FONDAMENTI DI BASI DI DATI

Antonio Albano, Giorgio Ghelli, Renzo Orsini

Copyright © 2005-2024 A. Albano, G. Ghelli, R. Orsini

Si concede il diritto di riprodurre gratuitamente questo materiale con qualsiasi mezzo o formato, in parte o nella sua interezza, per uso personale o per uso didattico alle seguenti condizioni: le copie non sono fatte per profitto o a scopo commerciale; la prima pagina di ogni copia deve riportare questa nota e la citazione completa, incluso il titolo e gli autori. Altri usi di questo materiale inclusa la ripubblicazione, anche di versioni modificate o derivate, la diffusione su server o su liste di posta, richiede un permesso esplicito preventivo dai detentori del copyright.

Revisione del 2 marzo 2024

INDICE

Prefazione

Dopo molti anni dalla pubblicazione della prima edizione del volume *Fondamenti di Basi di Dati* presso l'Editore Zanichelli, aggiornato con una seconda versione uscita nel 2005, è tempo di un'ulteriore ammodernamento, che coincide con la sua diffusione con un canale diverso da quello della carta stampata: il web, attraverso il sito http://fondamentidibasididati.it.

Riteniamo che questo materiale possa essere utile non solo per i classici corsi di *Basi di Dati*, fondamentali per le lauree in *Informatica* o *Ingegneria Informatica*, ma, data l'attenzione cha sta oggi avendo l'informatica in sempre più ampi settori della formazione e dell'istruzione ad ogni livello, in molti altri corsi di laurea e momenti formativi, in forma anche parziale, come abbiamo potuto sperimentare di persona durante questi ultimi anni.

Il passaggio alla nuova modalità di distribuzione, permettendo di mantenere aggiornati i contenuti, ci ha richiesto di modificare la struttura del sito di supporto al libro, che non avrà più la parte di errata corrige e di approfondimenti, ma conterrà materiale aggiuntivo utile per lo studio, come le soluzioni agli esercizi, i collegamenti ai software gratuiti, gli appunti del corso, e gli esempi scaricabili.

Organizzazione del testo

Il libro inizia presentando le ragioni che motivano la tecnologia delle basi di dati, ed i concetti principali che caratterizzano le basi di dati ed i sistemi per la loro gestione.

In maggior dettaglio, il Capitolo 2 si sofferma sulle nozioni fondamentali di modello informatico finalizzato al trattamento delle informazioni di interesse dei sistemi informativi delle organizzazioni e sui meccanismi d'astrazione per costruire modelli informatici. Il modello di riferimento è il modello ad oggetti, motivato non solo dalla sua naturalezza per la progettazione di basi di dati, ma anche per essere il modello dei dati dell'attuale tecnologia relazionale ad oggetti per basi di dati. Il formalismo grafico adottato si ispira all'*Unified Modeling Language*, UML, ormai uno standard dell'ingegneria del software.

Il Capitolo 3 presenta una panoramica del problema della progettazione di basi di dati, si sofferma sulle fasi dell'analisi dei requisiti e della progettazione concettua-

le usando il modello ad oggetti e il formalismo grafico proposto nel Capitolo 2, e descrive un metodo di lavoro per queste fasi.

I Capitoli 4 e 5 sono dedicati alla presentazione rigorosa del modello relazionale dei dati e ad un'introduzione alla teoria della normalizzazione. La scelta di dedicare questo spazio all'argomento è giustificata dal ruolo fondamentale che svolge il modello relazionale per la comprensione della tecnologia delle basi di dati e per la formazione degli addetti.

I Capitoli 6, 7 e 8 trattano il linguaggio relazionale SQL da tre punti di vista, che corrispondono ad altrettante categorie di utenti: (1) gli utenti interessati ad usare interattivamente basi di dati relazionali, (2) i responsabili di basi di dati interessati a definire e gestire basi di dati, (3) i programmatori delle applicazioni.

Il Capitolo 9 presenta una panoramica delle principali tecniche per la realizzazione dei sistemi relazionali. Vengono presentate in particolare la gestione delle interrogazioni e dei metodi di accesso e la gestione dell'atomicità e della concorrenza in sistemi centralizzati.

Ringraziamenti

L'organizzazione del materiale di questo testo è il risultato di molti anni di insegnamento dei corsi di *Basi di Dati* per le lauree in *Scienze dell'Informazione* e in *Informatica*.

Molte persone hanno contribuito con i loro suggerimenti e critiche costruttive a migliorare la qualità del materiale. In particolare si ringraziano i numerosi studenti che nel passato hanno usato le precedenti versioni del testo e Gualtiero Leoni per la sua collaborazione.

Per la versione rivista per la pubblicazione su Web si ringrazia in particolare Alessandra Raffaetà, che ci ha aiutato nella revisione e ha suggerito molti miglioramenti.

Si ringrazia infine l'Editore Zanichelli per il supporto che ci ha dato in questi anni, e, adesso che il libro è uscito dal suo catalogo, per il permesso di diffondere la nuova versione attraverso il web.

A. A.
G. G.
R. O.

Capitolo 1

SISTEMI PER BASI DI DATI

Spesso interagiamo con *basi di dati* senza saperlo: quando facciamo un prelievo dal conto corrente con il bancomat, quando facciamo un acquisto con la carta di credito, quando acquistiamo un biglietto ferroviario o prenotiamo un viaggio presso un'agenzia, quando usiamo il sito web di un dipartimento universitario per iscriversi ad un esame ecc. In tutte queste situazioni è richiesto l'accesso a certe raccolte di dati o la loro modifica per registrare l'effetto dell'operazione compiuta. In generale queste raccolte di dati sono gestite da organizzazioni che dedicano molte attività e risorse alla raccolta, archiviazione ed elaborazione di informazioni per fornire opportuni servizi. Negli ultimi anni, per l'evoluzione della tecnologia elettronica e la conseguente riduzione dei costi, si è assistito ad una crescente diffusione degli elaboratori elettronici per agevolare e potenziare le possibilità di trattamento delle informazioni. Questo fenomeno ha interessato organizzazioni di ogni tipo e dimensioni ed ha portato ad una richiesta sempre più ampia di sistemi dedicati a questo scopo. In seguito, dopo una precisazione sul ruolo del *sistema informativo* e del *sistema informatico* all'interno di un'organizzazione, vengono definite le nozioni di *base di dati* e di *sistema per la gestione di basi di dati* e vengono presentati i concetti fondamentali che verranno sviluppati nei capitoli successivi.

1.1 Sistemi informativi e informatici

Ogni organizzazione, per il proprio funzionamento, ha bisogno di disporre di informazioni che costituiscono una risorsa gestita da un proprio *sistema informativo*, del quale si propone la seguente definizione.

Definizione 1.1
Il sistema informativo di un'organizzazione è una combinazione di risorse, umane e materiali, e di procedure organizzate per la raccolta, l'archiviazione, l'elaborazione e lo scambio delle informazioni necessarie alle attività operative (*informazioni di servizio*), alle attività di programmazione e controllo (*informazioni di gestione*), e alle attività di pianificazione strategica (*informazioni di governo*).

Con il termine *sistema* si evidenzia il fatto che esiste un insieme organizzato di elementi, di natura diversa, che interagiscono in modo coordinato, mentre con *informativo* si precisa che tutto ciò è finalizzato alla gestione delle informazioni e quindi le inte-

razioni che preme evidenziare sono quelle dovute a scambi di informazioni (*flussi informativi*).

Per esempio, un'industria manifatturiera gestisce informazioni per svolgere attività che includono:

1. gestione degli ordini e dei pagamenti dei prodotti venduti;
2. gestione degli ordini e dei pagamenti ai fornitori di materiali;
3. gestione del magazzino;
4. programmazione della produzione;
5. controllo di gestione;
6. pianificazione di nuovi prodotti.

Le informazioni di un'organizzazione, una volta ridotte a dati mediante un processo di interpretazione, quantificazione e formalizzazione, possono essere trattate automaticamente con gli elaboratori elettronici. La riduzione dei costi della tecnologia informatica ha diffuso largamente questa possibilità, rendendo più accurate e rapide le procedure e potenziando i modi di elaborazione delle informazioni.

Con il termine *sistema informativo automatizzato* si indica la parte del sistema informativo realizzata impiegando strumenti informatici e della comunicazione. In generale, raramente il sistema informativo di un'organizzazione viene completamente automatizzato sia a causa di problemi tecnici che per motivi di convenienza economica.

Spesso nella letteratura si parla di sistemi informativi pensando alla parte automatizzata. Per brevità e per evitare confusione useremo il termine *sistema informatico* per riferirci ai sistemi informativi automatizzati.

Un sistema informatico, per fornire i servizi attesi dagli utenti, prevede i seguenti componenti principali (Figura 1.1):

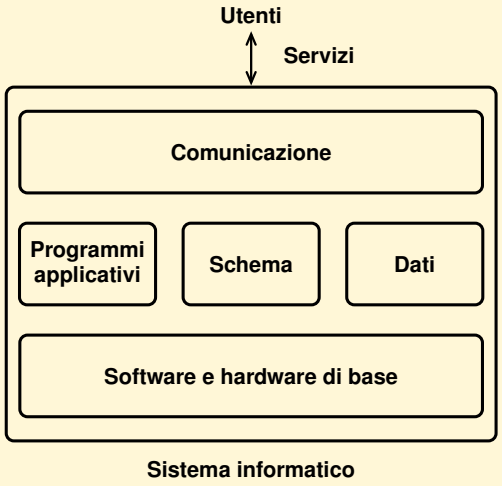


Figura 1.1: Componenti del sistema informatico

- il *software* e *l'hardware di base*;
- una *base di dati*, che contiene una rappresentazione del patrimonio informativo dell'organizzazione;
- uno *schema*, che descrive la struttura della base di dati, le operazioni per agire su di essa e le restrizioni sui valori memorizzabili nella base di dati e sui modi in cui essi possono evolvere nel tempo (*vincoli di integrità*). Lo schema viene usato dal sistema per garantire un uso corretto della base di dati;
- i *programmi applicativi*, che forniscono i servizi agli utenti eseguendo un certo insieme di operazioni sulla base di dati.
- la *comunicazione*, che permette l'accesso ai servizi del sistema informatico ad utenti e programmi.

In questo testo vengono prese in considerazione, tra le informazioni trattate da un'organizzazione, solo quelle strutturate, con un formato predeterminato, e di carattere prevalentemente globale, cioè utili a più reparti.

1.2 Evoluzione dei sistemi informatici

L'architettura dei sistemi informatici per la gestione di informazioni strutturate è cambiata molto negli ultimi quarant'anni con l'evolvere della tecnologia degli elaboratori elettronici e degli strumenti per la gestione di dati permanenti. Vediamo i passaggi più significativi.

Primo stadio: applicazioni operative

Agli inizi degli anni '60, le applicazioni riguardavano le attività delle funzioni amministrative che richiedevano l'elaborazione sistematica e ripetitiva di grandi quantità di dati, come il calcolo delle paghe e degli stipendi o l'emissione delle fatture. Successivamente, a questo primo nucleo di applicazioni, si sono aggiunte altre più complesse, come la gestione del magazzino e la contabilità dei clienti, che ancora oggi costituiscono il nucleo essenziale di ogni sistema informatico nelle aziende.

Secondo stadio: servizi informatici di funzione

Dalla fine degli anni '60, l'interesse si è spostato anche sull'elaborazione delle informazioni di supporto ai responsabili delle varie funzioni aziendali, con lo sviluppo di applicazioni per la contabilità generale, per il controllo di gestione e per la valutazione del funzionamento dell'azienda.

Questi tipi di sistemi informatici rispondevano quindi a due precise esigenze:

- rendere più efficienti le attività ripetitive dei livelli esecutivi dell'azienda;
- permettere una migliore gestione dell'azienda fornendo ai responsabili le informazioni sintetiche sull'andamente delle attività controllate.

Terzo stadio: servizi informatici per l'organizzazione

La realizzazione di servizi informatici per le funzioni aziendali non presupponeva l'integrazione dei dati in comune alle diverse funzioni e quindi comportava sia la duplicazione di dati, con il rischio di copie incoerenti, sia una limitata possibilità di correlare dati settoriali per generare informazioni di interesse globale per l'organizzazione.

A partire dagli anni '70, il progresso della tecnologia ha reso disponibili nuovi strumenti informatici, i *sistemi per la gestione di basi di dati* (*Data Base Management System, DBMS*), che, rendendo possibile una gestione integrata dei dati, interessavano ogni livello delle organizzazioni: i dati trattati automaticamente non erano suddivisi per interessi settoriali, ma venivano trattati globalmente, in modo che ciascuna informazione, benché rappresentata una sola volta, era utilizzabile per attività diverse del sistema informativo. Si è passati quindi da *sistemi informatici settoriali* a *sistemi informatici per l'organizzazione*, con notevoli riflessi sulla struttura dell'organizzazione stessa, in quanto un impiego razionale della tecnologia informatica comporta necessariamente una revisione del modo di funzionare della struttura organizzativa che deve utilizzarla.

I vantaggi più evidenti di questa soluzione sono:

1. *Integrazione dei dati*. Invece di avere per ogni applicazione una coppia (dati, programmi), con dati in generale non completamente distinti da quelli usati da un'altra applicazione, si vuole prevedere un'unica raccolta di dati comuni, che costituiscono le informazioni di base, e tanti programmi che realizzano le applicazioni operando solo sui dati di loro interesse. Questo obiettivo comporta i seguenti vantaggi:

   Disponibilità dei dati. Quando i dati non sono organizzati in funzione di una specifica applicazione, è più semplice renderli disponibili ad usi diversi.

   Limitazione delle ridondanze. L'esistenza di un'unica raccolta di dati, accessibili con modalità diverse, elimina la necessità di duplicazioni, che comportano maggiore occupazione di memoria e possibilità di inconsistenze fra i dati.

   Efficienza. La gestione integrata dei dati si presta allo sviluppo di strumenti per ottimizzare globalmente la loro organizzazione fisica e, quindi, l'efficienza complessiva del sistema.

2. *Flessibilità della realizzazione*. Le modifiche ad un sistema informatico funzionante sono inevitabili e devono poter essere fatte senza dover intervenire sulla realizzazione in modo massiccio. Infatti, il progetto e la realizzazione di un sistema informatico sono, in generale, dei compiti complessi: dal momento in cui si raccolgono le specifiche, al momento in cui si ha una prima versione funzionante della realizzazione, può trascorrere un intervallo di tempo sufficientemente lungo perché ci sia una modifica dei requisiti iniziali. Un'altra ragione, che richiede un'evoluzione della realizzazione, è che un sistema informatico di supporto a sistemi informativi si progetta in modo incrementale: si parte automatizzando un nucleo di funzioni essenziali e, successivamente, il sistema si amplia inglobando nuovi dati e funzioni. Esiste, infine, un'altra ragione, meno prevedibile, che può causare un'evoluzione

’60	Sistemi gerarchici (IMS e System 2000)
’70	Sistemi reticolari secondo la proposta CODASYL (IDMS, IDS II, DM IV, DMS 1100 ecc.) E.F. Codd propone il modello relazionale
’80	Sistemi relazionali (System R, Ingres, Oracle, SQL/DS, DB2, Informix ecc.)
’90	Sistemi relazionali distribuiti Architetture cliente/servente e parallele Sistemi ad oggetti e relazionali ad oggetti (GEMSTONE, ONTOS, Objectstore, O_2, Illustra, UniSQL ecc.)
1995	Integrazione con Internet

Tabella 1.1: Prospetto cronologico dell’evoluzione dei sistemi per basi di dati.

delle specifiche: quando il sistema funziona in modo soddisfacente, gli utenti vengono stimolati nella loro immaginazione e intravedono nuove possibilità d’impiego del sistema informatico, modificando così i requisiti iniziali.

Inizialmente i sistemi per basi di dati erano di tipo centralizzato, ovvero la base di dati era gestita da un unico elaboratore. Agli inizi degli anni ’90, invece, l’evoluzione della tecnologia ha reso possibile anche la realizzazione di sistemi per basi di dati distribuite, su rete locale o geografica, che consentono di avere una visione unica dei dati, indipendentemente da dove essi siano fisicamente memorizzati.

In Tabella 1.1 sono riportati i passaggi più significativi dell’evoluzione della tecnologia delle basi di dati.

Quarto stadio: servizi informatici per la pianificazione strategica

A partire dagli anni ’80, infine, lo sviluppo della tecnologia ha permesso una progressiva copertura delle esigenze informative per le attività di programmazione e di pianificazione strategica, allargando ulteriormente lo spettro delle possibilità di applicazione della tecnologia informatica nelle organizzazioni per l’automazione dei sistemi informativi.

Quinto stadio: servizi informatici per la cooperazione fra organizzazioni

A partire dalla fine degli anni ’90, infine, con la grande diffusione di standard di comunicazione dovuti allo sviluppo della rete Internet e del Web, sono iniziati a diffondersi protocolli di cooperazione fra sistemi informatici diversi, e strumenti che semplificano lo scambio di dati e le richieste di servizi fra di loro (*Web services*). L’obiettivo è di facilitare lo sviluppo di applicazioni di commercio elettronico che prevedano l’interazione di organizzazioni separate, con sistemi informatici eterogenei, come le aziende produttrici, quelle di intermediazione, di logistica, pubbliche amministrazioni ecc.

1.3 Tipi di sistemi informatici

Illustriamo le differenze fra le finalità di due tipici sistemi informatici delle organizzazioni, usando come esempio il caso di una catena di supermercati con negozi sul territorio nazionale.

