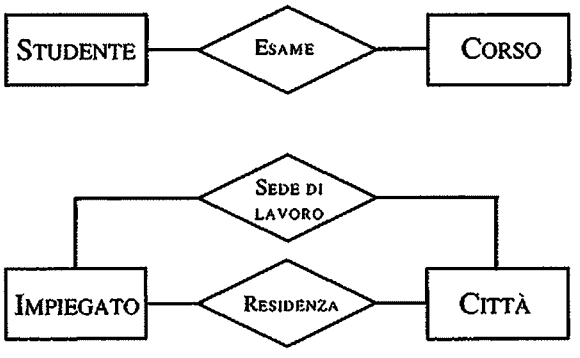


Figura 7.7 Esempi di relazioni tra entità nel modello E-R

Un aspetto molto importante delle associazioni è il seguente: come risulta evidente osservando la figura 7.6, 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, ovvero, 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 7.7 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 7.8 la relazione ricorsiva 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

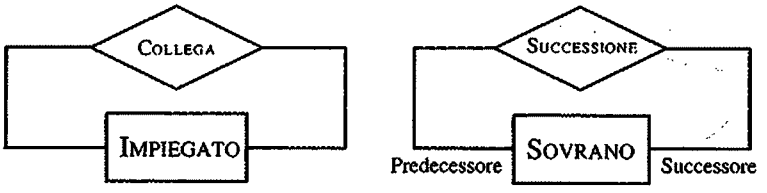


Figura 7.8 Esempi di relazioni ricorsive nel modello E-R

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 7.9: la relazione FORNITURA tra le tre entità FORNITORE, PRODOTTO e DIPARTIMENTO descrive il fatto che un fornitore rifornisce un dipartimento di un certo prodotto.

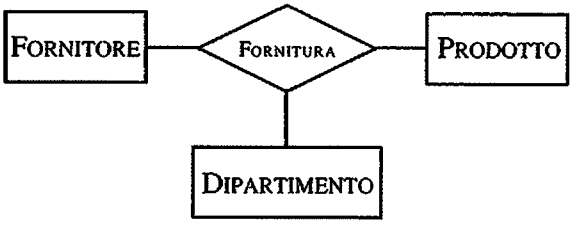


Figura 7.9 Esempio di relazione ternaria nel modello E-R

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 7.10 (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 e Età

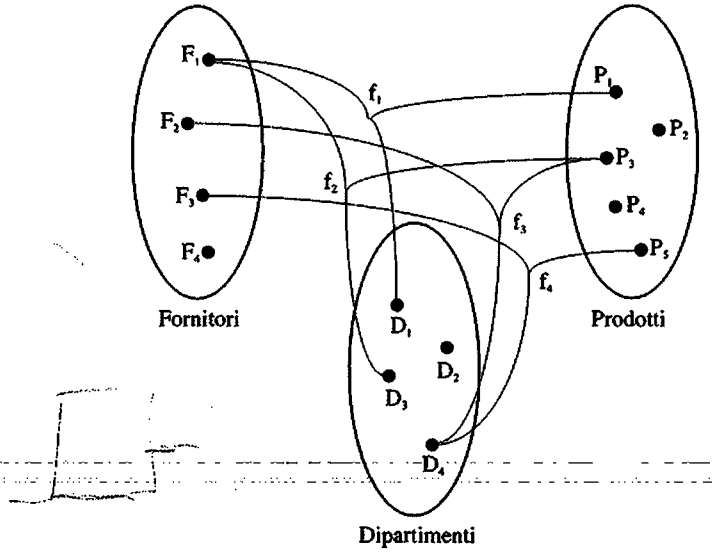


Figura 7.10 Esempio di occorrenze della relazione FORNITURA

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 7.11 viene mostrato come vengono rappresentati graficamente gli attributi. I domini non vengono riportati nello schema, ma sono generalmente descritti nella documentazione associata.

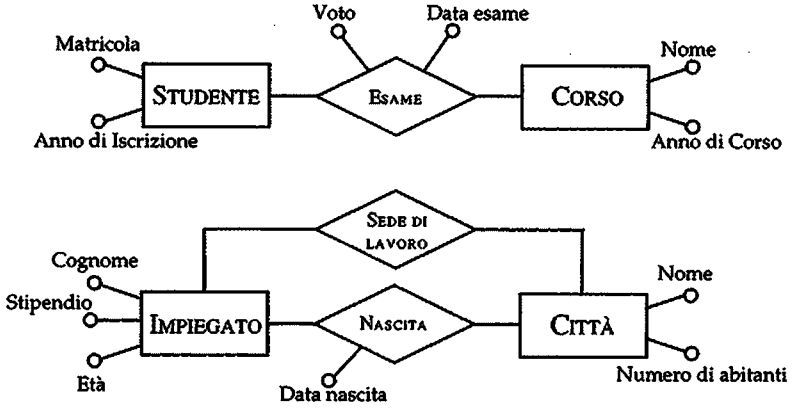


Figura 7.11 Schemi E-R con relazioni, entità e attributi

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 composto viene mostrata nell'esempio in figura 7.12. Per ridurre la complessità degli schemi, gli attributi composti verranno usati raramente nel seguito preferendo usare, per quanto possibile, attributi atomici.

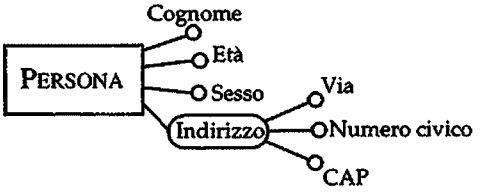


Figura 7.12 Un esempio di entità con attributo composto

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 7.13. 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 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 (entità 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).

7.2.2 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, ovvero proprietà che occorrenze di entità e di relazioni devono soddisfare per poter essere considerate "valide".

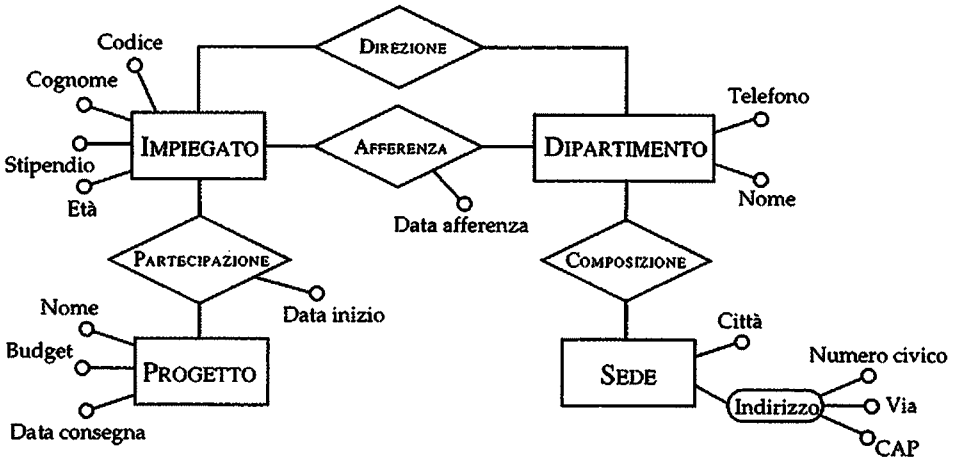


Figura 7.13 Uno schema Entità-Relazione

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 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 7.14.

Figura 7.14 Cardinalità di una relazione nel modello E-R

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 una associazione con un numero arbitrario di occorrenze dell'altra entità.

Se analizziamo la figura 7.6 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 ad e_2 ed e_3).

In figura 7.15 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.

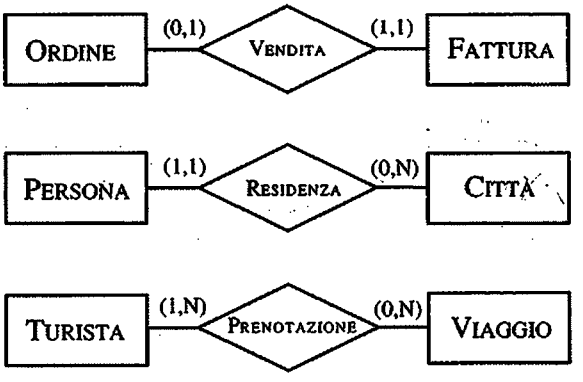


Figura 7.15 Esempi di cardinalità di relazioni

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 7.15, 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 7.15, 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 7.15, 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 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 7.15, 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 ad N. Un esempio concreto viene fornito dalla relazione ternaria FORNITURA di figura 7.9: come si può osservare dalla figura 7.10, 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 una entità partecipa a una relazione n-aria con cardinalità massima pari ad 1, significa che ogni sua occorrenza può essere legata ad una sola occorrenza della relazione, e quindi a una 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 comunque questo argomento nel capitolo 10 dedicato alla normalizzazione, nel quale 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 ad 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 minima pari a zero nel primo caso, e una cardinalità massima pari a molti (N), nel secondo. In figura 7.16 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

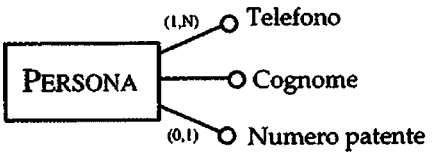


Figura 7.16 Esempio di attributi di entità con cardinalità

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 7.16, 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 ai paragrafo 8.2, 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 7.17 viene mostrata la simbologia usata per rappresentare 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.

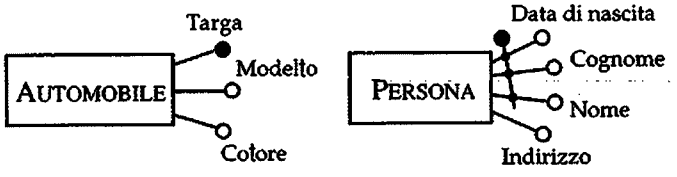


Figura 7.17 Esempi di identificatori interni di entità

Alcune volte però gli attributi di una entità non sono sufficienti a identifica-

re univocamente le sue occorrenze. Si consideri per esempio l'entità STUDENTE nello schema in figura 7.18. 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, una 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 una entità è ottenuta utilizzando altre entità si parla di identificatore *esterno*. La rappresentazione diagrammatica di un identificatore esterno è riportata nell'esempio di figura 7.18.

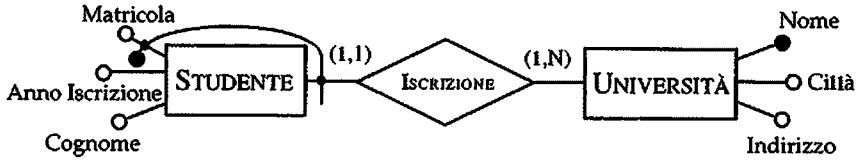


Figura 7.18 Esempio di identificatore esterno di entità

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);
- una identificazione esterna può coinvolgere uno o più entità, ognuna delle deve essere membro di una relazione alla quale l'entità da identificare partecipa con cardinalità (1,1);
- una identificazione esterna può coinvolgere una 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 7.13 introducendo cardinalità e identificatori. Lo schema risultante viene proposto in figura 7.19.

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

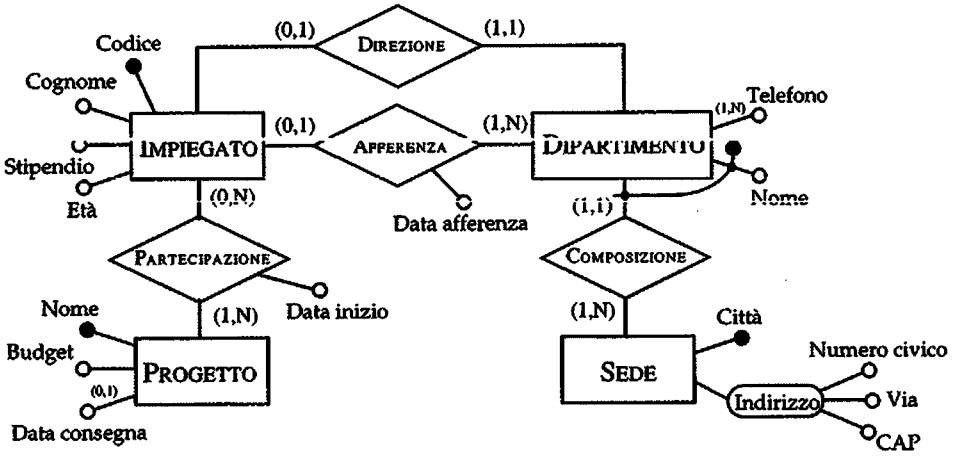


Figura 7.19 Lo schema di figura 7.13 completato con identificatori e cardinalità

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 una 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 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 una 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 7.20. Si osservi che, per le entità figlie, le proprietà ereditate non vanno rappresentate esplicitamente.

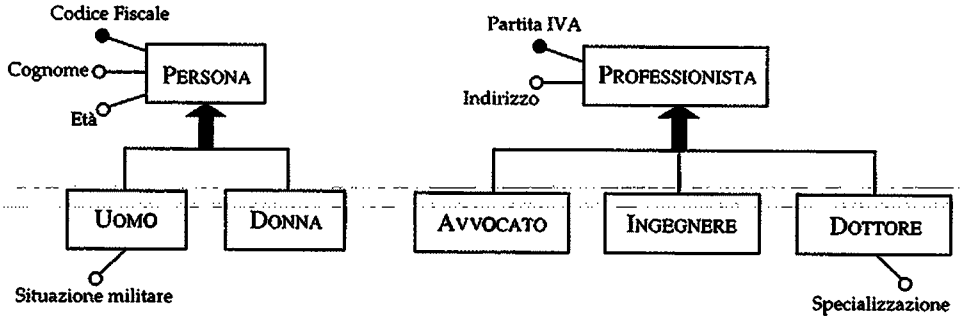


Figura 7.20 Esempi di generalizzazioni tra entità

Le generalizzazioni possono essere classificate sulla base di due proprietà tra loro ortogonali.

- Una generalizzazione è *totale* se ogni occorrenza della entità genitore è una occorrenza di almeno una delle entità figlie, altrimenti è *parziale*.
- Una generalizzazione è *esclusiva* se ogni occorrenza della entità genitore è al più una occorrenza di una delle entità figlie, altrimenti è *sovrapposta*.

La generalizzazione tra PERSONA, UOMO e DONNA in figura 7.20 è, per esempio, totale (gli uomini e le donne costituiscono "tutte" le persone) ed esclusiva (una persona è o uomo o donna). Una generalizzazione tra l'entità VEICOLO e le entità AUTOMOBILE e BICICLETTA è invece parziale ed esclusiva, perché ci sono altri tipi di veicoli (per esempio le moto) che non sono né automobili, né biciclette. 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 (vedi un esempio in figura 7.21). 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 7.21 viene mostrata una gerarchia di generalizzazioni. Il legame che esiste tra l'entità RESPONSABILE DI PROGETTO e PROGETTISTA è un esempio di sottoinsieme.

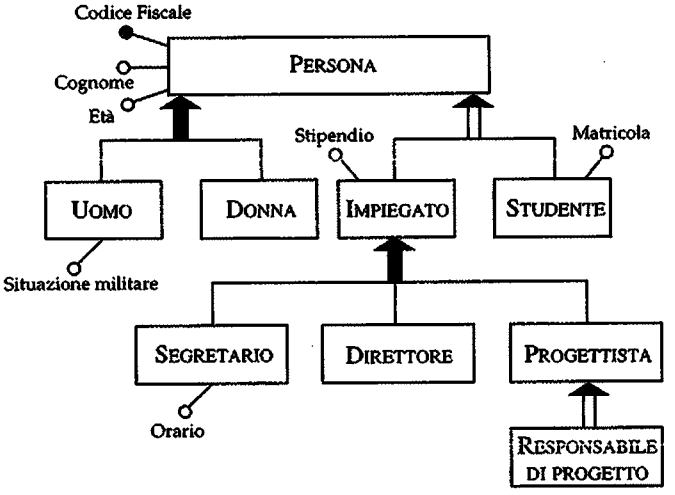


Figura 7.21 Gerarchia di generalizzazioni tra entità

7.2.3 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 7.22 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. Una 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

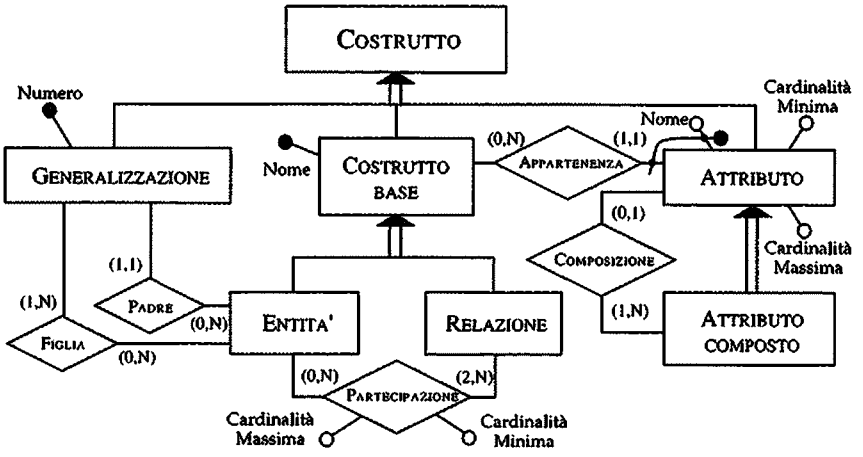


Figura 7.22 Descrizione del modello E-R con il modello E-R

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 concetto di base, cioè a una 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. Una 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 esterna). Ci possono essere cioè attributi con lo stesso nome ma devono appartenere a relazioni o entità diverse (vedi per esempio l'attributo Nome in figura 7.19). 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 nella 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.

7.3 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 7.19, 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 7.19, 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 7.19 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 ed 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 sempli-

ci 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 ad essa.

7.3.1 Regole aziendali

Uno degli strumenti più usati dagli analisti di sistemi informativi per la descrizione di proprietà di una 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 una 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 stati 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 7.19, 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 ottenere il concetto derivato. Una possibile struttura è quindi:

< *concetto* > `si ottiene` < *operazione su concetti* >

Per esempio, se nel nostro esempio 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 e generici o asserzioni di SQL-2 (come descritto nei capitoli 4 e 5);
- mediante trigger o *regole attive*;
- con opportune procedure scritte in qualche linguaggio di programmazione.

7.3.2 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 7.19 è riportato in figura 7.23. 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 7.19 viene riportata in figura 7.24.

7.4 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

Entità	Descrizione	Attributi	Identificatore
Impiegato	Impiegato che lavora nella 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 7.23 Il dizionario dei dati per lo schema in figura 7.19

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 7.24 Regole aziendali per lo schema in figura 7.19

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.

A causa del 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 sulla 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 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 7.1.2 di questo capitolo (e approfondita nei prossimi capitoli) rimane quella di riferimento.

7.4.1 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, ovvero 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 (vedi 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 una applicazione software, si prestano, con opportuni accorgimenti, anche alla descrizione dello schema concettuale di una base di dati.

7.4.2 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 una 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 7.25, 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.

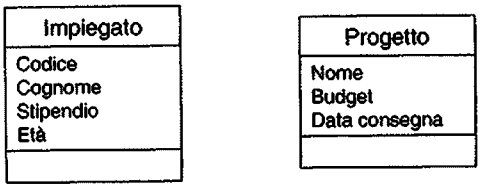


Figura 7.25 Rappresentazione di classi in UML

C'è da aggiungere che, a differenza del modello E-R, per una classe è possibile specificare nel riquadro in basso del rettangolo (lasciato vuoto in figura) che la rappresenta 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 7.25 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.

Associazioni Corrispondono alle relazioni del modello E-R e vengono rappresentate come indicato negli esempi in figura 7.26.

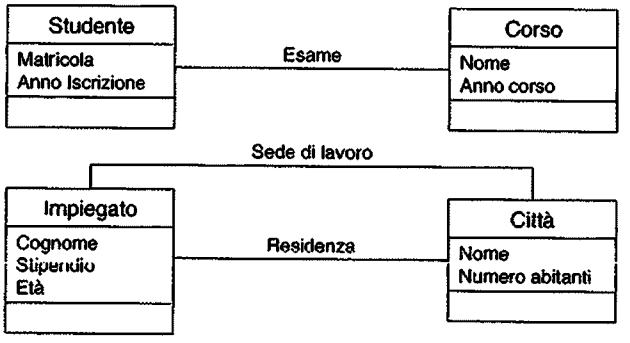


Figura 7.26 Associazioni binarie in UML

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 (vedi per esempio le due associazioni esistenti tra le classi `Impiegato` e `Città` in figura 7.26) 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, alla associazione da descrivere. La classe `Esame` in figura 7.27 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.

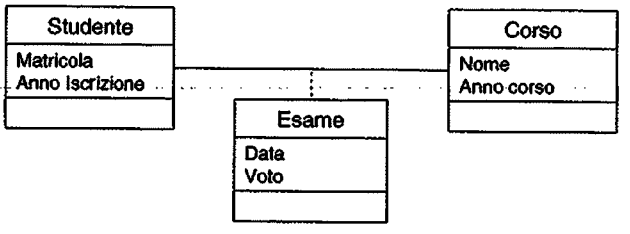


Figura 7.27 Una classe di associazione in UML

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 7.28 nella quale 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.

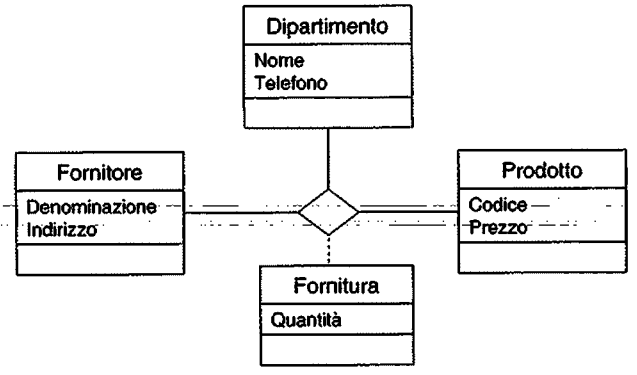


Figura 7.28 Una associazione ternaria in UML

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*, ovvero di trasformare l'associazione in una classe legata alle classi originarie con associazioni binarie. Per esempio, la reificazione dell'associazione in figura 7.28 produce lo schema riportato in figura 7.29. Nel seguito della trattazione, considereremo quindi solo associazioni binarie.

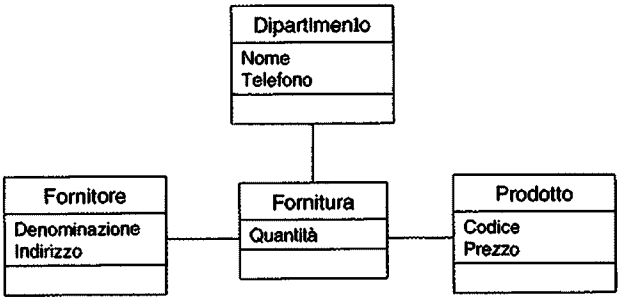


Figura 7.29 Una associazione ternaria in UML

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 una 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 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 7.30. La prima ci dice che un tecnico fa parte di un team, la seconda che una filiale è parte di un'azienda.

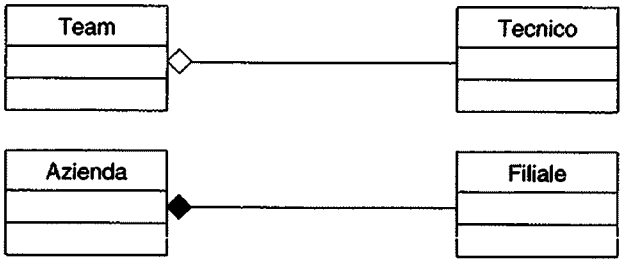


Figura 7.30 Aggregazioni e composizioni in UML

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 7.30 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 la 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, un 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 una 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 ad uno schema E-R, le cardinalità delle associazioni binarie nei diagrammi delle classi risultano invertite. Esempi di uso di molteplicità in associazioni

vengono riportate in figura 7.31, nella quale sono ripresi gli esempi discussi a pagina 214 e rappresentati nel modello E-R in figura 7.15.

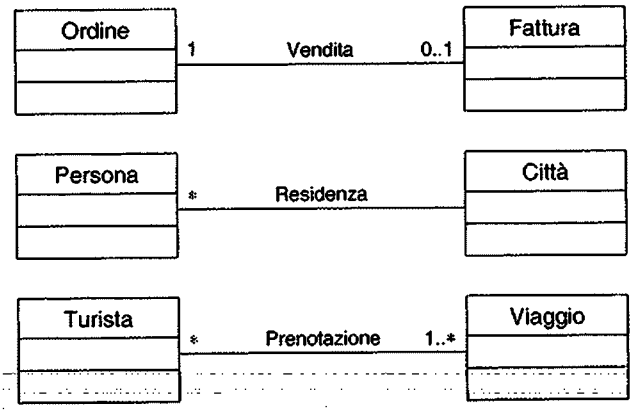


Figura 7.31 Associazioni con molteplicità in UML

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 a uno e un solo ordine associato. L'assenza di molteplicità nella associazione `Afferenza` 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 7.31 con la figura 7.15 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 è 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 7.32 dove un identificatore costituito da un so-

lo attributo viene specificato con il vincolo utente {id}, mentre 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.

Figura 7.32 Identificatori in UML

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 7.33 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 7.18 a pagina 217.

Figura 7.33 Identificatore esterno in UML

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. Ad 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. Ad esempio, la figura 7.34 ripropone, in termini di UML, gli schemi presentati in figura 7.20 a pagina 219. 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.

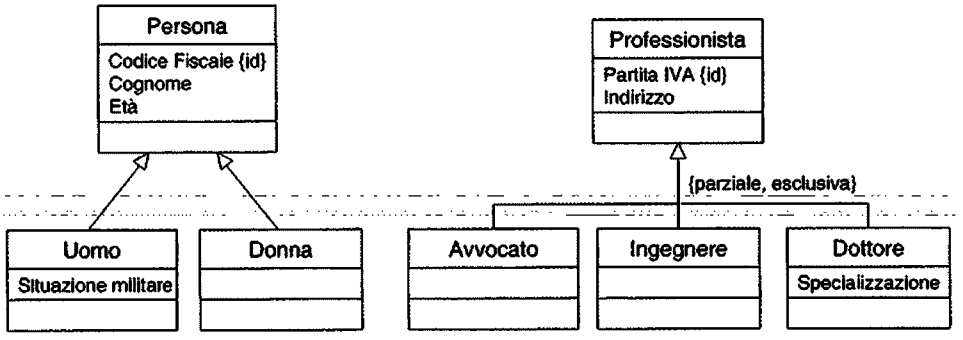


Figura 7.34 Generalizzazioni in UML

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 7.34, 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 7.35 viene riproposto, sotto forma di diagramma delle classi UML, lo schema E-R in figura 7.19 (pagina 218).

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

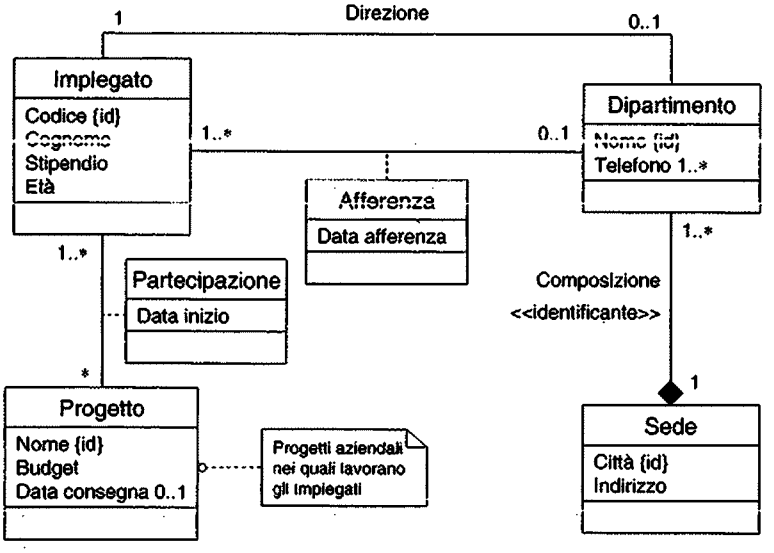


Figura 7.35 Il diagramma delle classi UML.

`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.

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 9.5.

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 e altri [36], quello di Pressman [52] e quello di Sommerville [59].

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 e altri [41] come risultato di un workshop

tenuto nel 1979. Un trattamento dettagliato della progettazione concettuale e logica è offerto dal libro in italiano di Batini e altri [8] e da quello in inglese di Batini, Ceri e Navathe [7]. Altre letture interessanti sono i testi di Mannila e Raiha [45], Teorey [61] e Wiederhold [68]. 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 [31] e di Ramakrishnan [53].

Il modello Entità-Relazione è di solito attribuito a Chen [21], 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 [58]. Un libro che presenta, oltre al modello E-R, altri modelli di dati è quello di Tsichritzis e Lochovski [63]. Infine, in un articolo di rassegna molto interessante, Hull e King hanno confrontato diversi modelli concettuali [38].

Le regole aziendali sono trattate diffusamente nel libro di Fleming e von Halle [32].

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 [9]. Un libro più snello che introduce gli aspetti essenziali di UML è quello di Fowler [33]. Infine, un ottimo testo sull'uso di UML nella progettazione di applicazioni software è quello di Larman [42].

Esercizi

Esercizio 7.1 Considerare lo schema E-R in figura 7.36: lo schema rappresenta varie proprietà di uomini e donne.

- 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.

Esercizio 7.2 Aggiungere le cardinalità minime e massime allo schema prodotto nell'esercizio 7.1 e gli identificatori principali. Dire se esistono dei vincoli di integrità sullo schema che non possono essere espressi con il modello Entità-Relazione.

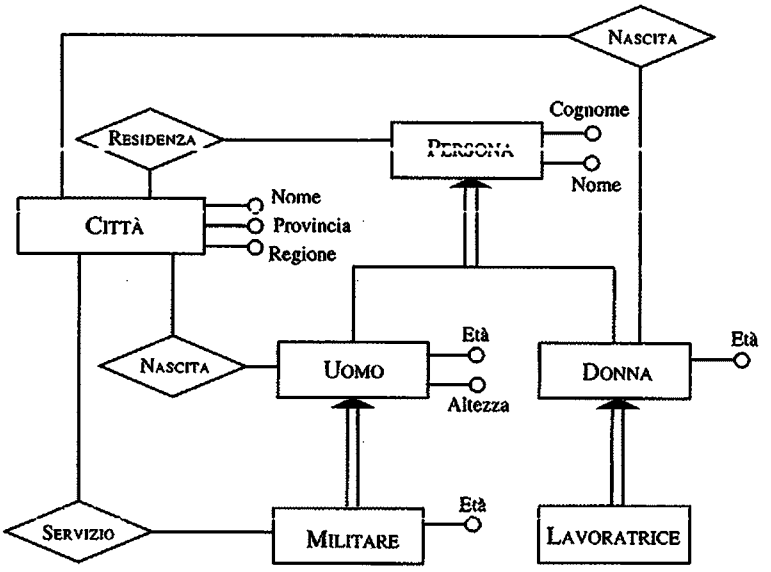


Figura 7.36 Schema E-R per l'esercizio 7.1

Esercizio 7.3 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, un indirizzo e, qualcuno, un nome d'arte.

Esercizio 7.4 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.

Esercizio 7.5 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 2001), 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.

Esercizio 7.6 Si consideri lo schema Entità-Relazione in figura 7.37. Descrivere le informazioni che esso rappresenta utilizzando il linguaggio naturale.

Esercizio 7.7 Tradurre in regole aziendali le seguenti proprietà sui concetti dello schema di figura 7.37.

- 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.

Esercizio 7.8 Modificare lo schema Entità-Relazione in figura 7.37 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.

Esercizio 7.9 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.

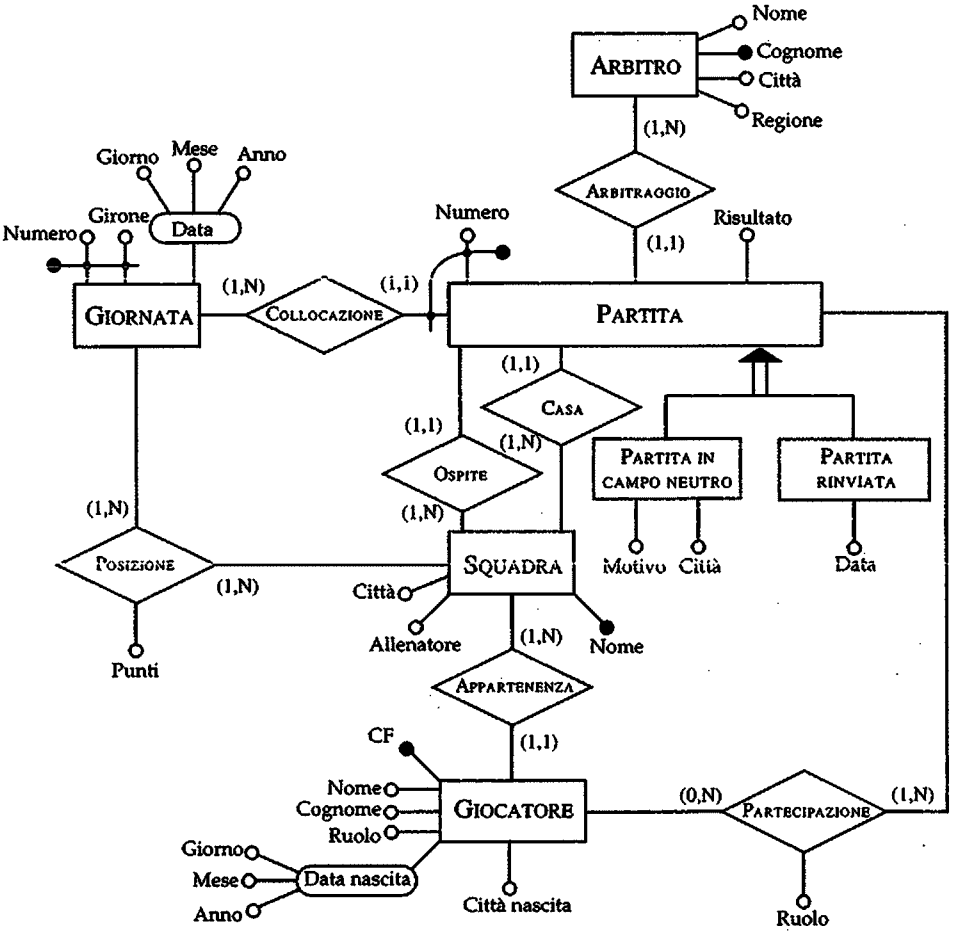


Figura 7.37 Schema E-R per l'esercizio 7.6

- Entità: sport, nazione e superficie. Concetto: il fatto che uno sport si pratichi in una nazione su una certa superficie (ad 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 ciascun azienda, con il relativo costo totale.

Esercizio 7.10 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.

Esercizio 7.11 Rappresentare lo schema Entità-Relazione in figura 7.37 con un diagramma della classi UML.

8

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 una 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 possibili 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 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 una applicazione per la gestione dei dati di una società di formazione.

8.1 La raccolta e l'analisi dei requisiti

Va detto innanzitutto che il reperimento e l'analisi dei requisiti di una 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) che 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 attività di raccolta. In molti casi è l'attività stessa di analisi dei requisiti che suggerisce successive attività di raccolta.

I requisiti di una applicazione provengono, nella maggior parte dei casi, da fonti diverse. Le principali fonti di informazione sono, in genere, le seguenti.

- Gli *utenti della 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, una 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 8.1. Si noti che abbiamo acquisito in questa fase anche informazioni sul carico previsto dei dati a regime.