1.3.1 Sistemi informatici operativi

Ogni negozio utilizza una base di dati operazionale (o transazionale) che raccoglie informazioni sugli effetti delle operazioni di routine necessarie quotidianamente per condurre le attività aziendali. Per esempio, per avere informazioni sui prodotti di cui dispone, per stabilire a quale prezzo vendere il prodotto quando viene fatto un acquisto, come stampare lo scontrino, come aggiornare il saldo del venduto ad ogni cassa e come aggiornare lo stato del magazzino. È facile immaginare come le cose si complichino quando sia necessario tener conto anche del fatto che i pagamenti possono essere fatti non solo in contanti, ma anche con bancomat o con carta di credito. Quando il cliente ha ricevuto lo scontrino, il sistema garantisce che la base di dati ha subito tutte le modifiche necessarie per tener conto di tutti gli effetti che ha prodotto l'evento che si è verificato alla cassa. Il termine *transazionale* si usa per riferirsi al fatto che le interazioni con il sistema avvengono mediante *transazioni*, un evento che innesca sulla base di dati una sequenza S di azioni che devono tutte andare a buon fine perché essa rimanga coerente, ovvero rifletta correttamente gli effetti dell'evento. In caso di un malfunzionamento che interrompa l'esecuzione di S, il meccanismo della transazione garantisce che la base di dati si troverà nello stato in cui si trovava prima che S iniziasse.

Le basi di dati operazionali sono gestite dalle applicazioni che fanno parte dei cosiddetti *sistemi informatici operativi* o *sistemi di elaborazione di transazioni* (*Transaction Processing Systems, TPS*). Le applicazioni assistono i dipendenti al livello operativo nello svolgimento delle attività di loro competenza (Figura 1.2).

Un esempio di questo tipo di soluzione sono i sistemi ERP, *Enterprise Resource Planning*, acronimo che non spiega la finalità di questi sistemi, che non è la pianificazione delle risorse aziendali, ma l'integrazione dei processi aziendali in un unico sistema software che possa soddisfare tutti i requisiti informativi dell'azienda usando una base di dati centralizzata.

1.3.2 Sistemi informatici direzionali

Le direzioni intermedia e operativa della catena di supermercati hanno bisogno di analizzare i dati di ogni negozio per produrre rapporti di riepilogo sull'andamento delle vendite e sul funzionamento del supermercato per prendere decisioni che riguardano tutta l'azienda al livello nazionale. Assumiamo che i rapporti da produrre riguardino (a) l'andamento delle operazioni di base dell'azienda nel breve periodo (settimanale, mensile e annuale), (b) siano standard e ripetitivi, (c) siano relativamente poco complessi, (d) siano producibili a partire dalle basi di dati operazionali. Per

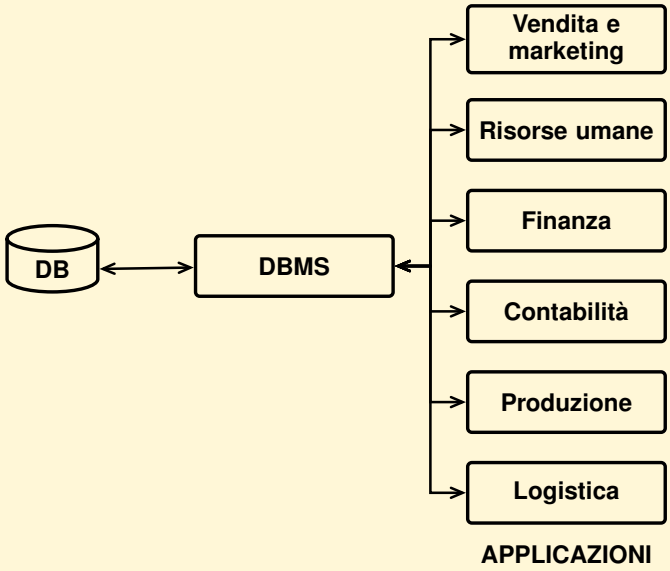


Figura 1.2: Sistemi informatici operativi

esempio, conoscere le vendite del supermercato con sede a Pisa nelle ultime quattro settimane.

Le informazioni sono ricavate dalle basi di dati operazionali usando applicazioni specializzate per assistere le direzioni intermedia e operativa dell'azienda nelle funzioni di programmazione e controllo.

Per prendere decisioni, le direzioni intermedia ed alta della catena di supermercati hanno invece bisogno di analisi storiche dell'andamento degli affari che comportano la produzione interattiva di rapporti di sintesi non programmati, più complessi e da punti di vista diversi, per scoprire situazioni anomali o tendenze interessanti, che non possono essere prodotti a partire dalle basi di dati operazionali perché esse di solito rendono disponibili solo i dati più recenti oppure perché le operazioni necessarie per produrre i rapporti sono complesse e rallenterebbero le operazioni di cassa oltre il tollerabile.

Per soddisfare queste esigenze, i dati storici vengono integrati con dati provenienti da fonti esterne e organizzati opportunamente in un altro tipo di base di dati, detta *data warehouse*, gestita separatamente con un opportuno sistema che metta a disposizione strumenti adatti per fare facilmente analisi interattive dei dati. Mentre una base di dati operazionale è aggiornata tempestivamente ad ogni verificarsi di un evento aziendale, il *data warehouse* viene aggiornato periodicamente con nuovi dati storici, o esterni, perché ai fini delle analisi di supporto alle decisioni non è necessario disporre di dati accurati al 100%.

Mentre nel caso precedente le esigenze sono soddisfacibili con richieste specifiche ("Trovare i possessori di carta di credito che hanno speso più di 50 euro in alcolici nel

mese di gennaio"), un'altra esigenza dell'alta dirigenza è di scoprire automaticamente qualche aspetto interessante di un insieme di dati con tecniche di *data mining* ("Qual è il profilo generale dei possessori di carta di credito che traggono profitto dalle promozioni?"). Il *data mining* è una fase di un processo interattivo e iterativo che cerca di estrarre modelli da un insieme di dati utili ai dirigenti per prendere decisioni. Un modello, in questo contesto, è una rappresentazione concettuale che evidenzia in una forma opportuna certe caratteristiche di informazioni implicite, sconosciute a priori e potenzialmente utili, presenti nei dati.

I dati per produrre le informazioni che aiutino i dirigenti a prendere decisioni sono gestiti dai cosiddetti *sistemi di supporto alle decisioni* (*Decision Support Systems, DSS*) (Figura 1.3).

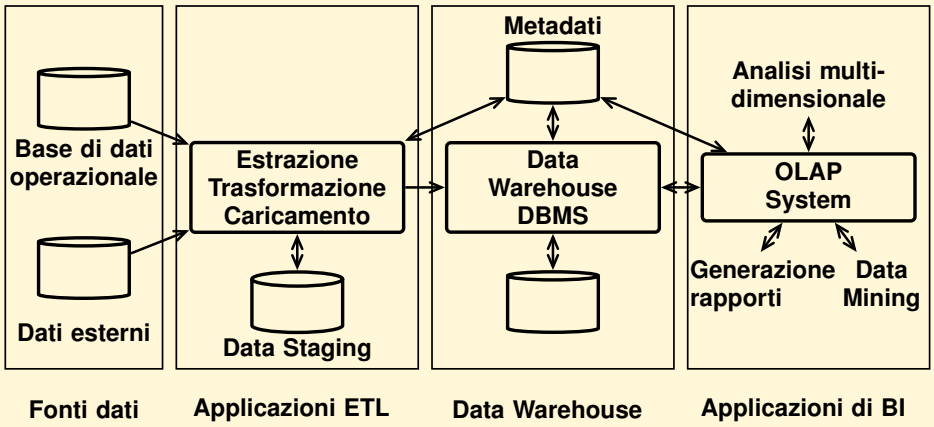


Figura 1.3: Sistemi informatici direzionali

1.4 I sistemi per basi di dati

Il termine *base di dati* viene spesso usato per riferirsi ad un qualsiasi insieme di dati permanenti trattati con un elaboratore elettronico, ma qui verrà usato con il seguente significato.

Definizione 1.2

Una base di dati è una raccolta di dati permanenti, gestiti da un elaboratore elettronico, suddivisi in due categorie:

1. i *metadati*, ovvero lo *schema* della base di dati (*database schema*), una raccolta di definizioni che descrivono la struttura dei dati, le restrizioni sui valori ammissibili dei dati (*vincoli d'integrità*), le relazioni esistenti fra gli insiemi,

e a volte anche alcune operazioni eseguibili sui dati. Lo schema va definito prima di creare i dati ed è indipendente dalle applicazioni che usano la base di dati;

2. i *dati*, le rappresentazioni di certi fatti conformi alle definizioni dello schema, con le seguenti caratteristiche:
   - sono organizzati in insiemi omogenei, fra i quali sono definite delle relazioni. La struttura dei dati e le relazioni sono descritte nello schema con opportuni meccanismi di astrazione dipendenti dal *modello dei dati* (*data model*) utilizzato, che prevede anche operatori per estrarre elementi da un insieme e per conoscere quelli che, in altri insiemi, sono in relazione con loro;
   - sono molti, in assoluto e rispetto ai metadati, e non possono essere gestiti tutti contemporaneamente in memoria temporanea;
   - sono permanenti, cioè, una volta creati, continuano ad esistere finché non sono esplicitamente rimossi; la loro vita quindi non dipende dalla durata delle applicazioni che ne fanno uso;
   - sono accessibili mediante *transazioni* (*transactions*), unità di lavoro atomiche che non possono avere effetti parziali;
   - sono protetti sia dall'accesso di utenti non autorizzati, sia da corruzione dovuta a malfunzionamenti hardware e software;
   - sono utilizzabili contemporaneamente da utenti diversi.[1]

Esempio 1.1
Per chiarire i concetti esposti, si mostra un semplice esempio di base di dati che memorizzi informazioni relative a studenti ed esami di un'università. Ogni insieme è visto come una tabella con tante colonne quanti sono gli attributi d'interesse. Questo tipo di base di dati è detto *relazionale* e sarà discusso in modo approfondito nel Capitolo 4.

Studenti

Nome	Matricola	Provincia	Nascita
Isaia	71523	Pisa	1980
Rossi	67459	Lucca	1981
Bianchi	79856	Livorno	1980
Bonini	75649	Pisa	1981

1. Il termine "utente" viene usato sia con il significato di persona che accede ai dati da un terminale in modo interattivo usando un opportuno linguaggio, sia con il significato di programma applicativo che contiene istruzioni per l'accesso ai dati.

ProveEsami

Materia	Candidato	Data	Voto
BD	71523	12/01/01	28
BD	67459	15/09/02	30
IA	79856	25/10/03	30
BD	75649	27/06/01	25
IS	71523	10/10/02	18

I metadati riguardano le seguenti informazioni:

1. il fatto che esistono due collezioni di interesse Studenti e ProveEsami;
2. la struttura degli elementi di queste due collezioni (intestazione delle tabelle): ogni studente ha una matricola, di tipo intero, che lo contraddistingue, un nome, di tipo stringa, un anno di nascita, una città di residenza; ogni esame ha una materia, la matricola dello studente, la data dell'esame e il voto ottenuto;
3. il fatto che ad ogni esame (inteso come l'evento in cui uno studente viene esaminato) corrisponde uno studente con la matricola specificata, e ad ogni studente corrispondono uno, nessuno, o più esami;
4. alcuni vincoli sui valori ammissibili, quali il fatto che il valore della matricola identifica una riga della tabella Studenti e i valori della matricola e della materia identificano una riga della tabella ProveEsami, oppure che il voto deve essere un numero intero compreso fra 18 e 30.

I dati invece sono le righe delle tabelle che contengono le informazioni relative ai singoli studenti ed esami.

Le caratteristiche delle basi di dati sono garantite da un *sistema per la gestione di basi di dati* (*Data Base Management System, DBMS*), che ha il controllo dei dati e li rende accessibili agli utenti autorizzati.

Definizione 1.3
Un DBMS è un sistema centralizzato o distribuito che consente (*a*) di definire schemi di basi di dati, (*b*) di scegliere le strutture dati per la memorizzazione e l'accesso ai dati, (*c*) di memorizzare, recuperare e modificare i dati, interattivamente o da programmi, ad utenti autorizzati e rispettando i vincoli definiti nello schema.

In Figura 1.4 è mostrata una versione semplificata della struttura di un DBMS. È interessante ricorrere ad un'analogia con concetti dei linguaggi di programmazione per chiarire alcuni dei concetti finora introdotti: modello dei dati, linguaggio per basi di dati, base di dati e schema, sistema per la gestione di basi di dati.

Modello dei dati

Un *modello dei dati* è un insieme di meccanismi di astrazione per definire una base di dati, con associato un insieme predefinito di operatori e di vincoli d'integrità.

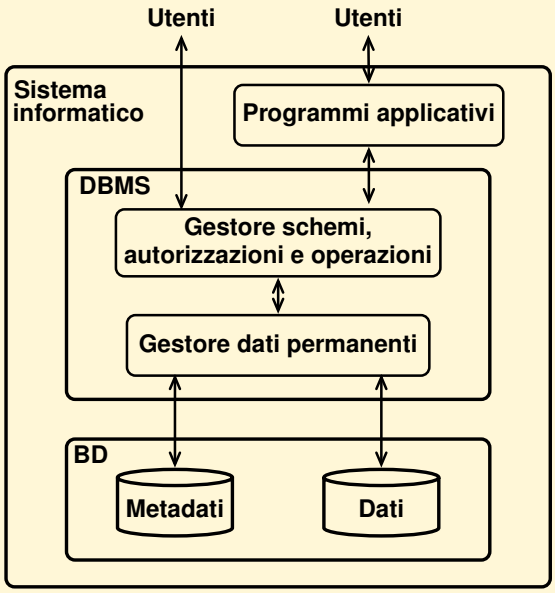


Figura 1.4: Il sistema informatico e il sistema per la gestione di basi di dati

I modelli dei dati adottati dai moderni sistemi commerciali sono il *relazionale* e quello *ad oggetti*. Le loro caratteristiche verranno presentate nel prossimo capitolo. I meccanismi di astrazione di un modello dei dati corrispondono ai meccanismi di astrazione per rappresentare i dati in un linguaggio di programmazione e vanno tenuti distinti dal modo in cui sono trattati in un particolare linguaggio per basi di dati, nello stesso modo in cui nei linguaggi di programmazione si parla, ad esempio, del meccanismo di astrazione del *record* e del *file*, prescindendo dalla forma sintattica che prende in un particolare linguaggio.

Un buon modello dei dati dovrebbe essere caratterizzato da:

- *espressività*: il modello dovrebbe permettere di rappresentare in modo naturale e diretto il significato di ciò che si sta modellando;
- *semplicità d'uso*: il modello dovrebbe essere basato su di un numero minimo di meccanismi semplici da utilizzare e da comprendere;
- *realizzabilità*: i meccanismi del modello di astrazione, e i relativi operatori per la manipolazione dei dati, devono essere realizzabili in modo efficiente su di un elaboratore elettronico.

Linguaggio per basi di dati

Quando si pensa ad un linguaggio di programmazione si pensa ad un linguaggio che prevede un sistema di tipi con opportuni operatori e una struttura del controllo.

Quando si pensa ad un linguaggio per base di dati, invece, per motivi storici, si usa distinguere fra:

1. il linguaggio per la definizione dello schema, ovvero della struttura delle collezioni dei dati (*Data definition language, DDL*);
2. il linguaggio degli operatori del modello dei dati, che permette di accedere ai dati e di modificarli, ma è in genere privo dei costrutti tipici dei linguaggi di programmazione (funzioni, struttura del controllo, variabili ecc.) (*Data manipulation language, DML*);
3. il linguaggio per la codifica delle applicazioni che richiedono lo sviluppo di programmi che usano la base di dati. Si tratta di solito di un linguaggio tradizionale, tipo C o Java, esteso aggiungendovi gli operatori del DML (*host language*);
4. il linguaggio di interrogazione (*query language*) per il recupero e la modifica di dati interattivamente, ma non per lo sviluppo di applicazioni.

Sono stati proposti sia linguaggi che offrono alcune delle funzionalità sopra elencate, come il linguaggio SQL che verrà descritto nei Capitoli 6-8, sia linguaggi di programmazione completi per basi di dati che offrono tutte le funzionalità sopra descritte: definizione dello schema, interrogazione e modifica dei dati, realizzazione di applicazioni complete. Questi linguaggi possono essere visti come dei linguaggi di programmazione arricchiti con la possibilità di definire alcuni dati come persistenti e la disponibilità di operatori e tipi di dati adatti a manipolare collezioni di dati omogenei. Un esempio di questi linguaggi verrà presentato nel Capitolo 8.

Base di dati e schema

Secondo la visione tradizionale, i dati di una base di dati ed il relativo schema corrispondono rispettivamente ad un insieme di variabili (che denotano insiemi modificabili di valori permanenti) che più applicazioni possono leggere e modificare in maniera concorrente, ed alla definizione del tipo di tali variabili, che tuttavia in questo caso è anch'essa permanente e in generale utilizzabile dalle applicazioni.

Gli orientamenti più recenti, invece, prevedono di trattare nello schema sia dati che procedure. In questa visione, uno schema è analogo ad un insieme di definizioni non solo di tipi e variabili, ma anche di procedure scritte nel linguaggio del DBMS, accessibili da tutte le applicazioni che fanno riferimento a tale schema. Esempi verranno mostrati nel Capitolo 7.

Sistema per la gestione di basi di dati

È il sistema, hardware e software, che consente la definizione e l'uso di basi di dati in un opportuno linguaggio. In altre parole, è la macchina astratta il cui linguaggio è il *linguaggio per basi di dati*. Esso, quindi, è analogo al compilatore, o l'interprete, di un certo linguaggio, più l'insieme dei moduli attivi durante l'esecuzione dei programmi (*run time system*). L'architettura e le tecniche di realizzazione dei DBMS saranno discusse nel Capitolo 9.

Si passa ora ad esaminare le funzionalità che caratterizzano un DBMS. Non tutti i sistemi offrono tutte le funzionalità che si prenderanno in considerazione, in particolare i DBMS previsti per calcolatori personali ne sacrificano alcune per ragioni di costo (tipicamente la gestione delle transazioni e l'accesso concorrente ai dati), ma l'elenco che segue include quelle funzionalità da considerarsi irrinunciabili per prodotti da usare nella gestione delle informazioni nelle organizzazioni.

1.5 Funzionalità dei DBMS

Si analizzano le principali funzionalità che un DBMS mette a disposizione per i seguenti scopi:

1. definizione della base di dati;
2. uso della base di dati;
3. controllo della base di dati;
4. distribuzione della base di dati;
5. amministrazione della base di dati.

1.5.1 Definizione della base di dati

Nei DBMS la base di dati è descritta separatamente dai programmi applicativi che ne fanno uso ed è utile distinguere tre diversi livelli di descrizione dei dati: il *livello fisico*, il *livello logico* e il *livello di vista logica*.