	Società di formazione
1	*Si vuole realizzare una base di dati per una società che eroga cor-*
2	*si, di cui vogliamo rappresentare i dati dei partecipanti ai corsi e*
3	*dei docenti. Per i partecipanti (circa 5000), identificati da un codi-*
4	*ce, si vuole memorizzare il codice fiscale, il cognome, l'età, il sesso,*
5	*il luogo di nascita, il nome dei loro attuali datori di lavoro, i posti*
6	*dove hanno lavorato in precedenza insieme al periodo, l'indirizzo e*
7	*il numero di telefono, i corsi che hanno frequentato (i corsi sono in*
8	*tutto circa 200) e il giudizio finale. Rappresentiamo anche i seminari*
9	*che stanno attualmente frequentando e, per ogni giorno, i luoghi e*
10	*le ore dove sono tenute le lezioni. I corsi hanno un codice, un titolo*
11	*e possono avere varie edizioni con date di inizio e fine e numero di*
12	*partecipanti. Se gli studenti sono liberi professionisti, vogliamo co-*
13	*noscere l'area di interesse e, se lo possiedono, il titolo. Per quelli*
14	*che lavorano alle dipendenze di altri, vogliamo conoscere invece il*
15	*loro livello e la posizione ricoperta. Per gli insegnanti (circa 300),*
16	*rappresentiamo il cognome, l'età, il posto dove sono nati, il nome del*
17	*corso che insegnano, quelli che hanno insegnato nel passato e quel-*
18	*li che possono insegnare. Rappresentiamo anche tutti i loro recapiti*
19	*telefonici. I docenti possono essere dipendenti interni della società o*
20	*collaboratori esterni.*

Figura 8.1 Esempio di requisiti espressi in linguaggio naturale

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*.

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 utiliz-

zato anche per indicare un concetto diverso a riga 10) e *giudizio* (riga 8), 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* (righe 13–14).
- **Individuare sinonimi/omonimi e unificare i termini**. I *sinonimi* indicano termini diversi con lo stesso significato (per esempio, *docente* a riga 3 e *insegnante* a riga 15, oppure *partecipante* a riga 2 e *studente* a riga 12); gli *omonimi* indicano termini uguali con diversi significati (per esempio *posto*, riferito a impiego a riga 5 e a città a riga 16, e *luogo*, riferito a città a riga 5 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, nelle righe 6 e 7, 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 8.2.

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 5) 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 telefono* fanno riferimento ai datori di lavoro dei partecipanti. Il *giudizio* (riga 8) 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

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 8.2 Un esempio di glossario dei termini

12) 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 15) è sinonimo di *docente*, *posto* (riga 16) indica la *città* di nascita, il *nome* del corso che insegnano (riga 16) è un sinonimo di *titolo* del corso e il *recapito telefonico* (riga 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 su i dati riportata in figura 8.3.

Naturalmente, accanto alle specifiche sui dati, vanno raccolte le specifiche sulle operazioni da effettuare su questi dati. Bisogna cercare di utilizzare 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);

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 nel 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 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 nel passato e di quelli che possono insegnare. I docenti possono essere dipendenti interni della società di formazione o collaboratori esterni.*

Figura 8.3 Esempio di strutturazione dei requisiti

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, sulla edizione alla quale hanno partecipato e la 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.

8.2 Criteri generali di rappresentazione

Prima di affrontare le metodologie di progetto, cerchiamo di stabilire alcuni criteri generali per tradurre una specifica informale in un costrutto del modello Entità-Relazione. Va precisato che spesso non esiste una rappresentazione univoca di un insieme di specifiche, perché le stesse informazioni 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 una 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 e già citato è il concetto di *esame* relativo a studenti e corsi. Questo concetto può essere rappresentato con una relazione

tra studente e corso nel caso sia di interesse, per esempio, solo la votazione. Ma se dell'esame sono di interesse anche la data, la sessione, la commissione e, soprattutto, vogliamo descrivere il fatto che lo stesso studente può sostenere più volte un esame per lo stesso corso, allora l'esame va rappresentato con una entità collegata da relazioni uno a molti con le entità che rappresentano gli studenti e i corsi.

- *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.

8.3 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.

8.3.1 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 8.4 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.

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*. In figura 8.5 vengono riportati alcuni esempi di primitive di trasformazione top-down. Come si può vedere, queste primitive operano su un singolo concetto dello schema e lo trasformano in una struttura più complessa in grado di descrivere il concetto di partenza con maggior dettaglio.

- Trasformazione T_1: si applica quando si comprende che una entità descrive in realtà due concetti diversi legati logicamente tra di loro. Per esempio, nell'applicazione relativa alla società di formazione, questo può accadere quando ci si

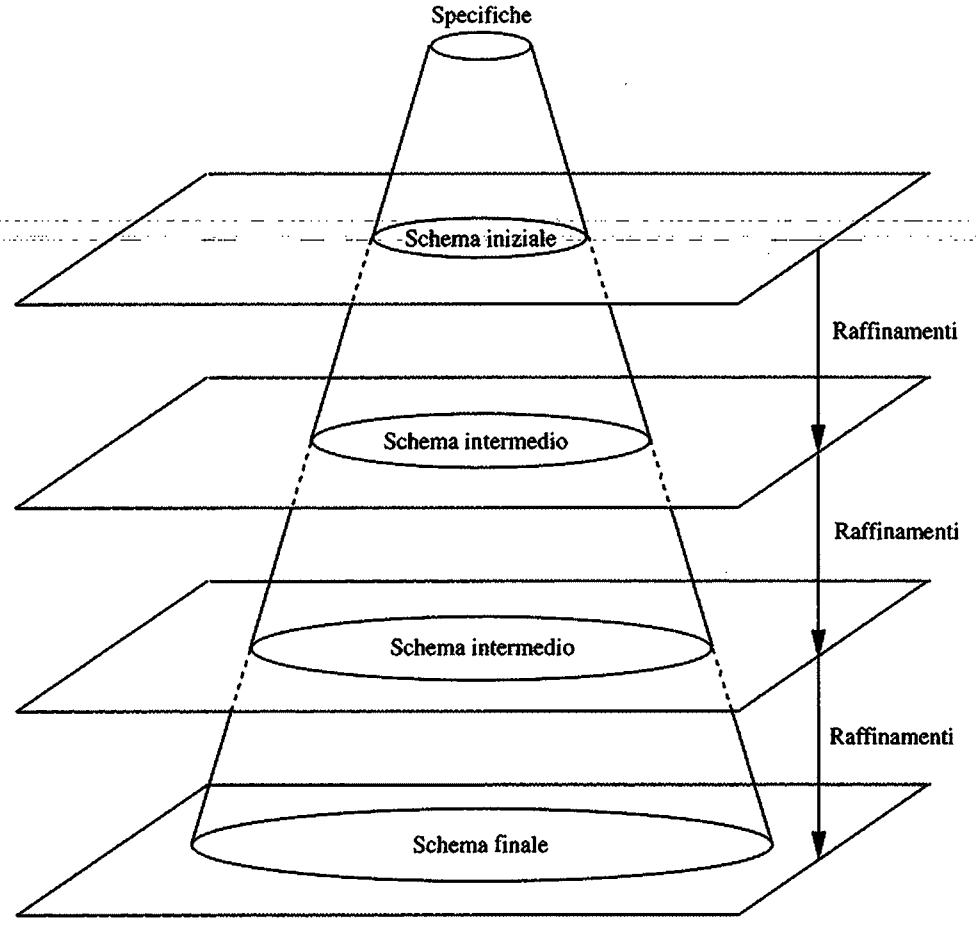


Figura 8.4 La strategia top-down

Primitiva di trasformazione	Concetto iniziale	Risultato
T_1: Da entità a relazione tra entità		
T_2: Da entità a generalizzazione		
T_3: Da relazione a insieme di relazioni		
T_4: Da relazione a entità con relazioni		
T_5: Introduzione di attributi su entità		
T_6: Introduzione di attributi su relazioni		

Figura 8.5 Primitive di trasformazione top-down

rende conto che in una entità astratta CORSO vanno distinti il TIPO DI CORSO (aventi un codice e un titolo) dalla EDIZIONE DI CORSO (avente una data di inizio e una di fine) e che queste entità sono correlate da un relazione che possiamo denominare TIPOLOGIA.

- Trasformazione T_2: si applica quando si comprende che una entità è composta da sotto-entità distinte. Per esempio, nella nostra applicazione, questo accade quando ci rendiamo conto che tra i partecipanti vanno distinte le tipologie DIPENDENTE e PROFESSIONISTA.
- Trasformazione T_3: si applica quando si comprende che una relazione descrive in realtà due relazioni diverse tra le medesime entità. Per esempio, questo può accadere quando ci rendiamo conto che in una relazione DOCENZA tra docenti e corsi, vanno distinti i casi di DOCENZA CORRENTE e DOCENZA PASSATA.
- Trasformazione T_4: si applica quando si comprende che una relazione descrive in realtà un concetto con esistenza autonoma ai fini della applicazione. Per esempio, questo può accadere quando ci rendiamo conto che una relazione CONTRATTO con molte proprietà tra una entità CONSULENTE e una entità AZIENDA è meglio rappresentata da una entità legata alle altre entità da relazioni binarie. Un altro esempio è il caso in cui, in una relazione ESAME tra le entità STUDENTE e CORSO, ci rendiamo conto che vogliamo rappresentare anche il fatto che uno studente può sostenere più volte l'esame per un certo corso. Come spiegato nel paragrafo 7.2.1, per le caratteristiche delle relazioni e delle entità nel modello E-R, bisogna trasformare la relazione ESAME in una entità.
- Trasformazione T_5: si applica per aggiungere proprietà (attributi) a entità. Questo accade, per esempio, quando aggiungiamo alla entità PARTECIPANTE della nostra applicazione i suoi attributi Codice Fiscale, Cognome, Età, Sesso e Città di nascita.
- Trasformazione T_6: si applica per aggiungere proprietà a relazioni, in maniera simile alla trasformazione T_5.

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.

8.3.2 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 8.6, nella quale

vengono rappresentate: la fase di decomposizione delle specifiche, la successiva fase di rappresentazione delle componenti di base e la fase finale di 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.

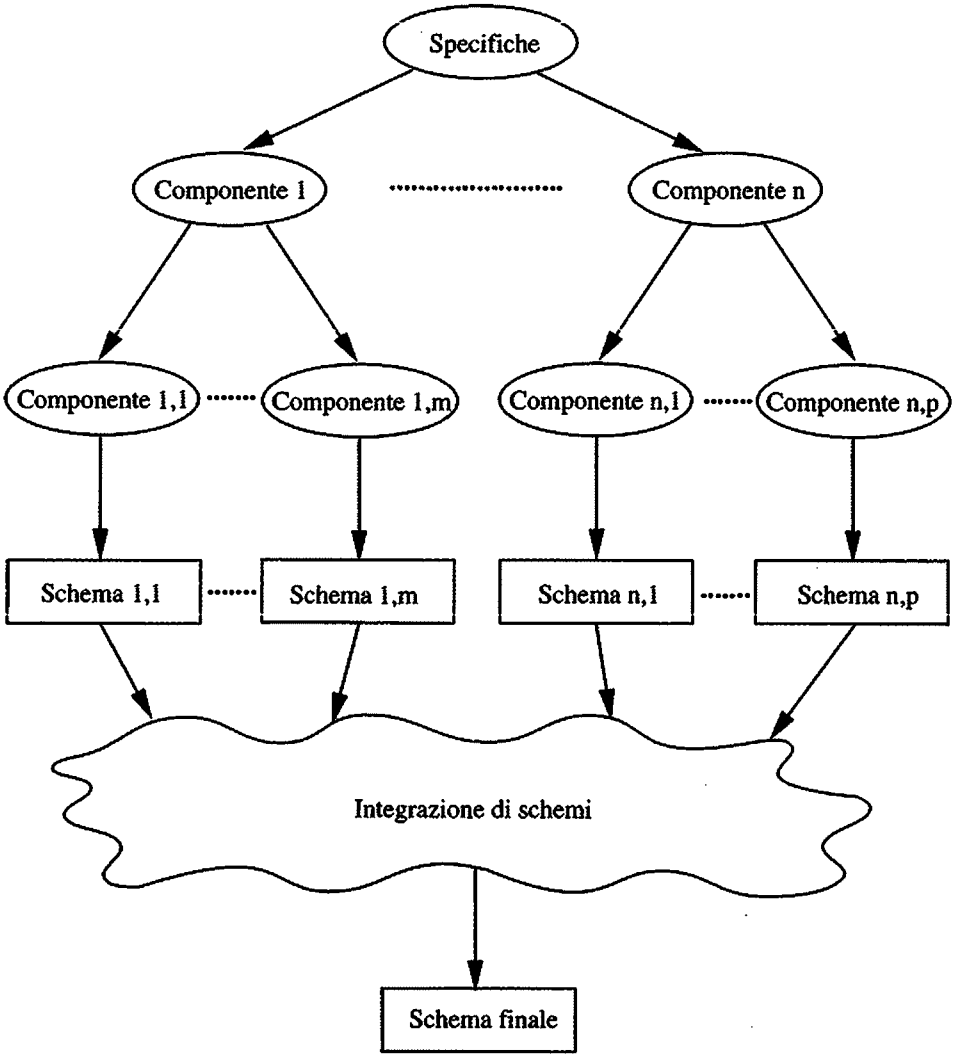


Figura 8.6 La strategia bottom-up

Anche in questo caso, lo schema finale si ottiene attraverso alcune trasformazioni elementari che vengono denominate *primitive di trasformazione bottom-up*. In figura 8.7 vengono riportati alcuni esempi di primitive di trasformazione bottom-up. Come si può vedere, queste primitive introducono nello schema nuovi

concetti non presenti precedentemente e in grado di descrivere aspetti della realtà di interesse che non erano ancora stati rappresentati.

- Trasformazione T_1: si applica quando si individua nelle specifiche una classe di oggetti con proprietà comuni. Per esempio, nell'applicazione relativa alla società di formazione, questo può accadere quando individuiamo l'entità LEZIONE (tenuta per un corso in una certa aula e in un certo orario).
- Trasformazione T_2: si applica quando si individua nelle specifiche un legame logico tra due entità. Per esempio, nella nostra applicazione, questo può accadere quando individuiamo la relazione ABILITAZIONE tra le entità DOCENTE e CORSO.
- Trasformazione T_3: si applica quando si individua nelle specifiche un legame tra diverse entità riconducibile a una generalizzazione. Per esempio, nella nostra applicazione, questo può accadere quando comprendiamo che l'entità DOCENTE è una generalizzazione delle entità INTERNO (dipendente della società di formazione) e COLLABORATORE.
- Trasformazione T_4: si applica quando, a partire da una serie di attributi si individua una entità che può essere vista come aggregazione di tali attributi. Per esempio, questo può accadere nel nostro esempio quando comprendiamo che le proprietà Codice Fiscale, Cognome, Età, Sesso e Città di nascita sono attributi dell'entità PARTECIPANTE.
- Trasformazione T_5: si applica in maniera simile alla trasformazione T_4, quando si individua una relazione che può essere vista come aggregazione di alcuni attributi.

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 concettuali diversi che, nel caso di schemi complessi, presentano quasi sempre grosse difficoltà.

8.3.3 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.

Un esempio di sviluppo inside-out di uno schema concettuale è mostrato in figura 8.8 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. Successivamen-

Primitiva di trasformazione	Concetto iniziale	Risultato
T_1: Generazione di entità		
T_2: Generazione di relazione		
T_3: Generazione di generalizzazione		
T_4: Aggregazione di attributi su entità		
T_5: Aggregazione di attributi su relazione		

Figura 8.7 Primitive di trasformazione bottom-up

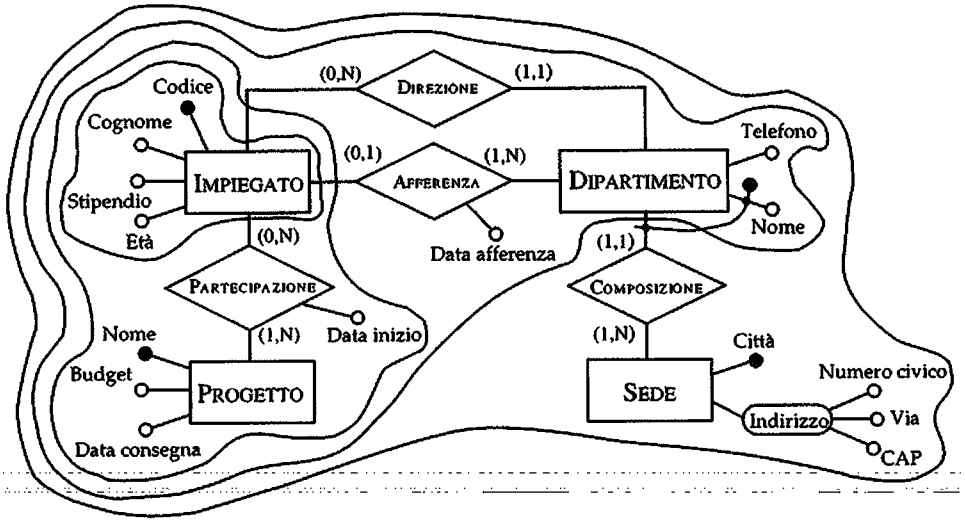


Figura 8.8 Un esempio di uso della strategia inside-out

te, 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.

8.3.4 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 della 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 8.9 viene riportato un possibile schema scheletro per la nostra applicazione relativa alla società di formazione. Da una semplice ispe-

zione delle specifiche strutturate dei requisiti contenute nel paragrafo 8.1, 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 sui 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 il (sotto)schema con concetti non ancora rappresentati (procedendo quindi in maniera bottom-up).

Figura 8.9 Schema scheletro per la società di formazione

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 perchè, 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 spesso delle conoscenze a livelli di dettaglio non omogenei.

8.4 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.

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 una 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 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.

[1] Torneremo su questo punto quando affronteremo la progettazione logica.

Nel prossimo paragrafo vedremo come la verifica delle qualità di uno schema concettuale appena viste, possa essere inglobata in una metodologia di progettazione generale.

8.5 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), finchè 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 7.3, 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, quello dell'analisi della qualità del progetto. Innanzitutto va precisato che questa attività non va relegata alla 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 8.4, non è detto che questa analisi porti necessariamente a delle ristrutturazioni, ma solo a una riorganizzazione dello schema che ne aumenti la leggibilità.

8.6 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 8.9. 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.

Eseguiamo quindi il passo iterativo, 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 8.10. 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.

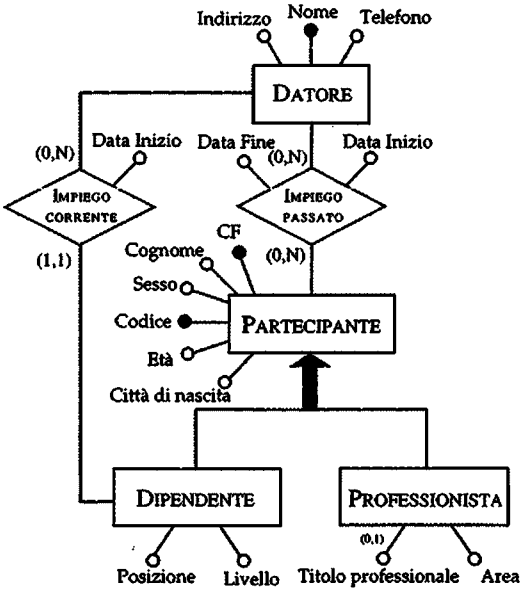


Figura 8.10 Il raffinamento di una porzione dello schema scheletro

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 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 8.11.

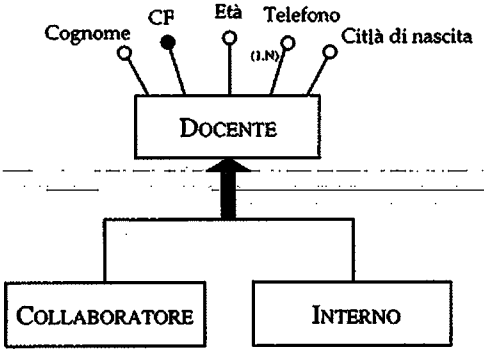


Figura 8.11 Il raffinamento di un'altra porzione dello schema scheletro

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) dalla edizione di un corso che ha una data di inizio, una data di fine e un numero di partecipanti. Rappresentiamo questi due concetti da 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 8.12.

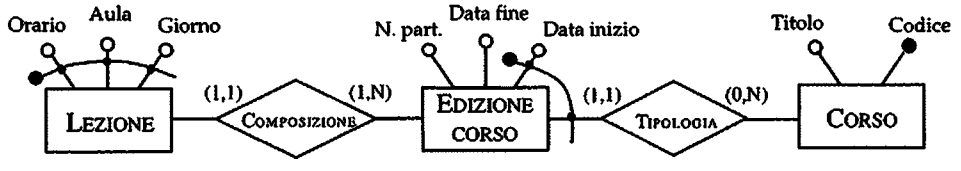


Figura 8.12 Il raffinamento di un'altra porzione dello schema scheletro

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 8.11 e in figura 8.12 rispettivamente: dallo schema scheletro si intuisce che il 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 (perchè un docente viene abilitato a insegnare un generico corso). Va ora integrato lo schema ottenuto con la porzione relativa ai partecipanti, riportata in figura 8.10. 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 di interesse la votazione che rappresentiamo con un attributo. Aggiungendo le varie cardinalità si ottiene lo schema finale riportato in figura 8.13.

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.

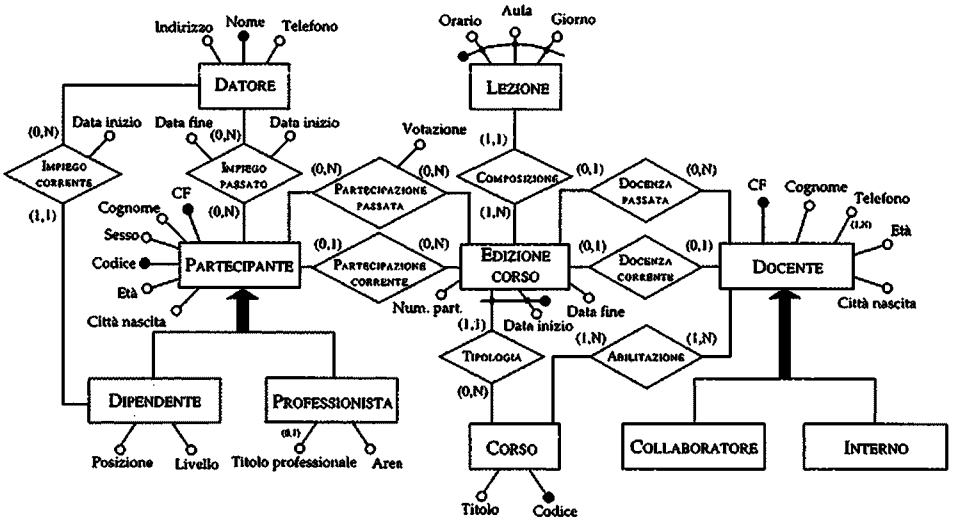


Figura 8.13 Lo schema E-R finale per la società di formazione

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 possono essere eseguite mediante una navigazione sullo sche-

ma. 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 8.10 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, arrivare 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 della 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.

Va infine ricordato che lo schema va corredato con una 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.

8.7 Strumenti CASE per la progettazione di basi di dati

La progettazione di basi di dati è una 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 interfaccie 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 alla 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 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 una 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, ovvero 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 8.14.

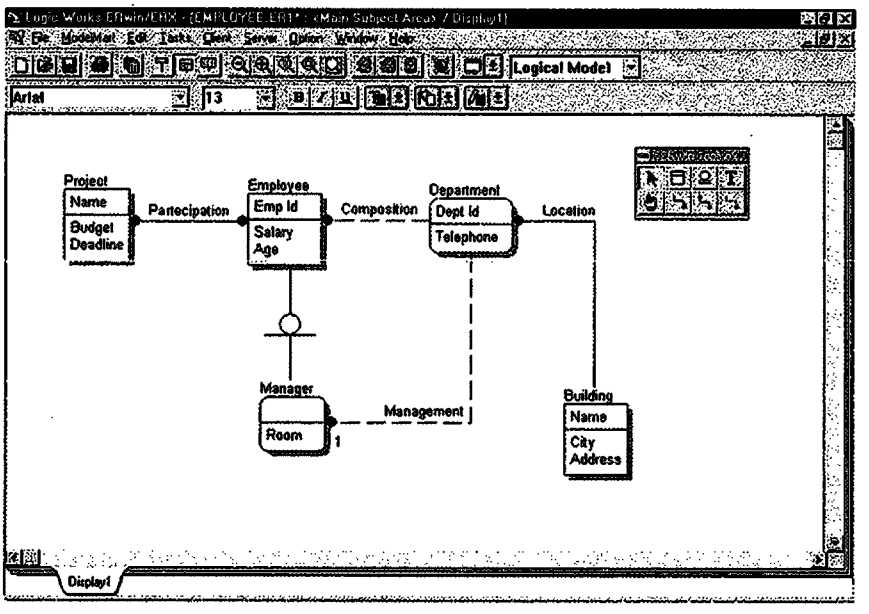


Figura 8.14 Progettazione concettuale fatta con ER-Win

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 8.15.

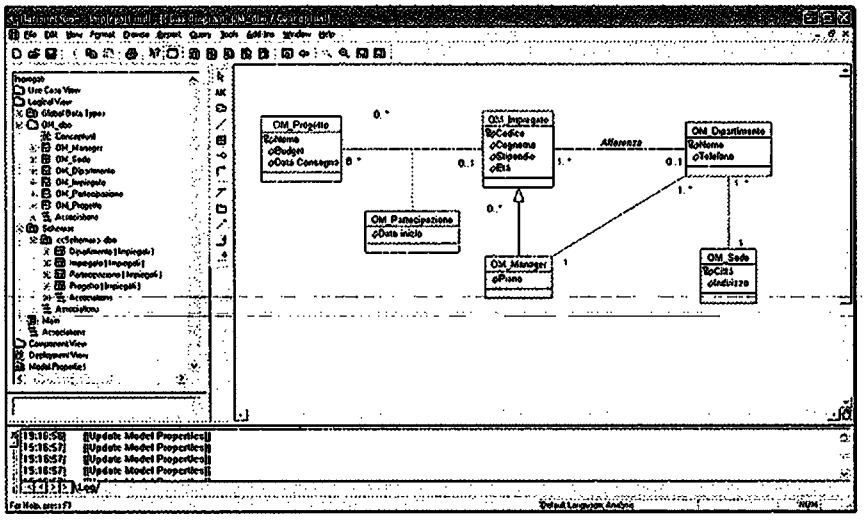


Figura 8.15 Progettazione concettuale fatta con Rational

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 7.4, 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 e altri [8] e nel testo in inglese di Batini, Ceri e Navathe [7] 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 [3, 14]. 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 [7].

Come testi di esercizi sulla progettazione di basi di dati suggeriamo il testo di Cabibbo, Torlone e Batini [11], quello di Francalanci, Schreiber e Tanca [34] e quello di Maio e Rizzi [44].

Esercizi

Esercizio 8.1 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 8.16. 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.

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 8.16 Specifiche per l'esercizio 8.1

Esercizio 8.2 Rappresentare le specifiche dell'esercizio precedente (dopo la fase di riorganizzazione) con uno schema del modello Entità-Relazione.

Esercizio 8.3 Definire uno schema Entità-Relazione che descriva i dati di una applicazione relativa a una catena di officine. Sono di 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.

- 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.

Esercizio 8.4 Definire uno schema E-R che descriva i dati di una applicazione relativa all'anagrafe del comune di Chissadove, con cittadini e famiglie. 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.

Esercizio 8.5 Analizzare le specifiche relative a partite di un campionato riportate in figura 8.17 e costruire un glossario dei termini a esse relativo.

Esercizio 8.6 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 7.37 rappresenta una possibile soluzione di questo esercizio.

Esercizio 8.7 Provare a rappresentare di nuovo le specifiche dell'esercizio 8.5 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 8.6.

Campionato di calcio
Per ogni partita, descrivere il girone e la giornata in cui si è svolta, il numero progressivo nella giornata (es. 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 8.17 Specifiche per l'esercizio 8.5

Esercizio 8.8 Si vuole effettuare una 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 una applicazione su treni e stazioni ferroviarie ed è composta dalle seguenti relazioni:

- STAZIONE(Codice,Nome,Citlà), con il vincolo di integrità referenziale fra l'attributo Città e la relazione CITTÀ;
- CITTÀ(Codice,Nome,Regione);
- TRATTA(Da,A,Dislanza), 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,Dala, OrarioDiParlenza,OrarioDiArrivo) con il vincolo di integrità referenziale tra l'attributo Numero e la relazione ORARIOTRENI;
- FERMATEREALI(NumeroTreno,Dala, Slazione,Amvo,Parlenza) con il vin-

colo di integrità referenziale tra gli attributi NumeroTreno e Stazione e la relazione ORARIOFERMATE.

Segnalare eventuali ridondanze. In particolare, qualora si tratti di relazioni derivate.

Esercizio 8.9 Definire uno schema Entità-Relazione che descriva i dati di una applicazione relativa a un reparto ospedaliero. Sono di 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, etc.), 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.

Esercizio 8.10 Definire uno schema Entità-Relazione che descriva i dati di una 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 di 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.

9

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 una 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 di 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). 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 9.1.

- **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

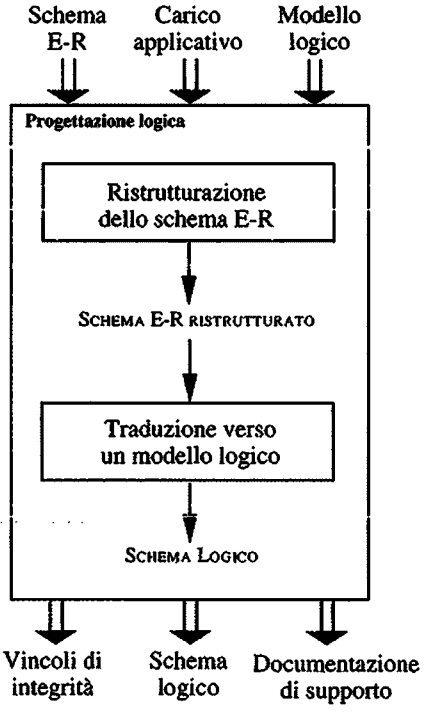


Figura 9.1 Progettazione logica di basi di dati

ristrutturato, che non è 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 prossimo capitolo. Lo schema logico finale, i vincoli di integrità definiti su di esso e la relativa documentazione, costituiscono i prodotti finali della progettazione logica.

Nel resto di questo capitolo presenteremo separatamente le due fasi che compongono la progettazione logica di una base di dati. 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.

9.1 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**. Ovvero:
  - numero di occorrenze di ogni entità e associazione dello schema,
  - dimensioni di ciascun attributo (di entità o associazione)
- **Caratteristiche delle operazioni**. Ovvero:
  - 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 9.2. 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 una analisi delle prestazioni che fa riferimento a un numero ristretto di operazioni può sembrare riduttiva rispetto al reale carico della base di dati, va

[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.

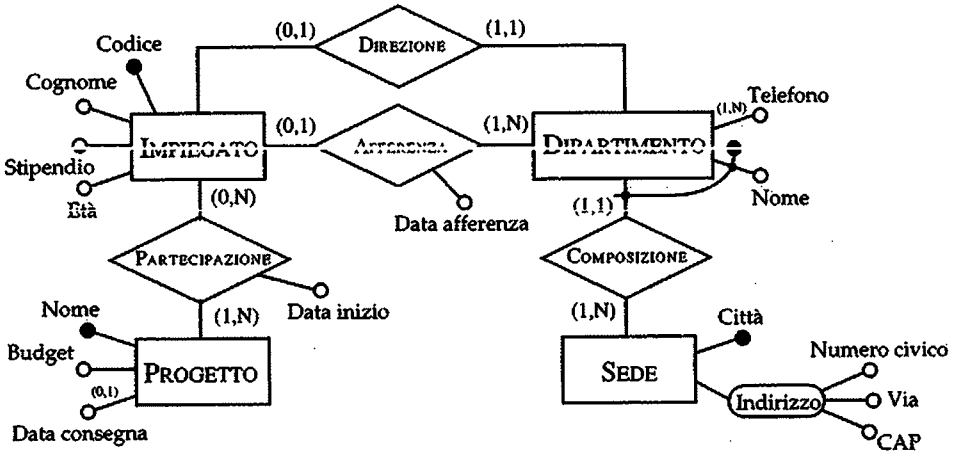


Figura 9.2 Uno schema E-R sul personale di un'azienda

notato che le operazioni sulle basi di dati seguono la cosiddetta regola "ottanta–venti". In base a questa regola, l'ottanta percento 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.

Il volume dei dati e le caratteristiche generali delle operazioni possono essere descritti facendo uso di tabelle come quelle in figura 9.3. 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 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 associazioni. 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 di interesse. Un esempio di schema di operazione viene proposto in figura 9.4 con riferimento all'operazione 2: per ottenere le informazioni di interesse su un impiegato si parte dall'entità IMPIEGATO per accedere, attraverso l'associazione AFFERENZA, al suo dipartimento e, attraverso

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 9.3 Esempio di tavole dei volumi e delle operazioni

l'associazione PARTECIPAZIONE, ai progetti ai quali partecipa. Avendo a dispo-

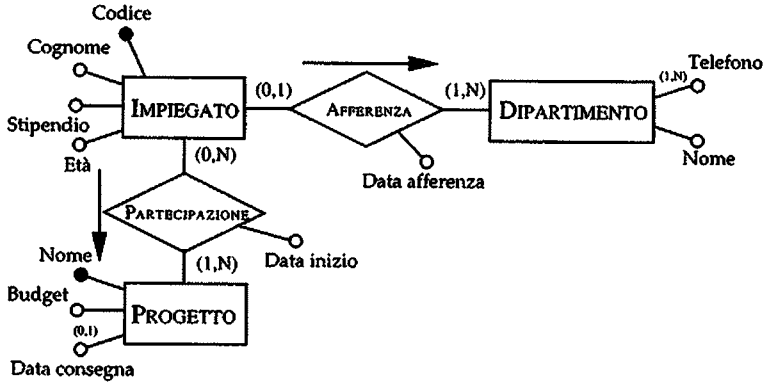


Figura 9.4 Esempio di schema di operazione

sizione 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 relazioni 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 9.5. Nell'ultima colonna di questa tabella viene riportato il tipo di accesso: L per 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

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 9.5 Tavola degli accessi per l'operazione 2

vedremo come questi strumenti di analisi possono essere utilizzati per prendere delle decisioni durante la ristrutturazione di schemi Entità-Relazione.

9.2 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 (vedi figura 9.6).

- **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 primari.** 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.

9.2.1 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 ope-

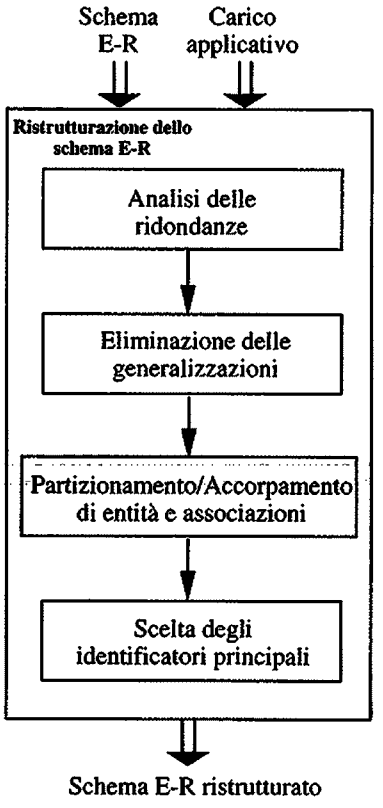


Figura 9.6 Fasi della ristrutturazione di uno schema E-R

razioni) da altri dati. In particolare, in uno schema Entità-Relazione si possono presentare varie forme di ridondanza. I casi più frequenti sono i seguenti.

- Attributi derivabili, occorrenza per occorrenza, da altri attributi della stessa entità (o associazione). Per esempio, il primo schema in figura 9.7 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 9.7, nel quale l'entità ACQUISTO ha un attributo Importo totale che 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 9.7 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.

- Associazioni derivabili dalla composizione di altre associazioni in presenza di cicli. L'ultimo schema in figura 9.7 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.

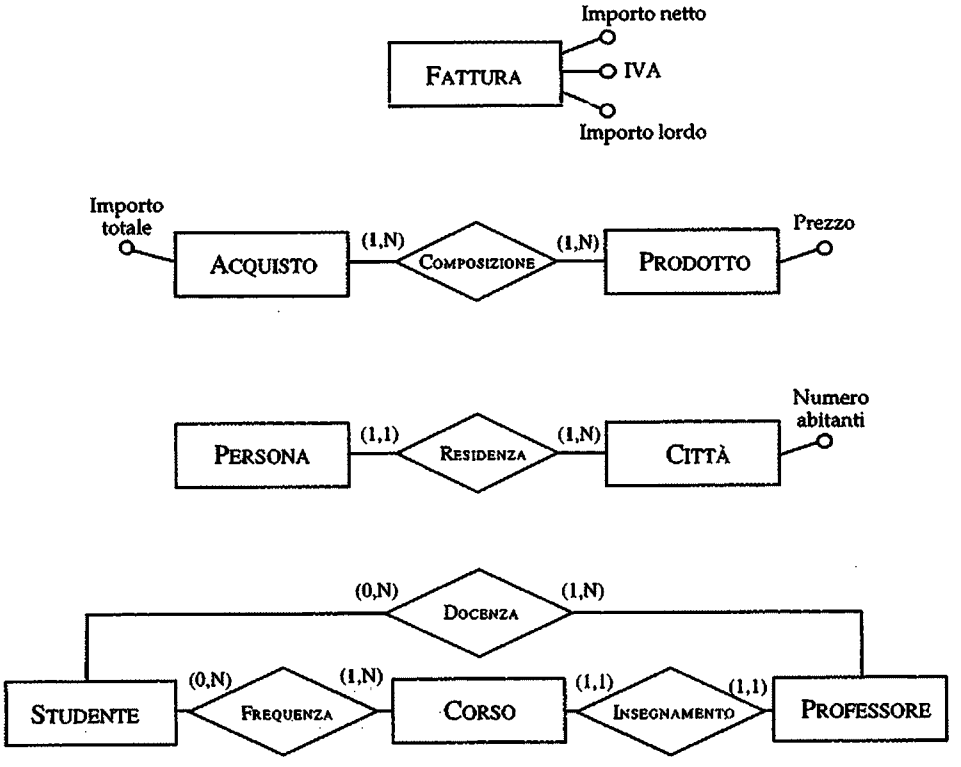


Figura 9.7 Esempi di schemi con ridondanze

La presenza di un dato derivato in uno schema E-R 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 decisione di questo tipo. Consideriamo lo schema su persone e città in figura 9.7 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 9.8.

Tavola dei volumi

Concetto	Tipo	Volume
Città	E	200
Persona	E	1000000
Residenza	R	1000000

Tavola delle operazioni

Operazione	Tipo	Frequenza
Op. 1	I	500 al giorno
Op. 2	I	2 al giorno

Figura 9.8 Tavole dei volumi e delle operazioni per lo schema in figura 9.7 relativo a dati anagrafici

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 KByte di memoria aggiuntiva. Passiamo ora alla stima del costo delle operazioni. Come descritto nella tavola degli accessi in figura 9.9, 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 di 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

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 9.9 Tavole degli accessi per lo schema su dati anagrafici in figura 9.7

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 10000 accessi in lettura al giorno. Contando doppi gli accessi in scrittura abbiamo un totale di 12000 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 Kbyte. 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.

9.2.2 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 9.10 che contiene una generalizzazione e alcune associazioni tra entità.

I metodi per eliminare la generalizzazione di questo schema sono mostrati in figura 9.11 e si ottengono attraverso le seguenti ristrutturazioni.

(1) **Accorpamento delle figlie della generalizzazione nel genitore.** Le entità E_1

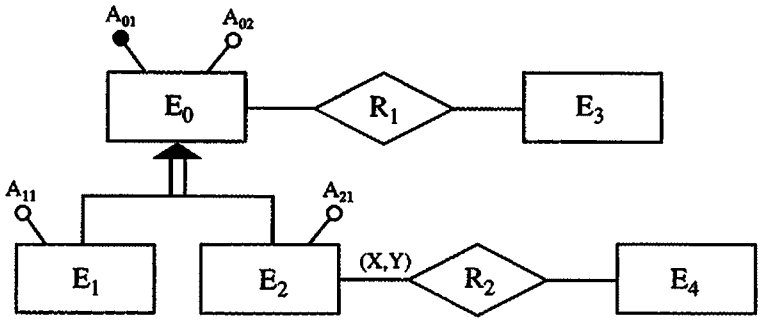


Figura 9.10 Esempio di schema con generalizzazione

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 generalizzazione 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 9.11, 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

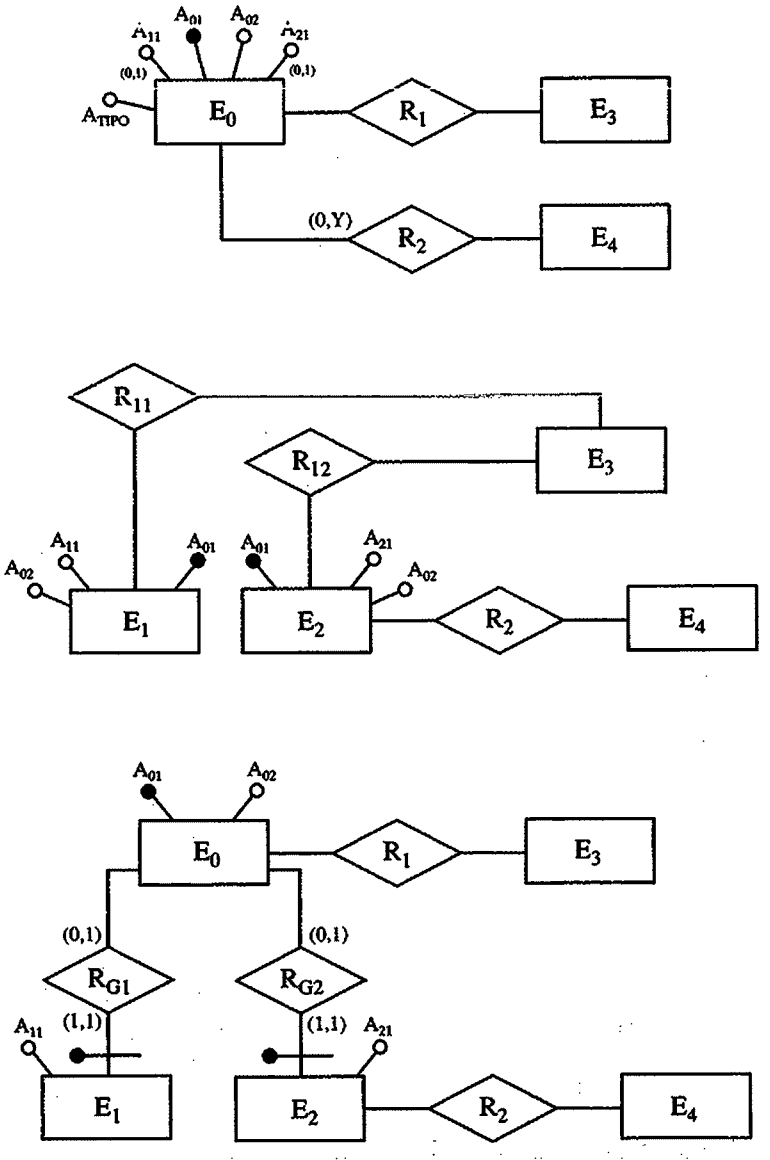


Figura 9.11 Possibili ristrutturazioni dello schema in figura 9.10

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 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 rappresentate. 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 conviene 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 9.12, sempre con riferimento allo schema originale in figura 9.10: 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,

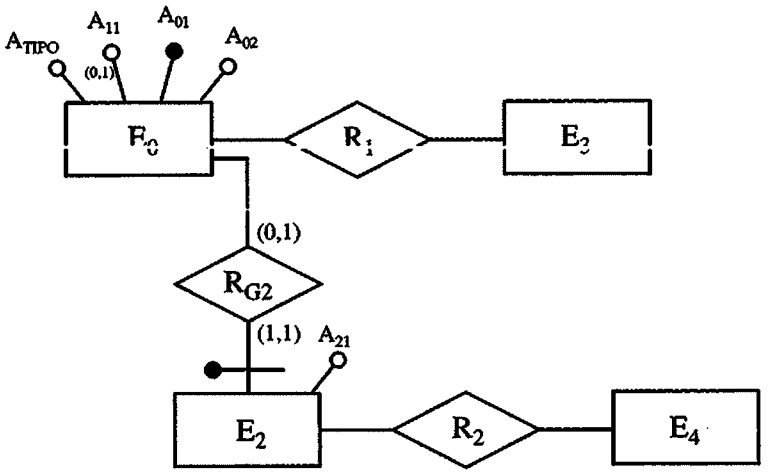


Figura 9.12 Possibile ristrutturazione dello schema in figura 9.10

ottenibili per combinazione delle ristrutturazioni di base, sia a livello della singola generalizzazione, che lungo i vari livelli della gerarchia.

9.2.3 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 ridondanze possono essere usate per prendere decisioni di questo tipo.

Partizionamenti di entità Un esempio di partizionamento di entità viene mostrato in figura 9.13: l'entità IMPIEGATO viene sostituita da due entità, collegate da una 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

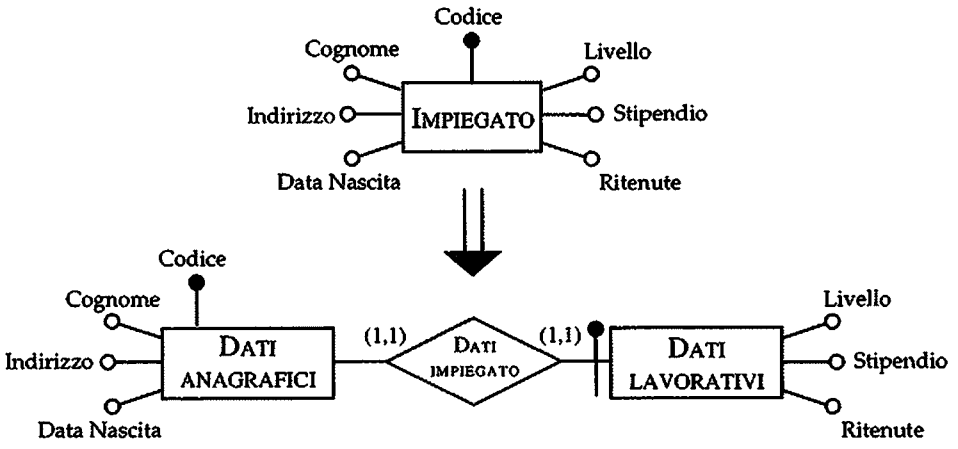


Figura 9.13 Esempio di partizionamento di entità

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.

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 9.14.

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 eccetto l'attributo multivalore, e l'entità TELEFONO, con il solo attributo Numero, legata mediante una 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.

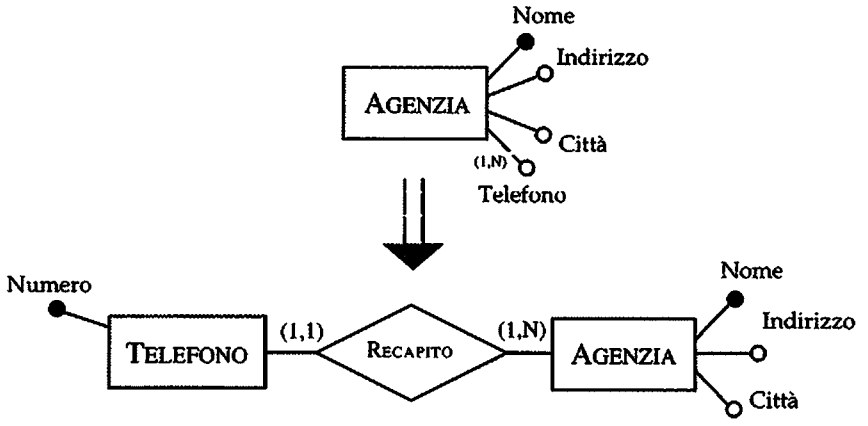


Figura 9.14 Esempio di eliminazione di attributo multivalore

Accorpamento di entità L'accorpamento è l'operazione inversa del partizionamento. Un esempio di accorpamento di entità viene mostrato in figura 9.15 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 ed 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 una 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 prossimo capitolo 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 associazioni. Può convenire cioè, in alcuni casi, decomporre una 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 re-

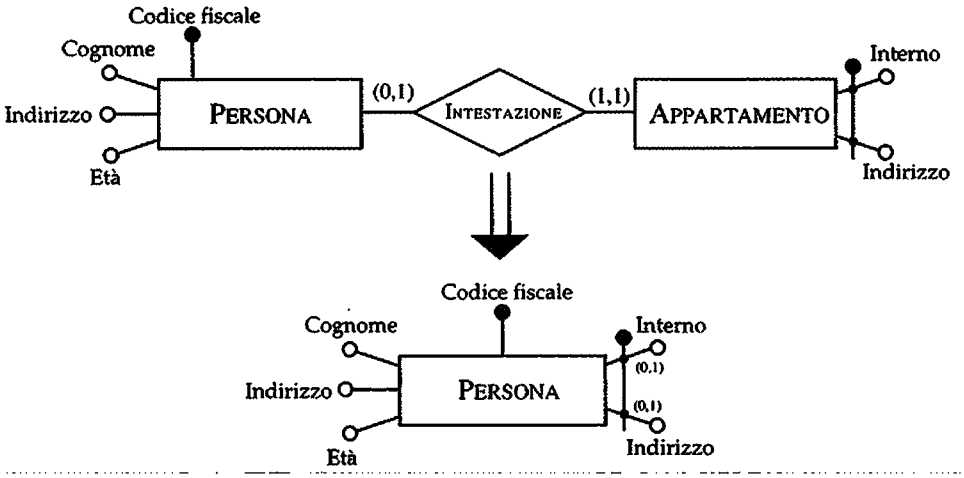


Figura 9.15 Esempio di accorpamento di entità

lative occorrenze vengono sempre accedute contemporaneamente. Un esempio di accorpamento di associazioni viene fornito in figura 9.16 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.

9.2.4 Scelta degli identificatori principali

La scelta degli identificatori principali è essenziale nelle traduzioni verso il modello relazionale perchè, 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 di basi di dati richiedono generalmente di specificare una *chiave primaria* sulla quale vengono costruiti 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 principale.

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 della entità corrispondente, come sottolineato quando abbiamo discusso le chiavi nel modello relazionale.

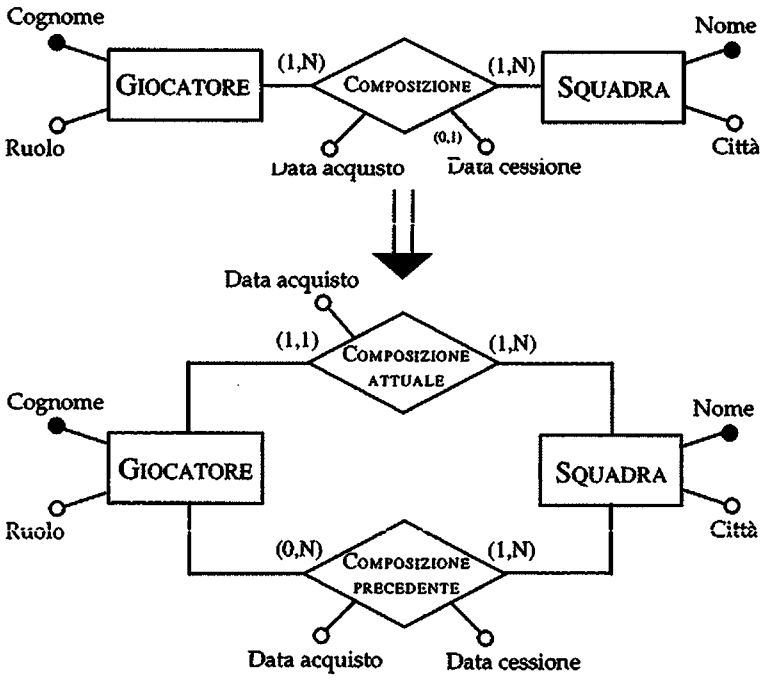


Figura 9.16 Esempio di partizionamento di associazione

- 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 una 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 identifica-

tori non selezionati come primari 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 (primari) definiti automaticamente sugli identificatori principali.

9.3 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 esistono solo identificatori primari. 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.

9.3.1 Entità e associazioni molti a molti

Consideriamo lo schema in figura 9.17. 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.

Figura 9.17 Schema E-R con associazione molti a molti

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.

La ridenominazione è in effetti essenziale in alcuni casi. Per esempio, nel caso di associazioni ricorsive come quella in figura 9.18.

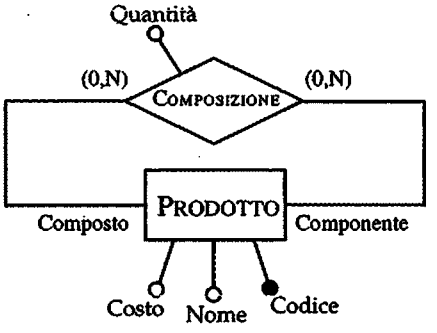


Figura 9.18 Schema E-R con associazione ricorsiva

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 9.19. Questo schema si traduce nelle seguenti tre relazioni:

FORNITORE(PartitaIVA, NomeDitta)
PRODOTTO(Codice, Genere), DIPARTIMENTO(Nome, Telefono)
FORNITURA(Fornitore, Prodotto, Dipartimento, Quantità).

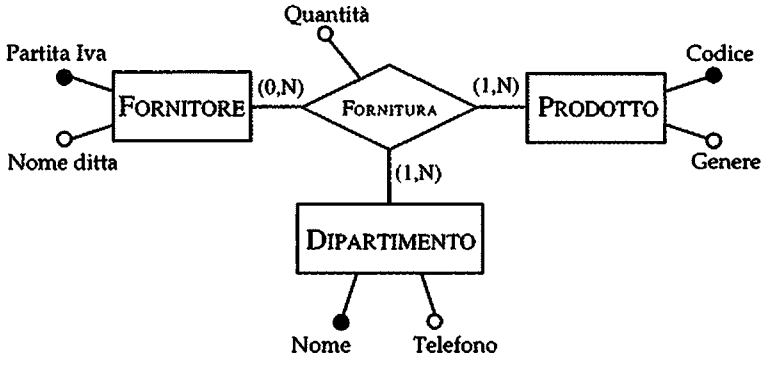


Figura 9.19 Schema E-R con associazione ternaria

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 9.19, c'è un solo fornitore che fornisce un certo prodotto a un dipartimento. Si noti che le cardinalità sono ancora valide, perchè 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.

9.3.2 Associazioni uno a molti

Consideriamo lo schema con associazione uno a molti in figura 9.20.

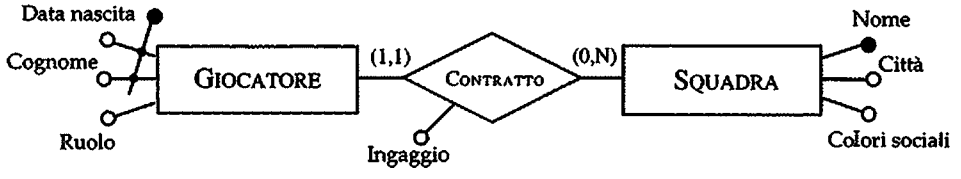


Figura 9.20 Schema E-R con associazione uno a molti

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 biunivoca tra le rispettive occorrenze). È quindi preferibile, per lo schema in figura 9.20, la traduzione che segue, nella quale la relazione GIOCATORE rappresenta sia l'entità relativa che 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 7.2.2, che le associazioni n-arie sono quasi sempre di tipo molti a molti. Comunque, nel caso in cui una delle entità partecipa con cardinalità massima pari a uno, le cose non cambiano molto rispetto a quanto detto per le associazioni binarie. L'entità che partecipa alla associazione con cardinalità massima pari ad 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 9.19 con cardinalità pari a $(1, 1)$ (e quindi, per ogni prodotto, esiste un solo fornitore che lo fornisce e un solo dipartimento al quale viene fornito), allora lo schema si traduce 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.

9.3.3 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 9.21.

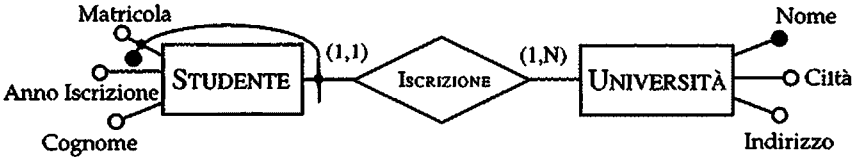


Figura 9.21 Schema E-R con identificatore esterno

Lo schema relazionale corrispondente a questo schema è il seguente:

STUDENTE(Matricola, NomeUniversità, Cognome, AnnoIscrizione)
UNIVERSITÀ(Nome, Città, indirizzo)

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.

9.3.4 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 9.22. Per questo tipo di

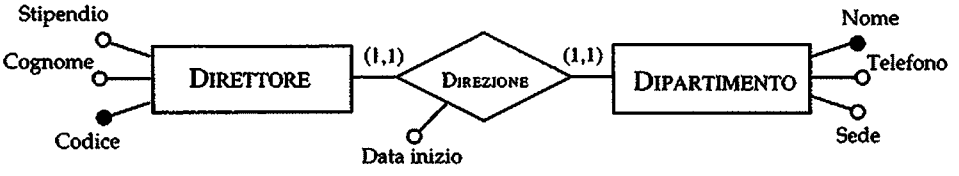


Figura 9.22 Schema E-R con associazione uno a uno

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.

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 perchè 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 9.23. In questo

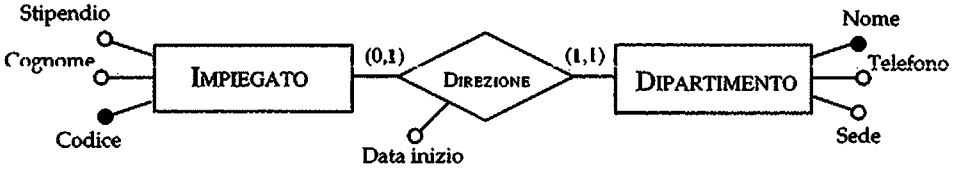


Figura 9.23 Schema E-R con associazione uno a uno

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 9.23, possono esistere dipartimenti senza direttori (e quindi la cardinalità dell'entità DIPARTIMENTO diventa $(0,1)$). In questo caso esiste una 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.

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.

9.3.5 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 9.24.

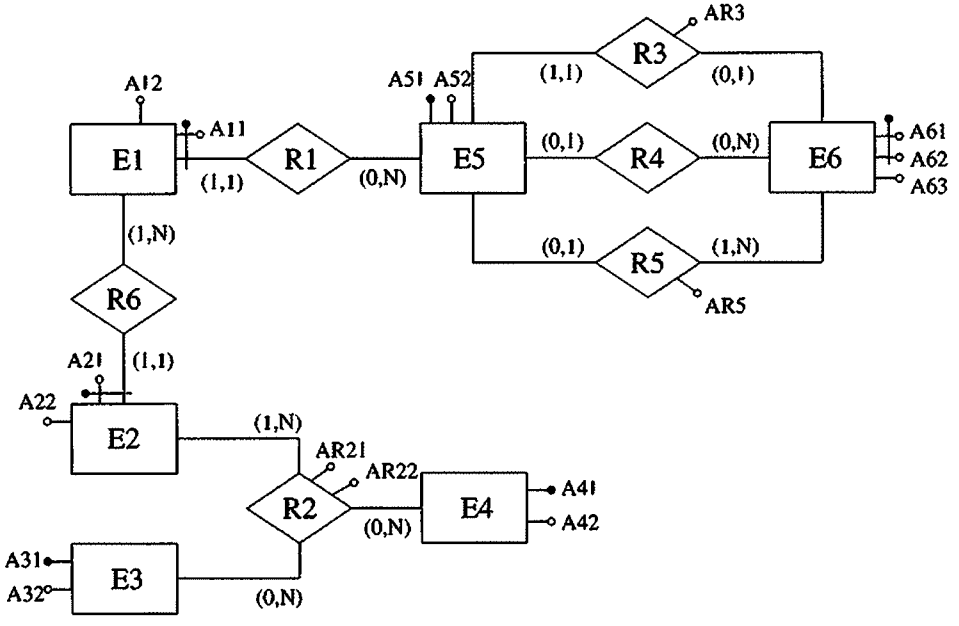


Figura 9.24 Uno schema E-R da tradurre

In una prima fase, traduciamo ciascuna entità con una relazione. La traduzione delle entità dotate di identificatore interno è immediata:

3(A31, A32) E4(A41, A42) E5(A51, A52)
E6 (A61, A62, A63)

le entità con le identificazioni esterne. Otteniamo le seguenti relazioni:

E1(A11, A51, A12) E2(A21, A11, A51, A22)

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 di 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 E2 ed E1 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(A21, A11, A51, A31, A41, AR21, AR22)

Lo schema relazionale ottenuto è il seguente:

E1(A11, A51, A12) E2(A21, A11, A51, A22)
E3(A31, A32) E4(A41, A42),
E5(A51, A52, A61R3, A62R3, AR3, A61R4, A62R4, A61R5, A62R5, AR5)
E6(A61, A62, A63) R2(A21, A11, A51, A31, A41, 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 9.2.4.

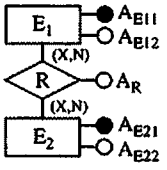
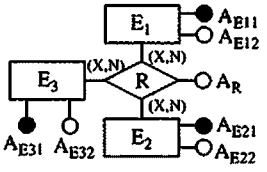
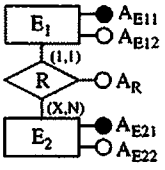
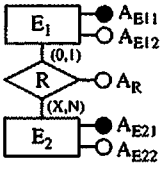
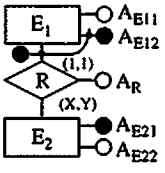
Tipologia	Concetto iniziale	Risultati possibili
Associazione binaria molti a molti		$E_1(\underline{A_{E11}},A_{E12})$ $E_2(\underline{A_{E21}},A_{E22})$ $R(\underline{A_{E11},A_{E21}},A_R)$
Associazione ternaria molti a molti		$E_1(\underline{A_{E11}},A_{E12})$ $E_2(\underline{A_{E21}},A_{E22})$ $E_3(\underline{A_{E31}},A_{E32})$ $R(\underline{A_{E11},A_{E21},A_{E31}},A_R)$
Associazione uno a molti con partecipazione obbligatoria		$E_1(\underline{A_{E11}},A_{E12},A_{E21},A_R)$ $E_2(\underline{A_{E21}},A_{E22})$
Associazione uno a molti con partecipazione opzionale		$E_1(\underline{A_{E11}},A_{E12})$ $E_2(\underline{A_{E21}},A_{E22})$ $R(\underline{A_{E11}},A_{E21},A_R)$ Oppure: $E_1(\underline{A_{E11}},A_{E12},A_{E21}^*,A_R^*)$ $E_2(\underline{A_{E21}},A_{E22})$
Associazione con identificatore esterno		$E_1(\underline{A_{E12},A_{E21}},A_{E11},A_R)$ $E_2(\underline{A_{E21}},A_{E22})$

Figura 9.25 Traduzioni dal modello E-R al relazionale

9.3.6 Tabelle riassuntive

Le traduzioni viste vengono riassunte nelle tabelle in figura 9.25 e 9.26.

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 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 9.26 Traduzioni dal modello E-R al relazionale

9.3.7 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 relazionale, 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.

A tale riguardo, è possibile adottare un semplice formalismo grafico che permette di rappresentare sia le relazioni con i relativi attributi, che i vincoli di integrità referenziale esistenti tra le varie relazioni. Un esempio di questo tipo di rappresentazione, di facile comprensione, viene dato in figura 9.27, con riferimento alla traduzione dello schema in figura 9.17. 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.

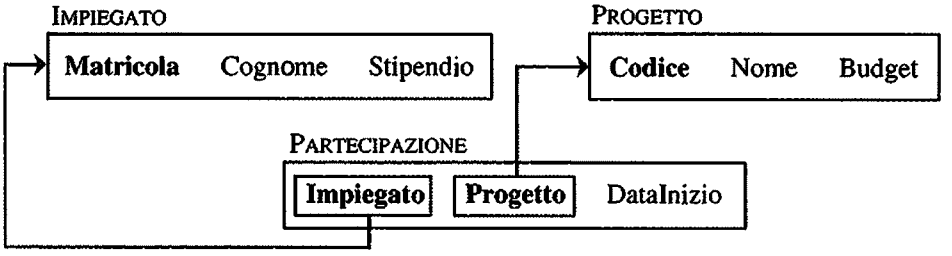


Figura 9.27 Rappresentazione grafica della traduzione nel modello relazionale dello schema in figura 9.17

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. Quindi, nel caso dell'esempio, le informazioni sui progetti ai quali gli impiegati partecipano, attraverso il join tra la IMPIEGATO, PARTECIPAZIONE e PROGETTO.

Un altro esempio di questi tipi di rappresentazione viene fornito in figura 9.28, con riferimento alla traduzione dello schema in figura 9.20.

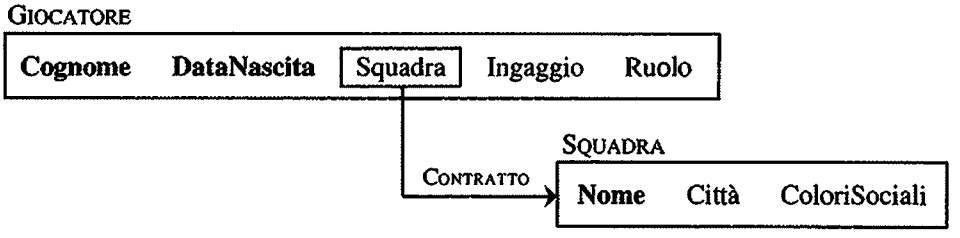


Figura 9.28 Rappresentazione grafica di una traduzione dello schema in figura 9.20

È interessante osservare come, con questo tipo di rappresentazione, è 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 9.29 viene riportata la rappresentazione dello schema relazionale ottenuto nel paragrafo 9.3.5. I legami logici tra le varie relazioni possono essere ora facilmente identificati.

Nell'appendice A verrà mostrato come una variante di questo formalismo grafico viene adottato da un sistema di gestione di basi di dati reale (Access), sia

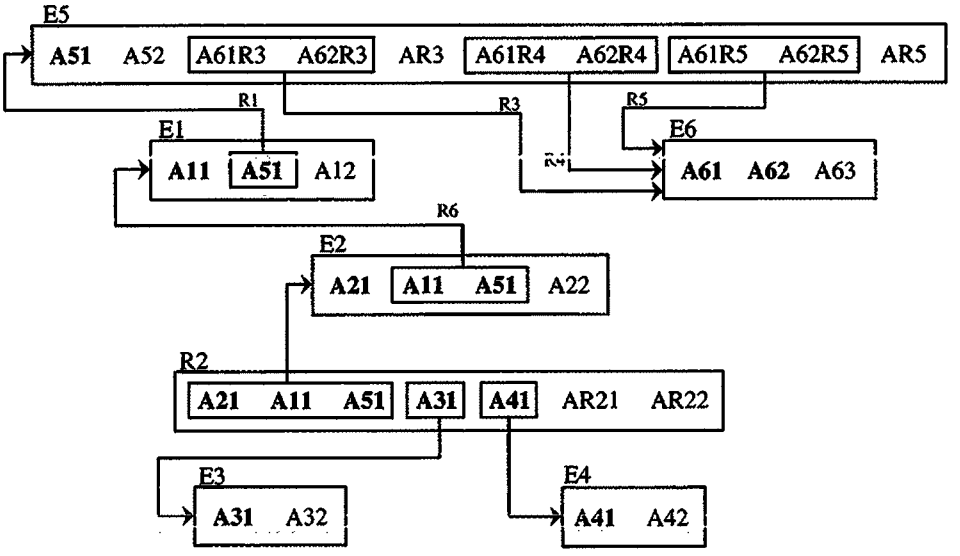


Figura 9.29 Rappresentazione grafica dello schema ottenuto nel paragrafo 9.3.5

per rappresentare schemi di relazione che per esprimere graficamente operazioni di join. Inoltre, nel prossimo capitolo 9.5 verrà mostrato che, come avviene per gli schemi concettuali (vedi paragrafo 7.4), alcuni strumenti fanno uso di un formalismo basato su UML anche per la documentazione di schemi logici di basi di dati.

9.4 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, in figura 9.30. Le varie ristrutturazioni che discuteremo sono riportate nello schema finale in figura 9.33.
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.

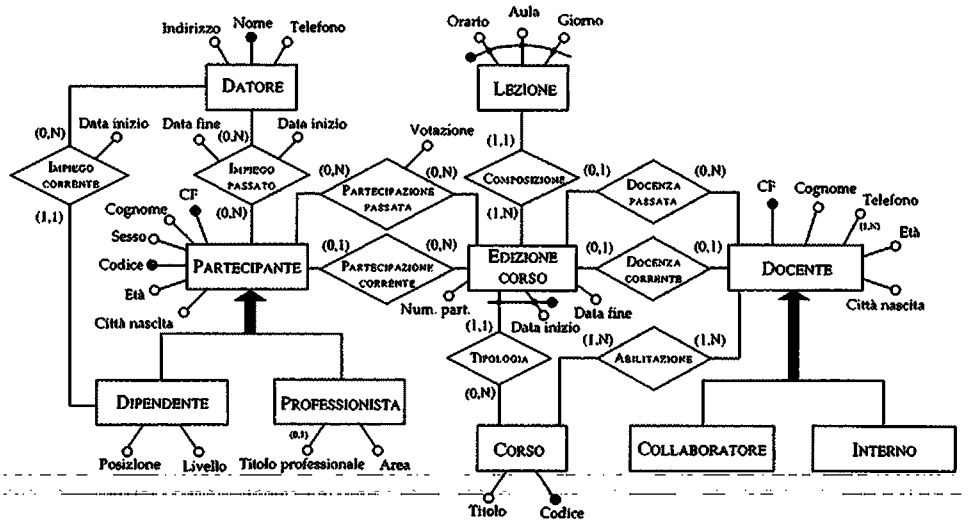


Figura 9.30 Lo schema E-R di una società di formazione

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.

9.4.1 Fase di ristrutturazione

Supponiamo che i dati di carico siano quelli riportati in figura 9.31. 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 $4 \times 1.000 = 4000$ bytes, avendo assunto che sono necessari 4 byte per ogni occorrenza di EDIZIONE CORSO 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 9.32.

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

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	10000
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 9.31 Tavole dei volumi e delle operazioni per lo schema in figura 9.30

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 9.32 Tavole degli accessi per lo schema in figura 9.30

$19 \times 10 = 190$ 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 che 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 operazioni che li riguardano, ovvero 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 dagli altri. Risulta quindi 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 9.33.

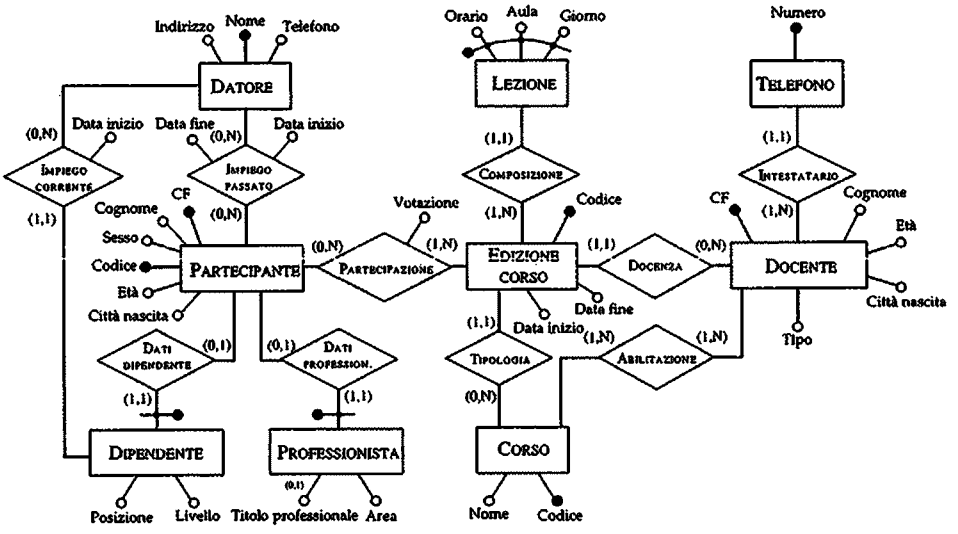


Figura 9.33 Lo schema E-R di figura 9.30 dopo la fase di ristrutturazione

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 associazioni 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 una associazione a un'altra quando una 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 due Kbyte. Decidiamo quindi di accorpare le due coppie di relazioni come descritto in figura 9.33. 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 rappresentare nella maniera migliore la realtà di 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 9.33.