Al *livello fisico* viene descritto il modo in cui vanno organizzati fisicamente i dati nelle memorie permanenti e quali strutture dati ausiliarie definire per facilitarne l'uso. La descrizione di questi aspetti viene detta *schema fisico* o *interno* (*physical* o *internal schema*).

Al *livello logico* viene descritta la struttura degli insiemi di dati e delle relazioni fra loro, secondo un certo modello dei dati, senza nessun riferimento alla loro organizzazione fisica nella memoria permanente. La descrizione della struttura della base di dati viene detta *schema logico* (*logical schema*).

Al *livello di vista logica* viene definito come deve apparire la struttura della base di dati ad una certa applicazione. Questa descrizione viene anche detta *schema esterno* o *vista* (*external schema* o *user view*), per evidenziare il fatto che essa si riferisce a ciò che un utente immagina che sia la base di dati. Le differenze fra una vista e lo schema logico della base di dati riguardano sia gli insiemi di dati accessibili che la struttura dei dati. Mentre lo schema logico è unico, esistono in genere più schemi esterni, uno per ogni applicazione (o gruppo di applicazioni correlate), che permettono di vedere e modificare sottoinsiemi diversi, in generale non disgiunti, della base di dati (Figura 1.5).

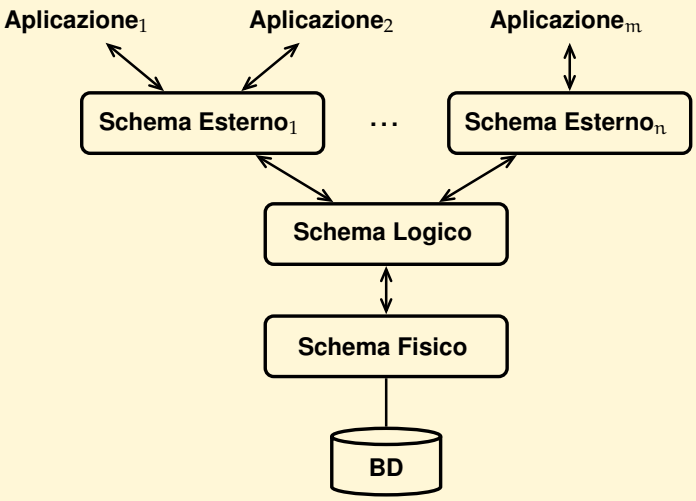


Figura 1.5: Livelli di descrizione dei dati

Esempio 1.2
Per chiarire la differenza fra i tre livelli di descrizione dei dati, si consideri una base di dati per gestire informazioni sui docenti di un'università, di supporto alle attività dell'ufficio stipendi e della biblioteca. Al livello di vista logica, l'ufficio stipendi richiede una vista dei dati sui docenti che include i seguenti attributi: nome e cognome, codice fiscale, parametro e stipendio. La biblioteca richiede invece una vista dei dati sui docenti che include i seguenti attributi: nome e cognome, recapito telefonico. Al livello logico, i dati sui docenti sono descritti da un unico insieme di registrazioni che includeranno gli attributi diversi che occorrono nelle due viste. Grazie al meccanismo degli schemi esterni, ogni applicazione vedrà poi solo i dati di sua competenza.

Ad esempio, lo schema logico di una base di dati relazionale è dichiarato come segue:

```
CREATE  TABLE Personale (
        Nome            CHAR(30),
        CodiceFiscale   CHAR(15),
        Stipendio       INTEGER,
        Parametro       CHAR(6),
        Recapito        CHAR(8) );
```

Al livello di vista logica, con un opportuno meccanismo di autorizzazioni, all'ufficio stipendi e alla biblioteca non viene consentito di accedere alla tabella Personale, ma di accedere solo ai dati di loro competenza definiti come

```
CREATE   VIEW     PersonalePerUfficioStipendi
      AS SELECT   Nome, CodiceFiscale, Stipendio, Parametro
         FROM     Personale;

CREATE   VIEW     PersonalePerLaBiblioteca
      AS SELECT   Nome, Recapito
         FROM     Personale;
```

Una *view* è una tabella calcolata da altre con opportuni operatori, che vedremo più avanti. Nell'esempio è stato usato solo l'operatore che proietta una tabella su alcune colonne rendendo così inaccessibili le altre; pertanto se agli addetti della biblioteca viene concessa l'autorizzazione ad usare solo i dati della tabella PersonalePerLaBiblioteca, essi potranno conoscere solo gli attributi Nome e Recapito del personale.

Infine, al livello fisico, il progettista della base di dati fisserà un'organizzazione fisica per l'insieme dei dati dei docenti descritto al livello logico, scegliendone una fra quelle previste dal DBMS: ad esempio, deciderà di memorizzare l'insieme in modo sequenziale oppure con una tecnica *hash* usando come chiave il nome:

```
MODIFY Personale TO HASH ON Nome
```

Naturalmente fra i tre livelli di descrizione dei dati devono esistere delle precise e dichiarate corrispondenze. Esse vengono utilizzate dal DBMS per convertire le operazioni sui dati "virtuali" accessibili da uno schema esterno, in quelle sui dati dello schema logico e, quindi, sui dati realmente presenti nel sistema, memorizzati secondo lo schema fisico.

I tre schemi dei dati sono gestiti dal sistema e non fanno parte dei programmi delle applicazioni. Un programma, prima di accedere alla base di dati, deve comunicare al sistema, con opportune modalità, a quale schema esterno fa riferimento. È come se, per programmare in un linguaggio ad alto livello, prima si definisse un modulo contenente le definizioni dei dati e poi si scrivessero le procedure specificando le dichiarazioni a cui fanno riferimento. Questo approccio ai DBMS è stato proposto nel '75 dal comitato ANSI/X3/SPARC (*Standards Planning and Requirements*), dell'*American National Standards Institute*, costituito con lo scopo di proporre una standardizzazione dell'architettura dei DBMS che aggiornasse la precedente del Codasyl-DBTG fatta nel 1969, ma i DBMS esistenti in commercio hanno finito per adottare approcci diversi.

L'approccio con tre livelli di descrizione dei dati è stato proposto come un modo per garantire le proprietà di *indipendenza logica* e *fisica* dei DBMS, che sono un obiettivo importante di questi sistemi.

Per *indipendenza fisica*, da leggere "indipendenza delle applicazioni dall'organizzazione fisica dei dati", si intende il fatto che non è necessario modificare i programmi applicativi quando si modifica l'organizzazione fisica dei dati. Il caso più frequente di modifica dell'organizzazione fisica dei dati si presenta quando occorre intervenire sulle strutture dati ausiliarie che agevolano il reperimento dei dati per migliorare le

prestazioni di certe applicazioni, oppure, nel caso di sistemi distribuiti, quando occorre cambiare il nodo della rete dove certi dati sono memorizzati per ridurre i costi di trasferimento.

Esempio 1.3
Si supponga che l'ufficio stipendi esegua con frequenza l'operazione di recupero dei dati riguardanti i docenti che abbiano un certo parametro. Se il numero dei docenti è basso, l'operazione potrebbe essere eseguita con ritardi accettabili visitando serialmente tutti i dati presenti, ma se il numero dei docenti cresce nel tempo, ad un certo punto questo modo di procedere comporterebbe tempi di risposta intollerabili. Occorre allora modificare lo schema fisico aggiungendo un indice sull'attributo Parametro:

```
CREATE INDEX IndiceParametro ON Personale(Parametro)
```

Per garantire l'indipendenza fisica non è necessario che il DBMS abbia un'architettura con tre livelli di descrizione dei dati, ma è sufficiente che gli operatori sulla base di dati disponibili agli utenti non dipendano dall'organizzazione fisica dei dati. L'indipendenza fisica corrisponde alla proprietà di indipendenza dei programmi dalla rappresentazione di un tipo di dato. Ad esempio, la rappresentazione di un tipo insieme può essere modificata da una struttura *hash* ad una ad albero senza effetti sui programmi che ne fanno uso, a patto di lasciare inalterata l'interfaccia del tipo di dato.

Per *indipendenza logica*, da leggere "indipendenza delle applicazioni dall'organizzazione logica dei dati", si intende il fatto che non è necessario modificare i programmi applicativi quando si modifica lo schema logico. Le modifiche possono essere l'aggiunta di nuove definizioni, la modifica o l'eliminazione di alcune di quelle esistenti.

Quindi, mentre l'indipendenza fisica rende le applicazioni indipendenti da modifiche dell'organizzazione fisica dei dati, l'indipendenza logica rende le applicazioni indipendenti da modifiche delle esigenze informative. Quanto sia ampio quest'ultimo tipo di indipendenza dipende dai meccanismi che offre il DBMS per definire la corrispondenza fra uno schema esterno, al quale fanno riferimento i programmi applicativi, e lo schema logico. Infatti, o la modifica non interessa un certo schema esterno, oppure, perché non vengano modificati i programmi, occorre poter ridefinire la relazione tra lo schema logico e lo schema esterno in modo da lasciare inalterata la visione della base di dati.

Esempio 1.4
Supponiamo che si decida di cambiare l'organizzazione logica dei dati sul personale memorizzandoli in due tabelle, Docenti e TecniciEAmministrativi. Per rendere

le applicazioni che usano la tabella Personale indipendenti da questa modifica, la base di dati si ridefinisce come segue:

```
CREATE   TABLE  Docenti (
         Nome           CHAR(30),
         CodiceFiscale  CHAR(15),
         Stipendio      INTEGER,
         Parametro      CHAR(6),
         Recapito       CHAR(8) );

CREATE   TABLE  TecniciEAmministrativi (
         Nome           CHAR(30),
         CodiceFiscale  CHAR(15),
         Stipendio      INTEGER,
         Parametro      CHAR(6),
         Recapito       CHAR(8) );

CREATE   VIEW       Personale
      AS SELECT     *
         FROM       Docenti
           UNION
         SELECT     *
         FROM       TecniciEAmministrativi;
```

Nei sistemi commerciali relazionali l'indipendenza fisica è di solito garantita e quella logica è garantita in una certa misura.

1.5.2 Uso della base di dati

Come conseguenza dell'integrazione dei dati, un DBMS deve prevedere più modalità d'uso per soddisfare le esigenze delle diverse categorie di utenti che possono accedere alla base di dati. Il valore della base di dati, infatti, dipende dalla facilità con cui può essere utilizzata e quindi dalle possibilità di accesso offerte dal sistema. Prendiamo in considerazione le esigenze di tre categorie di utenti: i *programmatori delle applicazioni*, gli *utenti non programmatori* e gli *utenti delle applicazioni*.

Programmatori delle applicazioni

Questa categoria comprende coloro che sviluppano le applicazioni che verranno poi utilizzate dagli "utenti delle applicazioni" per accedere o modificare i dati. Per lo sviluppo di queste applicazioni, un DBMS mette normalmente a disposizione alcune delle seguenti modalità:

1. utilizzo di "generatori di applicazioni", ovvero di strumenti i quali generano automaticamente il codice a partire da alcune definizioni date dal programmatore in modo dichiarativo o grafico. Ad esempio, questi strumenti permettono di indicare graficamente l'aspetto di menu e maschere nonché il codice da associare a determinate azioni dell'utente, e generano poi automaticamente il codice che realizza la modalità di interazione così specificata. Questi strumenti sono molto indicati per gestire gli aspetti di interazione delle applicazioni, ma il codice generato può avere bisogno di essere integrato manualmente quando si realizzano applicazioni complesse;
2. programmazione in un linguaggio tradizionale esteso con gli operatori del DML del sistema; normalmente il programmatore può scegliere tra alcuni linguaggi diversi, quali, ad esempio, C e Java. Gli operatori predefiniti del modello dei dati possono essere codificati nel linguaggio ospite secondo diverse modalità:
   - come procedure predefinite;
   - come costrutti da preelaborare, per convertire i nuovi costrutti in chiamate a procedure predefinite, prima di sottoporre il programma alla traduzione con il compilatore tradizionale del linguaggio ospite;
   - come nuovi costrutti, e il compilatore del linguaggio ospite viene esteso per trattare anche questi, traducendoli in chiamate a procedure predefinite.

   Questo approccio presenta alcuni problemi, dovuti alla necessità di effettuare conversioni improduttive tra il formato con cui i dati vengono rappresentati nel linguaggio ospite ed il formato con cui essi sono gestiti dal DBMS. Ad esempio, il DBMS gestisce collezioni di dati, che non hanno un corrispettivo diretto nei linguaggi di programmazione tradizionali, i quali a loro volta offrono tipi di dati, quali i puntatori o le liste, che non possono essere trasferiti direttamente nella base di dati. Per quanto riguarda invece la *comunicazione* fra il programma applicativo e il DBMS (scambio di dati e messaggi sull'esito delle operazioni), in generale si ricorre ad aree di lavoro comuni (*user working area, UWA*), viste dai programmi come un insieme di variabili predefinite. Il trasferimento dei dati dalla base di dati al programma, e viceversa, è causato dall'attivazione degli operatori sulla base di dati (Figura 1.6);

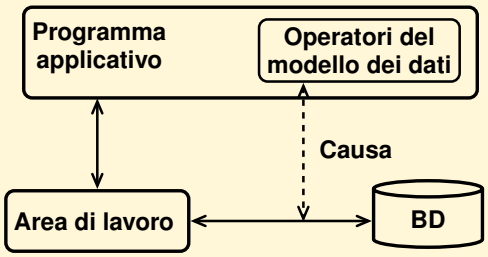


Figura 1.6: Scambio dei dati fra programmi e base di dati

3. programmazione con un "linguaggio per basi di dati", ovvero con un linguaggio completo che integri le caratteristiche di un linguaggio di programmazione con i meccanismi di definizione ed uso della base di dati, quali cosiddetti *linguaggi della quarta generazione* (*fourth generation languages, 4GL*) definiti per i sistemi relazionali. Questi linguaggi risolvono i problemi di cattiva integrazione tra linguaggio ospite e DML discussi al punto precedente.

Un'ultima caratteristica che distingue un linguaggio per l'uso dei dati, in particolare per la parte riguardante la ricerca, consiste nel fatto che è *dichiarativo* (*non procedurale*). Con questo termine si intende che gli operatori del linguaggio agiscono su insiemi di dati, mentre l'utente, per individuare i dati che interessano, specifica la condizione che essi devono soddisfare senza dettagliare i passi della ricerca, che vengono invece stabiliti automaticamente dal sistema.

Utenti non programmatori

In questa categoria rientrano coloro che richiedono un linguaggio interattivo a sé stante, di facile uso, per fare principalmente ricerche di dati. Sono quindi utili anche strumenti per (*a*) formulare le richieste, (*b*) definire in modo dichiarativo il formato in cui vanno stampati i risultati delle ricerche (*report generators*), (*c*) visualizzare il contenuto della base di dati e i risultati di ricerche in forma grafica (istogrammi, diagrammi, torte ecc.). I linguaggi più efficaci, in particolare quelli che prevedono l'uso della grafica e interfacce amichevoli, sono stati sviluppati grazie alla diffusione dei sistemi relazionali sui calcolatori personali. Sono inoltre disponibili per questi utenti semplici strumenti interattivi per l'importazione dei dati di una base di dati all'interno di fogli di calcolo o di altri programmi per la generazione di grafici e tabelle.

Utenti delle applicazioni

Sono coloro che richiedono delle modalità molto semplici per attivare un numero predefinito di operazioni, senza avere nessuna competenza informatica. Esempi sono gli impiegati addetti agli sportelli di una banca o alle prenotazioni di una compagnia aerea. In questi casi un'operazione è invocata interattivamente, con uso di un terminale video: l'utente agisce selezionando una delle possibili scelte proposte (menu), e fornisce i valori degli argomenti riempendo campi di opportune "maschere".

1.5.3 Controllo della base di dati

Una caratteristica molto importante dei DBMS è il tipo di meccanismi offerti per garantire le seguenti proprietà di una base di dati: *integrità*, *affidabilità* e *sicurezza*.

Integrità

I DBMS prevedono dei meccanismi per controllare che i dati inseriti, o modificati, siano conformi alle definizioni date nello schema, in modo da garantire che la base di

dati si trovi sempre in uno stato *consistente*. Il linguaggio per la definizione dello schema logico consente di definire non solo la struttura dei dati, ma anche le condizioni a cui essi devono sottostare per essere significativi (*vincoli d'integrità*), quando queste condizioni vanno verificate e cosa fare in caso di violazioni. Vedremo degli esempi nel Capitolo 7.

Affidabilità

I DBMS devono disporre di meccanismi per proteggere i dati da malfunzionamenti hardware o software e da interferenze indesiderate dovute ad accessi concorrenti ai dati da parte di più utenti. A tal fine, un DBMS prevede che le interazioni con la base di dati avvengano per mezzo di *transazioni* (Figura 1.7), cioè con un meccanismo che:

- esclude effetti parziali dovuti all'interruzione delle applicazioni per una qualsiasi ragione;
- garantisce l'assenza di interferenze nel caso di accessi concorrenti ai dati.

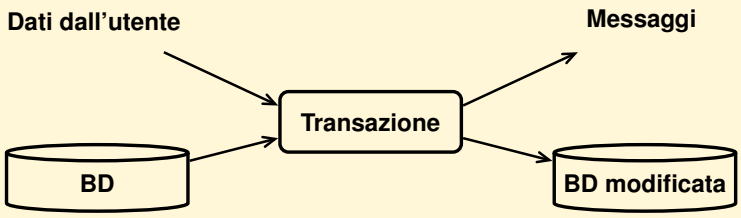


Figura 1.7: Dati e risultati di una transazione

Più precisamente vale la seguente definizione:

Definizione 1.4

Una transazione è una sequenza di azioni di lettura e scrittura della base di dati e di elaborazioni di dati in memoria temporanea, che il DBMS esegue garantendo le seguenti proprietà:[2]

Atomicità. Solo le transazioni che terminano normalmente (*committed transactions*) fanno transitare la base di dati in un nuovo stato. Le transazioni che terminano prematuramente (*aborted transactions*) sono trattate dal sistema come se non fossero mai iniziate; pertanto eventuali loro effetti parziali sulla base di dati sono annullati.