9.4.2 Traduzione verso il relazionale

Seguendo la strategia di traduzione descritta in questo capitolo, lo schema E-R in figura 9.33 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 9.3.7.

9.5 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 una 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 9.2.2. 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 9.34. L'esempio fa riferimento allo schema concettuale riportato nella figura 8.14 del capitolo precedente. Lo schema risultato viene rappresentato in forma grafica che rappresenta le tabelle relazionali insieme alle associazioni dello schema di partenza. Si osservi come l'associazione

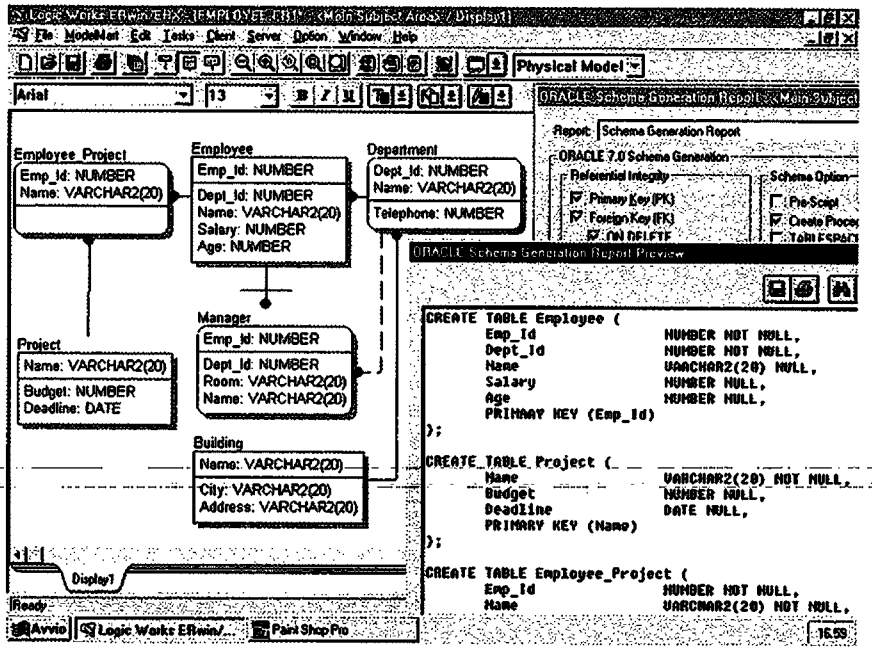


Figura 9.34 Progettazione logica fatta con uno strumento CASE

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 9.35 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 8.15, nel paragrafo 8.7 del capitolo precedente. 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.

Certi sistemi 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.

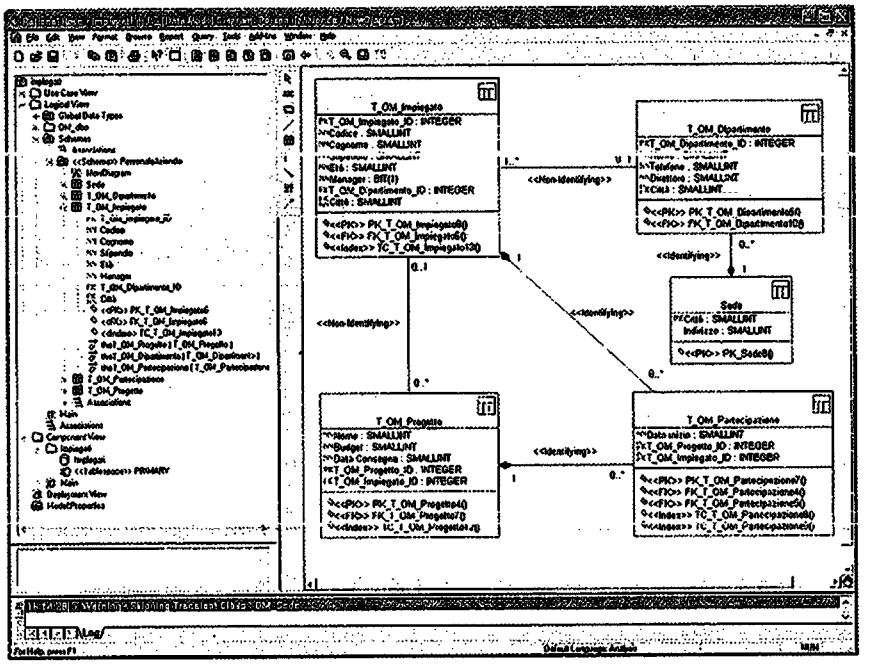


Figura 9.35 Progettazione logica con UML

9.6 Progettazione fisica di una base di dati

Nell'ambito del processo di progettazione di una base di dati, 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 (per esempio in forma di istruzioni `create table` in SQL) e soprattutto delle strutture fisiche di accesso 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, 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 delle aree di transito per scambio di informazioni tra memoria principale e memoria secondaria. Alcuni sistemi mettono a disposizione varie decine di parametri, i cui valori possono risultare significativi ai fini delle prestazioni delle applicazioni. Peraltro, esistono praticamente 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, oltre che dallo specifico sistema di gestione utilizzato, anche da aspetti

tecnologici che non sono stati approfonditi in questo testo. Risulta quindi difficile fornire qui una panoramica completa. 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. In questo contesto, a parte l'attività materiale, ma semplice, di specifica degli schemi delle relazioni nel particolare DDL del sistema utilizzato, con i tipi dei dati ammessi per i vari attributi e dimensioni dei campi, la progettazione fisica può essere ricondotta a una attività di individuazione degli *indici*. Senza entrare nei dettagli tecnici che non sono rilevanti a questo livello, possiamo dire che gli indici sono speciali strutture di accesso a relazioni che vengono definiti su attributi delle relazioni stesse e garantiscono l'accesso diretto, e quindi efficiente, ai dati quando l'interrogazione si basa su tali attributi.

Per orientarci nella scelta degli indici, è opportuno citare che 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, che permette un accesso diretto.

Consideriamo ad esempio una base di dati su due relazioni: la relazione IMPIEGATI(Matricola, Cognome, Nome, Dipartimento) (nella quale l'attributo Dipartimento indica il codice del dipartimento di afferenza) e la relazione DIPARTIMENTI(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 IMPIEGATI. Se sulla relazione è presente un indice su tale attributo 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 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 sulla chiave Codice della relazione DIPARTIMENTI, può essere effettuato in modo molto efficiente tramite il metodo di *nested loop*: si scandisce sequenzialmente la relazione IMPIEGATI (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 DIPARTIMENTI sulla base dell'indice. Se l'indice non è stato definito, l'accesso alla relazione DIPARTIMENTI risulta inefficiente, e tutto il join risulta molto più costoso (sono possibili leggeri miglioramenti con altri algoritmi, ma senza raggiungere le prestazioni che si hanno con l'indice).

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). Pertanto, possiamo dire che è ragionevole definire, su ciascuna relazione, un indice in corrispondenza della relativa chiave; la maggior parte dei DBMS costruisce questo indice automaticamente. 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.

Definiti in questo modo gli indici, 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

Anche la progettazione logica dei dati è affrontata in dettaglio nei libri in inglese di Batini, Ceri e Navathe [7] e Teorey [61]. Il problema della traduzione di uno schema Entità-Relazione nel modello relazionale è discusso nell'articolo originale di Chen [21] e in un articolo di Teorey, Yang e Fry [62]. Un buon testo di introduzione al progetto delle strutture fisiche e al loro dimensionamento è *Database Tuning: a Principled Approach* di Shasha [56].

Esercizi di varia complessità sulla progettazione logica di basi di dati si possono trovare sui testi di Cabibbo, Torlone e Batini [11] e di Francalanci, Schreiber e Tanca [34].

Un buon testo di introduzione alla progettazione fisica di una base di dati e al dimensionamento dei vari parametri è il libro di Shasha [56].

Esercizi

Esercizio 9.1 Si consideri lo schema Entità-Relazione ottenuto come soluzione dell'esercizio 8.4. 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.

Esercizio 9.2 Tradurre lo schema Entità-Relazione che abbiamo più volte incontrato sul personale di un'azienda (riportato per comodità in figura 9.36) in uno schema del modello relazionale.

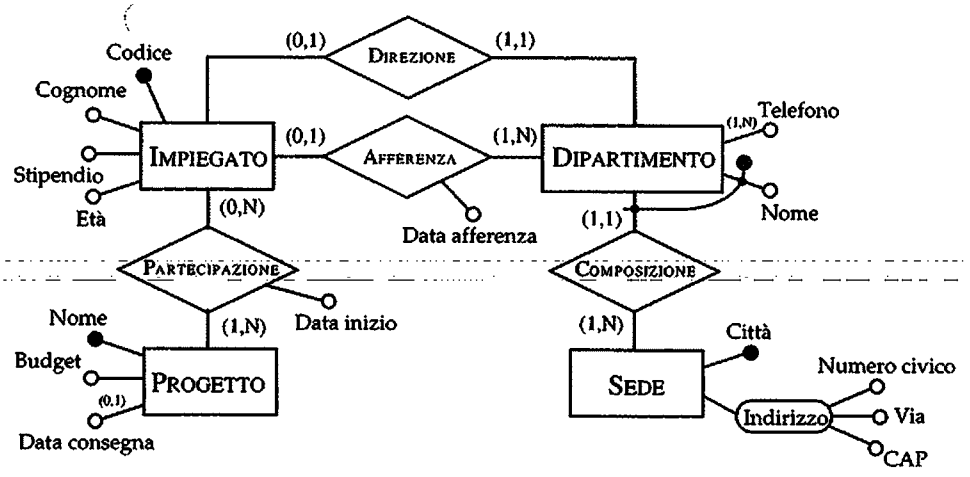


Figura 9.36 Uno schema E-R sul personale di un'azienda

Esercizio 9.3 Tradurre lo schema Entità-Relazione ottenuto nell'esercizio 8.6 in uno schema del modello relazionale.

Esercizio 9.4 Definire uno schema logico relazionale corrispondente allo schema E-R ottenuto nell'esercizio 8.10. 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 cento 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).

Esercizio 9.5 Tradurre lo schema Entità-Relazione di figura 9.37 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).

Tavola dei volumi

Concetto	Tipo	Volume
Cliente	E	15000
Conto	E	20000
Transazione	E	600000
Persona Fisica	E	14000
Società	E	1000
Titolarità	R	30000
Operazione	R	800000

Tavola delle operazioni

Operazione	Tipo	Frequenza
Op. 1	100/giorno	I
Op. 2	2000/giorno	I
Op. 3	1000/giorno	I
Op. 4	2000/giorno	I
Op. 5	1000/giorno	I
Op. 6	200/giorno	I
Op. 7	1500/giorno	B
Op. 8	1/mese	B
Op. 9	75/giorno	B
Op. 10	20/giorno	I

Figura 9.40 Tavole dei volumi e delle operazioni per lo schema in figura 9.39

10

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, ad 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.

10.1 Ridondanze e anomalie

Introduciamo i primi concetti attraverso un esempio. Consideriamo la relazione di figura 10.1. 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	20000	Marte	2000	tecnico
Verdi	35000	Giove	15000	progettista
Verdi	35000	Venere	15000	progettista
Neri	55000	Venere	15000	direttore
Neri	55000	Giove	15000	consulente
Neri	55000	Marte	2000	consulente
Mori	48000	Marte	2000	direttore
Mori	48000	Venere	15000	progettista
Bianchi	48000	Venere	15000	progettista
Bianchi	48000	Giove	15000	direttore

Figura 10.1 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, ad 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, ad 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).

10.2 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 suggerisce il nome, descrive legami di tipo funzionale tra gli attributi di una relazione.

Consideriamo ancora la relazione in figura 10.1. 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 10.1 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 alla 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 perchè 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 10.6, 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, ad 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 $R(X)$.

10.3 Forma normale di Boyce e Codd

10.3.1 Definizione di forma normale di Boyce e Codd

In questo paragrafo rivisitiamo i concetti illustrati nel paragrafo 10.1, 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 10.1, 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:

$$\text{Impiegato} \rightarrow \text{Stipendio}$$

$$\text{Progetto} \rightarrow \text{Bilancio}$$

sono causa di anomalie, mentre la dipendenza

$$\text{Impiegato Progetto} \rightarrow \text{Funzione}$$

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 della applicazione di 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, perchè i concetti indipendenti sono separati, uno per relazione.

10.3.2 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 10.1 sostituiamo le tre relazioni in figura 10.2, 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.

Impiegato	Stipendio
Rossi	20000
Verdi	35000
Neri	55000
Mori	48000
Bianchi	48000

Progetto	Bilancio
Marte	2000
Giove	15000
Venere	15000

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 10.2 Decomposizione della relazione in figura 10.1

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 10.6.

10.4 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.

10.4.1 Decomposizione senza perdita

Per discutere la prima proprietà, esaminiamo la relazione in figura 10.3.

Impiegato	Progetto	Sede
Rossi	Marte	Roma
Verdi	Giove	Milano
Verdi	Venere	Milano
Neri	Saturno	Milano
Neri	Venere	Milano

Figura 10.3 Relazione per la discussione sulla decomposizione senza perdita

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.

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;
- t'altra sugli attributi Progetto e Sede, in corrispondenza alla dipendenza funzionale Progetto → Sede.

L'istanza in figura 10.3 verrebbe decomposta, per mezzo di proiezioni sugli attributi coinvolti, nelle due relazioni in figura 10.4.

Impiegato	Sede
Rossi	Roma
Verdi	Milano
Neri	Milano

Progetto	Sede
Marte	Roma
Giove	Milano
Saturno	Milano
Venere	Milano

Figura 10.4 Relazioni ottenute per proiezione dalla relazione in figura 10.3

Esaminiamo in dettagtio 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 t'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: ad 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 10.4 produce la relazione in figura 10.5, che è diversa dalla relazione in figura 10.3.

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 10.5 Il risultato del join delle relazioni in figura 10.4

In effetti, la relazione in figura 10.5 contiene tutte le tuple della relazione originaria (in figura 10.3) 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 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 così 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 è 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. Ad esempio, la relazione in figura 10.5 (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.

10.4.2 Conservazione delle dipendenze

Per introdurre la seconda proprietà possiamo esaminare di nuovo la relazione in figura 10.3. 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 10.3 verrebbe così decomposta nelle relazioni in figura 10.6.

Il join delle due relazioni in figura 10.6 produce effettivamente la relazione in figura 10.3, per cui possiamo dire che la relazione in figura 10.3 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 10.6 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 pro-

Impiegato	Sede
Rossi	Roma
Verdi	Milano
Neri	Milano

Impiegato	Progetto
Rossi	Marte
Verdi	Giove
Verdi	Venere
Neri	Saturno
Neri	Venere

Figura 10.6 Un'altra decomposizione per la relazione in figura 10.3

getto Marte. Sulla relazione originaria, cioè quella in figura 10.3, 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.

10.4.3 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 (ri-

spettivamente, 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 10.3.2, con lo scopo di sostituire tre relazioni normalizzate a una non normalizzata, presenta tutte e tre le qualità.

10.5 Terza forma normale

10.5.1 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 10.7.

Dirigente	Progetto	Sede
Rossi	Marte	Roma
Verdi	Giove	Milano
Verdi	Marte	Milano
Neri	Saturno	Milano
Neri	Venere	Milano

Figura 10.7 Relazione per la discussione di una decomposizione problematica

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 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."

10.5.2 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 10.6.

10.5.3 Decomposizione in terza forma normale

Mostreremo nel paragrafo 10.6 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 10.8, per la quale vale la sola dipendenza funzionale Impiegato → Stipendio.

Impiegato	Progetto	Stipendio
Rossi	Marte	30000
Verdi	Giove	30000
Verdi	Venere	30000
Neri	Saturno	40000
Neri	Venere	40000

Figura 10.8 Relazione per la discussione sulla decomposizione in terza forma normale

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 10.9; 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.

Impiegato	Progetto
Rossi	Marte
Verdi	Giove
Verdi	Venere
Neri	Saturno
Neri	Venere

Impiegato	Stipendio
Rossi	30000
Verdi	30000
Neri	40000

Figura 10.9 Una decomposizione in terza forma normale

Per concludere torniamo all'esempio di figura 10.1 e osserviamo che la relazione non soddisfa neanche la terza forma normale. Procedendo come abbiamo suggerito otteniamo ancora la decomposizione di figura 10.2 che, incidentalmente, è anche in forma normale di Boyce e Codd. Questo è in effetti un risultato di validità generale: una decomposizione intesa 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.

10.5.4 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 [5], 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 10.10. Essa soddisfa le dipendenze Impiegato → Categoria e Categoria → Stipendio e quindi viola la terza forma normale, perchè Categoria non è chiave. La seconda forma normale tollera la

Impiegato	Categoria	Stipendio
Neri	3	30000
Verdi	3	30000
Rossi	4	50000
Mori	4	50000
Bianchi	5	72000

Figura 10.10 Una relazione con varie dipendenze funzionali

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 10.1 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 10.10 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.
- 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 10.10 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 la 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.

10.5.5 Normalizzazione e scelta degli attributi

Con riferimento alla relazione in figura 10.7, 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 (vedi la relazione in figura 10.11). 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 10.11 Modifica della relazione in figura 10.7

Per questo schema, esiste una buona decomposizione, come mostrato dall'istanza in figura 10.12. 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;

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 10.12 Una buona decomposizione della relazione in figura 10.11

- 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 una analisi non sufficientemente accurata dell'applicazione.

10.6 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.

10.6.1 Implicazione di dipendenze funzionali

Come abbiamo visto per mezzo di alcuni esempi nel paragrafo 10.3.2, 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 10.10, 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 tupie hanno lo stesso valore su Impiegato, facciamo vedere che hanno stesso valore su Stipendio; infatti, se hanno 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 ed 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 esiste una dipendenza $Y \rightarrow A \in F$ con $Y \subseteq X_i$ e $X_{i+1} = X_iA$; 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 $X_{i+1} = X_iA$.

- $X_F^+ \subseteq X_P$. 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 ad 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 $Z \not\subseteq X_P$. 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 una 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 una applicazione dell'algoritmo, consideriamo la relazione in figura 10.1, 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.

10.6.2 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 ed 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 ed 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, ad 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 (i) è non ridondante e (ii) 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 $X \rightarrow A \in F$, 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 una 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, possiamo nel passo (b) 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: $M \rightarrow D$, $M \rightarrow G$, $M \rightarrow C$, $G \rightarrow R$, $D \rightarrow S$, $S \rightarrow D$, $MP \rightarrow A$.

10.6.3 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 base di 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 10.4.1, 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 $R(MCGRDSPA)$ con le dipendenze mostrate nell'esempio alla fine del paragrafo 10.6.2, 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 $M \rightarrow D,\ M \rightarrow G,\ M \rightarrow C$
- $R_2(GR)$ 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 10.3.2, 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.

10.7 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 fase 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 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 alla 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 una entità o una 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). Ad esempio, la relazione in figura 10.10 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.

10.7.1 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 una 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). Ad esempio, consideriamo (vedi figura 10.13) una 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

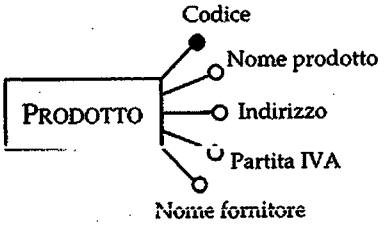


Figura 10.13 Una entità da sottoporre a verifica di normalizzazione

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 fatto 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 una associazione che le collega. Si tratta chiaramente di una 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. Ad 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 10.14, 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 (ad esempio, la dipendenza fra i codici dei prodotti e i nomi dei fornitori è ricostruibile tramite l'associazione FORNITURA, che associa a un prodotto un 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.

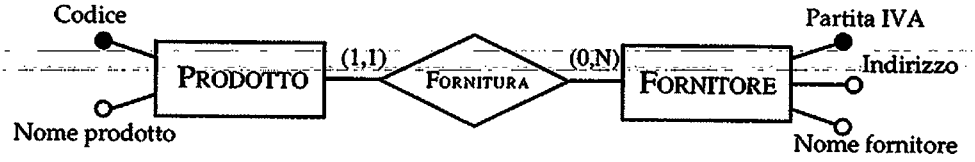


Figura 10.14 Il risultato della decomposizione di una entità

10.7.2 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 della 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 ad esempio l'associazione TESI in figura 10.15, che coinvolge le entità STUDENTE, PROFESSORE, CORSODILAUREA e DIPARTIMENTO, il cui scopo 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);

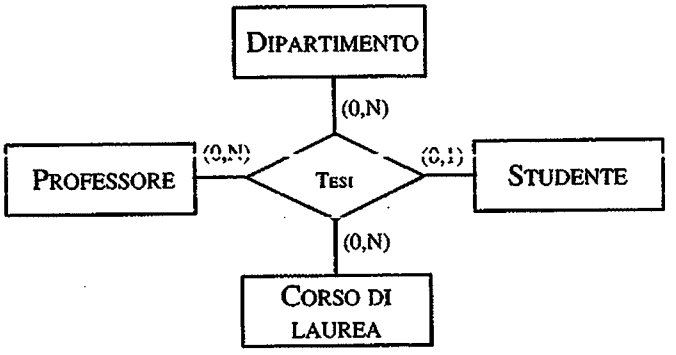


Figura 10.15 Una associazione da sottoporre a verifica di normalizzazione

- 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 10.16, 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.

10.7.3 Ulteriori decomposizioni di associazioni

Sullo schema in figura 10.16 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

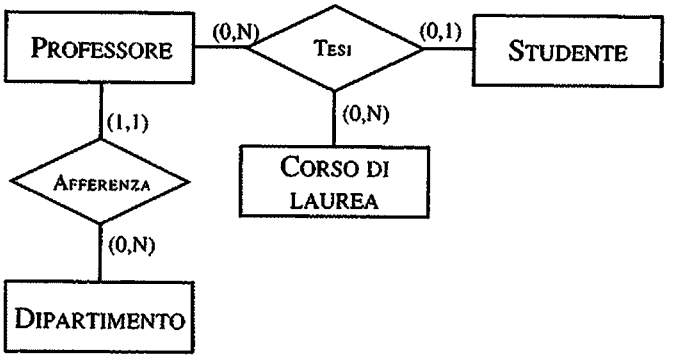


Figura 10.16 Il risultato della decomposizione di una associazione

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 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 una associazione"). Attraverso le dipendenze possiamo quindi notare che sarebbe opportuno decomporre ulteriormente la associazione, ottenendo due associazioni, una per ciascuno dei due concetti. La figura 10.17 mostra lo schema decomposto. La decomposizione è anche in questo caso accettabile, perché conserva le dipendenze ed è senza perdita.

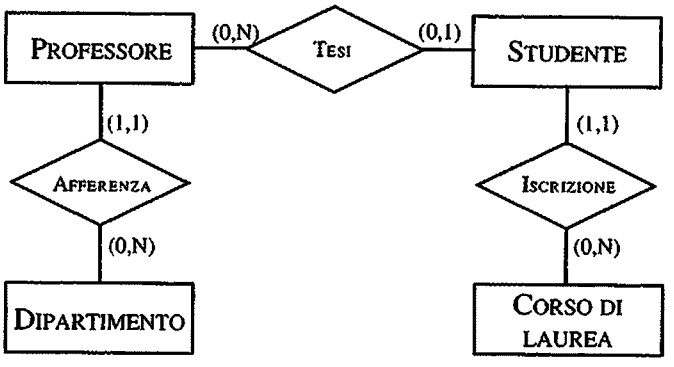


Figura 10.17 Il risultato di una ulteriore decomposizione di una associazione

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 coinvolgano più di tre entità, possiamo affermare che è di solito opportuno decomporre una 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.

10.7.4 Ulteriori decomposizioni di schemi concettuali

Anche il caso, discusso alla fine del paragrafo 10.4 di questo capitolo, 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. Ad esempio, nel corso dell'analisi, possiamo supporre di avere definito lo schema in figura 10.18 per il quale valgono le dipendenze fra entità corrispondenti alle dipendenze fra attributi discusse nel paragrafo 10.4.

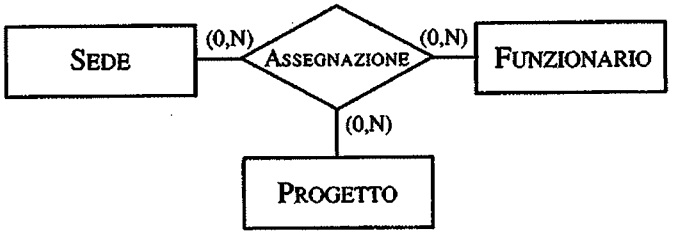


Figura 10.18 Una associazione difficile da decomporre

Esaminando l'associazione, possiamo notare che essa non è in forma normale di Boyce e Codd e non può essere utilmente decomposta. Proprio in questo contesto, possiamo individuare la possibilità di introdurre il concetto di reparto, per mezzo di una nuova entità, come mostrato nello schema figura 10.19, 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 10.12.

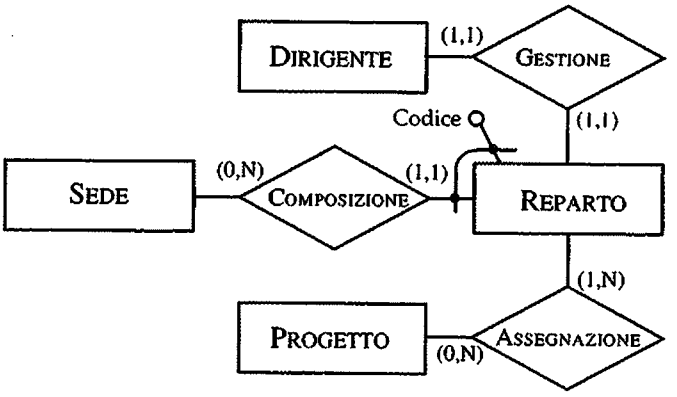


Figura 10.19 Una ristrutturazione dello schema in figura 10.18

Note bibliografiche

Le nozioni di base sulla normalizzazione, con la definizione di terza forma normale, sono state proposte da Codd [24]. La teoria della normalizzazione può essere approfondita nel testo in italiano di Atzeni, Batini e De Antonellis [4] oppure nel testo in inglese di Atzeni e De Antonellis [6].

Altri tre testi che studiano in maniera approfondita e formale aspetti legati alle dipendenze funzionali e alla teoria della normalizzazione sono quelli di Maier [43], Ullman [64] e Abiteboul, Hull e Vianu [1].

Sul testo di Cabibbo, Torlone e Batini [11], si trovano diversi esercizi di normalizzazione che fanno riferimento alla forma normale di Boyce e Codd.

Esercizi

Esercizio 10.1 Considerare la relazione in figura 10.20 e individuare le proprietà della corrispondente applicazione. Individuare inoltre eventuali ridondanze e anomalie nella 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 10.20 Relazione per l'esercizio 10.1

Esercizio 10.2 Individuare la chiave e le dipendenze funzionali della relazione considerata nell'esercizio 10.1 e individuare poi una decomposizione in forma normale di Boyce e Codd.

Esercizio 10.3 Si consideri la relazione riportata in figura 10.21 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.

Prodotto	Componente	Tipo	Q	PC	Fornitore	PT
Libreria	Legno	Noce	50	10000	Forrest	400000
Libreria	Bulloni	B212	200	100	Bolt	400000
Libreria	Vetro	Cristal	3	5000	Clean	400000
Scaffale	Legno	Mogano	5	15000	Forrest	300000
Scaffale	Bulloni	B212	250	100	Bolt	300000
Scaffale	Bulloni	B412	150	300	Bolt	300000
Scrivania	Legno	Noce	10	8000	Wood	250000
Scrivania	Maniglie	H621	10	20000	Bolt	250000
Tavolo	Legno	Noce	4	10000	Forrest	200000

Figura 10.21 Una relazione contenente dati di una falegnameria

Esercizio 10.4 Con riferimento alla relazione in figura 10.21 si considerino le seguenti operazioni di aggiornamento:

- inserimento di un nuovo prodotto;
- cancellazione di un prodotto;
- aggiunta di una componente a un prodotto;
- modifica del prezzo di un prodotto.

Discutere i tipi di anomalia che possono essere causati da tali operazioni.

Esercizio 10.5 Si consideri sempre la relazione in figura 10.21. 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.

Esercizio 10.6 Individuare le dipendenze funzionali definite sulla relazione in figura 10.21 e decomporre la relazione con l'algoritmo di sintesi di schemi in terza forma normale illustrato nel paragrafo 10.6.3.

Esercizio 10.7 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.

Esercizio 10.8 Si consideri lo schema della relazione in figura 10.22. La chiave di questa relazione è costituita dagli attributi Titolo e Copia, e su di esso è definita la dipendenza 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 10.22 Relazione per l'esercizio 10.8

Esercizio 10.9 Si consideri lo schema Entità-Relazione in figura 10.23. Sui dati

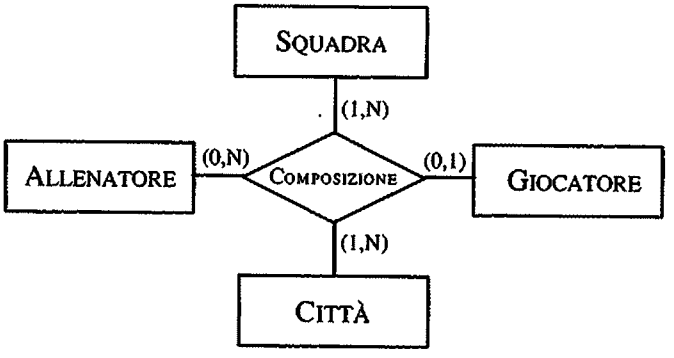


Figura 10.23 Uno schema da sottoporre a verifica di normalizzazione

descritti da questo schema valgono le seguenti proprietà:

- 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.

Esercizio 10.10 Consideriamo la relazione in figura 10.24 e le sue seguenti possibile decomposizioni:

- Reparto, Cognome in una relazione e Cognome, Nome, Indirizzo nell'altra;
- Reparto, Cognome, Nome in una relazione e Nome, Indirizzo nell'altra;
- Reparto, Cognome, Nome 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 10.24 Relazione per l'esercizio 10.10

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.

Esercizio 10.11 Consideriamo nuovamente la relazione in figura 10.24. 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.

A

Microsoft Access

Access, prodotto dalla Microsoft, è il più diffuso sistema di gestione di basi di dati per l'ambiente Microsoft Windows. Access può essere utilizzato in due modalità: come gestore di basi di dati autonomo su personal computer e come interfaccia verso altri sistemi.

Come gestore di basi di dati autonomo risente dei limiti dell'architettura dei personal computer: offre un supporto transazionale limitato, con meccanismi di sicurezza, protezione dei dati e gestione della concorrenza piuttosto semplici e incompleti. D'altra parte ha un costo assai ridotto, che giustifica queste limitazioni, e le applicazioni cui è destinato non hanno tipicamente bisogno di fare un uso sofisticato di questi servizi. L'interfaccia del sistema sfrutta le potenzialità dell'ambiente grafico e offre un ambiente facile da usare, sia per l'utente applicativo sia per il progettista della base di dati.

Quando è usato come client di database server relazionali, Access mette a disposizione le proprie funzionalità di dialogo per l'interazione con questi sistemi. Access, in questo contesto, può essere visto come uno strumento che permette di evitare di scrivere codice SQL, in quanto acquisisce schemi e semplici interrogazioni tramite una rappresentazione grafica facilmente comprensibile; questi input vengono tradotti in opportuni comandi SQL in modo trasparente. Il protocollo ODBC, descritto nel capitolo 6, viene usato per la comunicazione tra Access e il database server.

La descrizione di Access si concentrerà sulle funzioni di gestore di basi di dati, ponendo particolarmente l'accento sulle funzionalità di definizione di schemi e interrogazioni. La presentazione si baserà sulla versione italiana di Access 2003, disponibile come un prodotto a sé stante o come un componente di Microsoft Office. Per una descrizione completa del sistema, invitiamo a consultare i manuali forniti con il programma e la guida in linea raggiungibile dal menu di aiuto (l'ultimo a destra, caratterizzato dall'etichetta '?'). Sono anche facilmente reperibili un gran numero di testi dedicati alla descrizione di questo sistema (ad esempio, [55]).

MSACCESS

Figura A.1 L'icona del programma

A.1 Caratteristiche del sistema

Il sistema viene attivato nel modo tradizionale in cui partono le applicazioni Windows, ovvero selezionando l'icona del programma in una finestra o in un menu; l'icona è rappresentata in figura A.1.

All'avvio, il programma chiede se si vuole creare un nuovo database o aprire un database esistente, fornendo una lista dei database che sono stati utilizzati precedentemente dal programma. La creazione di un database può partire da un database vuoto o seguire una serie di modelli predefiniti (utili per utenti inesperti). A ogni database corrisponde un file con estensione standard mdb; per aprire un database preesistente bisogna selezionare il corrispondente file. Per creare un nuovo database o aprirne uno preesistente si possono anche utilizzare i comandi Nuovo database e Apri database del menu File. Il database appena creato o aperto è rappresentato da una piccola finestra, che compare all'interno della finestra principale dell'applicazione. La figura A.2 illustra questa situazione. La finestra principale contiene un insieme di menu a tendina (File, Modifica, ecc.) e, immediatamente sotto, una riga che contiene un insieme di pulsanti, premendo i quali vengono eseguiti particolari comandi. La riga di pulsanti viene detta *barra degli strumenti* o *toolbar*. I menu e la barra degli strumenti variano al variare della finestra in primo piano all'interno di Access. La figura A.2 mostra il contenuto della barra degli strumenti quando è selezionata la finestra che rappresenta la base di dati.

La finestra della base di dati contiene sul lato sinistro (sotto forma di opzioni) un elenco delle famiglie degli oggetti del sistema: *Tabelle*, *Query*, *Maschere*, *Report*, *Pagine*, *Macro* e *Moduli*. In ogni momento è selezionata una delle famiglie. Per passare da una famiglia all'altra, si seleziona il nome corrispondente. Quando una famiglia è selezionata, nella finestra compaiono tutti i nomi degli elementi della famiglia presenti nel database (se vi sono più elementi di quanti la finestra ne possa contenere, viene automaticamente aggiunta una barra di scorrimento con la quale è possibile navigare lungo la lista). In figura A.2 si vede l'insieme di elementi della famiglia *Tabelle* presenti nella base di dati di riferimento. Oltre le prime tre voci, che rappresentano tre diverse modalità per la creazione di una tabella, osserviamo che la base di dati contiene una tabella CITTÀ e una tabella PERSONA.

A questo punto, per ogni famiglia, è possibile premere uno dei tre pulsanti che compaiono sulla parte destra della finestra.

- Con il pulsante APRI si visualizza il contenuto di un elemento selezionato;
- Con il pulsante STRUTTURA si riapre la fase di progetto su un elemento selezionato;

- Con il pulsante NUOVO si crea un nuovo elemento della famiglia.

I comandi NUOVO e STRUTTURA vanno quindi ad accedere alle informazioni di progetto del componente (al suo schema), mentre APRI accede al suo contenuto (alla sua istanza).