2. Si possono scegliere altre proprietà per caratterizzare le transazioni. Nella letteratura in lingua inglese di solito si preferiscono le seguenti: *Atomicity*, *Consistency*, *Isolation*, e *Durability*, che producono l'acronimo ACID.

Serializzabilità. L'effetto sulla base di dati dell'esecuzione concorrente di più transazioni è equivalente ad un'esecuzione seriale delle transazioni, cioè ad una esecuzione in cui le transazioni vengono eseguite una dopo l'altra in un qualche ordine.

Persistenza. Le modifiche sulla base di dati di una transazione terminata normalmente sono permanenti, cioè non sono alterabili da eventuali malfunzionamenti successivi alla terminazione.

Una transazione può essere espressa come un insieme di espressioni in un linguaggio di interrogazione, oppure come un programma sequenziale che opera sulla base di dati; in entrambi i casi esiste qualche meccanismo per segnalare al sistema il punto di inizio ed il punto finale della transazione. L'esecuzione di una transazione comporta il trasferimento di dati fra le memorie temporanea (buffer) e permanente.

Un *malfunzionamento* (*failure*) è un evento a causa del quale la base di dati può trovarsi in uno stato scorretto. Verrà fatta l'ipotesi che la manifestazione di un'anomalia sia sempre rilevata e ciò comporti:

1. l'interruzione istantanea di una transazione o dell'intero sistema, a seconda del tipo di malfunzionamento verificatosi;
2. l'attivazione di opportune procedure che permettano di riportare la base di dati allo stato corretto precedente alla manifestazione del malfunzionamento (*procedure di ripristino della base di dati, "recovery"*).

Si distinguono tre tipi di malfunzionamenti in base al tipo di memoria interessata: *fallimenti di transazioni* (*transaction failure*), *fallimento di sistema* (*system failure*) e *disastri* (*media* (*disk*) *failure*).

Definizione 1.5

I *fallimenti di transazioni* sono interruzioni di transazioni che non comportano perdite di dati né in memoria temporanea (*buffer*) né in memoria permanente.

I fallimenti di transazioni sono dovuti a situazioni già previste nei programmi, il cui verificarsi comporta la terminazione prematura della transazione; oppure a situazioni non previste nei programmi, il cui verificarsi causa la terminazione prematura della transazione da parte del sistema. Esempi sono violazione di vincoli di integrità, tentativo di accesso a dati protetti, condizioni di stallo.

Definizione 1.6

I *fallimenti di sistema* sono interruzioni del suo funzionamento dovuti ad un'anomalia hardware o software dell'unità centrale o di una periferica, con conseguente interruzione di tutte le transazioni attive. Si assume che il contenuto della memoria permanente sopravviva, mentre si considera perso il contenuto della memoria temporanea.

Esempi di questo genere di malfunzionamento sono l'interruzione dell'alimentazione elettrica del sistema ed errori del software di base (DBMS e sistema operativo).

Definizione 1.7
I *disastri* sono malfunzionamenti che danneggiano la memoria permanente contenente la base di dati.

Possibili cause possono essere il danneggiamento delle periferiche o un errore umano che rende irrecuperabile il contenuto della base di dati. I disastri influenzano tutte le transazioni che stanno usando la porzione di base di dati danneggiata.

Compito delle procedure di ripristino è di garantire che la base di dati contenga tutte e sole le modifiche apportate dalle transazioni terminate con successo prima dell'occorrenza del malfunzionamento.

Per poter eseguire queste procedure un DBMS mantiene una copia di sicurezza della base di dati e tiene traccia di tutte le modifiche fatte sulla base di dati dal momento in cui è stata eseguita l'ultima copia di sicurezza. Grazie a questi dati ausiliari, quando si verifica un malfunzionamento il DBMS può ricostruire una versione corretta dei dati utilizzando l'ultima copia e rieseguendo tutte le operazioni che hanno modificato i dati e di cui ha mantenuto traccia.

Altra funzionalità di un DBMS è di consentire l'esecuzione *concorrente* di più transazioni, risolvendo il problema di farle funzionare correttamente, senza interferenze indesiderate quando esse operano sugli stessi dati (*controllo della concorrenza*). Il classico esempio di interferenza è quello che porta alla *perdita di modifiche*.

Esempio 1.5
Si supponga che Antonio e Giovanna condividano un conto corrente e che contemporaneamente facciano un prelievo e un versamento da sportelli diversi. Sia 350 il saldo, 400 la somma che Giovanna versa e 50 la somma che Antonio preleva. Supponiamo che sulla base di dati si verifichino i seguenti eventi, nell'ordine mostrato:

1. Il cassiere di Giovanna legge il saldo 350.
2. Il cassiere di Antonio legge il saldo 350.
3. Il cassiere di Giovanna modifica il saldo in 750.
4. Il cassiere di Antonio modifica il saldo in 300.

L'effetto dell'operazione di Giovanna è annullato da quello dell'operazione di Antonio e il saldo finale è di 300.

Per evitare interferenze indesiderate il DBMS coordina opportunamente l'esecuzione concorrente di un insieme di transazioni $\{T_1, \ldots, T_n\}$, intercalando opportunamente nel tempo le azioni sulla base di dati di ogni transazione, in modo che l'effetto dell'esecuzione sia quello ottenibile eseguendo le transazioni isolatamente, in un qualche ordine.

Un modo molto semplice di risolvere il problema sarebbe quello di eseguire le transazioni isolatamente, cioè in modo tale che, per ogni coppia di transazioni T_i e T_j, tutte le azioni di T_i precedono quelle di T_j, o viceversa (*esecuzione seriale*). Questa soluzione impedirebbe però ogni forma di concorrenza riducendo il DBMS ad un elaboratore

seriale di transazioni. Vedremo nel Capitolo 9 le tecniche solitamente usate dai DBMS sia per gestire le transazioni che la concorrenza.

Sicurezza

I DBMS prevedono meccanismi sia per controllare che ai dati accedano solo persone autorizzate, sia per restringere i dati accessibili e le operazioni che si possono fare su di essi. Il problema presenta diversi aspetti, alcuni dei quali simili a quelli affrontati nell'ambito dei sistemi operativi, altri tipici di questa classe di applicazioni.

Aspetti del primo tipo sono ad esempio l'identificazione degli utenti autorizzati, con parole di riconoscimento, oppure la possibilità di proteggere i dati da furti mediante crittografia. Per mostrare aspetti specifici dell'area base di dati, immaginiamo di disporre di dati riguardanti cittadini, fra cui il codice fiscale, dati anagrafici e il reddito. Eventuali restrizioni che si potrebbero imporre a categorie diverse di utenti sono:

1. certi utenti non possono accedere a questi dati, ma solo ad altri presenti nella base di dati;
2. certi utenti possono accedere ai dati, ma non possono modificarli;
3. certi utenti possono accedere solo ai dati che li riguardano, senza modificarli;
4. come nel caso precedente, ma si possono modificare solo i dati anagrafici;
5. certi utenti possono accedere solo ai dati anagrafici, da certi uffici e in certe ore del giorno;
6. certi utenti possono applicare solo operazioni statistiche sul reddito, ma non possono accedere a dati singoli né fare modifiche.

Questa lista potrebbe continuare e diventare ancora più significativa se si considerassero più insiemi di dati in relazione logica, ma è sufficiente per dare un'idea della complessità e flessibilità di un meccanismo per garantire la sicurezza dei dati. Una soluzione ad alcuni dei problemi è data dallo schema esterno ed altre verranno viste quando presenteremo i sistemi commerciali.

Questo problema diventa ancora più complesso nel caso di basi di dati per uso statistico: si vuol concedere la possibilità di conoscere, ad esempio, la media dei redditi delle persone, ma non il reddito di una particolare persona. Notare che non è sufficiente imporre che si possano soddisfare solo richieste riguardanti insiemi perché certe informazioni riservate potrebbero essere ricavate lo stesso per deduzione. Ad esempio, per conoscere il reddito di Albano, Orsini potrebbe chiedere il reddito medio dell'insieme {Albano, Orsini} e, noto il proprio reddito, dedurre ciò che cercava di sapere.

Si noti, infine, che la legislazione negli ultimi anni è diventata sempre più attenta al trattamento dei dati dei cittadini (leggi sulla *privacy*), in particolare dei cosiddetti dati *sensibili* (ad esempio i dati sanitari), richiedendo particolari accorgimenti per l'accesso e la protezione.

1.5.4 Distribuzione della base di dati

Un DBMS moderno deve garantire la possibilità di distribuire i dati gestiti dal sistema su più elaboratori collegati tra di loro da una rete locale o geografica, eventualmente replicando alcuni dati. La distribuzione dei dati su reti geografiche è utile ad aziende con più sedi per poter gestire in ognuna di queste i dati di interesse locale, mantenendo la possibilità di accedere a tutti i dati dell'organizzazione. Infine, entrambi i tipi di distribuzione, se combinati con la replicazione di alcuni dati, permettono di continuare ad operare sui dati anche quando un elaboratore sia fuori servizio. Il sistema mette a disposizione in questo caso:

1. strumenti per la progettazione e definizione dello schema che permettono di definire la locazione dei dati e di valutare gli effetti di una diversa distribuzione degli stessi;
2. linguaggi per l'uso dei dati che permettono di scrivere applicazioni prescindendo dalla locazione dei dati stessi;
3. strumenti per il controllo dei dati che permettono di gestire la concorrenza ed i fallimenti di transazioni che vengono eseguite su più elaboratori, garantendo in particolare la coerenza dei dati replicati;
4. strumenti per l'amministrazione di un sistema distribuito, in particolare per verificare gli effetti della distribuzione dei dati sulle prestazioni delle applicazioni.

1.5.5 Amministrazione della base di dati

L'amministratore della base di dati (*Data Base Administrator, DBA*) è una persona (o un gruppo di persone) che, dopo aver partecipato allo studio di fattibilità per decidere l'impiego di un DBMS, ne seleziona uno, lo mette in funzione, lo mantiene in esercizio e segue ogni base di dati dalla progettazione all'impiego. In particolare, quindi, ha bisogno di strumenti per svolgere le seguenti attività:

1. analisi dei requisiti di nuove applicazioni e progettazione, sviluppo e manutenzione di basi di dati e delle applicazioni che ne fanno uso;
2. definizione degli schemi di basi di dati (logici, fisici ed esterni), delle autorizzazioni e modalità di accesso ai dati per ogni classe di utenti e delle politiche per la sicurezza dei dati;
3. manutenzione della struttura logica e fisica dei dati per correggere errori di progettazione o per adeguarle a nuove esigenze;
4. definizione delle procedure per il caricamento dei dati, la creazione di copie di sicurezza, il ripristino dei dati dopo malfunzionamenti di sistema o disastri;
5. controllo del funzionamento del sistema per decidere eventuali riorganizzazioni della struttura logica e fisica dei dati al fine di migliorare le prestazioni delle applicazioni;
6. pianificazione della formazione del personale e definizione di standard per la programmazione e prova delle applicazioni.

Un importante strumento di ausilio per l'amministrazione di basi di dati è il cosiddetto *catalogo* del sistema, che contiene informazioni su ciò che è definito nella base di dati, su quali definizioni operano le applicazioni e su come sono memorizzati i dati. In altre parole, mentre con gli schemi si descrivono i dati presenti nel sistema e le relazioni fra loro, con il catalogo si raccolgono dati riguardanti gli oggetti descritti. Un catalogo contiene pertanto sia informazioni d'interesse degli utenti e dell'amministratore della base di dati, sia informazioni sui dati memorizzati utilizzabili dal DBMS per il suo funzionamento. Un catalogo è organizzato come una base di dati aggiornata automaticamente dal sistema. Altri utili strumenti per assistere l'amministratore della base di dati nello svolgimento di tutte le attività elencate sono forniti dai produttori di DBMS, o da ditte indipendenti, e saranno discussi nel Capitolo 7.

La progettazione di basi di dati viene discussa nel Capitolo 3 con un approccio indipendente dal tipo di DBMS da usare (progettazione concettuale). La realizzazione di basi di dati relazionali viene discussa nel Capitolo 4 con un approccio che trasforma il progetto concettuale dei dati in uno schema relazionale e nel Capitolo 5 con un approccio formale detto di *normalizzazione*, proposto inizialmente per essere usato in alternativa all'approccio per trasformazione, ma che di solito si usa in modo complementare. La definizione e amministrazione di basi di dati relazionali sono discusse nel Capitolo 7 .

1.6 Vantaggi e problemi nell'uso dei DBMS

La tecnologia delle basi di dati, oltre ai vantaggi visti (integrazione dei dati e flessibilità della base di dati), ne offre anche altri, in particolare quello di stabilire degli standard riguardo alla strutturazione e nomenclatura dell'informazione, e di ridurre notevolmente tempi di sviluppo delle applicazioni rispetto a quelli richiesti usando la tecnologia degli archivi. Il problema degli standard è molto sentito in grandi organizzazioni dove la molteplicità delle esigenze richiede una terminologia e un modo comune di definire i dati per facilitare le comunicazioni e la cooperazione fra gli utenti.

L'impiego dei DBMS comporta anche dei problemi che vanno tenuti presenti nel valutare l'opportunità della loro adozione. Questi problemi si avvertono in particolare quando si tratta di realizzare sistemi informatici complessi, ma non vanno sottovalutati nemmeno nei casi più semplici, sebbene la continua riduzione dei costi della tecnologia informatica renda sempre più conveniente l'impiego di questi prodotti.

Problemi gestionali e organizzativi

1. I DBMS più sofisticati richiedono un notevole impegno di risorse hardware e software per la loro messa a punto ed il loro funzionamento.
2. Il costo di questi sistemi, ed i costi di esercizio, possono essere facilmente sottostimati.
3. È importante acquisire e mantenere personale qualificato e con competenze specifiche sul sistema impiegato.

4. L'introduzione del sistema ha un impatto sulla struttura organizzativa.

Problemi di produzione del software

1. Il progetto di base di dati e la messa a punto delle applicazioni richiedono personale qualificato e strumenti opportuni, che non sono messi a disposizione da tutti i sistemi.
2. L'impiego di una base di dati richiede una ristrutturazione dei dati in archivi già esistenti e la riscrittura dei programmi applicativi.
3. L'impiego di un DBMS può aumentare la dipendenza da ditte esterne all'impresa per lo sviluppo delle applicazioni.

Problemi di funzionamento

1. Una conseguenza della centralizzazione è l'aumento del rischio di interruzione dei servizi. Infatti, se il sistema non è disponibile, non lo è per nessuna applicazione.
2. Per non degradare i tempi di risposta, i dati vanno organizzati con la massima attenzione.

Problemi di pianificazione

I costi di acquisizione della tecnologia delle basi di dati e di sviluppo delle applicazioni sono tali da rendere non praticabile la sostituzione di un sistema con un nuovo prodotto non compatibile. Pertanto la scelta del tipo di DBMS va fatta pianificando l'intervento accuratamente.

1.7 Conclusioni

È stata introdotta la nozione di base di dati e sono state discusse le funzionalità che caratterizzano un sistema per la gestione di basi di dati. Questo tipo di sistema è la tecnologia più adatta per sviluppare applicazioni che usano in modo concorrente dati persistenti, in contesti in cui sia necessario accedere alle stesse informazioni da parte di più applicazioni e di più settori di un'organizzazione, e dove è prevista un'evoluzione nel tempo delle esigenze di archiviazione e gestione di informazioni. La complessità di questo tipo di sistema è dovuta alla varietà di funzionalità che deve offrire per consentire di organizzare, e proteggere da malfunzionamenti, dati persistenti da rendere facilmente accessibili a utenti specialistici e non, senza sacrificare l'efficienza delle operazioni.

I concetti introdotti in questa rapida panoramica sono molti e riguardano aspetti fondamentali delle basi di dati che saranno ripresi e trattati diffusamente nei prossimi capitoli.

Esercizi

1. Discutere le differenze fra un sistema per la gestione di basi di dati e un sistema di archiviazione.
2. Elencare alcune domande (fra cinque e dieci) da fare ad un produttore di sistemi per la gestione di dati al fine di stabilire se il sistema che propone può essere classificato come un sistema per la gestione basi di dati centralizzate.
3. Discutere vantaggi e svantaggi di un sistema per la gestione di basi di dati.
4. Spiegare la differenza tra i seguenti termini: base di dati e sistema per la gestione di basi di dati.
5. Discutere i concetti di indipendenza logica e fisica, confrontandoli con i concetti di modulo e tipo di dato astratto dei linguaggi di programmazione.
6. Discutere le differenze tra il modello dei dati di un sistema di archiviazione e di un sistema per basi di dati.
7. Discutere i compiti del programmatore delle applicazioni e dell'amministratore della base di dati.
8. Quali delle seguenti affermazioni è vera?
   a) Sistema informatico è un sinonimo di sistema informativo.
   b) Un linguaggio di interrogazione richiede la conoscenza di un linguaggio di programmazione.
   c) Per usare correttamente una base di dati l'utente (persona o programma) deve conoscere l'organizzazione fisica dei dati.
   d) L'organizzazione fisica di una base di dati va programmata dall'amministratore della base di dati.
   e) Schema logico, schema fisico e schema esterno sono sinonimi.
   f) Per soddisfare le esigenze degli utenti delle applicazioni non occorre un linguaggio di programmazione.
   g) Le transazioni nei sistemi per basi di dati hanno le stesse proprietà dei programmi nei linguaggi con archivi.
   h) Per realizzare un sistema informatico il personale tecnico realizza innanzitutto applicazioni che usano basi di dati.
   i) Il programmatore delle applicazioni decide quali sono i dati accessibili agli utenti.

Note bibliografiche

Sono numerosi i libri che trattano i sistemi per la gestione di basi di dati. Fra i più recenti in italiano si segnalano [Atzeni et al., 2002], [Ramakrishnan and Gehrke, 2003], e [Elmasri and Navathe, 2001]. Per approfondimenti, riferimenti interessanti in lingua inglese sono [Abiteboul et al., 1995], [Silberschatz et al., 2002], [Kifer et al., 2005], [Ullman and Widom, 2001].

Capitolo 2

I MODELLI DEI DATI

Sfruttare la tecnologia informatica nella gestione delle informazioni significa essenzialmente costruire un modello di una realtà di interesse, che chiameremo *universo del discorso*, con l'obiettivo di lavorare poi sul modello per riprodurre o predire l'evoluzione della situazione oggetto di studio. I modelli ricorrono ampiamente nella tecnologia e in ogni campo che richiede un'attività di progettazione. Essi permettono di riprodurre le caratteristiche essenziali di fenomeni reali, omettendo quei dettagli che, ai fini dello studio che ci si prefigge, costituirebbero un'inutile complicazione.

In questo capitolo si chiariranno innanzitutto i diversi aspetti del processo di modellazione, poi si esamineranno i fatti che si modellano, per passare infine a vedere come questi si rappresentano utilizzando un modello dei dati a oggetti ed un semplice formalismo grafico, adatto per costruire modelli di analisi.

2.1 Progettazione e modellazione

Nel progettare una base di dati è conveniente immaginare che l'organizzazione committente voglia raccogliere informazioni riguardanti lo stato di una porzione della realtà, l'universo del discorso. In questa situazione la base di dati deve essere strutturata in modo tale che essa rappresenti un *modello astratto* o *simbolico* di questo universo del discorso, dove il termine modello astratto si definisce come segue.

Definizione 2.1
Un modello astratto è la rappresentazione formale di idee e conoscenze relative ad un fenomeno.

Questa definizione evidenzia tre aspetti fondamentali di un modello astratto:

1. un modello è la *rappresentazione* di alcuni fatti di un fenomeno;
2. la rappresentazione è data con un *linguaggio formale*;
3. il modello è il risultato di un processo di *interpretazione* di un fenomeno, guidato dalle *idee* e *conoscenze* già possedute dal soggetto che interpreta.

La progettazione della base di dati avviene in più fasi, come verrà descritto nel prossimo capitolo, ciascuna delle quali produce un progetto sempre più dettagliato, fino ad arrivare alla realizzazione del sistema. È utile descrivere queste fasi dicendo che ciascuna di esse costruisce un modello dell'universo del discorso raffinando il modello emerso dalla fase precedente. Nelle prime fasi si utilizzano formalismi più astratti

e più vicini a come la realtà è percepita dagli esseri umani, mentre nelle fasi finali si utilizzano formalismi in cui è possibile specificare un maggior numero di dettagli, e che sono direttamente interpretabili da un elaboratore.

2.2 Considerazioni preliminari alla modellazione

Prima di affrontare la costruzione del modello di un universo del discorso è necessario porsi le seguenti domande: "cosa si modella?", "come si modella? ovvero con quali strumenti concettuali si modella?", "con quale linguaggio formale si definisce il modello?", "come si procede nella costruzione del modello?". Le risposte a queste domande caratterizzano i seguenti aspetti della modellazione: *ontologico*, *linguistico astratto*, *linguistico concreto* e *pragmatico*.

2.2.1 Aspetto ontologico

L'aspetto ontologico (studio di ciò che esiste) riguarda ciò che si suppone esistere nell'universo del discorso e quindi sia da modellare. Molte ipotesi sono possibili a questo riguardo, e quella di solito preferita quando si utilizza la tecnologia delle basi di dati è che vadano modellati i seguenti fatti: la *conoscenza concreta*, la *conoscenza astratta*, la *conoscenza procedurale* e la *comunicazione*.

La conoscenza concreta

La *conoscenza concreta* riguarda i fatti specifici che si vogliono rappresentare. Adottando un approccio semplificato, si suppone che la realtà consista di *entità*, che hanno un *tipo* ed alcune *proprietà*. Si suppone inoltre che queste entità siano classificabili all'interno di *collezioni* di entità omogenee, e che esistano *istanze di associazioni* fra queste entità.

Questa visione della realtà, pur così schematica, non è lontana da alcune proposte sviluppate nell'ambito della riflessione filosofica; un esempio, tra i tanti possibili, è il seguente:

> Tutte le cose nominabili, esterne alla mente, si ritengono appartenenti o alla classe delle sostanze o a quella degli attributi.
>
> Un attributo, dicono i logici Scolastici, dev'essere attributo di qualche cosa; un colore, per esempio, dev'essere colore di qualche cosa. Se questo qualche cosa cessasse di esistere, o cessasse di essere connesso con quell'attributo, l'esistenza dell'attributo sarebbe finita.
>
> Una sostanza, invece, esiste di per sé; parlandone non occorre che poniamo "di" dopo il suo nome. Una pietra non è la pietra di qualcosa.
>
> Abbiamo detto che le qualità d'un corpo sono gli attributi, fondati sulle sensazioni che la presenza di quel particolare corpo ai nostri organi eccita nelle nostre menti. Ma quando attribuiamo ad un qualche oggetto lo speciale attributo chiamato relazione, il fondamento dell'attributo dev'essere qualche cosa in cui sono interessati altri oggetti, oltre all'oggetto stesso e al soggetto percipiente.

— John Stuart Mill, *System of Logic, Ratiocinative and Inductive*, Parker, Son, and Bourn, West Strand, London, Fifth Edition, 1862 (trad. it. di G. Facchi, *Sistema di Logica, raziocinativa e induttiva*, Ubaldini Editore, Roma, 1984).

Si dà ora una descrizione dei termini entità, proprietà e associazione, avvertendo che questa descrizione sarà in qualche misura discutibile e imprecisa, data la natura essenzialmente filosofica del problema ontologico.

Definizione 2.2
Un'*entità* è qualcosa di concreto o astratto di cui interessa rappresentare alcuni fatti.

Esempi tipici di entità sono oggetti concreti (un libro, un utente della biblioteca), oggetti astratti (la descrizione bibliografica di un libro) ed eventi (un prestito). Ciò che interessa di un'entità sono le sue *proprietà*.

Definizione 2.3
Il valore di una *proprietà* è un fatto che descrive una qualità di un'entità.

Esempi di proprietà sono il nome ed il recapito di un utente oppure l'autore, il titolo, l'editore e l'anno di pubblicazione di un libro.

La differenza che esiste tra una proprietà ed un'entità scaturisce dalla diversa interpretazione del loro ruolo nel modello: i valori delle proprietà non sono fatti che interessano di per sé, ma solo come caratterizzazione di altri concetti interpretati come entità.

Un'entità non coincide con l'insieme dei valori assunti dalle sue proprietà, poiché:

- i valori delle proprietà di un'entità cambiano in generale nel tempo, anche se l'entità resta la stessa (si pensi, ad esempio, all'età di una persona);
- due entità possono avere le stesse proprietà con gli stessi valori ma essere ugualmente due entità distinte (si pensi, ad esempio, a due persone diverse con lo stesso nome, cognome ed età).

Il concetto di proprietà è strettamente collegato al concetto di *tipo*.

Definizione 2.4
Un *tipo* è una descrizione astratta di ciò che accomuna un insieme di entità omogenee (della stessa natura), esistenti o possibili.

Ad esempio, Persona è il tipo di Giovanna, Mario ecc.

Un tipo non è una collezione di entità esistenti, ma descrive la natura di tutte le entità "possibili" o "concepibili" con tale natura. Ad esempio, il tipo Persona descrive non solo tutte le persone esistenti, ma anche quelle che esisteranno o che potrebbero esistere; quindi un tipo può essere visto come una collezione *infinita* di entità possibili. Ad un tipo sono associate le proprietà che interessano delle entità che appartengono a tale tipo, nonché le *caratteristiche* di tali proprietà. Ad esempio, il tipo Utente ha le proprietà Nome e Recapito intendendo con questo che ogni utente ha un nome e

un recapito, ma con un valore in generale diverso da quello di tutti gli altri. Per ciò che riguarda le *caratteristiche* delle proprietà, ogni proprietà ha associato un *dominio*, ovvero l'insieme dei possibili valori che tale proprietà può assumere, e può essere inoltre classificata come segue:

1. proprietà *atomica*, se il suo valore non è scomponibile (ad esempio, il nome di una persona); altrimenti è detta *strutturata* (ad esempio, la proprietà residenza è scomponibile in indirizzo, CAP, città);
2. proprietà *univoca*, se il suo valore è unico (ad esempio, il nome di un utente ha un unico valore); altrimenti è detta *multivalore* (ad esempio, la proprietà recapiti telefonici di una persona è multivalore se ammettiamo che le persone possano essere raggiungibili attraverso diversi numeri telefonici);
3. proprietà *totale* (*obbligatoria*), se ogni entità dell'universo del discorso ha per essa un valore specificato, altrimenti è detta *parziale* (*opzionale*) (ad esempio, si può considerare il nome di un utente una proprietà totale ed il suo recapito telefonico una proprietà parziale).

Si osservi che non solo tutte queste caratteristiche sono indipendenti tra di loro, ma anche i campi di una proprietà strutturata possono essere nuovamente classificati nello stesso modo; ad esempio, nella residenza, il campo indirizzo può essere a sua volta strutturato in via, o piazza, e numero civico.

Altro concetto interessante è quello di *collezione*.

Definizione 2.5

Una *collezione* è un insieme variabile nel tempo di entità omogenee interessanti dell'universo del discorso.

Ad esempio, i libri di una biblioteca sono la collezione dei libri da essa posseduta. L'insieme degli elementi di una collezione in un determinato momento è detto *estensione* della collezione.

Quindi, mentre un tipo descrive tutte le infinite possibili entità con certe qualità, una collezione è un insieme finito e variabile di entità, dello stesso tipo, che fanno effettivamente parte dell'universo del discorso. Si osservi che, nel realizzare un modello, non si rappresentano tutte le possibili collezioni di entità, così come non si rappresentano tutte le entità che esistono nell'universo del discorso; nel modello si rappresentano solo quelle entità e quelle collezioni che sono considerate interessanti rispetto agli obiettivi per cui viene costruito il modello.

Un altro importante tipo di informazione da modellare è l'eventuale esistenza di *gerarchie di specializzazione* (o *di generalizzazione*, a seconda del verso di percorrenza della gerarchia), tra diverse collezioni di entità: le collezioni in gerarchia modellano insiemi di entità ad un diverso livello di dettaglio. Se una collezione C_{sub} della gerarchia è una specializzazione della collezione C_{sup}, C_{sub} è un sottoinsieme di C_{sup} e gli elementi di C_{sub} ereditano le caratteristiche degli elementi C_{sup}, oltre ad averne altre proprie.

L'uso delle gerarchie di collezioni è molto comune, ad esempio, per classificare gli organismi animali e vegetali: quando diciamo che i marsupiali sono mammiferi

intendiamo che le femmine sono dotate di ghiandole mammarie per l'allattamento dei piccoli, proprietà di ogni mammifero, ma hanno come proprietà specifica il marsupio.

Nell'esempio della biblioteca, la collezione degli Utenti può essere pensata come una generalizzazione delle collezioni Studenti e Docenti, per rappresentare il fatto che entità classificate come elementi di Studenti e Docenti possono essere classificate anche come elementi di Utenti prescindendo dalle proprietà che le rendono semanticamente diverse, ed evidenziando invece che sono entità omogenee ad un diverso livello di astrazione, ad esempio con cognome, residenza e data di nascita come proprietà comuni. Si dice anche che Studenti e Docenti sono *specializzazioni* di Utenti.

Nel processo di modellazione è critico sia stabilire che cosa descrivere come una proprietà o come un'entità, sia stabilire una corretta gerarchia di collezioni per il problema in esame.

Due o più entità possono essere collegate da un'*istanza di associazione*.

Definizione 2.6

Un'*istanza di associazione* è un fatto che correla due o più entità, stabilendo un legame logico fra di loro.

Esempi di istanze di associazioni sono quelle descritte dai seguenti fatti: "lo studente Rossi ha in prestito il libro la Divina Commedia", "lo studente Bianchi frequenta il corso di Basi di dati". Nel primo caso le entità Rossi e la Divina Commedia sono associate dal fatto espresso dal verbo "ha in prestito", nel secondo caso le entità Bianchi e Basi di dati sono associate dal fatto espresso dal verbo "frequenta".

Un'istanza di associazione può correlare anche più di due entità (associazione n-aria), come nel fatto: "lo studente Bianchi frequenta il corso di Basi di dati in aula A1", che correla le tre entità Bianchi, Basi di dati, e aula A1.

Un'istanza di associazione può avere delle proprietà, come nel fatto: "lo studente Bianchi ha acquistato il libro la Divina Commedia con uno sconto del 5%", dove l'ammontare dello sconto è una proprietà che non caratterizza né l'acquirente, che può avere sconti diversi su diversi libri, né il testo, che può essere acquistato con sconti diversi, ma caratterizza proprio la specifica coppia acquirente-libro. Si osservi che lo stesso fatto potrebbe essere anche considerato non un'istanza di associazione con proprietà, ma un evento, ovvero un'entità di tipo acquisto, associato a Bianchi e alla Divina Commedia, con una proprietà sconto.

Così come si è interessati all'esistenza di collezioni omogenee di entità, allo stesso modo si è in genere interessati all'esistenza di insiemi di istanze di associazione.

Definizione 2.7

Un'*associazione* tra le collezioni $C_1, \ldots, C_n$ è un insieme di istanze di associazione tra elementi di $C_1, \ldots, C_n$, che varia in generale nel tempo. Il prodotto cartesiano delle estensioni di $C_1, \ldots, C_n$ è detto il *dominio dell'associazione*.[1]

1. Si osservi che, mentre il dominio di una proprietà è definito da un tipo, ed è quindi un insieme immutabile e in generale infinito, il dominio di un'associazione, essendo il prodotto delle estensioni

Per chiarire le idee, se vediamo due collezioni X ed Y come due insiemi, un'istanza di associazione tra X ed Y può essere vista come una coppia di elementi (x, y), con $x \in X$ ed $y \in Y$, e quindi un'associazione R tra X ed Y può essere vista come un sottoinsieme del prodotto $X \times Y$, ovvero come una relazione (nel senso matematico del termine) tra tali insiemi.

Un'associazione è caratterizzata, oltre che dal suo dominio e dalle caratteristiche delle eventuali proprietà, anche dalle seguenti *proprietà strutturali*: la *molteplicità* e la *totalità*, che definiamo, per semplicità, solo per le associazioni binarie.

Definizione 2.8

La *molteplicità* di un'associazione fra X ed Y riguarda il numero massimo di elementi di Y che possono trovarsi in relazione con un elemento di X e viceversa. Si dice che l'associazione è *univoca* da X ad Y se ogni elemento di X può essere in relazione con *al più* un elemento di Y. Se non esiste tale vincolo si dice che l'associazione è *multivalore* da X ad Y. Allo stesso modo si definisce il vincolo di univocità da Y ad X.

Si osservi che il vincolo di univocità da X ad Y è indipendente dal vincolo di univocità da Y ad X, dando luogo a quattro possibili combinazioni di presenza ed assenza dei due vincoli. Queste combinazioni si esprimono in modo compatto come segue.

Definizione 2.9

La *cardinalità* di un'associazione fra X ed Y descrive contemporaneamente la molteplicità dell'associazione e della sua inversa. Si dice che la cardinalità è *uno a molti* $(1 : N)$ se l'associazione è multivalore da X ad Y ed univoca da Y ad X. La cardinalità è *molti ad uno* $(N : 1)$ se l'associazione è univoca da X ad Y e multivalore da Y ad X. La cardinalità è *molti a molti* $(N : M)$ se l'associazione è multivalore in entrambe le direzioni, ed è *uno ad uno* $(1 : 1)$ se l'associazione è univoca in entrambe le direzioni.

Ad esempio, in un universo del discorso popolato da studenti, dipartimenti, corsi del piano di studi e professori, l'associazione Frequenta(Studenti, Corsi) (dove indichiamo con la notazione $A(C_1, \ldots, C_n)$ un'associazione A con dominio $C_1 \times \ldots \times C_n$) ha cardinalità $(M : N)$, l'associazione Insegna(Professori, Corsi) ha cardinalità $(1 : N)$, l'associazione Dirige(Professori, Dipartimenti) ha cardinalità $(1 : 1)$.

Definizione 2.10

La *totalità* di un'associazione fra due collezioni X ed Y riguarda il numero *minimo* di elementi di Y che sono associati ad ogni elemento di X. Se *almeno* un'entità di Y deve essere associata ad ogni entità di X, si dice che l'associazione è *totale* su

di alcune collezioni, è un insieme finito che dipende dal tempo, ovvero dallo stato dell'universo del discorso.

X, e viceversa sostituendo X con Y ed Y con X. Quando non sussiste il vincolo di totalità, si dice che l'associazione è *parziale*.

Ad esempio, l'associazione Dirige(Professori, Dipartimenti) è totale su Dipartimenti, in quanto ogni dipartimento ha un direttore, ma non su Professori.

Le definizioni precedenti possono essere generalizzate per definire molteplicità e totalità di un'associazione tra $X_1, \ldots, X_n$ rispetto ad X_1, sostituendo "X" con "X_1", ed "elemento di Y" con "ennupla di elementi di $X_2, \ldots, X_n$".

Definizione 2.11

La *struttura della conoscenza concreta* riguarda la conoscenza dei seguenti fatti:

1. esistenza di alcune collezioni, nome delle collezioni, tipo degli elementi delle collezioni (e quindi, nome e caratteristiche delle proprietà di tali elementi);
2. esistenza di alcune associazioni, nome delle associazioni, collezioni correlate da ogni associazione, proprietà strutturali delle associazioni.

Definizione 2.12

La *conoscenza concreta* presuppone che sia nota la struttura di tale conoscenza, e riguarda la conoscenza dei seguenti fatti:

1. quali entità esistono, che tipo ha ciascuna di queste entità, e quali sono i valori delle sue proprietà;
2. per ogni collezione esistente, quali entità appartengono a tale collezione (estensione della collezione);
3. per ogni associazione, quali istanze appartengono a tale associazione (estensione dell'associazione).

In questa classificazione, la conoscenza concreta comprende tutto ciò che riguarda le singole entità (esistenza, valori delle proprietà, appartenenza a collezioni, partecipazione ad associazioni), mentre ciò che descrive la struttura della conoscenza concreta fa parte di una categoria diversa, la conoscenza astratta, descritta nella prossima sezione.