Descriveremo ora la definizione di tabelle e interrogazioni, mentre tratteremo sommariamente le funzionalità offerte per la gestione di maschere, report, macro e moduli.

A.2 La definizione delle tabelle

Per definire lo schema di una nuova tabella, bisogna selezionare dalla finestra del database la famiglia *Tabelle* e premere il pulsante NUOVO. A questo punto Access propone la scelta tra cinque diverse modalità di definizione della tabella: Visualizzazione Foglio dati, Visualizzazione Struttura, Creazione guidata Tabella, Importa tabella e Collega tabella. Con Visualizzazione Foglio dati si definisce lo schema della tabella come se si definisse la struttura di un *foglio elettronico* (*spreadsheet*), ovvero gli oggetti gestiti da applicazioni come Microsoft Excel o Lotus 123. Questa interfaccia è dedicata a chi ha esperienza nell'uso di fogli elettronici e può rendere più facile per questa categoria d'utenti l'uso di un sistema relazionale. Le opzioni Importa tabella e Collega tabella permettono di importare dall'esterno una tabella, rispettivamente ricopiandone l'intero suo contenuto o costruendo un collegamento che permetta di recuperarne il contenuto dinamicamente dalla risorsa remota (in effetti si definisce una vista remota). Con

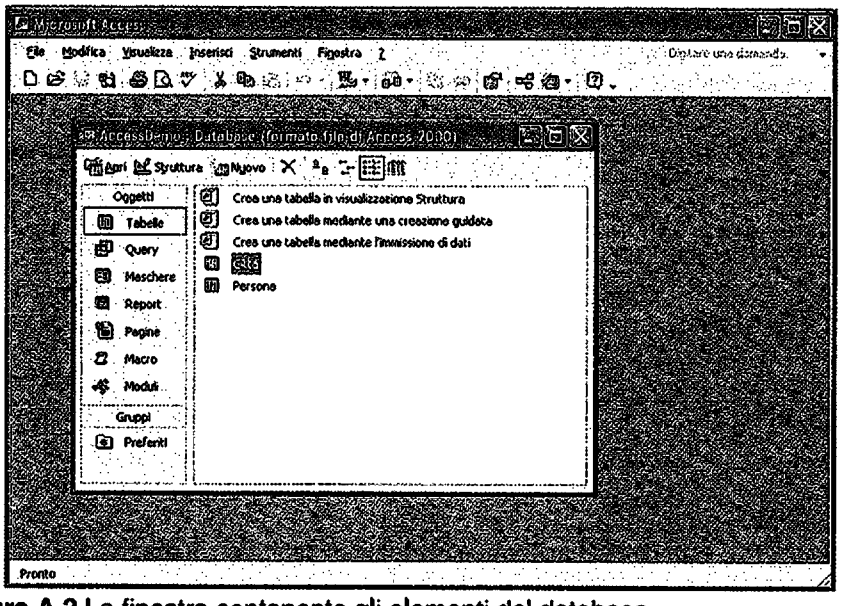


Figura A.2 La finestra contenente gli elementi del database

l'opzione Creazione guidala Tabella si consente l'uso di un "wizard", ovvero uno strumento di supporto che guida nella creazione di tabelle ponendo una serie di domande e fornendo una collezione di esempi cui ispirarsi. Access propone l'uso dei wizard anche negli altri contesti di definizione dei vari componenti. L'uso di wizard è consigliabile per la creazione di componenti dalla struttura regolare oppure nel caso di utenti poco esperti; per la gestione di situazioni particolari o nei caso di utenti sofisticati conviene invece fare uso delle funzionalità complete di definizione disponibili con la opzione Visualizzazione Struttura, su cui concentriamo l'analisi.

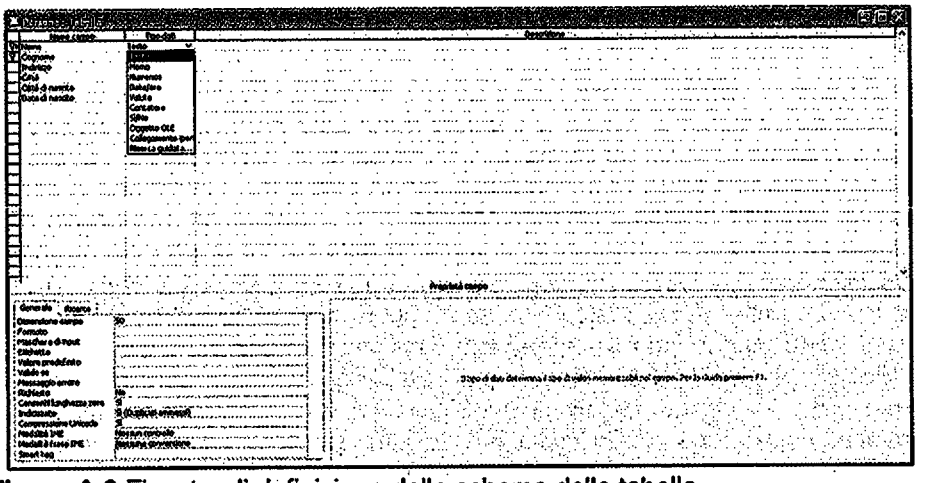


Figura A.3 Finestra di definizione dello schema delle tabelle

Per prima cosa si definiscono gli attributi, utilizzando la finestra che compare in figura A.3. Per ciascun attributo bisogna specificare il nome e il dominio. Vi è poi un insieme di ulteriori informazioni definibili, che variano a seconda del dominio associato all'attributo (nel programma gli attributi vengono chiamati *campi*, i domini *tipi*). I domini che possono essere associati a un attributo ricordano i domini dello standard SQL, con qualche arricchimento. I domini definibili sono:

- `Testo`: permette di rappresentare stringhe di caratteri (corrisponde al dominio `varchar` di SQL).
- `Memo`: permette di rappresentare stringhe di testo lunghe fino a 64.000 caratteri; non è un dominio esplicitamente previsto da SQL, ma può essere inteso come un caso particolare di `varchar`.
- `Numerico`: rappresenta la famiglia di domini numerici, interi e reali. Le proprietà del dominio permettono di specificare se l'attributo rappresenta valori precisi o approssimati, e il grado di precisione della rappresentazione. Corrisponde alle famiglie di domini `numeric`, `decimal`, `integer`, `smallint`, `float`, `double` e `real`.
- `Data/Ora`: rappresenta istanti temporali e corrisponde ai domini SQL `date`, `time` e `timestamp`, a seconda che si decida di rappresentare rispettivamente solo la data, solo l'ora o entrambe.

- `Valuta`: serve per rappresentare valori monetari. È essenzialmente un caso particolare del dominio `Numerico`, caratterizzato da una rappresentazione numerica esatta su 8 byte con due cifre decimali. È rappresentato con un dominio apposito a causa della particolare rilevanza applicativa che ha la rappresentazione di valori monetari.
- `Contatore`: è un particolare dominio che associa a ogni riga della tabella un valore unico. Questo dominio permette di associare una chiave compatta alla tabella.
- `Sì/No`: corrisponde al dominio `boolean` di SQL-3.
- `Oggetto OLE`: rappresenta un generico oggetto che può essere gestito tramite OLE (*Object Linking and Embedding*). OLE è un protocollo che permette di specificare quale applicazione deve gestire un oggetto nell'ambiente Windows. In questo modo si possono inserire all'interno di una base di dati informazioni dei tipi più vari, come documenti di un word processor, fogli elettronici, immagini o informazioni multimediali, lasciando all'applicazione invocata tramite OLE il compito di gestirne adeguatamente il contenuto.
- `Collegamento ipertestuale`: permette la definizione di un riferimento. Il riferimento può essere interno alla base di dati o anche esterno (come ad esempio l'URL di una risorsa disponibile in Internet).
- `Ricerca guidata...`: questa opzione permette la definizione di un meccanismo che in fase di inserimento consente di scegliere un valore tra quelli presenti in un elenco predefinito o estratti tramite una query dal database.

Per ciascun attributo si possono poi specificare un certo numero di proprietà, che qualificano ulteriormente il suo dominio e la sua rappresentazione interna. Le proprietà compaiono nella parte inferiore della finestra di definizione dello schema.

- *Dimensione campo*: rappresenta la dimensione dell'attributo. Si può specificare solo per i domini `Testo` e `Numerico`. Per il tipo `Testo` la dimensione è un valore che rappresenta la lunghezza massima della stringa di caratteri. Per il tipo `Numerico`, si possono specificare le seguenti dimensioni:
  - `Byte`: interi su 8 bit (valori tra 0 e 255);
  - `Intero`: interi su 16 bit (valori tra -32.768 e 32.767);
  - `Intero lungo`: interi su 32 bit;
  - `Precisione singola`: rappresentazione in virgola mobile su 32 bit;
  - `Precisione doppia`: rappresentazione in virgola mobile su 64 bit.
  - `ID replica`: identificatore su 128 bit, unico per ogni tupla, anche in un sistema distribuito.
  - `Decimale`: rappresentazione in virgola mobile, con la possibilità di specificare il numero di cifre decimali significative.
- *Formato*: descrive il formato di visualizzazione dei valori degli attributi. Access utilizza dove possibile i valori specificati nell'ambiente Windows (opzioni di internazionalizzazione del pannello di controllo). Per la rappresentazione di date, numeri e valori booleani, permette di scegliere tra vari formati predefiniti (sette formati predefiniti per le date, sette per i numeri, tre per i booleani). Si

possono poi definire altri formati. È possibile inoltre distinguere la rappresentazione dei valori a seconda che siano positivi, negativi, pari a zero, o abbiano associato il valore nullo.

- *Precisione*: questa informazione, definibile solo per attributi di dominio numerico `Decimale` specifica quante cifre decimali devono essere utilizzate nella rappresentazione.
- *Scala*: questa informazione, definibile solo per attributi di dominio numerico `Decimale` specifica quante cifre decimali devono essere utilizzate al massimo a destra del separatore decimale.
- *Posizioni decimali*: questa informazione, definibile solo per attributi di dominio numerico `Precisione singola`, `Precisione doppia` e `Decimale`, specifica il numero di cifre da utilizzare a destra del separatore decimale.
- *Maschera di input*: questo parametro specifica il formato che deve essere utilizzato per l'immissione dei dati. Se ad esempio un attributo registra un numero di telefono, composto da un prefisso di 3 cifre e da un numero di 7 cifre separati da un trattino, è possibile specificare una maschera di ingresso che individui le due parti e introduca immediatamente il trattino, permettendo all'utente di inserire solamente le cifre. Access mette a disposizione un wizard anche per la creazione della *Maschera di input*.
- *Etichetta*: rappresenta il nome che può essere dato all'attributo quando compare in una maschera o in un report. Può capitare di utilizzare come nome dell'attributo un nome compatto con cui sia possibile scrivere interrogazioni concise, mentre per la visualizzazione dei risultati si preferisca usare un nome che rappresenti meglio il significato dell'attributo.
- *Valore predefinito*: con questo parametro si specifica il valore di default per l'attributo. Corrisponde esattamente all'opzione `default` di SQL. Tutte le volte che si inserisce una nuova tupla, il valore di default comparirà come valore per l'attributo. Si può utilizzare come valore di default anche il risultato di un'espressione, come ad esempio `=Date()` che assegna a un campo di dominio `Data/Ora` il giorno corrente.
- *Valido se*: permette di specificare un vincolo sull'attributo. Access verifica automaticamente che ogni valore inserito appartenga al dominio dell'attributo. Oltre a questo controllo, Access permette di specificare un vincolo generico per ogni attributo (in modo analogo alla clausola `check` di SQL). Questo vincolo viene espresso utilizzando la sintassi che è utilizzata per la specifica delle condizioni in QBE, che vedremo nei prossimi paragrafi.
- *Messaggio errore*: specifica il messaggio che deve essere visualizzato quando viene inserito un valore che non rispetta il vincolo d'integrità.
- *Richiesto*: specifica se deve essere sempre presente un valore per l'attributo. La proprietà può essere vera o falsa e corrisponde al vincolo `not null` di SQL.
- *Consenti lunghezza zero*: è una proprietà che vale solo per gli attributi di tipo `Testo` e `Memo`. Specifica se devono essere ammesse stringhe di lunghezza nulla, o se una stringa di lunghezza nulla deve essere considerata come un valore nullo. A seconda dei contesti applicativi, può essere utile gestire in modo diverso le stringhe vuote dal valore nullo. Si tenga presente che il valore nullo viene trattato in modo particolare da SQL, per cui un confronto di disuguaglianza su

stringhe è soddisfatto da una stringa di lunghezza zero, ma non da una stringa nulla.

- *Indicizzato*: specifica se deve essere costruito un indice sull'attributo. Le opzioni possibili sono `No`, `Sì (Duplicati ammessi)` e `Sì (Duplicati non ammessi)`. La terza opzione definisce sull'attributo un indice di tipo *unique*. Questo è anche il modo in cui si rappresentano i vincoli di tipo `unique`. Non si possono definire indici su attributi `Oggetto OLE`. Con questa modalità si permette solo la definizione di indici su un solo attributo; per la definizione di indici più complicati bisogna operare al livello di tabella.
- *Compressione Unicode*: specifica per i campi di tipo testuale se la rappresentazione Unicode a 16 bit dei caratteri, che costituisce lo standard di Microsoft Access a partire dalla versione 2000, deve essere compressa nei casi in cui si rappresentano caratteri del normale alfabeto latino. La compressione avviene omettendo la rappresentazione del primo byte, il quale è pari a zero per tutti i caratteri del dominio ASCII.
- *Modalità IME*: specifica se sul campo deve essere utilizzato un *Input Method Extension*, ovvero una modalità particolare di acquisizione dell'input che consente di gestire gli alfabeti ideogrammatici che caratterizzano le lingue orientali. Non è un'opzione significativa se non si prevedono di utilizzare alfabeti di questo tipo.
- *Modalità frase IME*: specifica ulteriormente la modalità di input da adottare per l'acquisizione di frasi nei linguaggi ideogrammatici.
- *Smart tag*: specifica se all'attributo deve essere associata un'*etichetta intelligente*, che permette di invocare funzioni predefinite nella fase di input e consultazione del campo (ad esempio, per attributi che rappresentano date è possibile accedere a un calendario o invocare l'applicazione di gestione dell'agenda).

Dopo aver definito i vari attributi, si completa la sessione di definizione indicando quali attributi devono essere considerati la chiave primaria della tabella. L'indicazione degli attributi di chiave avviene selezionando gli attributi e premendo il pulsante che raffigura la chiave nella barra degli strumenti (o selezionando l'opzione Chiave primaria nel menu Modifica). Gli attributi che costituiscono la chiave vengono visualizzati con un'icona che rappresenta una chiave nella colonna che precede il nome. Automaticamente Access definirà un indice di tipo `unique` sugli attributi che costituiscono la chiave.

Si possono poi definire ulteriori proprietà a livello di tabella (cui si accede tramite l'opzione Proprietà del menu Visualizza). Le più significative sono:

- *Descrizione*: è una descrizione testuale del contenuto della tabella.
- *Valido se*: specifica un vincolo che deve essere soddisfatto da ogni tupla della tabella. A livello di tabella si definiscono vincoli che coinvolgono diversi attributi. Pure in questo caso, la sintassi è quella usata per l'espressione delle condizioni sulle interrogazioni. Il controllo viene effettuato al termine dell'inserimento di ogni tupla.
- *Messaggio errore*: rappresenta il messaggio che viene visualizzato quando il sistema rileva una violazione del vincolo.

- *Filtro*: specifica le condizioni che devono essere soddisfatte dagli elementi che si vogliono visualizzare.
- *Ordina per*: descrive gli attributi rispetto ai quali si devono ordinare le tuple della tabella.

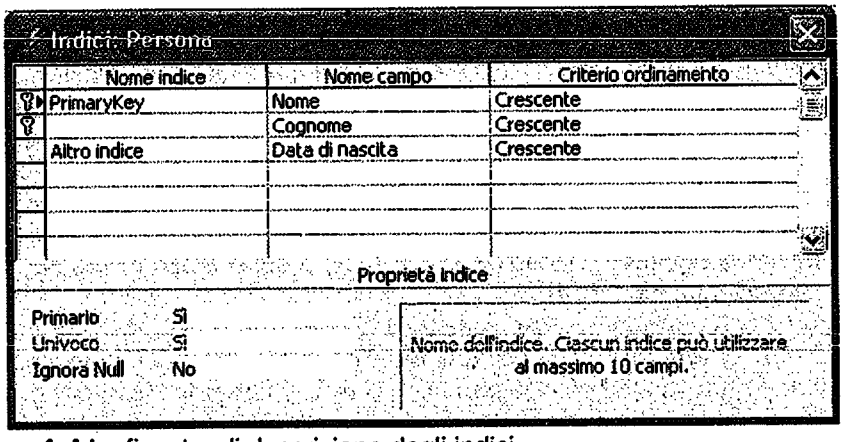


Figura A.4 La finestra di descrizione degli indici

Per specificare indici su più attributi si deve aprire la finestra di definizione degli indici, premendo il pulsante INDICI sulla barra degli strumenti, o selezionando la voce Indici dal menu Visualizza (il pulsante compare nella toolbar e la voce è presente nel menu solo quando si è nella fase di definizione della struttura della tabella). La finestra contiene una tabella (mostrata in figura A.4) con colonne Nome indice, Nome campo e Criterio ordinamento. Per definire un indice su più attributi, si inserisce in una riga il nome dell'indice, il nome del primo attributo e la direzione di ordinamento. Nella riga successiva si lascia vuoto il nome dell'indice e si introduce il nome del secondo attributo e la corrispondente direzione di ordinamento, proseguendo allo stesso modo per tutti gli attributi che caratterizzano l'indice.

Prima di terminare la sessione di definizione bisogna salvare il risultato (selezionando l'opzione Salva del menu File); Access a questo punto chiede quale nome deve essere associato alla tabella. Il nome può anche contenere al suo interno degli spazi.

A.2.1 Specifica dei cammini di join

Un cammino di join è un legame tra coppie di attributi di due tabelle e si usa per specificare che tra le due tabelle viene normalmente eseguito un join basato sull'uguaglianza di quegli attributi. Un cammino di join viene rappresentato graficamente da una linea che collega le due tabelle. Per esempio, nella base di dati "persone e città", esiste un cammino di join tra l'attributo Città di nascita di PERSONA e Nome di CITTÀ. Access permette la definizione di cammini di join (chiamati *relazioni*); per ogni cammino di join è inoltre possibile specificare se vi

è associato un vincolo di integrità referenziale. Tutto questo avviene senza che venga immesso del testo, operando esclusivamente in modo grafico. Descriviamo brevemente la procedura.

Si inizia selezionando l'opzione **Relazioni** dal menu **Strumenti**. Si apre a questo punto una finestra (figura A.5) in cui si possono inserire gli schemi delle tabelle create, selezionandole da una lista. È possibile definire i cammini di join tra gli schemi selezionando un attributo di uno schema e, tenendo premuto il pulsante del mouse, spostando il puntatore del mouse sull'attributo corrispondente dell'altra tabella. Una volta definito il legame tra gli attributi, Access apre una finestra che rappresenta gli attributi coinvolti nel join e che permette di estendere la condizione di join ad altri attributi o di modificare la relazione così definita. Questa finestra è rappresentata in figura A.6. Premendo il pulsante TIPO JOIN di questa finestra, si ha la possibilità di scegliere quale tipo di join, *inner*, *outer left* o *outer right*, deve essere utilizzato nel legame tra le due tabelle (a differenza di SQL, il *full outer join* non è offerto). In questo modo tutte le volte che si definisce una query che accede alle due tabelle, verranno assegnate di default le condizioni di join specificate.

Definito il cammino di join, è possibile specificare se a esso deve essere associato un vincolo di integrità referenziale. Nella stessa finestra, Access permette di definire una politica di reazione alle violazioni, per cui si può imporre che le modifiche o le cancellazioni siano seguite da corrispondenti modifiche e cancellazioni nelle altre tabelle. Se la politica non è specificata, ogni modifica che introduca una violazione viene semplicemente impedita.

Rispetto allo standard SQL-2, Access permette quindi un insieme di reazioni più limitato, corrispondente alle sole scelte di `cascade delete` e `cascade update`. Per specificare i vari vincoli di integrità referenziale, si devono seguire i criteri di progettazione descritti nel capitolo 7. Access non permette che nel grafo compaia più di un cammino tra due tabelle; se si devono rappresentare più cammini di join tra due tabelle, bisogna introdurre più esemplari della stessa tabella del grafo (come avviene in figura A.5).

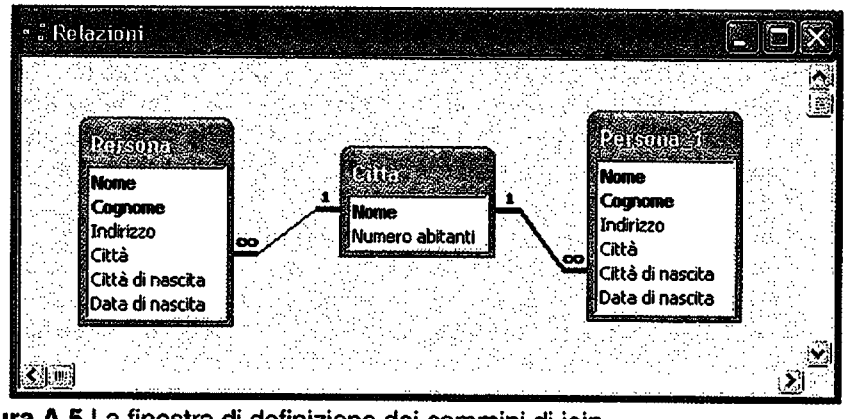


Figura A.5 La finestra di definizione dei cammini di join

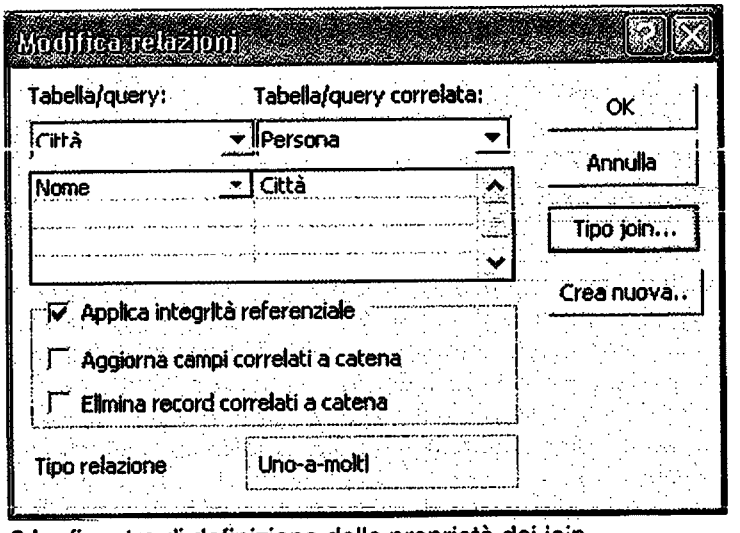
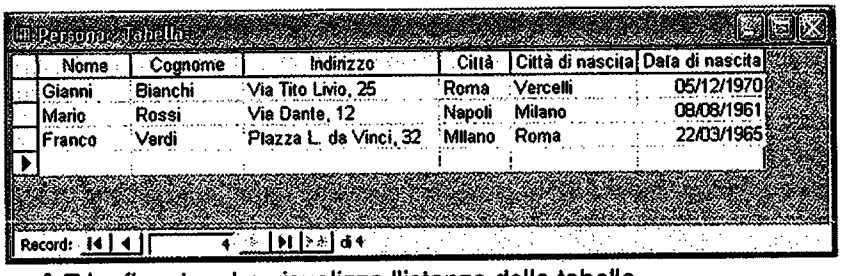


Figura A.6 La finestra di definizione delle proprietà dei join



Nome	Cognome	Indirizzo	Città	Città di nascita	Data di nascita
Gianni	Bianchi	Via Tito Livio, 25	Roma	Vercelli	05/12/1970
Mario	Rossi	Via Dante, 12	Napoli	Milano	08/08/1961
Franco	Verdi	Piazza L. da Vinci, 32	Milano	Roma	22/03/1965

Figura A.7 La finestra che visualizza l'istanza della tabella

A.2.2 Popolamento delle tabelle

Dopo aver definito lo schema delle tabelle, si possono inserire tuple nella tabelle per popolare l'istanza della base di dati. Anche per questo compito Access fornisce un'interfaccia grafica facile da usare. Aprendo una tabella dalla finestra di partenza premendo il pulsante APRI, compare una rappresentazione tabellare del contenuto della tabella. Questa consiste in una griglia con colonne con intestazione pari ai nomi degli attributi, e righe che descrivono le tuple della tabella, cui si somma una riga vuota che serve per inserire nuove tuple nella tabella. La figura A.7 rappresenta questa finestra.

L'inserimento avviene ponendo il cursore nell'ultima riga e digitando un valore per ogni attributo. Se il valore inserito non rispetta tutti i vincoli definiti sull'attributo, l'inserimento viene immediatamente rifiutato. Spostando il cursore al di fuori della riga, si indica implicitamente che l'inserimento è terminato. A questo punto il sistema controlla che tutti i vincoli siano rispettati, ovvero che i campi per cui è necessario fornire un valore siano stati specificati e che i valori degli attributi rispettino le regole di validazione definite. Per effettuare delle mo-

difiche di attributi basta andare a selezionare il valore da modificare con il cursore e inserire il nuovo valore. Terminata l'immissione del valore e spostato il cursore in una diversa posizione, i vincoli sono verificati e la modifica è completata.

A.3 La definizione di query

Per la definizione di interrogazioni, Access mette a dispozione due diversi strumenti: uno strumento grafico di formulazione di interrogazioni di tipo QBE (*Query By Example*) e un interprete SQL. Descriviamo dapprima le caratteristiche dell'interfaccia QBE, analizzando successivamente l'interprete SQL.

A.3.1 Query By Example

Il nome QBE fa riferimento a una famiglia molto vasta di linguaggi di interrogazione per basi di dati, che realizzano un'implementazione delle idee di base del calcolo relazionale dei domini (paragrafo 3.2.1). Il punto fondamentale è che un'interrogazione venga formulata descrivendo le caratteristiche che devono essere possedute dalle righe del risultato. La definizione di una query avviene riempiendo uno schema di tabella con tutti gli attributi e le condizioni che caratterizzano una riga "esemplare" del risultato.

Per definire una nuova interrogazione si seleziona l'opzione Query della finestra di partenza del database. Premendo il pulsante NUOVO e scegliendo l'opzione *Visualizzazione Struttura* si apre la finestra di progettazione di query. Questa finestra è divisa in due metà (si vedano le figure A.8 e A.12). La metà superiore è inizialmente vuota e viene riempita con una descrizione degli schemi delle tabelle selezionate da una lista. Le tabelle sono collegate dai cammini di join predefiniti (possiamo in effetti interpretare la metà superiore della finestra come la porzione del diagramma Relazioni rilevante per l'interrogazione che si deve definire). Nella metà inferiore della finestra compare una tabella inizialmente vuota, con un insieme di colonne senza nome e con righe etichettate Campo, Ordinamento, Mostra e Criteri.

Le celle della riga Ordinamento possono essere vuote o contenere una tra le opzioni Crescente o Decrescente. Quando la riga non è vuota viene imposto un ordinamento delle tuple del risultato, secondo i valori dell'attributo associato alla colonna in cui compare l'opzione. Se vi sono più colonne con la cella Ordinamento attivata, si applica per primo l'ordinamento che compare più a sinistra nelle colonne; a pari valori dell'attributo, si applicheranno man mano le successive condizioni di ordinamento, procedendo da sinistra a destra.

Le celle della riga Mostra contengono un quadratino che può o meno contenere un segno di spunta. Se il quadratino contiene tale segno, l'attributo che compare nella colonna dovrà far parte del risultato della interrogazione. Lo stato del quadratino commuta con un clic del mouse.

Le celle della riga Criteri contengono le condizioni che devono essere soddisfatte dalle tuple risultato della interrogazione. Le condizioni possono essere semplici confronti tra il valore dell'attributo che compare in cima alla colonna

e una costante; in questo caso basta inserire il valore della costante nella cella. La condizione può anche essere più complicata e includere confronti più ricchi, espressioni e riferimenti ad altri attributi, come vedremo negli esempi successivi.

Per mettere i nomi degli attributi in cima alle colonne si possono utilizzare due modalità: si può scrivere direttamente il nome dell'attributo, eventualmente qualificato col nome della tabella di appartenenza, o si possono selezionare gli attributi che compaiono nella rappresentazione degli schemi della metà superiore della finestra, "trascinandoli" nelle relative colonne o facendo un doppio clic su di essi (a quel punto l'attributo viene riportato nella prima colonna libera). Dopo che la query è stata formulata, per eseguirla bisogna premere il pulsante della barra degli strumenti che contiene il punto esclamativo; dopo l'esecuzione, la tabella risultato della query compare al posto della finestra di definizione della query.

Vediamo alcuni esempi di definizione di interrogazioni. Supponiamo di avere una base di dati con una tabella PERSONA(Nome,Cognome, Indirizzo, Città, Città di nascita, Data di nascita) e una tabella CITTÀ(Nome, Numero abitanti). Per trovare i nomi, ordinati alfabeticamente, delle persone aventi cognome Rossi possiamo riempire lo schema nel modo descritto in figura A.8.

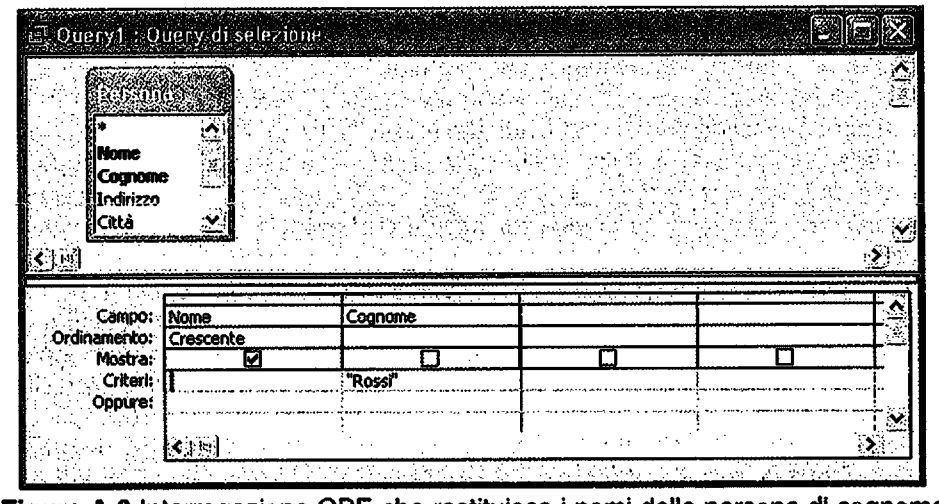


Figura A.8 Interrogazione QBE che restituisce i nomi delle persone di cognome Rossi

Quando sono riempiti più campi della riga Criteri, l'interrogazione considera tutte le condizioni in congiunzione. Se bisogna selezionare le tuple che soddisfano più condizioni in disgiunzione, bisogna riempire più righe con i diversi criteri. Access etichetta automaticamente con Oppure le righe aggiuntive. Così, per trovare i nomi, cognomi e indirizzi delle persone di Milano aventi cognome Rossi o Bianchi, si creerà uno schema come quello in figura A.9.

Nella lista degli attributi di una tabella compare anche il simbolo di asterisco che, con significato analogo a SQL, rappresenta tutti gli attributi. Per selezionare quindi tutti gli attributi delle tuple della tabella PERSONA che hanno la

Campo:	Nome	Cognome	Indirizzo	Città
Ordinamento:	Crescente			
Mostra:	☑	☑	☑	☐
Criteri:		"Rossi"		"Milano"
Oppure:		"Bianchi"		"Milano"

Figura A.9 Query che restituisce i milanesi chiamati "Rossi" o "Bianchi"

residenza nella città di nascita, potremo formulare la query QBE rappresentata in figura A.10.

Campo:	Persona.*	Città		
Ordinamento:				
Mostra:	☑	☐	☐	☐
Criteri:		[Città di nascita]		
Oppure:				

Figura A.10 Query che restituisce le persone con città di nascita e di residenza uguali

In questa interrogazione, per imporre l'uguaglianza di due attributi, abbiamo imposto come valore dell'attributo Città il nome dell'attributo Città di nascita racchiuso tra parentesi quadre. Le parentesi quadre sono il costrutto sintattico che permette ad Access di distinguere le stringhe di caratteri costanti dai riferimenti ai componenti dello schema. Access consente poi di formulare delle condizioni utilizzando i normali operatori di confronto (<, ≤, >, ≥ e <>) e l'operatore `Like`, per il confronto di stringhe con espressioni regolari che usano i caratteri speciali * e ? (che corrispondono rispettivamente ai caratteri % e _ della sintassi standard SQL). Per trovare i nomi e cognomi delle persone che sono nate prima del 31 gennaio 1965 e che hanno un cognome che inizia con la lettera 'C', si potrà formulare la query mostrata in figura A.11.

A ogni interrogazione è possibile associare delle proprietà, a diversi livelli. A livello di singola colonna, si può specificare un formato di visualizzazione diverso da quello immesso nella fase di definizione dello schema della tabella. Un'altra importante proprietà è l'eliminazione di eventuali duplicati presenti nel risultato. Per specificare che i duplicati devono essere rimossi bisogna accedere alla finestra di descrizione delle proprietà della query (tramite l'opzione Proprietà del menu Visualizza) e assegnare alla proprietà Valori univoci il valore *Sì*.

Per formulare delle interrogazioni che richiedono più tabelle, è opportuno far comparire nella finestra superiore le tabelle richieste (selezionandole dalla finestra di dialogo che compare appena si crea una nuova interrogazione). Le tabelle compariranno legate dai cammini di join che sono stati definiti al momento della definizione dello schema. Nella tabella nella metà inferiore della finestra di definizione della query è possibile aggiungere anche la riga Tabella, selezionando l'opzione Nomi tabelle dal menu Visualizza. La riga rappresenta il nome della

Campo:	Nome	Cognome	Data di nascita	
Ordinamento:				
Mostra:	☑	☑	☑	☐
Criteri:		Like "C*"	<#31/01/1965#	
Oppure:				

Figura A.11 Query che restituisce i nomi e cognomi delle persone con un cognome che inizia per 'C' nate prima del 31/1/65

tabella da cui viene prelevato l'attributo. Le condizioni di join tra le tuple delle tabelle non dovranno essere specificate se queste sono state predefinite. Quando le condizioni di join predefinite non sono quelle richieste dalla particolare query, è possibile modificare i cammini che collegano le tabelle intervenendo sulla rappresentazione grafica della parte superiore della finestra.

Volendo ad esempio formulare la query che trova le persone nate in una città con meno di centomila abitanti, se è stato predefinito il legame di join tra le tabelle PERSONA e CITTÀ, si può formulare la query che appare in figura A.12.

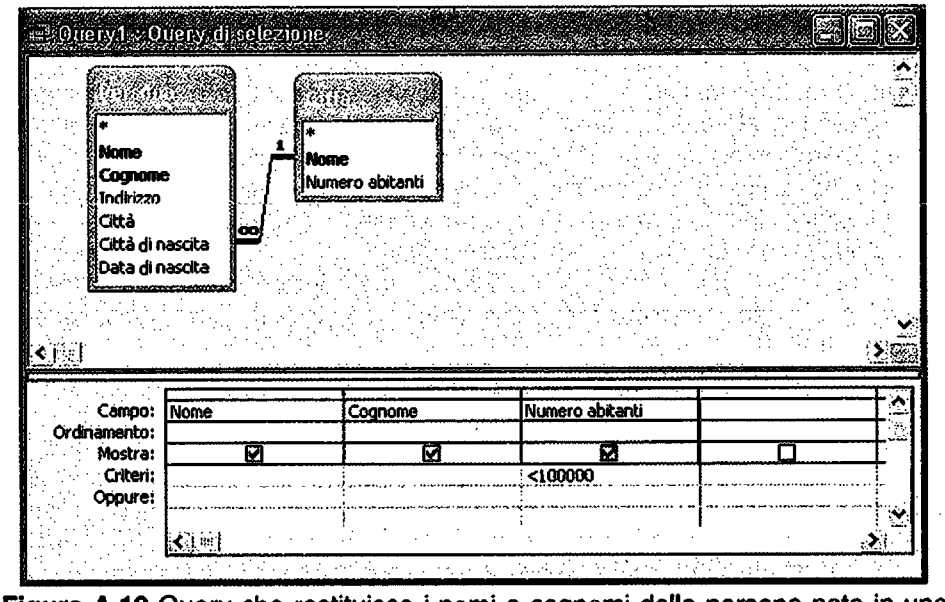


Figura A.12 Query che restituisce i nomi e cognomi delle persone nate in una città con meno di 100.000 abitanti (con join predefinito)

Qualora invece il legame di join non sia stato predefinito, si dovrà rendere esplicita la condizione nella tabella, formulando una interrogazione come quella in figura A.13, che usa i riferimenti agli attributi nelle righe Criteri.