In generale, la conoscenza concreta, che è anche detta *stato* dell'universo del discorso, si evolve nel tempo, in quanto le collezioni, le entità, i valori delle loro proprietà e le associazioni cambiano nel tempo per effetto di processi continui (dipendenti dal trascorrere del tempo), o di processi discreti (dipendenti dal verificarsi di eventi in certi istanti), come il cambio della residenza di un utente, l'acquisizione di un nuovo libro, la concessione di un nuovo prestito ecc. Anche la struttura della conoscenza si evolve nel tempo, perché nell'universo del discorso si possono creare nuovi tipi o collezioni di entità, ma soprattutto perché l'universo del discorso è definito come ciò che interessa modellare, e l'interesse dell'osservatore cambia, in generale, nel tempo.

La conoscenza astratta

La *conoscenza astratta* riguarda i fatti generali che descrivono (*a*) la struttura della conoscenza concreta, (*b*) le restrizioni sui valori possibili della conoscenza concreta

e sui modi in cui essi possono evolvere nel tempo (*vincoli d'integrità*), (*c*) regole per derivare nuovi fatti da altri noti.

È utile classificare i vincoli d'integrità in *vincoli d'integrità statici* e *dinamici*. I *vincoli d'integrità statici* definiscono condizioni sui valori della conoscenza concreta che devono essere soddisfatte indipendentemente da come evolve l'universo del discorso. Le condizioni possono riguardare:

1. I valori di una proprietà. Ad esempio, (*a*) un utente ha le proprietà codice fiscale, nome, residenza, con valori di tipo stringa di caratteri alfanumerici e anno di nascita, con valori di tipo intero; (*b*) uno studente universitario deve avere almeno diciassette anni; (*c*) lo stipendio di un impiegato è un numero positivo.
2. I valori di proprietà diverse di una stessa entità. Ad esempio, (*a*) per ogni impiegato le trattenute sulla paga devono essere inferiori ad un quinto dello stipendio; (*b*) se X è sposato con Y allora il suo stato civile è coniugato.
3. I valori di proprietà di entità diverse di uno stesso insieme. Ad esempio, (*a*) le matricole degli studenti sono tutte diverse; (*b*) se due persone hanno la stessa data di nascita, allora hanno anche la stessa età; (*c*) se una persona X è sposata con Y, allora Y è sposata con X. Un insieme di proprietà è detto *chiave*, rispetto ad una collezione di elementi, se (*a*) i suoi valori identificano univocamente un elemento della collezione, (*b*) ogni proprietà della chiave è necessaria a questo fine. Un esempio di chiave per la collezione degli utenti è il codice fiscale.
4. I valori di proprietà di entità di insiemi diversi. Ad esempio: il presidente della commissione degli esami deve essere il titolare del corrispondente corso.
5. Caratteristiche di insiemi di entità. Ad esempio, il numero degli studenti è inferiore ad un limite massimo prestabilito oppure un laureato in Informatica deve aver accumulato almeno 180 CFU.

I *vincoli d'integrità dinamici* definiscono delle condizioni sul modo in cui la conoscenza concreta può evolvere nel tempo. Ad esempio, una persona coniugata non potrà cambiare stato civile diventando scapolo o nubile, uno studente di un anno di corso non può iscriversi ad un anno precedente dello stesso corso di studio, una data di nascita non può essere modificata. In conclusione, mentre un vincolo statico riguarda ogni singolo stato dell'universo del discorso, un vincolo dinamico riguarda le transizioni da uno stato ad un altro.

Infine, esempi di fatti derivabili da altri sono l'età di una persona, ricavabile per differenza fra l'anno attuale e il suo anno di nascita, oppure la media dei voti degli esami superati da uno studente.

La conoscenza procedurale

L'universo del discorso è una realtà in evoluzione che interagisce con un ambiente. Mentre la conoscenza astratta riguarda la *struttura* dell'universo del discorso, la *conoscenza procedurale* riguarda le operazioni a cui può essere soggetta la conoscenza concreta, ed in particolare l'effetto di tali operazioni ed il modo in cui esse si svolgono.

È utile distinguere due tipi di conoscenza procedurale:

– le operazioni con le quali avvengono le interazioni con l'ambiente esterno per fornire i servizi previsti (*conoscenza delle operazioni degli utenti*);
– le operazioni elementari che interessano le entità dell'universo del discorso per produrre determinati effetti, in particolare le operazioni per la loro creazione, modifica e cancellazione che devono soddisfare le condizioni espresse dai vincoli d'integrità (*conoscenza delle operazioni di base*).

Ad esempio, le operazioni di base che possono interessare uno studente universitario sono: superamento di un esame, passaggio ad altro corso di laurea, conseguimento della laurea, trasferimento ad altra università ecc.

Le operazioni di base riguardanti un'entità ne completano il significato e ne caratterizzano il "comportamento". Secondo un punto di vista, un'entità potrebbe essere caratterizzata completamente in termini delle operazioni che si possono compiere su di essa, senza ricorrere alla nozione separata di proprietà, potendo vedere anche quest'ultime come esempi di operazioni.

Mentre le operazioni di base riguardano una singola entità, un'operazione degli utenti è finalizzata a fornire un servizio agli utenti del sistema informativo secondo modalità prestabilite codificate nelle procedure dell'organizzazione.

Un esempio di procedura nell'ambiente universitario è quella che viene eseguita quando un docente si trasferisce ad un'altra sede (operazione trasferimento docente): (*a*) si interrompe il pagamento dello stipendio; (*b*) si esclude dagli indirizzari per le comunicazioni; (*c*) per ogni corso tenuto dal docente, si inizia la procedura per assegnare il corso ad un altro docente; (*d*) per ogni commissione a cui partecipava il docente, si inizia la procedura per nominare un sostituto ecc.

Altri esempi di procedure sono quelle previste per la gestione del servizio conti correnti di una banca che prevede operazioni del tipo: apertura di un conto, versamento, prelievo, interrogazione saldo, interrogazione movimenti.

La comunicazione

La *comunicazione* riguarda le modalità offerte agli utenti del sistema informativo per scambiare informazioni con il sistema e per accedere alle risorse informative rispettando le regole e le possibilità loro concesse. In particolare, modellare la comunicazione significa anche rappresentare le *interfacce* del sistema informativo, quali ad esempio l'aspetto dei moduli cartacei e delle schermate che vanno riempite per comunicare con tale sistema.

Si osservi che la conoscenza concreta, astratta e delle operazioni di base può essere modellata facendo riferimento essenzialmente all'universo del discorso, sia pure visto in funzione delle necessità del sistema informativo di un'organizzazione. Passando invece alla conoscenza delle operazioni degli utenti, e ancor più alla comunicazione, l'interesse si sposta gradatamente dall'universo del discorso al sistema informativo stesso. Poiché la modellazione è in generale finalizzata alla progettazione di un sistema informativo che modifichi quello preesistente, gli strumenti per modellare procedure e comunicazione possono essere usati per rappresentare tanto il sistema informativo esistente quanto quello in via di progettazione.

L'interesse verso questo aspetto della progettazione e della modellazione, in particolare verso la modellazione dei dialoghi fra gli utenti e il sistema informatico, è molto cresciuto negli ultimi anni poiché, con lo sviluppo degli strumenti hardware e software che supportano la realizzazione di applicazioni con interfaccia grafica, la realizzazione di modalità di interazione gradevoli e personalizzate ai diversi tipi di utenti coinvolti ha assunto un'importanza sempre maggiore, sia come requisito, sia per ciò che riguarda la quantità di lavoro che i programmatori dedicano alla realizzazione dell'interfaccia stessa.

2.2.2 Aspetto linguistico astratto

L'aspetto linguistico astratto riguarda gli strumenti concettuali, o *meccanismi di astrazione*, adottati per modellare l'universo del discorso.

Non sorprende che negli studi sull'argomento sia stata posta la massima attenzione su questi meccanismi, perché l'astrazione è lo strumento concettuale principale per acquisire e organizzare conoscenza.

> Sì come a voler che i calcoli tornino sopra i zuccheri, le sete e le lane, bisogna che il computista faccia le sue tare di casse, invoglie e altre bagaglie, così, quando il filosofo geometra vuol riconoscere in concreto gli effetti dimostrati in astratto, bisogna che difalchi gli impedimenti della materia.
> — Galileo Galilei, *Dialogo sopra i due massimi sistemi del mondo.*

Un meccanismo di astrazione è lo strumento fondamentale per cogliere un aspetto della situazione da descrivere, tralasciando dettagli ritenuti in quel momento poco significativi (l'astrazione come forza semplificatrice). Vedremo nelle prossime sezioni i meccanismi proposti per modellare alcuni dei fatti precedentemente elencati.

2.2.3 Aspetto linguistico concreto

L'aspetto linguistico concreto riguarda le caratteristiche, e la definizione, del linguaggio formale usato per costruire il modello. Una volta fissati i meccanismi di astrazione da usare per modellare, si possono usare linguaggi formali con caratteristiche molto diverse che supportano i meccanismi di astrazione prescelti, a seconda degli obiettivi che ci si prefigge con la costruzione del modello. Si possono usare linguaggi di specifica non eseguibili, linguaggi logici o linguaggi di programmazione. In questo libro l'attenzione sarà sui formalismi grafici.

2.2.4 Aspetto pragmatico

L'aspetto pragmatico riguarda la metodologia da seguire nel processo di modellazione. Una metodologia è un insieme di regole finalizzate alla costruzione del modello informatico. La natura delle metodologie dipende dal tipo di modello da costruire e quindi dal livello di dettaglio al quale vanno trattati i vari aspetti del modello.

Nel prossimo capitolo ci limiteremo a mostrare un esempio di metodologia per progettare uno *schema concettuale* del modello, cioè una rappresentazione ad alto livello della struttura della conoscenza concreta da trattare. Un'analoga metodologia da utilizzare per la progettazione degli altri aspetti del modello informatico è molto più complessa e la sua presentazione esula dai fini di questo libro. Si rinvia alle note bibliografiche per riferimenti a testi specifici.

Nel resto del capitolo si approfondisce l'aspetto linguistico astratto della modellazione presentando dei meccanismi di astrazione per modellare alcuni dei fatti precedentemente elencati e si mostra un formalismo grafico per rappresentare tali fatti. Sebbene il formalismo grafico non consenta di rappresentare ogni aspetto di un modello informatico, vedremo come esso sia molto utile nelle fasi di specifica dei requisiti e di progettazione dello schema di una base di dati, quando occorre ragionare sui fatti più importanti da trattare.

2.3 Il modello dei dati ad oggetti

Nella costruzione di un modello informatico si procede in due stadi:

1. si "definisce" il modello, ovvero si rappresenta la struttura della conoscenza concreta, le altre parti della conoscenza astratta, la conoscenza procedurale, i tipi di comunicazione (definizione di schema e applicazioni);
2. si "costruisce" la rappresentazione di fatti specifici conformi alle definizioni date, ovvero la rappresentazione della conoscenza concreta (immissione dati).

Esempio 2.1
Per costruire un modello informatico per la gestione di informazioni sui libri, prima si devono definire, tra le molteplici proprietà che caratterizzano un libro, quelle che interessano ai fini dell'applicazione: possono essere titolo, autore, editore ecc. come si usa in una biblioteca, oppure dettagliate informazioni qualitative su materiale e stato di conservazione come può interessare ad un laboratorio di restauro.

Una volta definite le proprietà interessanti comuni a tutti i possibili libri, si passa a costruire per ogni entità "libro" dell'universo del discorso una rappresentazione nel modello informatico, assegnando un valore per ogni proprietà definita.

Per la definizione del modello si possono usare diversi tipi di formalismi, che si differenziano innanzitutto per il modello dei dati che supportano, cioè per i meccanismi di astrazione offerti per modellare. A loro volta, i modelli dei dati si differenziano per:

- i modi in cui si rappresenta la struttura della conoscenza concreta;
- la possibilità di rappresentare aspetti procedurali;
- gli operatori disponibili sui fatti rappresentati.

Nel seguito si esamineranno innanzitutto i meccanismi di astrazione di un *modello ad oggetti*, che sono ritenuti i più adatti per modellare in modo naturale e diretto situazioni reali. Per semplicità non si prenderanno in considerazione la conoscenza procedurale e la comunicazione. Successivamente verrà fatta una panoramica dei meccanismi di astrazione di altri modelli dei dati e in particolare del *modello relazionale*, che verrà poi approfondito nei capitoli successivi perché è ormai uno standard dei sistemi per basi di dati.

2.3.1 Rappresentazione della struttura della conoscenza concreta

I modelli dei dati sono stati pensati inizialmente per descrivere solo la struttura della conoscenza concreta ed alcuni vincoli d'integrità, e non per descrivere proprietà calcolate od operazioni di base. L'aspetto interessante del modello ad oggetti è invece la proposta di un insieme di meccanismi che permettono di modellare sia i dati che le operazioni di base per agire su di loro.

Un modello dei dati ad oggetti è caratterizzato dalle nozioni di *oggetto*, *tipo di oggetto*, *classe*, *gerarchie fra tipi*, *definizioni per ereditarietà* e *gerarchie fra classi*. Per rendere più chiara la presentazione, verranno dati esempi di utilizzo di questi meccanismi utilizzando un formalismo grafico adeguato per definire ad un primo livello di astrazione lo schema concettuale di una base di dati, ovvero la struttura della conoscenza concreta. Per un esempio di linguaggio testuale per definire lo schema nei dettagli si veda [Albano et al., 1985], [Albano et al., 2000].

Oggetto

Un *oggetto* è la rappresentazione nel modello informatico degli aspetti strutturali e comportamentali di un'entità della realtà. Una caratteristica essenziale del modello ad oggetti, che lo differenzia dagli altri modelli, è il fatto che non esistono limitazioni sulla complessità della struttura degli oggetti, così che è sempre possibile avere una corrispondenza biunivoca fra le entità e gli oggetti del modello che le rappresentano.[2]

Definizione 2.13

Un *oggetto* è un'entità software con *stato*, *comportamento* e *identità*, che modella un'entità dell'universo del discorso. Lo stato è costituito da un insieme di *campi*, o *componenti*, che possono assumere valori di qualsiasi complessità che modellano le proprietà dell'entità. Il comportamento è costituito da un insieme di procedure locali, eventualmente dotate di parametri, chiamate *metodi*, che modellano le operazioni di base che riguardano l'oggetto e le proprietà derivabili da altre. Le operazioni applicabili ad un oggetto sono dette *messaggi* a cui l'oggetto risponde, utilizzando i propri metodi e manipolando il proprio stato.

2. Il termine *entità* si usa per riferirsi ad uno specifico fatto interessante della realtà da modellare, mentre il termine *oggetto* si usa per riferirsi alla rappresentazione dell'entità nel modello informatico.

Definizione 2.14
L'*interfaccia* di un oggetto specifica l'insieme dei messaggi a cui esso può rispondere, con il tipo dei parametri e del risultato, e l'insieme dei campi dell'oggetto che sono accessibili dall'esterno dell'oggetto stesso, con il loro tipo.

Definizione 2.15
L'*identità* di un oggetto (*Object Identifier, OID*) è una caratteristica che è associata all'oggetto stesso fin dal momento in cui esso è creato, che non può essere modificata, e che non viene in particolare modificata dall'aggiornamento dello stato dell'oggetto. L'identità di due oggetti diversi è sempre diversa. L'identità modella quelle caratteristiche di un'entità del mondo reale che fanno sì che tale entità sia sempre percepita come la stessa anche quando cambiano i valori di alcune delle sue proprietà.

Di solito i componenti dello stato sono accessibili direttamente dall'esterno, ma sono modificabili solo mediante le procedure associate all'oggetto. Quando anche l'accesso ai componenti dello stato può avvenire solo mediante i metodi associati all'oggetto, si dice che l'oggetto *incapsula* lo stato.

Come è stato detto in precedenza, in questo capitolo si vuole proporre un formalismo grafico per descrivere gli aspetti essenziali di uno schema concettuale di una base di dati: le collezioni di oggetti e le associazioni fra loro che diano una visione immediata della struttura della base di dati. Per questa ragione i metodi degli oggetti non si prenderanno in considerazione.

Tipo oggetto

Un tipo definisce un insieme di possibili valori e le operazioni ad essi applicabili. Nella costruzione di un modello informatico i tipi scaturiscono dal processo di astrazione con il quale prima si raggruppano entità diverse, perché ritenute omogenee ai fini dell'applicazione in esame, e poi si prescinde dalle differenze tra le singole entità per evidenziare invece ciò che le accomuna e le rende omogenee, cioè la loro *struttura*.

Esempio 2.2
Entità diverse come "Tizio", "Caio" e "Sempronio", vengono considerate (classificate) di tipo Utente, per porre l'accento sul fatto che di esse interessano le proprietà tipiche delle persone in quanto fruitori dei servizi di una biblioteca, quali il nome, la residenza e il recapito telefonico.

Ogni oggetto è un valore di un tipo. Ad ogni tipo sono associati degli operatori per costruire oggetti di quel tipo (costruttori di oggetti) e l'interfaccia degli oggetti del tipo stesso.

Definizione 2.16
L'*interfaccia di un tipo oggetto* specifica i componenti dello stato che sono accessibili dall'esterno e il loro tipo.

I nomi dei componenti accessibili dello stato sono detti anche *attributi degli oggetti*.

Come accade per le proprietà delle entità, un attributo di un oggetto può avere valori di tipo *atomico* o *strutturato*, *univoco* o *multivalore*.

Esempio 2.3
Nell'esempio della biblioteca, alcuni possibili attributi del tipo Utente sono CodiceFiscale (CF), Nome, Residenza, AnnoDiNascita, con i primi due associati a valori di tipo string, sequenza di caratteri alfanumerici, il terzo strutturato e il quarto associato a valori di tipo int.

Rappresentazione grafica
Un tipo oggetto si rappresenta con una notazione simile a quella usata in UML (*Unified Modeling Language*):[3] un rettangolo diviso in tre parti; la prima contiene il nome del tipo; la seconda parte contiene i componenti dello stato e il loro tipo; la terza parte, che può mancare, contiene la descrizione di vincoli d'integrità. Per esempio, in Figura 2.1 è mostrata una rappresentazione per il tipo Persona.