Può essere necessario talvolta introdurre tra le tabelle che compaiono nella metà superiore anche tabelle di cui non vengono utilizzati attributi nella query. Un caso importante di questo tipo è quello in cui si devono estrarre informazioni

Campo:	Nome	Cognome	Nome	Numero abitanti
Tabella:	Persona	Persona	Città	Città
Ordinamento:				
Mostra:	☑	☑	☐	☐
Criteri:			[Città di nascita]	<100000
Oppure:				

Figura A.13 Query che restituisce i nomi e cognomi delle persone nate in una città con meno di 100.000 abitanti (senza join predefinito)

tra due tabelle che non posseggono un cammino di join diretto, ma in cui il join è realizzato tramite una tabella intermedia. In questo caso, anche se l'interrogazione utilizzerà per la visualizzazone e l'applicazione di condizioni solo attributi delle due tabelle esterne, la tabella intermedia dovrà comparire nella metà superiore, con un approccio simile a quello utilizzato dall'algebra, dal calcolo relazionale (capitolo 3) e da SQL (capitolo 4). Infatti, senza un cammino di join che leghi le tabelle, il sistema esegue il prodotto cartesiano delle tabelle che compaiono nella metà superiore, applicando poi le condizioni che sono state definite per la query.

Vediamo ora la formulazione di interrogazioni QBE facenti uso di operatori aggregati. Gli operatori aggregati che Access mette a disposizione sono `Somma`, `Media`, `Min`, `Max`, `Conteggio`, che corrispondono agli operatori standard SQL `sum`, `avg`, `min`, `max` e `count`. A questi si aggiungono gli operatori `DevSt` (la deviazione standard), `Var` (la varianza), `Primo` e `Ultimo` (il valore dell'attributo rispettivamente per la prima e per l'ultima tupla). Per utilizzare questi operatori è necessario introdurre una nuova riga nello schema della query, la riga Formula. Questa riga viene introdotta selezionando l'opzione Totali dal menu Visualizza. Vediamo un semplice esempio d'uso degli operatori, definendo, come mostrato in figura A.14, una query che permette di trovare il numero di tuple presenti nella tabella PERSONA.

Campo:	Nome
Tabella:	Persona
Formula:	Conteggio
Ordinamento:	
Mostra:	☑
Criteri:	
Oppure:	

Figura A.14 Query che restituisce il numero di persone

Il fatto che nella riga Formula compaia il valore `Conteggio` fa sì che il risultato della interrogazione non siano i valori dell'attributo Nome della tabella PERSONA, ma appunto il numero di tuple. Nella riga Campo deve comparire uno qualsiasi degli attributi della tabella PERSONA; sarebbe stato meglio, per coerenza con SQL-2, che Access permettesse in questo contesto l'uso dell'asterisco (*), utilizzando invece il nome di un attributo per contare i diversi valori dell'attributo; purtroppo nella valutazione degli altri operatori aggregati l'asterisco crea

delle complicazioni, per cui in Access si è scelta la soluzione di impedire l'uso dell'asterisco quando è abilitata la riga Formula.

Nella interrogazione in figura A.15 l'attributo Cognome viene caratterizzato dal valore `Raggruppamento` nella riga Formula, ovvero l'attributo viene utilizzato per raggruppare le tuple della tabella PERSONA. La seconda colonna rappresenta l'applicazione dell'operatore `Conteggio` a ogni singolo raggruppamento.

Campo:	Cognome	Nome		
Tabella:	Persona	Persona		
Formula:	Raggruppamento	Conteggio		
Ordinamento:				
Mostra:	☑	☑	☐	☐
Criteri:				
Oppure:				

Figura A.15 Query che restituisce il numero di persone che possiedono ciascun cognome

Access permette di esprimere delle condizioni sul risultato della valutazione degli operatori aggregati, in modo analogo alla clausola `having` in SQL. Per fare ciò, è sufficiente scrivere le condizioni nella riga Criteri, in modo analogo all'espressione di semplici condizioni sulle tuple. Così, per trovare i cognomi che sono posseduti da almeno due persone, si potrà scrivere l'interrogazione QBE in figura A.16, che restituisce i cognomi posseduti da più di una persona, indicando per ogni cognome il numero di volte che questo appare nella tabella PERSONA.

Campo:	Cognome	Nome		
Tabella:	Persona	Persona		
Formula:	Raggruppamento	Conteggio		
Ordinamento:				
Mostra:	☑	☑	☐	☐
Criteri:		>1		
Oppure:				

Figura A.16 Query che restituisce il numero di persone che possiedono ciascun cognome, per i cognomi posseduti da più persone

Se in una interrogazione con raggruppamento le tuple devono essere selezionate preliminarmente in base ai valori di attributi che non vengono utilizzati nel raggruppamento, diventa necessario distinguere le condizioni che devono essere valutate prima del raggruppamento da quelle che devono essere eseguite nella fase successiva. Questa distinzione in SQL avviene ponendo le condizioni preliminari nella clausola `where` e le condizioni successive nella clausola `having`; nel linguaggio QBE la distinzione avviene ponendo il valore `Dove` nella riga Formula per gli attributi che servono solo per la selezione delle tuple da raggruppare. La presenza del valore `Dove` è incompatibile con il flag di Mostra. Infatti, come in SQL, nel risultato delle interrogazioni che fanno uso di operatori aggregati possono comparire solo il risultato della valutazione degli operatori aggregati e gli

attributi su cui viene effettuato il raggruppamento. Ad esempio, per trovare le persone nate dopo l'1/1/75 che hanno lo stesso cognome di persone nate dopo la stessa data, si può utilizzare l'operatore `Count` e formulare l'interrogazione in figura A.17.

Ci sono dei modi alternativi per formulare questa interrogazione che non fanno uso del termine `Dove`. Uno di questi consiste nel formulare e salvare nella base di dati una interrogazione preliminare che estrae solo le persone nate dopo l'1/1/75; si può poi definire una seconda interrogazione, che parte dal risultato della prima andando a fare un raggruppamento in base al valore di Cognome, identificando così gli omonimi. A ogni interrogazione, infatti, Access associa un nome e un'interrogazione può estrarre le informazioni sia dalle tabelle del database, sia dalle interrogazioni già definite. Ogni interrogazione che viene salvata e resa persistente può quindi essere considerata come una definizione di una vista sul database.

A.3.2 L'interprete SQL

Oltre al linguaggio di interrogazione QBE, Access fornisce un interprete SQL, che può essere usato in alternativa a QBE. Access permette di passare rapidamente dal contesto QBE al contesto SQL e viceversa, premendo gli opportuni pulsanti presenti sull'estrema sinistra della barra degli strumenti o selezionando rispettivamente le opzioni VISUALIZZAZIONE SQL e VISUALIZZAZIONE STRUTTURA dal menu Visualizza. Il passaggio da un ambiente all'altro trasforma l'interrogazione corrente nella corrispondente interrogazione nell'altro ambiente.

Il passaggio da QBE a SQL è sempre possibile. In effetti, tutte le volte che una query QBE viene mandata in esecuzione, essa viene prima tradotta nella corrispondente forma SQL e quindi eseguita dall'interprete SQL. Il passaggio inverso non è invece sempre possibile, in quanto il linguaggio SQL è più potente di QBE, permettendo ad esempio l'espressione di query con l'operatore di unione. Il linguaggio QBE è un linguaggio molto potente e facile da usare quando si devono formulare interrogazioni che fanno uso solo di selezioni, proiezioni e join: in questo caso la possibilità di formulare le interrogazioni senza bisogno di scrivere del testo secondo una rigida sintassi rappresenta un notevole aiuto. D'altra parte QBE non mette a disposizione un meccanismo adeguato per la rappresentazione di interrogazioni complesse, come quelle che richiedono di formulare delle interrogazioni nidificate in SQL. Di fatto, quando una interrogazione SQL che fa uso

Campo:	Cognome	Data di nascita	Cognome	
Tabella:	Persona	Persona	Persona	
Formula:	Raggruppamento	Dove	Conteggio	
Ordinamento:				
Mostra:	☑	☐	☑	☐
Criteri:		>#01/01/1975#	>1	
Oppure:				

Figura A.17 Query che restituisce gli omonimi tra le persone nate dopo l'1/1/75

Campo:	Cognome	Cognome	Numero abitanti	
Tabella:	Persona	Persona	Città	
Formula:	Raggruppamento	Conteggio	Dove	
Ordinamento:		Decrescente		
Mostra:	☑	☑	☐	☐
Criteri:		>1	>200000	
Oppure:				

Figura A.18 Query che restituisce il numero di omonimi in città sopra i 200.000 abitanti

di query nidificate viene tradotta in QBE, la traduzione riporta semplicemente il testo dell'intera query nidificata nell'opportuna cella della riga Criteri.

Per quanto riguarda la sintassi riconosciuta dall'interprete, si tratta di un'estensione della sintassi standard SQL, con un supporto per particolari funzionalità e determinate differenze sintattiche e semantiche. Alcune delle differenze sono le seguenti:

- la clausola `top` può essere usata per selezionare un certo numero di tuple dal risultato;
- vengono usate le parentesi quadre per racchiudere gli identificativi di tabelle e attributi (necessarie quando compaiono degli spazi o caratteri speciali all'interno degli identificatori);
- l'operatore `join` deve essere sempre qualificato con il termine `inner` od `outer`;
- la valutazione dell'operatore `count` è diversa: se si dà come argomento un attributo, non vengono restituiti i distinti valori dell'attributo, bensì il numero di valori non nulli (come se fosse specificata implicitamente l'opzione `all`); l'opzione `distinct` non è riconosciuta.

Ad esempio, si consideri la query QBE in figura A.18, per cui è specificata la proprietà che vengano restituiti solo i primi 10 elementi e per cui è stato predefinito un cammino di join. A essa corrisponde la seguente query nel dialetto SQL di Access:

```
select top 10 Cognome, count(Cognome) as NumeroOmonimi
from Persona inner join Città
     on Persona.[Città di nascita] = Città.Nome
where [Numero abitanti] > 200000
group by Cognome
having count(Cognome) > 1
order by count(Cognome) desc
```

A.4 Maschere e report

Le *maschere* permettono di rappresentare il contenuto del database in modo molto più chiaro e comprensibile, rispetto alla piatta rappresentazione delle righe delle tabelle.

Le maschere sono analoghe ai "moduli prestampati", caratterizzati da un insieme di caselle in cui si devono inserire i dati, e un insieme di etichette che specificano quale dato deve essere inserito nella particolare casella. Le maschere possono servire per l'inserimento dei dati, realizzando una versione elettronica del modulo prestampato, e possono anche essere utilizzate per visualizzare e modificare il contenuto della base di dati.

Gli strumenti di generazione di maschere sono sempre stati uno degli strumenti di supporto più comuni tra quelli offerti dai DBMS commerciali. Access, al posto della semplice e tradizionale interfaccia a caratteri, sfrutta le prerogative dell'ambiente Microsoft Windows e permette la realizzazione di maschere grafiche. Lo strumento di definizione delle maschere permette di definire la posizione e il significato di ciascun componente della maschera, dando al progettista un'ampia libertà di personalizzazione a livello di fonti di caratteri, colori, disegni e simboli grafici.

Per creare una nuova maschera, bisogna selezionare il componente Maschere dalla finestra di base del database e premere il pulsante NUOVO. Il sistema chiede a questo punto se si vuole utilizzare lo strumento di progetto di base, o se si vuole utilizzare uno strumento per la creazione guidata; mediante la creazione guidata si può creare in pochi istanti una semplice maschera associata a una tabella. Access offre in questo contesto la possibilità di scegliere tra vari strumenti per la creazione di maschere; essi differiscono in base allo schema di maschera che producono (per cui uno strumento produce una maschera con tutti gli attributi della tabella in sequenza uno per riga, un altro produce una rappresentazione tabellare arricchita da spazi e colori, ecc.). Se non si utilizzano i servizi di uno strumento di creazione guidata, ci si trova davanti a una pagina bianca in cui si possono inserire man mano i componenti della maschera. La stessa interfaccia può essere usata per esplorare e modificare la struttura di una maschera preesistente, selezionando la maschera e premendo il pulsante STRUTTURA.

Una maschera è composta da diversi elementi, che devono essere definiti uno a uno. L'elemento di base di una maschera è il *controllo*, un oggetto cui corrisponde un'area rettangolare dello schermo e che può essere di tre tipi: *associato*, *non associato* e *calcolato*. Un controllo *associato* è un elemento della maschera cui viene associato un attributo di una tabella. Questo elemento rappresenterà il valore dell'attributo per la particolare tupla considerata. La rappresentazione del valore di un attributo in genere richiede di riprodurre una sequenza di caratteri; per attributi di tipo OLE, la rappresentazione viene affidata all'applicazione a esso associata. Un controllo *non associato* invece contiene un valore fisso, che non cambia al variare delle tuple; in questo modo si rappresentano tipicamente le etichette delle maschere (che anche in questo caso possono essere sequenze di caratteri, o altri oggetti di tipo generico; se si vuole ad esempio inserire un logo nella maschera, bisognerà associare a un controllo non associato la figura che rappresenta il logo). Infine, i controlli *calcolati* permettono di visualizzare i risultati di espressioni valutate su combinazioni arbitrarie di parametri costanti e valori degli attributi delle tuple. I controlli *calcolati* non possono essere usati per l'immissione o la modifica dei valori degli attributi.

Con una maschera è anche possibile inserire nuove tuple, riportando diret-

tamente i valori degli attributi nei controlli *associati*. Con le maschere è anche possibile interrogare la base di dati, utilizzando un semplice comando di ricerca (opzione Trova del menu Modifica). Per modificare il valore di un attributo, si seleziona la tupla e si apporta la modifica direttamente sulla casella che presenta il valore dell'attributo. Le modifiche vengono rese persistenti spostandosi su una diversa tupla, tipicamente premendo i tasti per l'avanzamento o l'arretramento di una pagina che permettono di scandire le tuple della tabella. La figura A.19 mostra una maschera sulle tuple di PERSONA, estese con un attributo che contiene una foto.

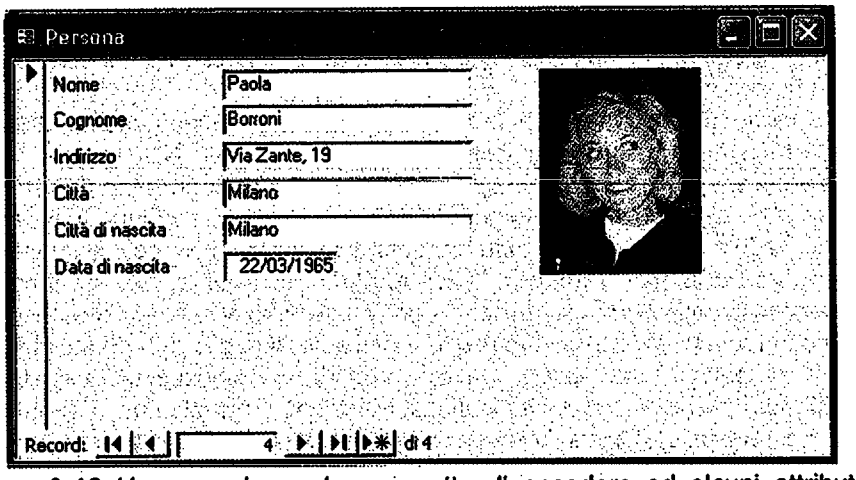


Figura A.19 Una maschera che permette di accedere ad alcuni attributi di PERSONA

Le maschere sono in genere costruite su una singola tabella. È possibile definire delle maschere che ne contengono altre al loro interno, specificando dei legami tra i valori visualizzati nelle maschere. In questo modo si possono ad esempio definire delle maschere per la gestione di tabelle con relazioni uno a molti, in cui la maschera interna rappresenta tutte le tuple della relazione di dettaglio associate all'elemento esterno (si pensi a una coppia di tabelle che descrivono gli ordini; una maschera può illustrare al livello più esterno i dati dell'ordine e in una sottomaschera può comparire la distinta degli ordinativi, appartenenti a due tabelle distinte).

Un *report* si definisce in modo analogo a una maschera, usando gli stessi strumenti e concetti. La differenza principale consiste nel fatto che un report ha normalmente l'obiettivo di fornire una descrizione del contenuto della base dati al livello globale. Per questo i report tipicamente contengono dei controlli *calcolati* che forniscono consuntivi, e non permettono la visione di tuple particolari. Un'altra differenza con le maschere è che in genere i report vengono stampati, invece di essere utilizzati in modo interattivo.

A.5 La definizione di macro

Le macro costituiscono un modo per specificare un insieme di azioni che il sistema deve compiere. In questo modo è possibile automatizzare l'esecuzione di un insieme di compiti. Mediante le macro si possono svolgere le seguenti azioni:

- Far interagire una maschera con altre maschere e con i report. Ad esempio, avendo una maschera che descrive i dati dei clienti e una che descrive gli ordini, è possibile aggiungere alla prima maschera un pulsante che attiva la seconda maschera visualizzando solo i dati degli ordini del cliente visualizzato. Si può anche aggiungere un pulsante che permette di stampare un report che elenca la situazione contabile nei confronti del cliente, con l'ammontare di merce rispettivamente ordinata, consegnata e pagata.
- Selezionare e raggruppare automaticamente le tuple. In una maschera si può inserire un pulsante che permette di selezionare immediatamente le tuple che rispettano particolari condizioni.
- Assegnare i valori agli attributi. Usando una macro, è possibile assegnare a un campo di una maschera un valore ottenuto da altri campi o altre tabelle della base di dati.
- Garantire l'accuratezza dei dati. Le macro sono molto utili per manipolare e validare i dati sulle maschere. Per esempio, si può definire una macro che reagisce a diversi valori di un attributo con diversi messaggi, e garantire in modo sofisticato che i dati inseriti siano corretti.
- Impostare le proprietà delle maschere, dei report e dei campi. Con le macro si possono automatizzare i cambiamenti di qualsiasi proprietà di questi oggetti. Ad esempio, è possibile rendere invisibile una maschera quando serve il suo contenuto ma non serve che questo venga visualizzato.
- Automatizzare i trasferimenti dei dati. Se si devono trasferire ripetutamente dati tra Access e altre applicazioni (sia in lettura sia in scrittura), si può automatizzare il compito.
- Creare un proprio ambiente di lavoro. Si può specificare una macro che apre tutto un insieme di tabelle, query, maschere e report tutte le volte che si apre un database, personalizzando eventualmente le barre degli strumenti.
- Specificare le reazioni a certi eventi. Per ogni campo di una maschera è possibile specificare quale macro debba essere eseguita in corrispondenza di ogni evento di accesso, selezione o modifica. Questa caratteristica costituisce la base per la definizione di comportamenti reattivi in Access, che però presentano diverse limitazioni rispetto a quanto è offerto dai trigger; infatti, tali comportamenti scattano in Access solo quando viene utilizzata una maschera particolare per la manipolazione della base di dati, mentre non scattano se la stessa operazione è eseguita direttamente in SQL o tramite un'altra maschera.

Per la definizione di macro bisogna selezionare la voce Macro dalla finestra di base del database e premere quindi il pulsante NUOVO. A questo punto si apre la finestra di progettazione di macro. La finestra contiene nella metà superiore una tabella con due colonne: Azione e Commento. La macro è composta da una

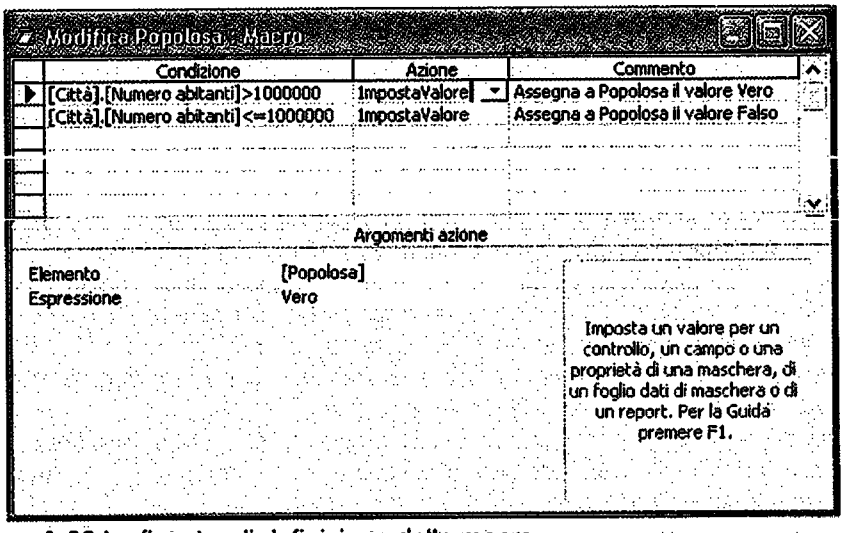


Figura A.20 La finestra di definizione delle macro

sequenza di azioni descritte nella colonna Azione, ciascuna su una diversa riga, cui può essere associata una breve descrizione. Nella metà inferiore della tabella compaiono un insieme di attributi che, per ogni singola azione, specificano i parametri dell'azione. È possibile specificare delle condizioni che devono essere verificate affinché un comando venga eseguito, aggiungendo la colonna Condizione tramite l'opzione presente nel menu Visualizza. Si può inoltre far uso di semplici strutture di controllo.

I comandi disponibili possono essere divisi in varie tipologie. Una prima tipologia è costituita dai comandi che permettono di chiedere l'esecuzione di altri servizi, che possono essere altre macro, query, generici comandi SQL, o anche applicazioni esterne. Un'altra tipologia sono i comandi che permettono di accedere ai dati, scandendo il contenuto di una tabella o di una maschera. Vi sono poi i comandi che permettono di manipolare il contenuto della base di dati. Infine, vi sono famiglie di comandi che permettono di trasferire dati tra Access e altre applicazioni, di modificare la dimensione delle finestre e di aprire delle finestre di dialogo con l'utente.

Una semplice macro è descritta in figura A.20. La macro è associata alle modifiche sull'attributo Numero abitanti di una città; la macro assegna all'attributo Popolosa il valore *Sì* se il numero di abitanti è superiore a un milione, altrimenti assegna all'attributo il valore *No*.

B

DB2 Universal Database

DB2 Universal Database (UDB)[1] appartiene a una storica famiglia di sistemi di gestione di basi di dati prodotti dalla IBM. Il capostipite di questa famiglia è SQL/DS, uno dei primi prodotti commerciali basati sul modello relazionale, reso disponibile dall'IBM agli inizi degli anni Ottanta. A sua volta, il sistema SQL/DS affonda le sue radici in System R, uno dei primi prototipi di DBMS relazionale sviluppato, negli anni Settanta, nei laboratori di ricerca dell'IBM di San José. Proprio nell'ambito dello sviluppo di questo prototipo è nato il linguaggio SQL che, come abbiamo detto nel capitolo 4, è presto diventato il linguaggio di riferimento per tutti i DBMS commerciali basati sul modello relazionale.

DB2 estende il modello relazionale con un ricco pacchetto di funzionalità avanzate, tra cui:

- un supporto alla gestione di dati multimediali quali testi, immagini, audio e video;
- funzionalità evolute di *business intelligence* per il supporto alle decisioni basate sulla costruzione di *data warehouse* e su operazioni OLAP (i sistemi di data warehouse sono illustrati nel secondo volume [5]);
- la possibilità di gestire dati secondo il paradigma a oggetti e dati semistrutturati basati sul linguaggio XML (anche questi argomenti sono trattati nel secondo volume);
- un supporto completo allo sviluppo di applicazioni distribuite e basate su Web che rende possibile l'accesso a basi di dati attraverso la rete Internet;
- un supporto sia a parallelismi a memoria condivisa (shared-memory), per basi di dati memorizzate su calcolatori a multiprocessori simmetrici (SMP), sia a parallelismi a memorie separate (shared-nothing), nei quali una base di dati viene partizionata tra diversi calcolatori connessi in rete (MPP).

La componente server di DB2 è disponibile su varie piattaforme software e hardware, in ambienti Windows e Unix (nelle loro varie versioni). Essa include la

[1] In tutto il capitolo chiameremo questo prodotto semplicemente DB2, sigla la quale è universalmente conosciuto.

componente client per l'accesso locale. Per l'accesso remoto al sistema è disponibile una componente client autonoma per la maggior parte dei sistemi operativi (che comunque non è necessario installare per applicazioni basate su JDBC livello 4). Client remoti possono comunicare con la componente server mediante i più diffusi protocolli di comunicazione (TCP/IP, NetBios, ecc.).

Con DB2 è possibile costruire componenti di federazioni distribuite ed eterogenee di basi di dati, usando un protocollo chiamato *Distributed Relational Database Architecture* (DRDA), adottato da diversi DBMS relazionali (le architetture distribuite sono anch'esse trattate nel secondo volume). Inoltre, DB2 fornisce un supporto per i principali standard di interfacce (quali ODBC, JDBC) e aderisce allo standard corrente di SQL.

Nel resto di questa appendice descriveremo le caratteristiche generali del sistema, prestando particolare attenzione agli strumenti di base e ad alcune sue funzionalità avanzate. Per ulteriori approfondimenti su questo sistema si rimanda il lettore al testo di Chamberlin [18], uno degli inventori di SQL.

B.1 Caratteristiche generali di DB2

B.1.1 Versioni del sistema

Sotto il termine DB2 si celano in realtà diversi prodotti tra cui: uno strumento per l'integrazione di basi di dati eterogenee (DB2 Information Integrator), uno strumento per lo sviluppo di data warehouse (DB2 Warehouse Manager), uno strumento per l'OLAP (DB2 OLAP Server), uno strumento per il data mining (DB2 Intelligent Miner) e uno strumento di *content management* per la gestione di dati non strutturati quali documenti e immagini (DB2 Content Manager).

Nel seguito ci occuperemo solo del *database server* di DB2, ovvero della componente del sistema che si occupa strettamente della gestione di basi di dati, e d'ora in poi utilizzeremo per semplicità la sigla DB2 per fare riferimento a questa componente.

DB2 è disponibile in cinque versioni, in grado di gestire architetture di complessità crescente.

- *Everyplace*: è una versione ridotta per dispositivi mobili (palmari, smart phone, portatili con risorse di calcolo limitate), disponibile sui diversi sistemi operativi per questa fascia di calcolatori (Palm OS, Symbian, Windows Mobile). Occupa solo 200 Kbyte di memoria e fornisce le funzionalità di base del sistema.
- *Personal* (PE): è una versione monoutente e monoprocessore, ma completa, disponibile in ambienti Windows e Linux. Consente la creazione e la manipolazione di basi di dati alle quali possono accedere più applicazioni, che però devono risiedere localmente. Questa versione può essere anche usata come client per l'accesso a una componente server remota.
- *Workgroup* (WSE): consente l'accesso condiviso a basi di dati locali da parte di utenti/applicazioni locali e remoti ed è disponibile in ambienti Windows,

Linux e Unix. È anche in grado di sfruttare le potenzialità di parallelismo di un calcolatore che contiene fino a quattro processori (SMP a quattro vie).

- *Express*: è una versione di facile installazione e semplificata rispetto alla WSE, che offre funzionalità di amministrazione automatica; è adatta allo sviluppo di applicazioni persistenti che richiedano una amministrazione minima della base di dati.
- *Enterprise*: è la versione completa del sistema che, oltre alle funzionalità delle altre versioni, consente la gestione di federazioni di basi di dati e il partizionamento di una base di dati tra diversi calcolatori connessi da una rete di comunicazione, ognuno dei quali può essere dotato di più processori (architetture SMP e MPP).

Il sistema è infine completamente integrato con i diversi ambienti di sviluppo commercializzati da IBM (VisualAge, WebSphere e Rational) che, insieme a DB2, offrono un supporto per l'intero ciclo di vita di applicazioni software realizzate con i più diffusi linguaggi di programmazione (C, C++, Java, ecc.).

B.1.2 Istanze e schemi di DB2

Su un medesimo calcolatore è possibile definire diverse *istanze*[2] indipendenti di server DB2, a ognuna delle quali viene assegnato un nome. Ogni istanza ha una propria configurazione e può gestire diverse basi di dati che restano di proprietà dell'istanza. È possibile in questa maniera adattare il sistema a specifiche necessità applicative; ad esempio, si può definire e configurare un'istanza DB2 per applicazioni operative e un'altra, con parametri di configurazione diversi, per applicazioni di supporto alle decisioni. Esiste comunque un'istanza predefinita, creata durante l'installazione di DB2, che si chiama semplicemente DB2. Le basi di dati di una istanza sono organizzate in *schemi* aventi un nome e costituiti da collezioni di tabelle. Lo schema di default nel quale vengono inserite nuove tabelle ha lo stesso nome dell'amministratore DB2.

I client DB2 possiedono una lista delle istanze e delle basi di dati DB2 alle quali possono accedere e sono dotati di strumenti per creare nuove istanze, nuovi schemi e nuove basi di dati. Per interagire con il server, essi devono prima accedere a una istanza di DB2 e successivamente stabilire una connessione con una base di dati di questa istanza. Un client può connettersi contemporaneamente a diverse basi di dati. Si possono poi inviare comandi DB2 sia a livello di istanza (ad esempio per creare una nuova base di dati) sia a livello di base di dati (tipicamente una istruzione SQL). Nelle istruzioni SQL si può far riferimento alla tabella di uno schema anteponendo al nome della tabella il nome dello schema, separati da un punto. Se il nome dello schema non è specificato, allora ci si riferisce allo schema di default.

[2] Il termine "istanza" qui usato denota una installazione di DB2 e quindi non ha niente a che vedere con il concetto di istanza di base di dati introdotto nel paragrafo 1.3.1

B.1.3 Interazione con DB2

Come in tutti sistemi di gestione di basi di dati moderni, l'accesso a una base di dati DB2 può avvenire secondo due modalità principali.

- In maniera interattiva, nella quale si inviano, tramite una opportuna interfaccia (comunemente detta *interfaccia utente*), comandi o istruzioni SQL che vengono immediatamente eseguiti dal sistema. Esistono sia una versione semplice dell'interfaccia, puramente testuale, che una versione grafica evoluta per ambienti Windows. Per l'amministrazione del sistema (creazione di istanze, basi di dati, schemi e gestione di autorizzazioni, prestazioni ecc.) si ricorre nella maggior parte dei casi a questa modalità di interazione.
- Tramite lo sviluppo di programmi in linguaggi di programmazione tradizionali, nei quali vengono immerse istruzioni SQL. È possibile sviluppare, secondo questa modalità, sia programmi statici, nei quali la struttura delle istruzioni SQL è nota a tempo di compilazione, sia programmi dinamici (vedi capitolo 6), nei quali le istruzioni SQL vengono generate a tempo di esecuzione.

A differenza di altri sistemi di gestione di basi di dati, DB2 non offre un linguaggio 4GL, cioè un linguaggio di sviluppo ad-hoc. Se da un lato questa scelta comporta la necessità di dover disporre, oltre che di DB2, anche di opportuni compilatori per lo sviluppo di applicazioni per basi di dati, dall'altra favorisce la realizzazione di software facilmente portabile da un sistema a un altro.

Nel prossimo paragrafo descriveremo, con maggior dettaglio, come si gestisce una base di dati DB2, secondo le suddette modalità.

B.2 Gestione di una base di dati con DB2

B.2.1 Strumenti per la gestione interattiva

La maniera più semplice per interagire con DB2 è attraverso la sua interfaccia utente. Questo avviene tipicamente in una architettura client-server classica nella quale una base di dati che risiede su un server (ad esempio su una macchina Unix) viene acceduta dall'utente tramite un client locale (sullo stesso computer sul quale risiede la base di dati) o remoto (ad esempio su un personal computer in ambiente Windows). L'interfaccia utente mette a disposizione diversi strumenti interattivi che sono classificati come segue.

- Strumenti di gestione generale (consentono l'amministrazione di una base di dati mediante un'interfaccia grafica di facile uso):
  - Il *Centro di controllo* è lo strumento principale e tipicamente si invoca all'avvio del sistema. Fornisce una interfaccia per le operazioni di amministrazione più importanti quali la creazione di basi di dati e di tabelle. Dal centro di controllo è possibile poi invocare tutti gli altri strumenti.

- Il *Giornale* tiene traccia di tutti le note informative prodotte dal sistema, inclusi i diagnostici e i messaggi di errore. È utile per verificare la presenza di un problema nel sistema.
- Il *Centro attività* consente di pianificare attività che il sistema esegue successivamente in maniera automatica, ad esempio dei backup periodici.
- Il *Centro di replica* permette la creazione e la gestione di copie di una base di dati, che vengono tenute aggiornate in maniera automatica dal sistema.

• Strumenti riga comandi (consentono di inviare istruzioni SQL o comandi di sistema).

- Il *Command Line Processor* (CLP) è un ambiente puramente testuale ed è disponibile su tutte le piattaforme.
- L'*Editor di comandi* è invece dotato di un'interfaccia grafica ed è disponibile per sistemi operativi Windows.

• Strumenti di sviluppo (consentono di realizzare piccole procedure per una base di dati).

- Il *Centro di sviluppo* consente la definizione e il testing di oggetti applicativi quali procedure (le stored procedure, descritte nel paragrafo 6.1), tipi utente e funzioni utente (che verranno descritte nei paragrafi B.3.3 e B.3.4).
- Gli *Strumenti di distribuzione progetto* permettono di esportare su altre basi di dati, eventualmente remote, oggetti applicativi definiti nel Centro di sviluppo.

• Strumenti di informazione:

- Il *Centro di informazioni* fornisce l'accesso rapido a tutti i manuali DB2.
- La *Verifica aggiornamenti di DB2* serve a tenere aggiornato il sistema.

• Strumenti di controllo (per il monitoraggio del sistema):

- Il *Centro di controllo stato* consente di individuare potenziali fonti di errore, ad esempio l'eccessivo uso di memoria principale, quando certi indicatori superano soglie di "guardia".
- L'*Analizzatore di eventi* memorizza tutti gli eventi che si verificano sul sistema e consente di monitorare il corretto funzionamento del sistema.
- Il *Controllo attività* fornisce informazioni relative alle prestazioni del sistema e all'uso delle risorse.
- Il *Memory Visualizer* genera grafici relativi all'uso della memoria da parte del sistema.
- L'*Indoubt Transaction Manager* consente di analizzare transazioni distribuite che si trovano in stati di incertezza (indoubt), o per cui è stata avviata la fase di chiusura ma per le quali non è stato effettuato il commit o il rollback (le transazioni sono descritte nel paragrafo 5.4 e approfondite nel secondo volume [5]).

• Strumenti di configurazione:

- L'*Assistente di configurazione* consente di configurare facilmente il sistema per la connessione a basi di dati remote.
- I *Primi passi* che fornisce un tutorial sull'uso del sistema.

- La *Registrazione Visual Studio Add-in* che permette l'installazione di un plug-in in Visual Studio che consente l'invocazione diretta di DB2 da questo ambiente.

• Altri strumenti, di vario tipo.
  - Il *Centro licenze* fornisce informazioni sul tipo di licenza installata sul computer.
  - Il *Centro di gestione satelliti* consente la gestione di satelliti, ovvero di gruppi di diversi server DB2 con configurazioni simili sui quali vengono eseguite le medesime applicazioni.
  - L'*SQL assist* offre un supporto in linea per la scrittura di istruzioni SQL.

Nel seguito vengono illustrati con maggior dettaglio gli ambienti che consentono la gestione di base del sistema.

Command Line Processor La versione spartana ma completa dell'interfaccia utente di DB2 è costituita da un ambiente testuale chiamato *Command Line Processor* (CLP) ed è disponibile su tutte le piattaforme. Il CLP si può invocare a livello di sistema operativo con il comando db2. Un esempio di interazione con il CLP è riportato in figura B.1. Il prompt db2 => indica che il sistema è pronto a ricevere comandi. Una volta entrati nell'ambiente si deve avviare il sistema con il

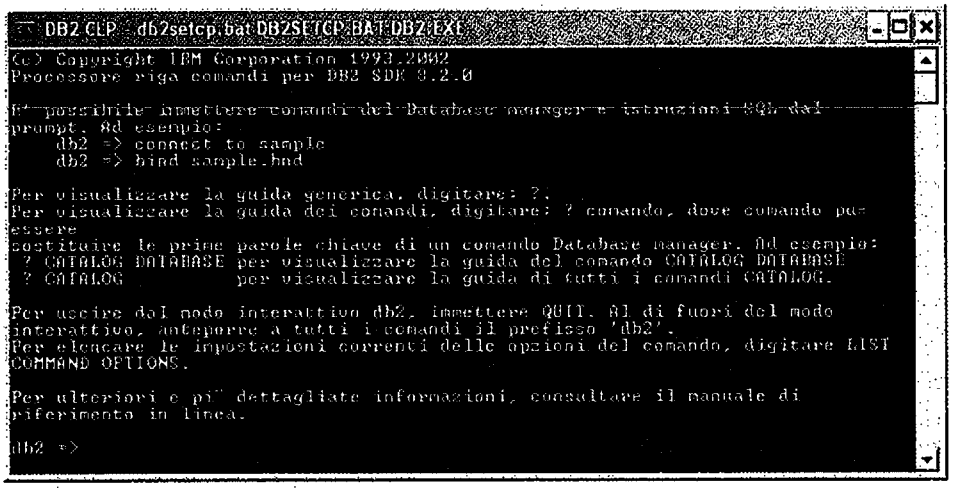


Figura B.1 Il Command Line Processor di DB2

comando `db2start` e poi si possono inviare, oltre alle istruzioni SQL, una serie di comandi per la gestione completa del sistema. Il comando inviato viene eseguito immediatamente e il risultato viene mostrato sul video. Se i comandi sono più lunghi di una riga, bisogna segnalare la continuazione sulla riga successiva con il carattere di *backslash* (\). Tutti i comandi possono essere inviati direttamente da sistema operativo premettendo il prefisso db2.

Centro di controllo Permette di amministrare una base di dati mediante un'interfaccia grafica che offre la possibilità di specificare operazioni di definizione di tabelle e di vincoli di integrità, controllo delle autorizzazioni, backup, ripristini, ecc. Come si può vedere in figura B.2, l'interfaccia del Centro di controllo ha tre finestre principali. In quella di sinistra vengono presentati, secondo una organizzazione gerarchica, gli oggetti DB2 (istanze, basi di dati, tabelle, ecc.) che l'utente ha a disposizione. Questo oggetti possono essere dichiarati esplicitamente dall'utente, oppure si può richiedere al Centro di controllo di cercare su rete tutti gli oggetti DB2 ai quali è possibile accedere.

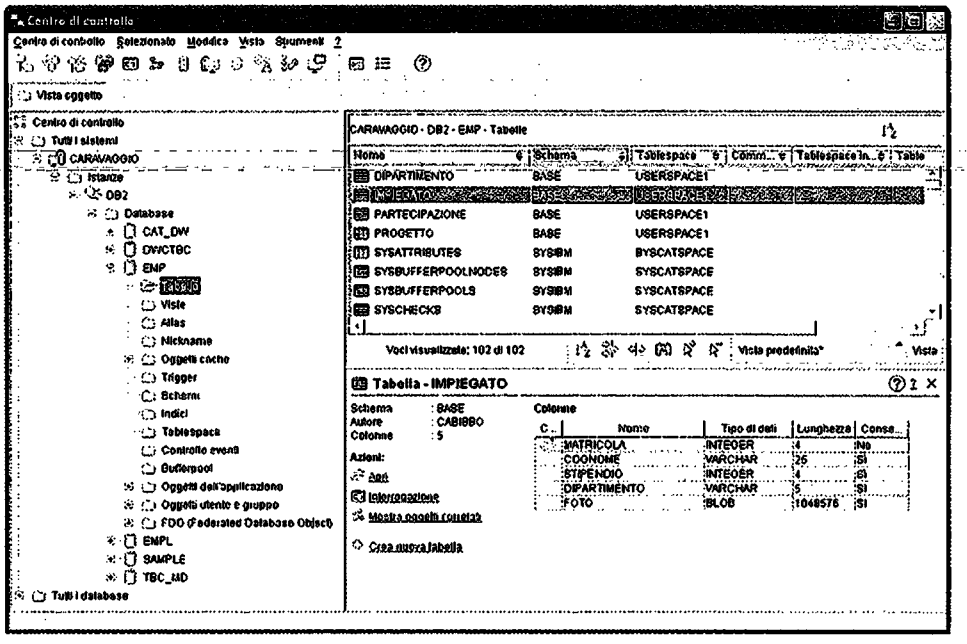


Figura B.2 Il Centro di controllo di DB2

Nel nostro caso, si può osservare che è disponibile un sistema con oggetti DB2 chiamato CARAVAGGIO. Le informazioni relative a questo sistema sono state espanse, come indicato dal segno meno davanti alla relativa icona. Su questo sistema risiedono due istanze DB2: una è quella di default, l'altra si chiama MyDB2. La prima contiene diverse basi di dati, tra cui una chiamata EMP. Per questa base di dati vengono mostrate tutte le sue componenti, mentre le informazioni relative alle altre non sono state espanse, come indicato dal segno '+' davanti alle relative icone. La finestra in alto a destra contiene informazioni dell'oggetto selezionato nella finestra di sinistra. Nel nostro caso, sono mostrate tutte le tabelle della base di dati EMP. Questa base di dati contiene due schemi: lo schema BASE e lo schema SYSIBM. Le tabelle dello schema SYSIBM sono tabelle di sistema. Nella finestra in basso a destra vengono infine riportate informazioni sulla tabella selezionata. Nel nostro esempio sono mostrati i dettagli della tabella IMPIEGATO.

Se si clicca con il bottone di destra del mouse su un qualunque oggetto sullo schermo, compare un menu che consente di effettuare una serie di azioni sull'oggetto puntato. Ad esempio, puntando l'icona `Tabelle` di una base di dati, è possibile creare una nuova tabella. Il sistema guida l'utente nell'esecuzione di queste operazioni.

Editor comandi Questo strumento consente di digitare ed eseguire istruzioni SQL e di comporre *script*, ovvero sequenze di istruzioni SQL da eseguire eventualmente in momenti prestabiliti. Può essere invocato dal Centro di controllo cliccando sull'icona `Interrogazione` della finestra in basso a destra. L'Editor comandi è costituito da tre ambienti tra loro integrati: *Comandi*, *Risultati* e *Plan di accesso*; per passare da un ambiente all'altro è sufficiente cliccare sul corrispondente nome. Nel primo è possibile digitare singole istruzioni SQL, script o comandi DB2 di amministrazione. In figura B.3 viene mostrato un esempio di interrogazione SQL formulata nell'Editor comandi di DB2 (invocato dal Centro di controllo).

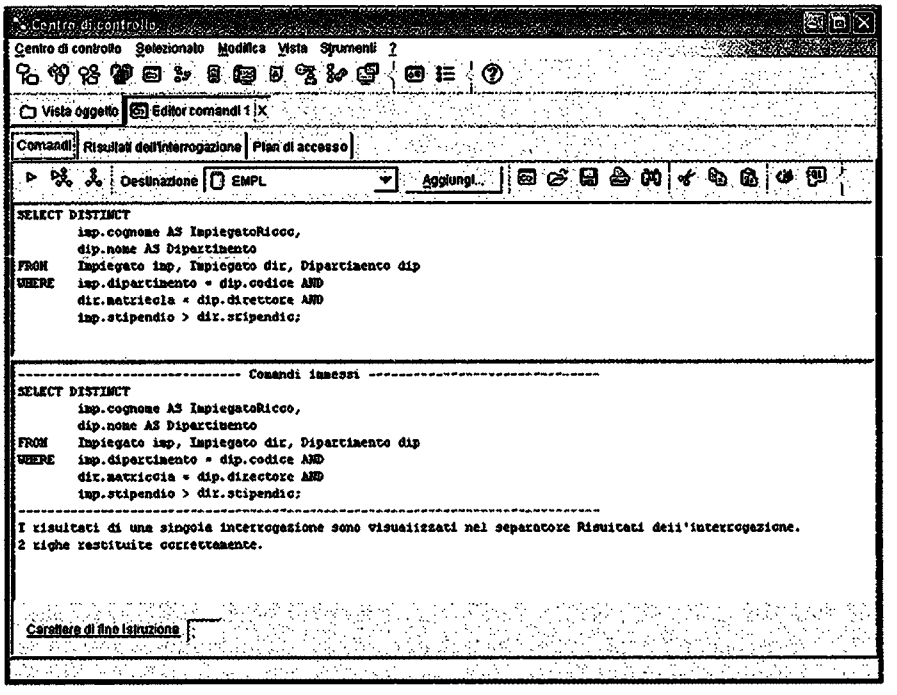


Figura B.3 Una interrogazione SQL fatta con l'Editor comandi di DB2

Si tratta di una interrogazione che cerca gli impiegati che guadagnano più del rispettivo direttore, in uno schema contenente le tabelle:

IMPIEGATO(Matricola, Cognome, Dipartimento, Stipendio)

DIPARTIMENTO(Codice, Nome, Sede, Direttore).

Per eseguire i comandi digitati è sufficiente cliccare sull'icona di esecuzione, costituita da un triangolo: un messaggio relativo all'esecuzione dell'istruzione (completamento con successo o eventuale diagnostica di errore) viene riportato nella finestra in basso, mentre il risultato viene visualizzato automaticamente nell'ambiente *Risultati dell'interrogazione*. La figura B.4 mostra un esempio del contenuto di questo ambiente dopo aver eseguito l'interrogazione in figura B.3.

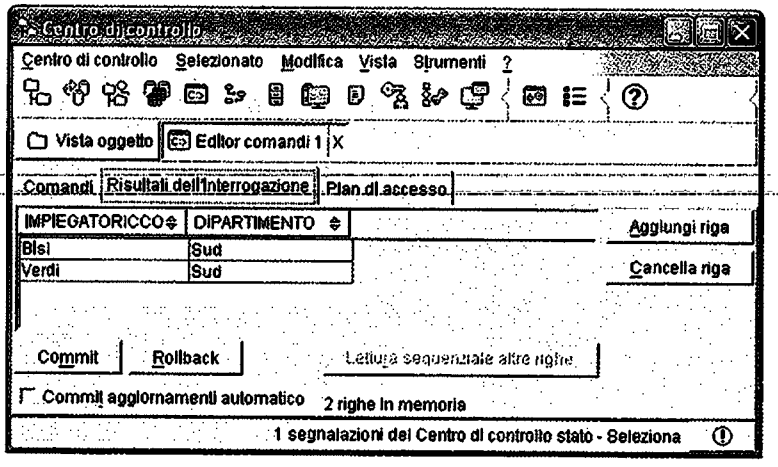


Figura B.4 Il risultato di una interrogazione SQL nell'Editor comandi di DB2

Passando dal primo al secondo ambiente è quindi possibile interagire in maniera interattiva con una base di dati. I comandi (o gli script) digitati nel primo ambiente possono essere salvati su un file per successive esecuzioni. Il terzo ambiente visualizza i piani di accesso creati dall'ottimizzatore di DB2 per l'esecuzione di istruzioni SQL digitate nel primo ambiente. I piani di accesso hanno la forma di alberi: le foglie degli alberi corrispondono alle tabelle coinvolte nell'istruzione SQL e i nodi interi a operazioni effettuate dal sistema; ad esempio, la scansione di una tabella, un certo tipo di join tra due tabelle, l'ordinamento di un risultato intermedio. La creazione di questi grafi deve essere richiesta esplicitamente in fase di esecuzione al sistema cliccando sull'icona che raffigura un albero. In figura B.5 viene mostrato una porzione del piano di accesso dell'interrogazione di figura B.3.

Nella finestra sono indicate delle operazioni di scansione (scan) che precedono le operazioni di join tra le tabelle DIPARTIMENTO ed IMPIEGATO e le operazioni che seguono (sort e scan) per la preparazione del risultato finale. Il valore indicato nei nodi fornisce una stima del costo previsto per l'operazione. Cliccando sui singoli nodi è possibile visualizzare altri dati relativi alla operazione corrispondente, quali stime più dettagliate sui costi di esecuzione e sulla cardinalità dei risultati. L'analisi dei piani di accesso consente di ottimizzare interrogazioni, ad esempio introducendo opportunamente degli indici per evitare una operazione di

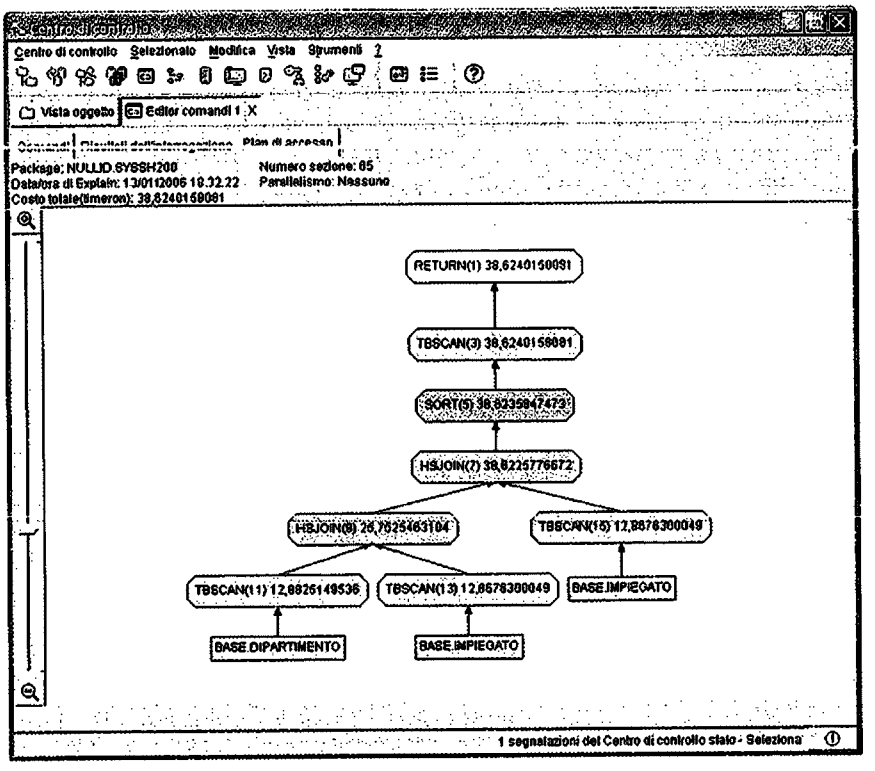


Figura B.5 Il piano di accesso generato da DB2 per una interrogazione SQL

ordinamento durante la sua esecuzione. Questi aspetti sono illustrati in dettaglio nel secondo volume [5].

Giornale Costituisce un utile strumento di verifica perché mantiene una traccia di tutte le operazioni svolte su una base di dati. In particolare, è possibile visualizzare con questo strumento l'esecuzione di attività pianificate (pannello `Cronologia attività`), le operazioni di backup/ripristino (pannello `Cronologia database`), tutti i messaggi generati dal sistema in seguito all'esecuzione delle varie operazioni svolte (pannello `Messaggi`) e i diagnostici di errore prodotti dal sistema (pannello `Registrazione notifiche`).

Centro di controllo stato Si tratta di un altro utile strumento che consente di analizzare lo stato del sistema e risolvere preventivamente potenziali problemi. In particolare è in grado di segnalare situazioni "limite", che non costituiscono ancora veri e propri errori, ma possono indicare possibili malfunzionamenti del sistema. Questi avvisi vengono generati quando alcuni indicatori di stato (ad esempio la dimensioni di memoria principale occupata) superano dei valori di soglia pre-

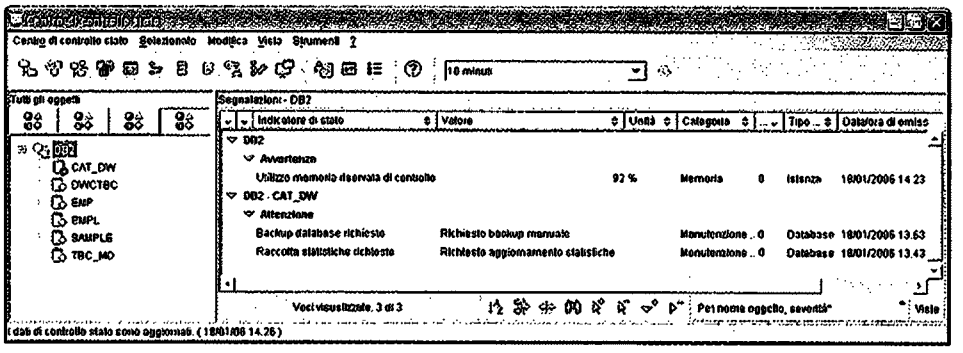


Figura B.6 Il centro di controllo stato di DB2

definiti (detti *threshold*). Un esempio di contenuto del Centro di controllo viene mostrato di figura B.6.

In questo esempio viene segnalato un eccessivo utilizzo della memoria riservata al controllo (92% del valore massimo) e la necessità di eseguire dei backup. Selezionando un avviso, vengono suggerite delle azioni da intraprendere per risolvere la segnalazione.

Centro informazioni Fornisce un ricco insieme di informazioni sul sistema DB2. Tramite questo strumento è possibile accedere a spiegazioni su come effettuare certe operazioni (ad esempio, "creazione di una tabella" o "backup di una base di dati"), a tutorial, ai vari manuali disponibili in linea, a chiarimenti sulla messaggistica di DB2 e sui codici di errore, e a una lista di programmi di esempio che illustrano le varie funzionalità di DB2.

B.2.2 Applicazioni

DB2 mette a disposizione diversi strumenti per lo sviluppo di applicazioni per basi di dati, nei quali le istruzioni SQL di interazione con la base di dati vengono usate insieme a istruzioni di linguaggi di programmazione tradizionali, detti linguaggi *host*, come C, C++, Java, FORTRAN e COBOL. Come descritto nel capitolo 6, esistono due modalità principali: l'SQL embedded (statico e dinamico) e l'uso delle Call Level Interface (CLI) quali ODBC e JDBC. Ricordiamo che nell'SQL statico i nomi delle relazioni e degli attributi coinvolti nelle istruzioni SQL sono fissati a priori; l'unica parte delle istruzioni che può rimanere non nota a tempo di compilazione sono gli specifici valori da ricercare o da aggiornare. Nell'SQL dinamico, invece, le istruzioni SQL sono generate a tempo di esecuzione; non è quindi necessario specificare relazioni e attributi coinvolti in una istruzione SQL prima della sua esecuzione.

Nel seguito parleremo prevalentemente dell'SQL statico, mentre accenneremo agli strumenti offerti da DB2 per realizzare programmi secondo le altre moda-

lità. Per rendere la trattazione più autocontenuta, vengono ripetute alcune nozioni già presentate nel capitolo 6.

SQL statico Un esempio completo di un programma SQL statico immerso in C per DB2 è riportato in figura B.7. Questo programma accede a una base di dati contenente le tabelle IMPIEGATO e DIPARTIMENTO citate nel paragrafo precedente, legge il nome di una località da input e stampa nome e stipendio degli impiegati dei dipartimenti che si trovano in tale località, ordinati per dipartimento.

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <sqlenv.h>
exec sql include sqlca;
void main() {
   char  CurrDept[10];              /* Dichiarazione variabile programma */
   exec sql begin declare section;      /* Dichiarazione variabili host */
      char  CognomeImp[10];                        /* Cognome Impiegato */
      char  NomeDip[10];                           /* Nome Dipartimento */
      char  SedeDip[15];                           /* Sede Dipartimento */
      long  StipendioImp;                        /* Stipendio Impiegato */
      char  msgbuffer[500];         /* Buffer per messaggi di errore DB2 */
   exec sql end declare section;
   exec sql declare C1 cursor for              /* Dichiarazione cursore */
        select distinct imp.cognome, imp.stipendio, dip.nome
        from     impiegato imp, dipartimento dip
        where    imp.departmento = dip.codice and
                 dip.sede = :SedeDip
        order by dip.nome;
   exec sql whenever sqlerror go to PrintError;
   exec sql connect to EMP;         /* Connessione alla basi di dati EMP */
   printf("Digita una localita':"); scanf("%s", SedeDip);
   exec sql open C1;                                /* Apre il cursore C1 */
   exec sql fetch C1 into :CognomeImp, :StipendioImp, :NomeDip;
   if (sqlca.sqlcode==100) printf("Nessun dipartimento trovato.\n");
   while (sqlca.sqlcode == 0) {
      if (strcmp(NomeDip,CurrDept)!=0) {
         printf("\nDipartimento: %s\nImpiegato\tStipendio\n",NomeDip);
         strcpy(CurrDept,NomeDip);
      }              /* end if */
      printf("%s\t%d\n", CognomeImp, StipendioImp);
      exec sql fetch C1 into :CognomeImp, :StipendioImp, :NomeDip;
   }               /* end while */
   exec sql close C1;                            /* Chiude il cursore C1 */
   exec sql connect reset;                     /* Rilascia la connessione */
   return;
PrintError:                        /* Trova e stampa un messaggio di errore */
   sqlaintp(msgbuffer, 500, 70, &sqlca);
   printf("Errore DB2 inatteso: %s\n", msgbuffer);
   return;
}                /*  end of main */
```

Figura B.7 Un programma SQL immerso in C per DB2

Si può innanzitutto osservare che il prefisso utilizzato in DB2 per segnalare una istruzione SQL è `exec sql`. Il programma inizia con una serie di direttive

di compilazione che includono la libreria di sistema DB2 (chiamata `sqlenv.h`) e il record `sqlca` nel quale vengono memorizzate automaticamente, di volta in volta, informazioni sullo stato del sistema. In particolare, il campo `sqlcode` di `sqlca` contiene un numero intero che codifica il comportamento dell'ultima istruzione SQL eseguita.

Fra le dichiarazioni delle variabili, notiamo che ve ne sono alcune racchiuse in una sezione delimitata dalle parole chiave `declare section`. Queste variabili sono dette *host* e sono quelle variabili del programma che possono essere usate nelle istruzioni SQL: tali variabili realizzano l'interfaccia tra programma e base di dati. Si osservi che quando una variabile host è usata in SQL, è preceduta da un prefisso (`:`); questo prefisso serve a distinguere le variabili dai nomi di attributo. L'istruzione `whenever` permette di stabilire il comportamento del programma al seguito dell'esecuzione di ogni istruzione SQL non andata a buon fine. L'istruzione non è quindi associata a una specifica operazione SQL, ma all'intero programma. Tre opzioni sono possibili per questa istruzione: `not found`, `sqlerror` e `sqlwarning`. Nel primo caso si può specificare cosa fare quando il risultato di una interrogazione è vuoto (`sqlcode` = 100); nel secondo, quando si è verificato un errore (`sqlcode` < 0); nell'ultimo, quando si è verificato un *warning*, ovvero la segnalazione di un evento imprevisto ma non grave (`sqlcode` > 0). Nel nostro programma è stato stabilito che, quando si verifica un errore, il controllo passi a una routine che carica in un buffer il messaggio di errore generato, lo stampa e poi termina il programma. Nulla è stato specificato per risultati nulli e warning.

Prima di interagire con una base di dati, va stabilita una connessione con essa tramite l'istruzione `connect to`. Successivamente, si possono eseguire liberamente istruzioni SQL i cui effetti sulla base di dati divengono però permanenti solo dopo l'esecuzione dell'istruzione `commit`. (Il programma di esempio non prevede istruzioni di aggiornamento della base di dati e quindi non fa uso di tale istruzione). Per eseguire le operazioni previste, il programma fa uso della tecnica del cursore presentata nel paragrafo 6.3.1. Viene inizialmente definito un cursore su una interrogazione SQL che contiene la variabile host `SedeDip`. Successivamente, dopo aver memorizzato in questa variabile un dato letto da input, il cursore viene aperto, l'interrogazione eseguita e, tramite l'istruzione `fetch`, il risultato viene copiato, una tupla alla volta, in variabili di host per poter essere visualizzato. Questa operazione viene coordinata esaminando il valore assunto dalla variabile di stato `sqlcode`. In particolare, viene visualizzato un messaggio quando il risultato è vuoto (`sqlcode` pari a 100) e viene ripetuta l'istruzione di copia e stampa finché ci sono ancora tuple da visualizzare (`sqlcode` pari a 0). Il cursore viene quindi chiuso e la connessione rilasciata.

Per creare una applicazione eseguibile per DB2, il programma va prima precompilato inviando il comando `prep <nome file>` nel CLP o nell'Editor comandi. Il risultato della precompilazione va poi compilato e collegato alle librerie usate, secondo le modalità adottate per un qualunque programma scritto nel linguaggio host scelto.

SQL dinamico Nell'SQL dinamico vengono messe a disposizione speciali istruzioni per compiere le seguenti operazioni principali.

- preparazione di una istruzione SQL con invocazione dell'ottimizzatore DB2 che crea il relativo piano di accesso;
- descrizione del risultato di una istruzione SQL con specifica del numero e del tipo degli attributi;
- esecuzione di una istruzione SQL precedentemente preparata, con assegnamento di valori a eventuali variabili usate nell'istruzione;
- caricamento del risultato, una tupla alla volta, in variabili di programma per il loro successivo uso.

Queste operazioni possono essere immerse in linguaggi di programmazione secondo le medesime modalità dell'SQL statico (in particolare mediante l'uso del prefisso `exec sql`).

La fase di preparazione di una istruzione SQL si realizza con il comando `prepare <nome> from <variabile>`, dove la variabile è di tipo stringa e tipicamente memorizza istruzioni SQL. Questa istruzione SQL può contenere dei punti interrogativi che indicano la presenza di parametri che verranno passati in fase di esecuzione. Nel caso di interrogazioni, la gestione del risultato si realizza con il meccanismo del cursore, come avviene con l'SQL statico.

La fase di descrizione si fonda sull'uso di un *descrittore*, ovvero di una struttura di dati che descrive tipo, lunghezza e nome di un numero variabile di attributi del risultato di un'interrogazione. Il sistema DB2 mette a disposizione a questo scopo il descrittore predefinito `sqlda`, che è un record contenente un numero variabile di campi di tipo `sqlvar`, uno per ogni attributo da descrivere. Il numero di questi campi di tipo viene memorizzato nel campo `sqld` di `sqlda`. I campi `sqlvar` sono a loro volta dei record contenenti, tra l'altro, un campo `sqltype`, che codifica il tipo dell'attributo, un campo `sqllen`, che memorizza la lunghezza dell'attributo, e un campo `sqlname`, che memorizza il nome dell'attributo. Nella fase di preparazione, si può indicare l'uso di questo descrittore con la seguente sintassi: `prepare <nome> into sqlda from <variabile>`. Dopo l'esecuzione di questa istruzione, in `sqlda` viene caricata la descrizione dell'istruzione memorizzata nella variabile specificata. Mediante questa tecnica, è possibile implementare per esempio una interfaccia utente personalizzata, in grado di accettare ed eseguire istruzioni SQL arbitrarie.

Per la fase di esecuzione e caricamento si possono utilizzare le medesime istruzioni viste per l'SQL statico, compreso l'uso di cursori. Accedendo alle informazioni contenute nel descrittore è possibile generare opportunamente la stampa dei risultati di un'interrogazione.

Call Level Interface DB2 offre delle interfacce CLI (Call Level Interface) basate sugli standard ODBC (Open DataBase Connectivity) e JDBC (Java DataBase Connectivity), illustrate nel paragrafo 6.4.

Ricordiamo che l'interfaccia basata su ODBC è disponibile per vari linguaggi di programmazione (escluso Java). Tale interfaccia mette a disposizione una serie

di funzioni che possono essere direttamente invocate dai programmi per accedere a una base di dati. Ad esempio la funzione `sqlconnect()` consente di connettersi a una base di dati DB2, la funzione `sqlprepare()` prepara una istruzione SQL alla sua esecuzione, la funzione `sqlexecute()` esegue una istruzione SQL precedentemente preparata e infine la funzione `sqlfetch()` carica una tupla in variabili host del programma.

L'interfaccia JDBC si basa sullo stesso principio della ODBC, ma è dedicata allo sviluppo di programmi in linguaggio Java ed è quindi orientata agli oggetti. In particolare, questa interfaccia è dotata del metodo `executequery(String)` che prende in ingresso una stringa contenente una istruzione SQL e restituisce un oggetto della classe predefinita `ResultSet`, costituito da un insieme di tuple. Questa classe possiede una serie di metodi che permettono di manipolare l'insieme di tuple contenute negli oggetti della classe. Oltre che applicazioni standard Java, questa interfaccia permette di sviluppare *applets*, ovvero programmi che possono essere caricati ed eseguiti da un browser Web abilitato allo scopo. In questa maniera, basi di dati DB2 possono essere accedute da una qualunque computer connesso a Internet, senza bisogno di installare su di esso la componente client del sistema DB2.

Il grosso vantaggio dei programmi che usano l'interfaccia CLI è che non hanno bisogno di essere precompilati; inoltre, essi possono essere utilizzati su qualunque DBMS che fornisce un supporto per gli standard citati.

B.3 Funzionalità avanzate di DB2

Come accennato all'inizio del capitolo, DB2 offre tutta una serie di funzionalità avanzate. Pur non essendo tutte standardizzate, queste funzionalità ci danno una interessante indicazione delle caratteristiche dei sistemi di gestione di basi di dati reali. Ne citiamo alcune.

B.3.1 Dati complessi

DB2 mette a disposizione tre tipi di dati predefiniti che possono essere usati per memorizzare in una tabella dati complessi (ad esempio documenti o immagini), che vengono chiamati genericamente LOB (Large OBjects).

- *Blob (Binary Large Object)*: rappresenta un dato in formato binario, grande fino a due gigabyte. I dati di tipo Blob non possono essere assegnati o confrontati con dati di altro tipo.
- *Clob (Character Large Object)*: rappresenta un dato composto da una sequenza di caratteri di un byte, grande fino a due gigabyte. I dati di tipo Clob possono essere confrontati con dati di tipo stringa (Char e Varchar).
- *Dbclob (Double-Byte Character Large Object)*: rappresenta un dato composto da una sequenza di caratteri di due byte, grande fino a due gigabyte. I dati di tipo Dbclob possono essere usati solo su basi di dati con una configurazione apposita.

Una possibile definizione di tabella che contiene dati di tipo LOB è la seguente.

```
create table Impiegato (
   Codice          integer not null unique,
   Nome            varchar(20),
   Stipendio       decimal(7,3),
   DataAssunzione  date,
   Foto            blob(5M) compact,
   Curriculum      clob(500K)
)
```

In questa tabella la colonna Foto conterrà immagini e la colonna Curriculum testi. La dimensione specificata tra parentesi indica un valore massimo (si possono usare i suffissi K, M, e G per indicare Kilobyte, Megabyte e Gigabyte). L'opzione `compact` specifica che il dato LOB dovrà occupare il minimo spazio sul disco, a costo di una minore efficienza nella sua gestione.

Esistono diverse limitazioni nell'uso di dati LOB in istruzioni SQL. In particolare non è possibile confrontare direttamente colonne di tipo LOB mediante operatori come =, >, < o `in`. È però possibile far uso dell'operatore `like`. Ad esempio, l'istruzione SQL:

```
select Codice, Nome
from Impiegato
where Curriculum like '%DBA%'
```

trova gli impiegati per i quali compare la stringa 'DBA' nel curriculum.

I dati LOB sono gestiti dal sistema in maniera da minimizzare i loro spostamenti da una locazione di memoria a un'altra. In particolare, i LOB possono essere manipolati nelle applicazioni per mezzo di opportune variabili dette *locatori*. I locatori rappresentano un LOB senza memorizzarlo fisicamente. Usando in maniera opportuna i locatori è possibile differire o addirittura evitare il caricamento di un LOB in un programma. Ad esempio, la copia di un LOB avviene semplicemente copiandone il locatore. Un locatore si definisce nella sezione di dichiarazione delle variabili host di un programma con il comando: `sql type is <tipo di LOB> <nome locatore>`. Successivamente, il locatore può essere usato come tutte le altre variabili host per gestire un LOB del tipo associato. È inoltre possibile manipolare locatori con funzioni speciali tra cui `posstr`, che trova la posizione della prima occorrenza di un pattern in un LOB, e `substr`, che restituisce la sottostringa di un LOB compresa tra le posizioni specificate.

B.3.2 Extender

Oltre ai tipi predefiniti di base, DB2 mette a disposizione, come prodotti ausiliari, degli *extender*, ovvero ulteriori tipi di dato, più specifici dei LOB, per la gestione di dati non tradizionali. A ciascun extender sono associate particolari funzioni ausiliarie per la manipolazione dei relativi tipi di dato. Generalmente queste funzioni possono essere utilizzate liberamente in comandi SQL.

Gli extender sono tipicamente sviluppati direttamente dalla IBM, ma esistono anche extender sviluppati da partner commerciali della IBM.

Tra gli extender DB2 più interessanti citiamo i seguenti.

- Gli *AIV (Audio, Image, Video) extenders* per la gestione di dati multimediali. Questi extender offrono funzionalità evolute di manipolazione di dati multimediali (tra cui la la riproduzione di suoni e video) e di ricerca avanzata (ad esempio ricerche basate su proprietà dei dati quali la presenza di un certo campione audio o immagine).
- Il *text extender* per la gestione di basi di dati documentali. Questo extender offre funzionalità tipiche dell'*information retrieval*, come l'indicizzazione di testi e la ricerca basata su parole chiavi.
- Lo *spatial extender* per la gestione di dati spaziali. Vengono offerti a tale riguardo diversi tipi di dato spaziale, che possono essere usati per modellare oggetti del mondo reale quali mappe, confini geografici, corsi di fiumi, ecc.
- L'*XML extender* per la gestione di dati in formato XML. Questo extender consente di memorizzare documenti XML in attributi di tabelle relazionali. Le funzioni messe a disposizione permettono di effettuare ricerche su interi documenti XML o su loro porzioni e di trasformare dati XML in basi di dati relazionali e viceversa.
- Il *Net search extender* per la gestione di ricerche testuali evolute. Offre un ricco insieme di funzioni di ricerca su testi e di indicizzazione di documenti memorizzati in una base di dati. Questi strumenti consentono la costruzione di motori di ricerca.

B.3.3 Tipi utente

Un tipo di dato utente (denominato *distinct* in DB2) è un tipo definito a partire dai tipi di dato di base e corrisponde al concetto di definizione di dominio utente introdotto nel capitolo 4. Ad esempio, la definizione dei tipi MONEY, IMAGE e TEXT può essere specificata con le seguenti istruzioni:

```
create distinct type Money as decimal(7,2)
                           with comparisons;
create distinct type Image as blob(100M);
create distinct type Text  as clob(500K) compact;
```

La clausola `as` specifica il tipo *sorgente* del tipo utente, mentre la clausola `with` serve a specificare che i relativi dati vanno confrontati come se fossero dati dei rispettivi tipi sorgente. Si possono usare come tipi sorgente solo tipi di dato DB2 predefiniti e non sono ammesse nidificazioni nelle definizioni.

Una volta definiti, i tipi utente possono essere liberamente usati nelle creazioni di tabelle. Ad esempio, la precedente definizione della tabella IMPIEGATO può essere riscritta come segue:

```
create table Impiegato (
   Codice          integer not null unique,
   Nome            varchar(20),
   Stipendio       money,
   DataAssunzione  date,
   Foto            image,
   Curriculum      text
)
```

Sugli attributi definiti su un tipo utente non è in generale possibile applicare le medesime operazioni che si possono applicare sui rispettivi tipi sorgente. Ad esempio, non è possibile sommare due dati di tipo Money. Si può però ovviare a questa limitazione con la definizione di opportune funzioni utente, come descritto nel seguito.

B.3.4 Funzioni utente

Le funzioni utente corrispondono alle procedure (o *stored procedure*) introdotte nel paragrafo 6.1. In DB2 possono essere dichiarate in maniera esplicita mediante l'istruzione `create function` che assegna un nome alla funzione e ne definisce la semantica. Tali funzioni sono associate a una base di dati e possono essere utilizzate solo nel contesto della rispettiva base di dati. Una caratteristica importante delle funzioni utente DB2 è che aderiscono al principio di *overloading* della programmazione orientata agli oggetti. È possibile definire cioè funzioni con lo stesso nome purché i parametri di ingresso delle varie definizioni differiscano in tipo e/o numero. In base allo stesso principio è anche possibile ridefinire una funzione DB2 predefinita, ad esempio gli operatori aritmetici.

Le funzioni utente DB2 possono essere classificate come segue.

- *Funzioni interne*: si costruiscono sulla base di funzioni DB2 predefinite, dette funzioni *sorgente*, in maniera simile a quanto avviene per i tipi utente.
- *Funzioni esterne*: corrispondono a programmi esterni alla base di dati e scritti in linguaggi di programmazione di alto livello (C o Java) che possono contenere o meno istruzioni SQL. La dichiarazione di una funzione esterna contiene la specifica della locazione su disco dove risiede il codice che implementa la funzione. Esistono due tipi di funzioni esterne:
  - *Funzioni scalari*: possono ricevere diversi valori in ingresso ma restituiscono un solo valore in uscita; se il nome della funzione ridefinisce un operatore (ad esempio "+") allora tali funzioni possono essere invocate utilizzando la notazione infissa;
  - *Funzioni tabella*: sono in grado di restituire una collezione di ennuple di valori, che può essere assimilata a una tabella: ogni volta che queste funzioni vengono invocate, esse restituiscono una nuova ennupla (vista come una riga di tabella), oppure un codice speciale che indica il completamento dell'operazione.

Le funzioni interne si usano principalmente con i tipi utente per poter applicare, in maniera selettiva, alcune funzioni dei relativi tipi sorgente. Ad esempio, con riferimento al tipo utente Money precedentemente definito, possiamo definire alcune funzioni interne con le seguenti dichiarazioni.