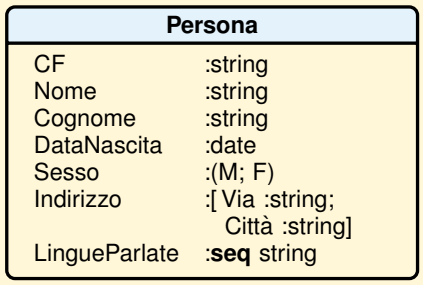


Figura 2.1: Esempio di tipo oggetto

Per definire i tipi degli attributi di un tipo oggetto si utilizzano tipi primitivi e non primitivi, ma non tipi oggetti, come invece accade in UML.[4]

I tipi primitivi comprendono int, real, bool, date e string.

3. Un tipo oggetto è detto *classe* in UML, ma noi useremo questo termine con un altro significato, come vedremo più avanti.
4. Questa modalità di definizione di un tipo oggetto non coincide con quella prevista nell'UML e, come vedremo anche più avanti, per modellare in modo semplice una base di dati si preferirà introdurre

Tipi non primitivi possono essere costruiti applicando i seguenti operatori ad altri tipi (non necessariamente primitivi):

- l'operatore *tipo record* costruisce un tipo i cui valori sono record, ovvero insiemi di coppie (etichetta, valore associato). Il tipo record specifica le etichette che devono essere presenti nei valori di quel tipo, ed il tipo del valore associato ad ogni etichetta, con la seguente sintassi:

  $[\ A_1{:}T_1; \ldots; A_n{:}T_n\ ]$

- l'operatore *tipo enumerazione*, insieme di etichette separate da un punto e virgola e racchiuse fra parentesi tonde. Per esempio, il tipo dell'attributo Sesso è (M; F).
- l'operatore *tipo sequenza*, seq T, costruisce un tipo i cui valori sono sequenze di valori con tipo T. Una sequenza si differenzia da un insieme perché gli elementi sono ordinati e possono essere ripetuti; una sequenza è quindi un multinsieme finito e ordinato di valori dello stesso tipo, eventualmente vuoto.

Una proprietà strutturata si modella con un attributo di tipo record, mentre il tipo sequenza modella le proprietà multivalore.

Classe

Come gli oggetti modellano entità, così le classi modellano collezioni. Si osservi che nel gergo dei linguaggi ad oggetti il termine classe si usa con il significato di tipo oggetto.

Definizione 2.17
Una *classe* è un insieme di oggetti dello stesso tipo, modificabile con operatori per includere o estrarre elementi dall'insieme, al quale sono associabili alcuni vincoli di integrità.

Rappresentazione grafica

Per non complicare la rappresentazione grafica dello schema concettuale di una basi di dati, si preferisce non introdurre una nuova notazione grafica per descrivere le classi, ma fare l'ipotesi che (*a*) gli unici tipi oggetti che si modellano sono quelli che rappresentano elementi di classi e (*b*) il nome del tipo degli oggetti è anche il nome della classe. Pertanto da ora in poi, una definizione come quella di Figura 2.1 si interpreterà come la classe Persona con elementi aventi la struttura mostrata. Useremo nel seguito la convenzione di usare nomi al plurale negli esempi di classi.

Quando lo schema concettuale contiene numerose classi, ed elementi con struttura complessa, per non appesantire la rappresentazione grafica, si preferisce descriver-

altre notazioni per trattare alcuni aspetti importanti delle basi di dati e non ancora previste nello standard UML.

le usando solo un rettangolo etichettato con il nome della classe e descrivere poi separatamente la struttura dei suoi elementi.[5]

Rappresentazione delle associazioni

Associazioni binarie senza proprietà

Un'associazione fra due collezioni C_1 e C_2 è una relazione binaria fra C_1 e C_2, e si rappresenta nella notazione grafica con una linea che collega le classi che rappresentano le due collezioni. La linea è etichettata con il nome dell'associazione che di solito viene scelto utilizzando un predicato che dia un significato alla frase con la struttura "soggetto predicato complemento" ottenuta leggendo il diagramma da sinistra a destra (o dall'alto al basso), dove il soggetto è il generico elemento della prima collezione e il complemento il generico elemento della seconda collezione. Ad esempio, con riferimento alla Figura 2.2: *uno studente ha sostenuto un esame*. Quando però non si riesce a trovare un verbo specifico per l'associazione, oppure quando più associazioni in uno schema finirebbero per avere lo stesso nome, si può utilizzare come nome dell'associazione la concatenazione dei nomi delle classi coinvolte.

L'univocità di un'associazione, rispetto ad una classe C_1, si rappresenta disegnando una freccia singola sulla linea che esce dalla classe C_1 ed entra nella classe C_2; l'assenza di tale vincolo è indicata da una freccia doppia. Una freccia singola che esce dalla classe C_1 si può quindi leggere come "ad ogni elemento della classe C_1 corrisponde un unico elemento nella classe C_2", ovvero come "ogni elemento della classe C_1 partecipa al più ad un'istanza dell'associazione". Similmente, la parzialità è rappresentata con un taglio sulla linea vicino alla freccia, mentre il vincolo di totalità è rappresentato dall'assenza di tale taglio.

Esempio 2.4
In Figura 2.2 è rappresentata l'associazione fra studenti ed esami superati dagli studenti (esami intesi come eventi di esaminazione): ogni esame riguarda uno ed un solo studente, mentre non ci sono vincoli di totalità né di univocità sugli esami superati da uno studente.

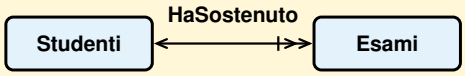


Figura 2.2: Rappresentazione di classi e associazioni

Alcuni autori propongono una specifica notazione per trattare associazioni (1:M) con inversa totale quando si modellano oggetti *composti*, ovvero oggetti con componenti

5. Gli editori grafici di schemi concettuali di solito consentono di passare con un clic sul rettangolo di una classe alla sua visualizzazione estesa con gli attributi.

che sono altri oggetti, come una bicicletta con componenti un sedile, due ruote, un telaio ecc. (associazione *parte-di* o *part-of*). Questa descrizione può essere utile per evidenziare che alcune operazioni su un oggetto composto si applicano anche a tutte le sue parti; ad esempio, spostare in un disegno una bicicletta vuol dire spostare tutte le sue componenti, oppure eliminare una bicicletta da una collezione vuol dire eliminare anche le sue componenti dalle rispettive classi.

Associazioni binarie con proprietà

Se l'associazione binaria ha delle proprietà, il suo nome si scrive in un rettangolo collegato alla linea che la rappresenta e contenente la definizione delle proprietà. In Figura 2.3 è mostrato un esempio di un'associazione tra i libri di una biblioteca e gli utenti, che modella i prestiti, e che ha una proprietà Data.

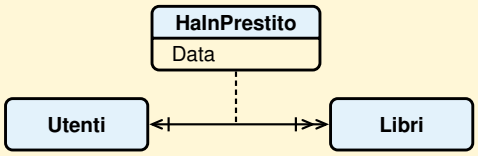


Figura 2.3: Rappresentazione di associazioni con proprietà

Esempio 2.5
Un'associazione con proprietà, come quella tra i libri di una biblioteca e gli utenti di Figura 2.3, può essere modellata interpretando un'istanza di associazione come un'entità e definendo così una classe Prestiti, associata in modo $(1:1)$ ai Libri e in modo $(N:1)$ agli Utenti, e aggiungendo un attributo Data alla classe Prestiti stessa (si veda la Figura 2.4).

Il fatto che un utente u di una biblioteca ha preso in prestito il libro l in una data d è modellato inserendo nella classe Prestiti un oggetto con attributo Data uguale a d, associato all'utente u e al libro l.

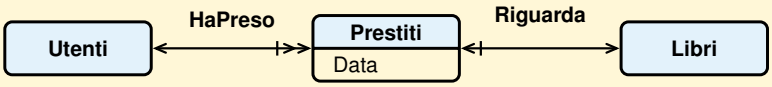


Figura 2.4: Trasformazione di associazioni con proprietà

Associazioni ricorsive

Le associazioni ricorsive sono relazioni binarie fra gli elementi di una stessa collezione. Un classico esempio è la relazione ÈMadreDi sulla collezione delle Persone: una persona può essere madre di più persone della collezione e può avere una madre nel-

la collezione. Nella rappresentazione di questo tipo di associazione, per una corretta interpretazione del fatto rappresentato occorre etichettare la linea non solo con il nome dell'associazione, ma anche con dei nomi per specificare il ruolo che hanno i due componenti in un'istanza di associazione (Figura 2.5).

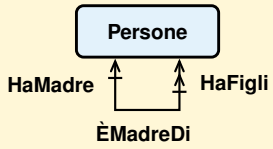


Figura 2.5: Associazione ricorsiva

Associazioni n-arie

In generale le associazioni possono essere di grado maggiore di due. Associazioni di grado tre qualche volta si usano, quelle di grado quattro si usano raramente e quelle di grado superiore sono una curiosità difficile da comprendere e da usare. Per semplicità non daremo una notazione grafica per rappresentare associazioni non binarie, e si mostra nell'esempio che segue come trattarle con un'opportuna trasformazione della rappresentazione, come si è fatto nel caso delle associazioni con proprietà.

Esempio 2.6

Si supponga di voler rappresentare l'associazione fra Voli, Passeggeri e Posti. Per ogni volo (ad esempio, il volo Pisa-Roma del 1-1-2021), al momento dell'imbarco, viene assegnato un posto a ciascun passeggero. L'associazione è multivalore rispetto ad ogni collezione, poiché ogni singolo volo, passeggero, e posto, partecipa in generale a diverse istanze di associazione, ma con i vincoli che, in un singolo volo, ogni posto è associato al più ad un passeggero ed ogni passeggero può occupare al più un posto.

Un'associazione ternaria può essere modellata interpretando un'istanza di associazione come un'entità e definendo così una classe Imbarchi, associata in modo $(1 : N)$ ai Voli, ai Passeggeri e ai Posti. Se l'associazione avesse delle proprietà, verrebbero aggiunti degli attributi alla classe Imbarchi (si veda la Figura 2.6).

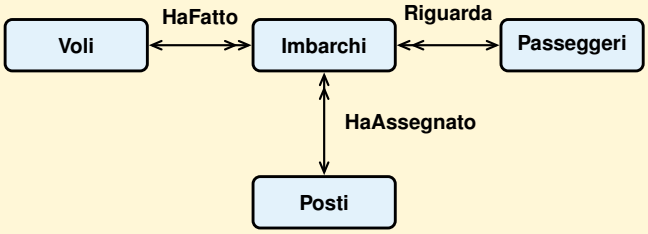


Figura 2.6: Associazione ternaria trasformata in classe

Esempio 2.7
Di una biblioteca universitaria vogliamo rappresentare i seguenti fatti:

- Le *Descrizioni Bibliografiche* dei libri a volume unico, sia già disponibili che quelli ordinati ma non ancora consegnati alla biblioteca. Delle descrizioni bibliografiche interessano: Codice ISBN (*International Standard Book Number*) che le identifica, titolo del libro, autori, editore, anno di pubblicazione e termini che le descrivono.
- I *termini* sono una rappresentazione di un *thesaurus*, o *parole chiave*, usati per descrivere (*indicizzare*) le descrizioni bibliografiche. Fra i termini interessano due particolari associazioni, fra le diverse possibili:
  - *Preferenza* (*ÈSinonimoDi*), che associa a termini non-standard il termine standard che viene usato da esperti del settore. Ad esempio:
    - Calcolatore *Standard* Computer;
    - Computer *Sinonimi* Calcolatore, Workstation, Server;
  - *Gerarchia* (*Specializza*), per sottolineare le relazioni di specificità o generalità fra due termini. Ad esempio:
    - Felino *TerminiPiùSpecifici* Gatto, Leone, Tigre;
    - Gatto *TerminePiùGenerale* Felino;
- I *libri*, disponibili in una o più copie ognuna identificata da un Codice, una stringa con la loro collocazione e numero.
- Gli *autori* dei libri, dei quali interessano: Codice Fiscale, che li identifica, nome e cognome, nazionalità e anno di nascita.
- Gli *utenti* della biblioteca. Quando un utente prende in prestito un libro, interessano le seguenti informazioni (se non già presenti): Codice Fiscale, nome e cognome, indirizzo e numeri dei telefoni. Un utente può avere più libri in prestito.
- I *prestiti* dei libri, dei quali interessano, fino a quando il libro non è restituito, le date del prestito e della restituzione prevista.

In Figura 2.7 è mostrato uno schema della base di dati usando il formalismo grafico visto fino ad ora.

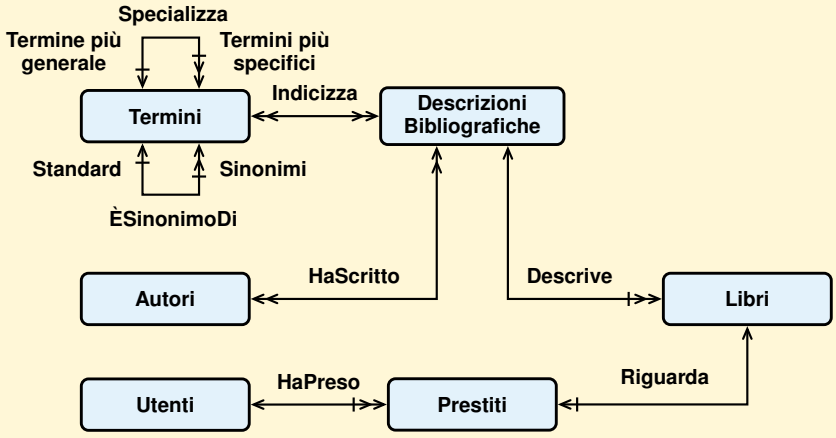


Figura 2.7: Un esempio di schema di base di dati

Gerarchie di tipi

La relazione di sottotipo è una relazione di ordinamento parziale sull'insieme dei tipi tale che un valore che appartiene al sottotipo può essere utilizzato in tutti i contesti in cui è previsto un valore del supertipo.

Ad esempio, il tipo Utente può essere definito come un supertipo dei tipi Studente e Docente, per rappresentare il fatto che un'entità classificata come Studente e Docente possa essere classificata anche come Utente, e apparire quindi in tutti i contesti in cui può apparire un Utente. Si prescinde così dalle proprietà che rendono queste entità semanticamente diverse, e si evidenzia invece che sono entità omogenee ad un particolare livello di astrazione, ad esempio con Nome, Residenza e DataDiNascita come proprietà comuni.

Qualche relazione di sottotipo è presente in molti linguaggi (tipicamente, gli interi sono un sottotipo dei numeri in virgola mobile), ma nei linguaggi ad oggetti assume un ruolo particolarmente importante poiché è collegata strettamente alla nozione di "ereditarietà".

Definizione per ereditarietà

Si dice che la definizione dell'interfaccia di un tipo oggetto è data per ereditarietà a partire da un tipo T quando tale definizione è fornita specificando cosa va aggiunto o modificato per ottenere il nuovo tipo a partire da T. In particolare, un'interfaccia è definita per ereditarietà da un'interfaccia $\mathcal{I}$ specificando quali attributi aggiungere ad $\mathcal{I}$ e come, eventualmente, modificare (*overriding*) il tipo degli attributi "ereditati", ovvero degli attributi già presenti in $\mathcal{I}$.

In genere si impone che, nel definire un'interfaccia per ereditarietà, se si modifica il tipo di un attributo, il nuovo tipo sia un sottotipo del tipo precedente (vincolo di *ereditarietà stretta*) [Albano et al., 1985], [Albano et al., 2000]. Grazie a questo vincolo,

un valore di un tipo definito per ereditarietà a partire da T ha tutti gli attributi di T e, se il tipo di un attributo è cambiato, il nuovo tipo è compatibile (un sottotipo) con il tipo precedente. Ad esempio, un oggetto di tipo Studente, definito per ereditarietà stretta dal tipo Utente, ha un Residenza, come ogni altro utente. Se il tipo di Residenza è ridefinito nel tipo Studente, sarà comunque un tipo utilizzabile in tutti i contesti in cui sono utilizzabili valori del tipo che aveva Residenza nel tipo Utente. Ne consegue che, in generale, un tipo definito per ereditarietà stretta costituisce anche un sottotipo del tipo originario.

Infine, un tipo può essere definito per ereditarietà a partire da un unico supertipo (*ereditarietà singola*) o da più supertipi (*ereditarietà multipla*). Questa seconda possibilità è molto utile ma può creare alcuni problemi quando lo stesso attributo viene ereditato, con tipi diversi, da più tipi antenato.

La possibilità di definire tipi oggetto per ereditarietà è di grande interesse dal punto di vista dell'ingegneria del software, sia perché rende il linguaggio più adatto per modellare in modo naturale situazioni complesse, sia perché consente di sviluppare le applicazioni incrementalmente per raffinamento di definizioni di tipi. Ciò che rende questo meccanismo interessante è, in particolare, la combinazione di sovraccarico dei messaggi (*overloading*) e risoluzione dei nomi dei messaggi a tempo di esecuzione (*late binding*).

Gerarchia di inclusione fra classi

La gerarchia di inclusione è una relazione di ordinamento parziale sull'insieme delle classi tale che se C_1 è una sottoclasse di C_2 (è inclusa in C_2), allora gli elementi di C_1 sono un sottoinsieme degli elementi di C_2 ed ereditano le associazioni definite su C_2 (*vincolo estensionale*). Come conseguenza, quando nel modello si aggiunge un elemento ad una sottoclasse, esso viene automaticamente a far parte anche delle superclassi. Affinché questo non provochi problemi di tipo, si impone anche il seguente *vincolo strutturale*: se C_1 è una sottoclasse di C_2, allora il tipo degli elementi di C_1 è un sottotipo del tipo degli elementi di C_2.

Quando si definiscono più sottoclassi di una stessa classe, su questo insieme di sottoclassi possono essere definiti i seguenti vincoli:

- un insieme di sottoclassi soddisfa il *vincolo di disgiunzione* se ogni coppia di sottoclassi in questo insieme è disgiunta, ovvero è priva di elementi comuni (*sottoclassi disgiunte*);
- un insieme di sottoclassi soddisfa il *vincolo di copertura* se l'unione degli elementi delle sottoclassi coincide con l'insieme degli elementi della superclasse (*sottoclassi copertura*).