```
create function "*"(Money, Decimal()) returns Money
   source "*"(Decimal(), Decimal())
create function Total(Money) returns Money
   source sum(Decimal())
```

In queste dichiarazioni viene definito il nome della funzione, i tipi dei parametri di ingresso (che possono essere anche tipi utente), il tipo del dato restituito e la funzione *sorgente*, ovvero la funzione DB2 che va applicata quando viene invocata la funzione utente che si sta definendo. Nel nostro caso è stato ridefinito l'operatore di prodotto ed è stata definita una nuova funzione basata sull'operatore aggregato `sum` di SQL. Queste dichiarazioni rendono legali istruzioni SQL come le seguenti.

```
select Dipartimento, Total(Stipendio)
from Impiegato
group by Dipartimento;

update Impiegato
set Stipendio =  Stipendio * 1.1
where Dipartimento = 'Produzione';
```

Non sono invece valide espressioni che coinvolgono, ad esempio, la somma di uno stipendio e di un numero decimale, perché l'operatore di somma risulta indefinito per il tipo Money.

Supponiamo ora di voler definire una funzione scalare esterna che calcola lo stipendio di riferimento dovuto a un impiegato in base alla sua anzianità di servizio, valutando l'anzianità sulla base della data di assunzione. Una possibile definizione per tale funzione esterna è la seguente:

```
create function StandardSalary(Date) returns Money
   external name '/usr/db2/bin/salary.exe!StdSal'
   deterministic
   no external action
   language c parameter style db2sql
   no sql;
```

questa dichiarazione definisce la funzione esterna `StandardSalary` che riceve una data in ingresso e restituisce un dato di tipo Money. La clausola `external` indica il nome e la posizione su disco del file che contiene il codice della funzione. Il nome dopo il punto esclamativo indica il modulo (funzione C nel nostro caso), all'interno del file, che implementa la funzione utente. La clausola `deterministic` specifica che diverse invocazioni della funzione sullo stesso valore restituiscono sempre lo stesso risultato. La clausola `external action` serve a specificare se la funzione coinvolge azioni esterne alla base di dati, come ad esempio la scrittura in un file. Questa informazione può essere utile all'ottimizzatore per decidere se limitare il numero di invocazioni della funzione stessa. La clausola `language` specifica il linguaggio di programmazione usato per implementare la funzione esterna e le modalità di passaggio di parametri con essa. La modalità standard per il linguaggio C è denominata `db2sql`, mentre quello per il linguaggio Java è `db2general`. Infine, l'ultima clausola indica che la funzione non coinvolge accessi a una base di dati.

Non discuteremo qui le modalità secondo le quali si implementa una funzione esterna. Accenniamo solo al fatto che esistono delle convenzioni per trasformare tipi di dato SQL in tipi di dato del linguaggio di programmazione prescelto e viceversa, e che l'implementazione ha in effetti più parametri della funzione esterna. Questi parametri aggiuntivi servono per scambiare ulteriori dati di servizio, come eventuali messaggi di errore.

Una volta definita e implementata una funzione esterna, è possibile usarla in istruzioni SQL nella stessa maniera in cui si usano le funzioni predefinite. Ad esempio, la seguente istruzione trova cognomi e stipendi degli impiegati che guadagnano più del 20% del rispettivo stipendio di riferimento, mostrando anche tale valore.

```
select Nome, Stipendio, StandardSalary(DataAssunzione)
from Impiegato
where Stipendio > StandardSalary(DataAssunzione) * 1.2;
```

Si osservi che la condizione nella clausola `where` è valida perché: (a) la funzione utente restituisce un dato di tipo Money, (b) il prodotto tra questo tipo di dato e un decimale è stato definito, (c) il valore restituito dal prodotto è di tipo Money e può quindi essere confrontato con l'attributo Stipendio, come stabilito all'atto della definizione del tipo utente Money.

Le funzioni tabella costituiscono una funzionalità particolarmente interessante di DB2. Permettono infatti di trasformare facilmente una qualunque sorgente di dati esterna in una tabella manipolabile con SQL. L'unica cosa da fare è scrivere un programma che

accede alla sorgente, per esempio un file di testo o un file nel formato Microsoft Excel, filtra eventualmente dei dati in base a parametri passati in ingresso, e infine li restituisce, una riga alla volta. La restituzione delle varie righe avviene mediante una tecnica particolare: viene allocata in maniera automatica una zona di memoria (detta *scratchpad*) che preserva il suo contenuto tra una invocazione e un'altra di una funzione. In questa maniera, una invocazione può accedere a informazioni relative all'invocazione precedente; ad esempio, l'ultima posizione del file acceduta.

Supponiamo di avere a disposizione una funzione di questo tipo che prende il nome di una località in ingresso e restituisce una ennupla di valori memorizzati su un file remoto, corrispondenti a dati di vendita di negozi che si trovano in tale località. La definizione della corrispondente funzione tabella potrebbe essere la seguente:

```
create function Vendite(Char(20))
   returns table (Negozio  char(20),
                  Prodotto char(20),
                  Incasso  Integer)
   external name '/usr/db2/bin/sales'
   deterministic
   no external action
   language c parameter style db2sql no sql
   scratchpad
   final call disallow parallel;
```

Si può notare che, a parte alcune clausole specifiche per le funzioni tabella, la definizione è simile alla definizione di funzioni scalari esterne. La differenza principale è che il risultato della funzione è composto da più parametri, che vengono interpretati come attributi di una tabella.

Una possibile istruzione SQL che fa uso della funzione tabella sopra definita è la seguente:

```
select Negozio, sum(Incasso)
from table(Vendite('Roma')) as VenditeRoma
where Product = 'Giocattoli'
group by Negozio;
```

Questa istruzione restituisce l'incasso totale dei negozi di Roma relativo alle vendite di giocattoli. La tabella interrogata non è memorizzata nella base di dati (come indicato dalla parola chiave `table`), ma viene generata dalla funzione tabella `Vendite`, alla quale viene passato, come parametro di ingresso, la stringa `'Roma'`.

Bibliografia

[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, V. De Antonellis, A. Di Leva (editors). *Computer-Aieded Database Design: The DATAID Project*. North-Holland, Amsterdam, 1985.

[4] P. Atzeni, C. Batini, V. De Antonellis. *Teoria relazionale dei dati*. Boringhieri, Torino, 1985.

[5] P. Atzeni, S. Ceri, P. Fraternali, S. Paraboschi, R. Torlone. *Basi di dati: architetture e linee di evoluzione*. McGraw-Hill, Milano, 2003.

[6] P. Atzeni, V. De Antonellis. *Relational Database Theory*. Benjamin-Cummings, Menlo Park, California, 1993.

[7] C. Batini, S. Ceri, S. B. Navathe. *Conceptual Database Design, an Entity-Relationship Approach*. Benjamin-Cummings, Menlo Park, California, 1992.

[8] C. Batini, G. De Petra, M. Lenzerini, G. Santucci. *La progettazione concettuale dei dati*. Franco Angeli, Milano, 1991.

[9] G. Booch, I. Jacobson, J. Rumbaugh. Second edition. *The Unified Modeling Language User Guide*. Addison-Wesley, Reading, Massachussetts, 2005.

[10] M.L. Brodie and M. Stonebraker. *Migrating Legacy Systems: Gateways, Interfaces & the Incremental Approach*. Morgan-Kaufmann, Los Altos, California, 1995.

[11] L. Cabibbo, R. Torlone, C. Batini. *Basi di dati: progetti ed esercizi svolti*. Pitagora editrice, Bologna, 1995.

[12] 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).

[13] S. Castano, M.G. Fugini, G. Martella, P. Samarati. *Database Security*. Addison Wesley, New York, 1994.

[14] S. Ceri (editor). *Methodology and Tools for Database Design*. North-Holland, Amsterdam, 1983.

[15] S. Ceri, P. Fraternali. *Designing Database Applications with Objects and Rules: The IDEA Methodology*. Addison-Wesley, Reading, Massachussetts, 1997.

[16] 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.

[17] S. Ceri, G. Gottlob, L. Tanca. *Logic Programming and Data Bases*. Springer-Verlag, Berlino, 1989.

[18] D. D. Chamberlin. *A Complete Guide to DB2 Universal Database*. Morgan-Kaufmann, San Mateo, California, 1998.

[19] D. D. Chamberlin, R. F. Boyce. SEQUEL: A structured English query language. *Proceedings of ACM SIGMOD Workshop*, vol. 1, pagg. 249–264, 1974.

[20] 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.

[21] 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.

[22] P. Ciaccia, D. Maio. *Lezioni di basi di dati*. Leonardo Editore, Bologna, 1995.

[23] E. F. Codd. A relational model for large shared data banks. *Communications of the ACM*, vol. 13, n. 6, pagg. 377–387, 1970.

[24] 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.

[25] E. F. Codd. Relational completeness of database sublanguages. In R. Rustin, *Database Systems*, pagg. 65–98, Prentice Hall, Englewood Cliffs, New Jersey, 1972.

[26] 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.

[27] E. F. Codd. Relational database: A practical foundation for productivity. *Communications of the ACM*, vol. 25, n. 2, pagg. 109–117, 1982.

[28] C. J. Date. *An Introduction to Database Systems*. Eighth edition. Addison-Wesley, Reading, Massachussetts, 2003.

[29] C. J. Date, H. Darwen. *A Guide to the SQL Standard*. Third edition. Addison-Wesley, Reading, Massachussetts, 1993.

[30] A. Eisenberg, J. Melton. Standards in practice. *ACM SIGMOD Record*, vol. 27, n. 3, pagg. 53–58, 1998.

[31] 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).

[32] C. C. Fleming. B. von Halle. *Handbook of Relational Database Design*. Addison-Wesley, Reading, Massachussetts, 1989.

[33] 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).

[34] C. Francalanci, F. Schreiber, L. Tanca. *Progetto di dati e funzioni*. Società Editrice Esculapio, Bologna, 2003.

[35] H. Garcia-Molina, J.D. Ullman, J. Widom. *Database Systems: The Complete Book*. Prentice Hall, 2002.

[36] C. Ghezzi, M. Jazayeri, D. Mandrioli. *Ingegneria del software Fondamenti e principi*. Seconda edizione. Pearson Education Italia, Milano).

[37] J. Gray, A. Reuter. *Transaction Processing Concepts and Techniques*. Morgan-Kaufmann, San Mateo, California, 1994.

[38] R. Hull, R. King. Semantic database modelling: survey, applications and research issues. *ACM Computing Surveys*, vol. 19, n. 3, pagg. 201–260, 1987.

[39] IBM Corporation. *IBM DATABASE 2 SQL Guide for Common Servers, Version 2*. 1995.

[40] W. Kim. On optimizing an SQL-like nested query. *ACM Transactions on Database Systems*, vol. 7, n. 3, pagg. 443–469, 1982.

[41] 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.

[42] 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).

[43] D. Maier. *The Theory of Relational Databases*. Computer Science Press, Potomac, Maryland, 1983.

[44] D. Maio, S. Rizzi. *Esercizi di progettazione di basi di dati*. Pitagora editrice, Bologna, 1995.

[45] H. Mannila, K. J. Raiha. *The Design of Relational Databases*. Addison-Wesley, Reading, Massachussetts, 1992.

[46] J. Melton. SQL-3 update. *Proceedings of the IEEE International Conference on Data Engineering 1996*, New Orleans, pagg. 666-672, 1996.

[47] J. Melton, A. R. Simon. *Understanding the New SQL*. Morgan-Kaufmann, San Mateo, California, 1993.

[48] J. Melton, A. R. Simon. *SQL:1999 - Understanding Relational Language Components*. Morgan-Kaufmann, San Mateo, California, 2001.

[49] Oracle Corporation. *Oracle 8 Server: Concepts Manual*. Redwood City, California, 1998.

[50] Oracle Corporation. *Oracle 8 Server: SQL Language Reference Manual*. Redwood City, California, 1998.

[51] J. Paredaens, P. De Bra, M. Gyssens, D. Van Gucht. *The Structure of the Relational Database Model*. Springer-Verlag, Berlino, 1989.

[52] 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).

[53] 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).

[54] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen. *Object-Oriented Modelling and Design*. Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[55] A. Salvaggio. *Access: Programmazione VBA*. Edizioni FAG, Milano, 2005.

[56] D. Shasha. *Database Tuning: A Principled Approach*. Morgan-Kaufmann, San Mateo, California, 1992.

[57] A. Silberschatz, H. F. Korth, S. Sudarshan. *Database System Concepts*. McGraw-Hill, New York, 1997.

[58] 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.

[59] I. Sommerville. *Software Engineering*. Seventh Edition. Addison-Wesley, Reading, Massachussetts, 2004. (edizione italiana: *Ingegneria del software, 7/ed.*, Pearson Education Italia, Milano).

[60] M. Stonebraker (editor). *Readings in Database Systems*. Second edition. Morgan-Kaufmann, San Mateo, California, 1994.

[61] T. J. Teorey. *Database Modeling and Design: the E-R Approach*. Morgan-Kaufmann, San Mateo, California, 1990.

[62] 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.

[63] D. Tsichritzis, F. H. Lochovsky. *Data Models.* Prentice-Hall, Englewood Cliffs, New Jersey, 1982.
[64] 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).
[65] J. D. Ullman, J. Widom. *A First Course in Database Systems.* Prentice-Hall, Upper Saddle River, New Jersey, 1997.
[66] *Vocabolario della lingua italiana.* Istituto della Enciclopedia Italiana, 1987.
[67] G. Vossen. *Data Models, Database Languages, and Database Management Systems.* Addison-Wesley, Reading, Massachussetts, 1990.
[68] G. Wiederhold. *Database Design.* McGraw-Hill, New York, 1983.
[69] C. Zaniolo. Database relations with null values. *Journal of Computer and System Science*, vol. 28, n. 1, pagg. 142–166, 1984.

Indice analitico

Abort, 166
Access, *349–370*
    Avvio di, 350
    Cammini di join in, 356
    Definizione di tabelle in, 351
    Domini in, 352–355
    Interprete SQL in, 365
    Macro in, 369
    Maschere in, 366
    Popolamento delle tabelle in, 358
    QBE (query by example) in, 359–365
    Report in, 368
Accesso, Controllo di, 163
Accorpamento
    di associazioni, 278, 288–289
    di entità, 278, 288
Acidità delle transazioni, 167
ADO, ActiveX Data Object, 187
ADO.NET, 188
Affidabilità dei dati, 5
Aggregazioni in UML, 232
Algebra relazionale, 45–72
    Espressione in, 61
    Interrogazione in, 61–64
    Operatori dell', 46
`alias`, 109, 116
Alias in SQL, 120
`all`, 116, 123, 131, 134
`alter`, 105
Amministratore della base di dati, 11
Analisi
    dei requisiti, 200, 243–249, 260, 266
    delle ridondanze, 278–282
    di prestazioni su schemi E-R, 274–278
Anomalie di aggiornamento, 318
ANSI, 91
`any`, 134
Applicazioni
    in DB2, 381–385
`asc`, 122
Asserzioni in SQL, 152
Associazioni
    Accorpamento di, 278, 288–289
    in UML, 228, 230–232
    nel modello E-R, 208–210
    Partizionamento di, 278, 288–289
    Verifica di normalizzazione di, 341–342
Atomicità, 167
Attributi
    Cardinalità di, 215–216
    composti, 212
    di relazione, 21
    Eliminazione di – multivalore, 287
    in UML, 229
    nel modello E-R, 210–212
Autorizzazioni, 5, 163
`avg`, 124

Backup, 5
Base di dati, 2, 4–6
    Amministratore della, 11
    Condivisione di, 4
    Dimensione di una, 4
    Estensione della, 8
    Intensione della, 8
    Istanza di, 8
    Progettazione di, 199–316
    Schema della, 8
    Utenti della, 11
Batch, 171
`bigint`, 97
`blob`, 97, 385
Blocco `select-from-where`, 109
`boolean`, 97
Bottom-up
    Primitive di trasformazione, 254
    Strategia, 253–255, 260
Boyce e Codd, Forma normale di , 321–323
Business rules, 223–225

Calcolo relazionale, 72–83
    su domini, 73–80
        Interrogazione nel, 74–78
    su tuple, 80–83, 115
Call Level Interface, 184
Cammini di join, 301–302
    in Access, 356
Caratteristiche delle operazioni, 273, 275

Cardinalità
di attributi, 215–216
di relazioni, 212–215, 232
cascade, 103–106, 141, 142
cascaded, 154, 155
case, 161
CASE, Strumenti, 265–267, 308–310
cast, 160
Cataloghi relazionali, 106
char_length, 160
character, 94
check, 151, 152
Chiave
nel modello E-R, 216
nel modello relazionale, 34–37, 101, 116, 128, 321
Ciclo di vita dei sistemi informativi, 199–202
Classi in UML, 228–230
CLI, 184, 384–385
clob, 97, 385
close, 180
coalesce, 160
CODASYL, 7
Collaudo di sistemi informativi, 200
commit work, 166
Completezza di uno schema E-R, 259
Composizioni in UML, 232
Condivisione dei dati, 4
Conflitto d'impedenza, 178
Consistenza, 167
Controllo di Accesso, 163
Correttezza di uno schema E-R, 258
Costrutti del modello E-R, 206
Costruttore di tupla in SQL, 139
count, 123, 129
create assertion, 153
create domain, 99
create function, 388–390
create role, 165
create schema, 97
create table, 98, 310
create view, 154
Criteri di rappresentazione, 249–250
current of, 180
current_date, 159
current_time, 159
current_timestamp, 160
Cursori, 179, 184

Datalog, 83–87
Interrogazioni in, 84–86
Negazione in, 86
date, 96
Dati
Affidabilità dei, 5
Indipendenza dei, 9
Privatezza dei, 5, 163
Progettazione dei, 200
Dato, 2
DB2, 371–390
Applicazioni in, 381–385
Definizione di tabelle in, 377–378
Funzioni utente in, 388–390
Gestione di una base di dati in, 374–385
Interazione con, 374
Interprete SQL in, 378–380
DBA, 11
dbclob, 385
DBMS, 4–6, 12, 204
DDL, Data Definition Language, 10, 91
deallocate, 183
decimal, 95
declare cursor, 179
declare section, 177
Decomposizione, Qualità della, 323–328
default, 99, 103, 107, 108, 143
deferred, 153
Definizione
di domini in SQL, 99
di schema in SQL, 97
di tabelle
in Access, 351
in SQL, 98
delete, 104, 141, 163
desc, 122
Descrittore, 384
Diagramma di
attività, 228
casi d'uso, 228
classi, 227–229, 236
collaborazione, 228
componenti, 228
comunicazione, 228
distribuzione dei componenti, 229
oggetti, 228
sequenza, 228
stati, 228
Differenza (operatore), 47
Dipendenza funzionale, 319–321, 333–338
Copertura di, 336–337
Implicazione di, 333–335
distinct, 116, 117, 123, 124
Dizionario dei dati
in SQL, 106
nel modello E-R, 225
DML, Data Manipulation Language, 10, 91
Documentazione di schemi E-R, 222–225
Domini

Definizione di – in SQL, 99, 387
elementari di SQL, 94–97
in Access, 352–355
introdotti in SQL-3, 97
`double precision`, 95
`drop`, 105
Duplicati in SQL, 116

ECA, paradigma, 174
ECPG, 177
Eliminazione
degli attributi multivalore, 287
delle generalizzazioni, 282–286
delle gerarchie, 278
Embedded, SQL, 177
Entità, 206–208
Accorpamento di, 278, 288
Identificatori di, 216–218
Partizionamento di, 278, 286–287
Verifica di normalizzazione di, 339–341
Entry SQL, 93
Equi-join, 61
Equivalenza
di espressioni algebriche, 65–68
di linguaggi di interrogazione, 79
Espressione
del calcolo relazionale su domini, 74–78
dell'algebra relazionale, 61
Equivalenza di, 65–68
Estensione della base di dati, 8
`except`, 130, 133, 135, 140
`exec sql`, 177, 181
`execute`, 182, 184
Extender, 386–387

`fetch`, 179
Fisica, Progettazione, 310–312
Fixpoint, 86
`float`, 95
Foreign key, 38, 102
Forma normale, 317
di Boyce e Codd, 321–323, 328–330
Seconda, 331
Terza, 328–331, 337–338
`from`, 109, 111
`full join`, 118, 119
Full SQL, 93
Funzionamento di sistemi informativi, 200
Funzioni
Progettazione delle, 200
scalari in SQL, 159–162
utente in DB2, 388–390

Generalizzazioni
Eliminazione delle, 282–286
esclusive
in UML, 235
nel modello E-R, 219
Gerarchie di, 220
in UML, 234–235
nel modello E-R, 218–220
parziali
in UML, 235
nel modello E-R, 219
sovrapposte
in UML, 235
nel modello E-R, 219
totali
in UML, 235
nel modello E-R, 219
Gestione di una base di dati in DB2, 374–385
Glossario dei termini, 246, 260
`grant`, 164
`group by`, 126–130

`having`, 129–130

Identificatori
di entità, 216–218
esterni e interni, 217
in UML, 233–234
Scelta degli – primari, 278, 289–291
`immediate`, 153
Implementazione di sistemi informativi, 200
Incastonamento, 177
Indici, 204, 206, 278, 311–312
di prestazione, 274, 281
secondari, 291
Indipendenza
dal dominio, 78
dei dati, 9
Informazione, 2
incompleta, 28–31
`inner join`, 118
`insert`, 140, 163
Inside-out, Strategia, 255–257
`integer`, 95
Integrazione di schemi, 254, 255, 257, 260
Intensione della base di dati, 8
Interazione con DB2, 374
Intermediate SQL, 93
Interpretazione algebrica di SQL, 115
Interprete SQL
in Access, 365
in DB2, 378–380
Interrogazioni
in algebra relazionale, 61–64
in Datalog, 84–86
in SQL, 108–140

insiemistiche in SQL, 130–133
nel calcolo relazionale su domini, 74–78
nidificate, 133–140
`intersect`, 130
Intersezione (operatore), 47, 59
`interval`, 96
`into`, 173, 178, 179, 181, 184
ISO, 91
Isolamento, 168
Istanza
di base di dati, 8
di DB2, 373
di relazione, 25
nel modello E-R, 206

JDBC, 190–194, 372, 384–385
Join, 53–61, 66, 68
Cammino di, 301–302
Equi-, 61
esterno, 57–58
interni ed esterni in SQL, 117–120
naturale, 54–55
Theta-, 59–61

`left join`, 118
Leggibilità di uno schema E-R, 259
`like`, 114
Linguaggio
di definizione dei dati, 10
di manipolazione dei dati, 10
dichiarativo, 45, 72
Equivalenza di, 79
Indipendenza dal dominio del, 78
per basi di dati, 10–11
procedurale, 45, 72
LOB, 385–386
Locatori, 386
Logica, Progettazione, 204, 273–310
`lower`, 160

Macro in Access, 369
Manipolazione dei dati in SQL, 140–144
Maschere in Access, 366
`max`, 124, 135
Metamodello, 228
Metodi in UML, 229
Metodologia di progettazione, 202–206
generale, 260–261
`min`, 124, 135
Minimalità di uno schema E-R, 259
`minus`, 130
Mista, Strategia, 257–258
Modellazione dei dati
in UML, 227–236
nel modello E-R, 206–222
Modello
a oggetti, 7
basato su record e puntatori, 23
basato su valori, 23
concettuale, 7, 202
Traduzione da – a modello logico, 204, 273, 291–297
dei dati, 6–8
dell'applicazione, 228
Entità-Relazione, 8, 204, 206–222
Attributi nel, 210–212
Chiave nel, 216
Costrutti del, 206
Dizionario dei dati nel, 225
Entità, 206–208
Generalizzazioni nel, 218–220
Identificatori nel, 216–218
Istanza nel, 206
Relazioni nel, 208–210
Vincoli di integrità nel, 212, 223–224
fisico, 204
gerarchico, 7
logico, 8, 204
Traduzione da modello concettuale a, 204, 273, 291–297
relazionale, 7, 17–43
Traduzione verso il, 291–297
reticolare, 7
Modifica degli schemi in SQL, 105
Molteplicità di associazioni, 232–233

n-upla, 20
Negazione in Datalog, 86
`no action`, 103
Normalizzazione, 204, 317–348
`not null`, 100
`nullif`, 161
Nullo, valore, 28–31, 36–37, 68–70
`numeric`, 95

Occorrenza nel modello E-R, 206
ODBC, 185–187, 384–385
OLE DB, 187
`open`, 179
Operatori
aggregati in SQL, 123–125
dell'algebra relazionale, 46
Operazioni batch e interattive, 275
`order by`, 122
Ordinamento in SQL, 122
`outer join`, 118, 119

Partecipazione obbligatoria e facoltativa a relazione, 214
Partizionamento
di associazioni, 278, 288–289
di entità, 278, 286–287
Persistenza, 168
PL/SQL, 174
Popolamento delle tabelle in Access, 358
PostgreSQL, 159, 177
Predicato
estensionale, 83
intensionale, 83
`prepare`, 183
Prestazioni su schemi E-R, Analisi di, 274–278
`primary key`, 101
Primitive di trasformazione
bottom-up, 254–255
top-down, 250–253
Privatezza dei dati, 5, 163
Privilegi in SQL, 163–167
Procedure in SQL, 172–174
Prodotto cartesiano (operatore), 59
Progettazione
concettuale, 202, 243–267, 339
dei dati, 200
delle funzioni, 200
di basi di dati, 199–316
di sistemi informativi, 199
fisica, 204, 310–312
logica, 204, 273–310, 339
Metodologia di, 202
generale, 260–261
Proiezione (operatore), 52–53, 66
Prolog, 83
Prototipizzazione di sistemi informativi, 200
Punto fisso, 86

QBE (query by example) in Access, 359–365
Qualità di uno schema E-R, 258–260
Quantificatore, 74, 78, 80

Raccolta dei requisiti, 200, 243–249
Raggruppamento in SQL, 126–130
Range list, 80
Rappresentazione
Criteri di, 249–250
di schemi relazionali, 301–302
`real`, 95
Recovery, 5
`references`, 102–104, 163
Referenziale, Vincoli di integrità, 38–41, 102–104
Regole
attive, 174
aziendali, 223–225
Datalog, 83
ricorsive, 84, 86
Relazione, 18–28
Attributo di, 21
derivata, 70
di base, 70
Istanza di, 25
nel modello E-R, 208–210
Cardinalità di, 212–215
molti a molti, 215
ricorsiva, 210
uno a molti, 214
uno a uno, 214
Schema di, 25
virtuale, 71
Report in Access, 368
Requisiti
Analisi dei, 200, 243–249
di una base di dati, 204
`revoke`, 164
Ricorsive
Interrogazioni – in SQL, 158–159
Relazioni, 210
Ridenominazione (operatore), 47–49
Ridondanze, 3, 319
Analisi delle, 278–282
in uno schema E-R, 259
`right join`, 118
Ristrutturazione di schemi E-R, 273, 278–291
`rollback work`, 166
Ruoli, 165

Scelta degli identificatori primari, 278, 289–291
Schema
concettuale, 202, 204
Proprietà di uno, 258–259
Definizione di – in SQL, 97
di base di dati, 8
di operazione, 276
di relazione, 25
Rappresentazione di uno, 301–302
E-R, 206
Analisi di prestazioni su, 274–278
Completezza di, 259
Correttezza di, 258
Documentazione di, 222–225
Leggibilità di, 259
Minimalità di, 259
Proprietà di uno, 258–259
Ristrutturazione di, 273, 278–291
esterno, 9

fisico, 18, 204
interno, 9
logico, 8, 18, 204
scheletro, 257, 261
`scroll`, 179
`select`, 109–111, 163
Selezione (operatore), 50–52, 66, 67
`set constraints`, 153
`set default`, 103
`set null`, 103
`set role`, 165
Sistema di gestione di basi di dati, 4–6
Sistema informatico, 2
Sistemi informativi, 1
Analisi dei requisiti nei, 200
Ciclo di vita dei, 199–202
Collaudo di, 200
Funzionamento di, 200
Implementazione di, 200
Progettazione di, 200
Prototipizzazione, 200
Raccolta dei requisiti nei, 200
Validazione di, 200
`smallint`, 95
Specializzazioni, 218
SQL, 371
Alias in, 120
Asserzioni in, 152
Costruttore di tupla in, 139
Definizione di domini in, 99
Definizione di schema in, 97
Definizione di tabella in, 98
dinamico, 383–384
Dizionario dei dati in, 106
Domini elementari di, 94–97
Duplicati in, 116
Embedded, 177
Funzioni scalari in, 159–162
Interpretazione algebrica di, 115
Interrogazioni, 108–140
insiemistiche in, 130–133
ricorsive in, 158–159
Join interni ed esterni in, 117–120
Manipolazione dei dati in, 140–144
Modifica degli schemi in, 105
Operatori aggregati in, 123–125
Ordinamento in, 122
Privilegi in, 163–167
Procedure, 172–174, 388
Raggruppamento in, 126–130
Standardizzazione di, 91–93
statico, 382–383
Trigger in, 174
Valori di default in, 99–100
Valori nulli in, 114–115
Variabili in, 120–122, 134, 136–138
Vincoli di integrità in, 100–104
Viste in, 154–159
SQL-2, 92, 96, 99, 107, 115, 117, 120, 151, 158–160, 172, 178
SQL-3, 92, 147, 158, 173, 175
Domini introdotti in, 97
Ruoli in, 165
SQL-89, 92, 93, 115
SQL:1999, 92
SQL:2003, 92, 97
`sqlca`, 177
`sqlcode`, 177
Standardizzazione di SQL, 91–93
`start transaction`, 166
Strategia di progetto, 250–258, 260
bottom-up, 253–255
inside-out, 255–257
mista, 257–258
top-down, 250–253
Studio di fattibilità, 200
`substring`, 160
`sum`, 124
Superchiave, 34, 293
Sviluppo basato sui dati, 202

Tabella, 18–20
Definizione di – in Access, 351
Definizione di – in DB2, 377–378
Definizione di – in SQL, 98
Target list, 74
in SQL, 109
Tavola
degli accessi, 278, 281
dei volumi, 276, 281
delle operazioni, 276, 281
Theta-join, 59–61
`time`, 96
`timestamp`, 96
Top-down
Primitive di trasformazione, 250
Strategia, 250–253, 260
Traduzione da modello concettuale a logico, 204, 273, 291–297
Traduzione equivalente, 291
Transazione, 166–168
Trigger, 174
Tupla, 22
Costruttore di – in SQL, 139
dangling, 56

UML, 228–236, 267
Aggregazioni in, 232
Associazioni in, 228, 230–232
Attributi in, 229

Classi in, 228–230
Composizioni in, 232
Generalizzazioni in, 234–235
Identificatori in, 233–234
Metodi in, 229
`union`, 130
Unione (operatore), 46
`unique`, 101
`update`, 104, 143–144, 163
`upper`, 160
`usage`, 164
Utenti, 11

Validazione di sistemi informativi, 200
Valore nullo, 28–31, 36–37, 68–70
in SQL, 114–115
Valori di default in SQL, 99–100
Variabili in SQL, 120–122, 134, 136–138
Vincoli di integrità, 31–33
di tupla, 33
in SQL, 100–104
intrarelazionali, 100–102
nel modello E-R, 212
referenziale, 38–41, 102–104
Virtuale, Relazione, 71
Vista, 9, 70–72
in SQL, 154–159
materializzata, 71
ricorsiva, 158–159
Volume dei dati, 275, 276

`where`, 112

Finito di stampare
nel mese di febbraio 2006

Paolo Atzeni
Stefano Ceri
Stefano Paraboschi
Riccardo Torlone

Basi di dati

Modelli e linguaggi di interrogazione

Il testo si rivolge sia agli studenti di basi di dati delle facoltà di Ingegneria, Informatica, Economia e altri corsi di laurea, sia ai professionisti del settore informatico interessati agli argomenti nei loro ruoli di utenti avanzati e di progettisti del sistema informativo.

Scopo del libro è fornire una descrizione delle basi di dati adatta a chi deve progettare e programmare applicazioni, offrendo una visione chiara e aggiornata sugli aspetti fondamentali delle basi di dati, sui linguaggi di interrogazione e gestione e sulle tecniche e i metodi di progettazione.

Il volume è la seconda edizione di un testo che è stato accolto con grande favore e tiene conto dell'evoluzione recente della tecnologia, realizza diversi suggerimenti provenienti dalla comunità dei lettori e aggiorna la descrizione degli standard SQL in seguito al rilascio di SQL:2003. Amplia inoltre la sezione dedicata a UML e presenta una versione profondamente rivista del capitolo dedicato alla normalizzazione.

Il libro *Basi di dati: Architetture e linee di evoluzione* degli stessi autori costituisce la naturale continuazione di questo volume, che può d'altra parte essere utilizzato in piena autonomia.

All'indirizzo web www.ateneonline.it/atzeni sono disponibili materiali di supporto al testo per studenti e docenti.

Paolo Atzeni è professore ordinario di Basi di dati presso la facoltà di Ingegneria dell'Università Roma Tre.
Stefano Ceri è professore ordinario di Basi di dati presso la facoltà di Ingegneria dell'Informazione del Politecnico di Milano.
Stefano Paraboschi è professore ordinario presso la facoltà di Ingegneria dell'Università di Bergamo.
Riccardo Torlone è professore straordinario presso la facoltà di Ingegneria dell'Università Roma Tre.

ISBN 88-386-6292-4

9 788838 662928

www.mcgraw-hill.it

www.ateneonline.it

€ 31,00 (i.i.)