I due vincoli sono indipendenti fra loro; quando sono entrambi soddisfatti, l'insieme di sottoclassi costituisce una *partizione* della superclasse.

Una sottoclasse può essere definita anche a partire da un'altra sottoclasse, modellando così gerarchie a più livelli. Ad esempio, la classe Studenti potrebbe essere a sua volta specializzata introducendo le sottoclassi partizione StudentiInCorso e StudentiFuoriCorso. In generale, una sottoclasse può essere popolata creando in essa nuovi

elementi, che diventano anche elementi delle superclassi, oppure spostando in essa elementi già presenti nella superclasse.

Infine, la gerarchia non è necessariamente ad albero (*gerarchie singole*), perché una sottoclasse può essere definita a partire da più classi (*gerarchie multiple*). Ad esempio, gli StudentiLavoratori potrebbero essere una sottoclasse sia degli Studenti che dei Dipendenti.

In generale, al meccanismo delle sottoclassi sono associati operatori per:

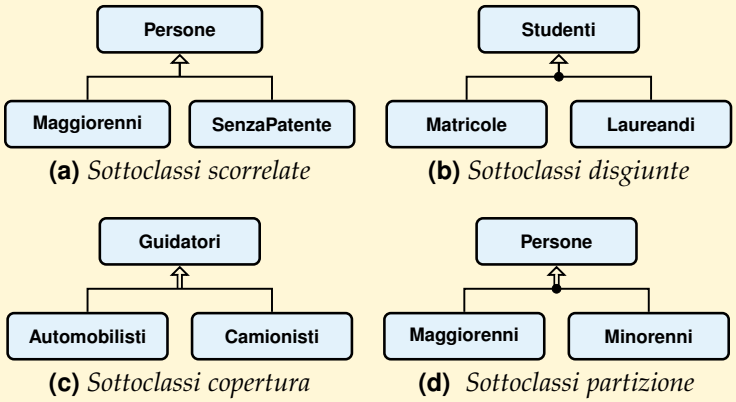
- creare un elemento in una classe, che diventa anche un elemento delle sue superclassi per il vincolo estensionale;
- rimuovere un elemento da una classe, con la conseguente rimozione da tutte le sue sottoclassi per il vincolo estensionale;
- controllare se un elemento di una classe è anche in una sua sottoclasse.

Osservazione

Di solito si assume che un oggetto una volta creato con un certo tipo (ad esempio, Persona) non possa acquisire nuovi tipi (ad esempio, Studente), e che non sia quindi possibile "spostare" un elemento di una classe in una o più sottoclassi. Nel seguito non si terrà conto di questa limitazione, perché sono stati proposti linguaggi ad oggetti per basi di dati con operatori che permettono questo tipo di spostamenti, come il linguaggio Galileo [Albano et al., 1985], [Albano et al., 2000].

Rappresentazione grafica

I quattro tipi di sottoclassi si descrivono come mostrato in Figura 2.8: il vincolo di disgiunzione viene rappresentato con il pallino nero, mentre il vincolo di copertura viene rappresentato con una freccia doppia verso la superclasse.



(a) *Sottoclassi scorrelate* **(b)** *Sottoclassi disgiunte*

(c) *Sottoclassi copertura* **(d)** *Sottoclassi partizione*

Figura 2.8: Rappresentazione grafica dei tipi di sottoclassi

Sottoclassi scorrelate, non richiedendo né il vincolo di copertura né quello di disgiunzione, si possono rappresentare anche con la notazione usata in Figura 2.9. Le gerarchie multiple si rappresentano come in Figura 2.10.

Figura 2.9: Rappresentazione grafica di sottoclassi scorrelate

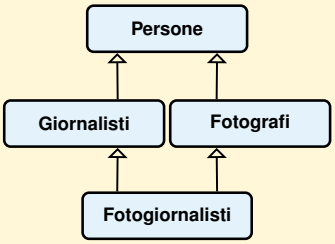


Figura 2.10: Rappresentazione grafica di gerarchie multiple

Esempio 2.8
Passando ad un successivo livello di dettaglio si può rendere più fedele lo schema per la base di dati della biblioteca di Figura 2.7, si definiscono le seguenti *sottoclassi partizione*:

- della classe *Utenti* le sottoclassi *Studenti*, dei quali interessa anche la matricola, e *Docenti*, dei quali interessa anche il numero del telefono dell'ufficio;
- della classe *Libri* le sottoclassi *Prestabili* e *Consultabili* che possono essere presi in prestito per un numero prefissato di giorni solo dagli utenti che sono docenti;
- della classe *Prestiti* le sottoclassi *Regolari*, dei prestiti di libri prestabili, e *InConsultazione* dei prestiti di libri consultabili (Figura 2.11).

Infine in Figura 2.12 lo schema viene completato con la struttura degli elementi delle classi.

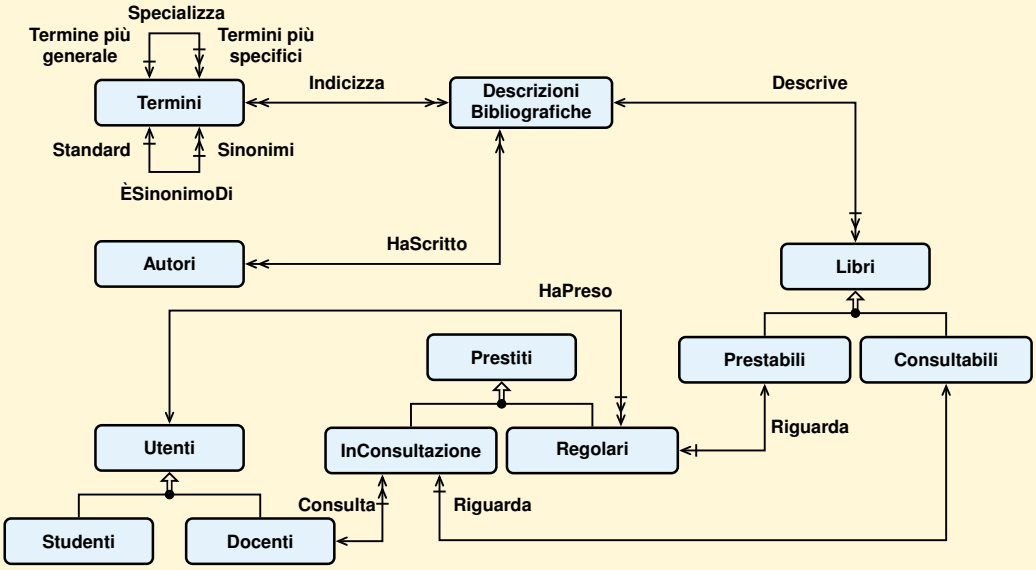


Figura 2.11: Raffinamento delle schema di Figura 2.7

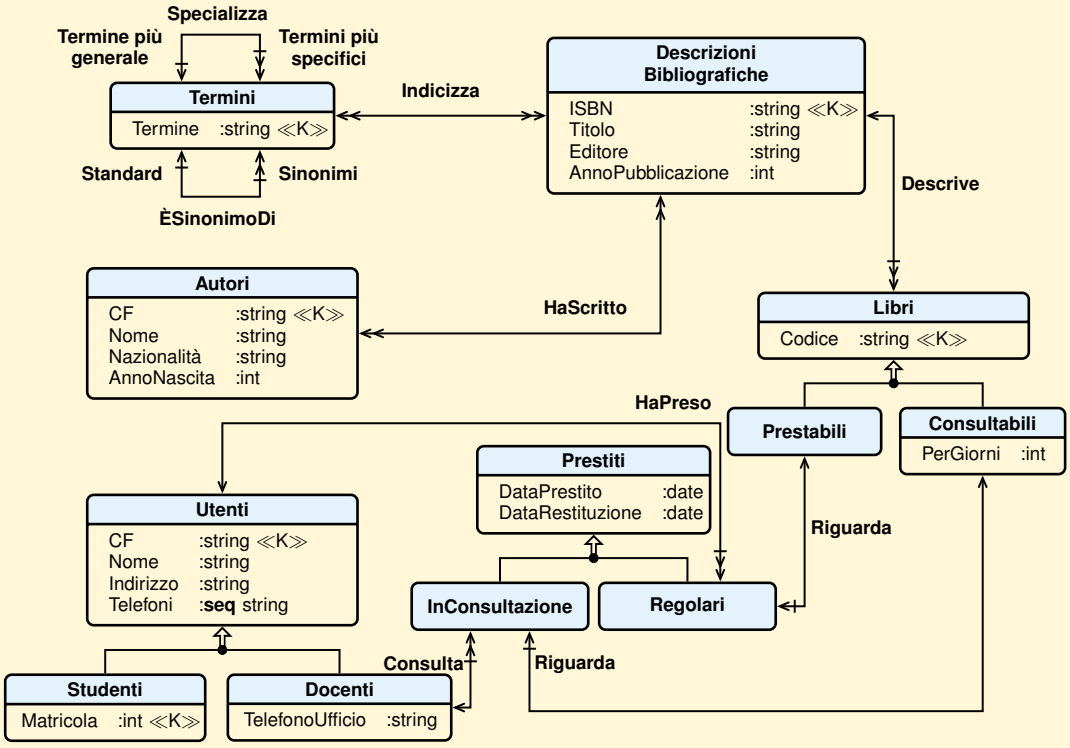


Figura 2.12: Schema di Figura 2.11 con gli attributi degli elementi delle classi

2.3.2 Rappresentazione degli altri aspetti della conoscenza astratta

Nel modellare una situazione reale, in generale non è sufficiente limitarsi alla descrizione della struttura della conoscenza concreta con i meccanismi di astrazione del modello dei dati prescelto, come è stato visto finora, ma è necessario descrivere anche vincoli d'integrità che impongono ulteriori restrizioni sui possibili valori della conoscenza concreta.

I vincoli possono essere descritti in modo dichiarativo, con formule del calcolo dei predicati, oppure mediante controlli da eseguire nelle operazioni (di base o degli utenti). La preferenza è per l'approccio dichiarativo per due ragioni principali. Innanzitutto è più semplice stabilire la coerenza dei vincoli con la definizione dei requisiti ed apportare modifiche per adeguarli a nuove esigenze. In secondo luogo, una descrizione dichiarativa dei vincoli si presta allo sviluppo di strumenti da usare nella fase di progettazione per stabilire eventuali inconsistenze o ridondanze, cioè vincoli logicamente implicati da altri. Infine si evita di ripetere alcuni controlli in più operazioni.

Rappresentazione grafica

Si è già mostrato come rappresentare graficamente i vincoli sulle proprietà strutturali delle associazioni e sul tipo di sottoclassi.

In Figura 2.13 è mostrato come la definizione di una classe si arricchisce di una sezione dedicata alla descrizione dei vincoli. Gli attributi marcati con la stringa ≪K≫ sono una chiave. Se vi sono altre n chiavi si usano le stringhe ≪K1≫ ... ≪Kn≫.

Vincoli generali andrebbero espressi usando un opportuno formalismo, come l'OCL (*Object Constraint Language*) proposto per l'UML, e in figura sono mostrati alcuni semplici esempi, dove si usa la notazione self.nomeAttributo per fare riferimento all'attributo nomeAttributo dell'oggetto. In uno stadio iniziale del progetto può essere sufficiente descrivere i vincoli usando il linguaggio naturale.

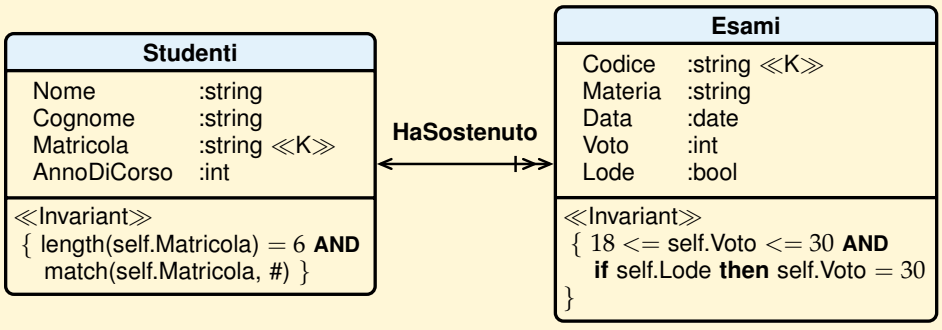


Figura 2.13: Esempio di classi con vincoli

2.3.3 Rappresentazione della conoscenza procedurale

Mentre la conoscenza delle operazioni di base viene modellata con il meccanismo dei metodi degli oggetti, le operazioni degli utenti vengono modellate con il meccanismo delle transazioni, programmi sequenziali che consentono di modellare l'evoluzione dell'universo del discorso astraendo da malfunzionamenti e da interferenze indesiderate con altre transazioni che accedono concorrentemente ai dati.

Un'operazione (di base o degli utenti) si specifica, a questo livello di astrazione, dando le seguenti informazioni:

- il *nome*, un identificatore non già utilizzato per altri elementi dello schema;
- lo *scopo* dell'operazione, descritto con un testo in linguaggio naturale;
- gli *argomenti* dell'operazione, un elenco di coppie "identificatore :tipo";
- il tipo del *risultato*;
- il testo di eventuali messaggi di errore;
- le classi e le associazioni che *usa* ma non modifica;
- le classi e le associazioni che *modifica*, per aggiunta di elementi o modifica di attributi degli elementi;
- due condizioni, la *precondizione* e la *postcondizione*, che devono essere vere nello stato di partenza e nello stato di arrivo della base di dati. In una condizione si possono usare i nomi dei parametri o le classi che l'operazione usa o modifica.

Queste informazioni si danno nel formato (*descrittore operazioni*) mostrato in Figura 2.14.

2.3.4 Rappresentazione della comunicazione

Nel modellare la comunicazione con il sistema informativo, ed in particolare con il sistema informatico, si suppone che questa comunicazione avvenga come un dialogo, nel quale il sistema presenta all'utente una *forma* che è caratterizzata da un certo aspetto, dalla presenza di alcuni *campi modificabili* e dalla presenza di alcuni *elementi attivi* (bottoni, menu, barre di scorrimento ecc.). L'utente modifica (o riempie) i campi modificabili oppure agisce sugli elementi attivi, ed il sistema reagisce a queste azioni dell'utente modificando la forma, o ritirandola, o presentando una nuova forma, fino a che l'interazione non termina. Questo modello può essere esteso, supponendo ad esempio che il sistema sia disponibile a presentare all'utente più forme in uno stesso momento, e a permettere all'utente di agire su queste forme in qualunque ordine. Un'ulteriore estensione riguarda la possibilità per l'utente di sottomettere al sistema non solo forme riempite ma anche interrogazioni scritte in un linguaggio formale opportuno, che il sistema esegue. Un'ultima estensione riguarda la possibilità per l'utente di effettuare interazioni meno strutturate, sottoponendo al sistema richieste in forma testuale o con dialoghi vocali in linguaggio naturale. Queste forme di comunicazione non strutturata sono ovviamente essenziali in un sistema informativo in cui siano presenti anche componenti umane.

Operazione	NuovoEsame
Scopo	Immissione dei dati di un nuovo esame
Argomenti	CodiceMateria :string, UnaMatricola :string, DataEsame :date, UnVoto :int, ConLode :bool;
Risultato	(OperazioneEseguita; Errore);
Errori	Materia sconosciuta; Candidato sconosciuto; Voto errato: deve essere fra 18 e 30; Lode solo con voto 30;
Usa	Esami, Studenti, HaSostenuto;
Modifica	Esami, HaSostenuto;
Prima	18 <= UnVoto <= 30; **if** ConLode **then** UnVoto = 30; **some** x **In** Studenti **with** Matricola = UnaMatricola; **Not some** e **In** HaSostenuto **with** (e.Esami.Materia = CodiceMateria **And** e.Studenti.Matricola = UnaMatricola);
Poi	**some** e **In** HaSostenuto **with** (e.Esami.Materia = CodiceMateria **And** e.Studenti.Matricola = UnaMatricola);

Figura 2.14: Esempio di descrittore di operazione.

Il modello ad oggetti è particolarmente adatto a modellare la comunicazione non solo per la possibilità di catturare la natura "attiva" dei componenti delle interfacce grafiche, ma anche perché esso permette di trarre vantaggio dal fatto che tra i tipi degli oggetti che costituiscono un'interfaccia grafica (detti *widget*) sono naturalmente definite delle ricche gerarchie di sottotipo e di eredità.

2.4 Altri modelli dei dati

Vengono presentate le caratteristiche principali di altri modelli dei dati, molto noti nell'area base di dati, per modellare la struttura della conoscenza concreta: il *modello entità-relazione* e il *modello relazionale*.

2.4.1 Il modello entità-relazione

Il modello entità-relazione, proposto da Chen nel 1976, è il modello più popolare per il disegno concettuale di basi di dati. Verrà presentato nella sua forma originaria, estesa successivamente per trattare altri aspetti, in particolare la generalizzazione.

Rispetto al modello ad oggetti, questo modello non tratta aspetti procedurali (metodi) né gerarchie di inclusione tra collezioni, non distingue collezioni e tipi, non supporta alcun meccanismo di ereditarietà. Definisce però un meccanismo per modellare direttamente le associazioni, e in particolare le associazioni non binarie o con proprietà.

Più in dettaglio, il modello prevede due meccanismi di astrazione distinti: uno per modellare insiemi di entità, con le relative proprietà, ed un altro per modellare le associazioni (chiamate "relazioni"). Le collezioni sono qui chiamate *tipi di entità*, e gli attributi dei loro elementi possono assumere solo valori di tipo primitivo.

Il formalismo grafico usato per rappresentare i meccanismi di astrazione del modello entità-relazione è chiamato *diagrammi entità-relazione* (*entity-relationship diagrams*), o *diagrammi ER*, e, come nel caso del modello ad oggetti, i rettangoli rappresentano tipi di entità e gli archi interrotti da un rombo rappresentano associazioni. Anche qui gli archi sono etichettati con le proprietà strutturali delle associazioni, codificate però da una coppia di interi (min, max) che rappresentano, come nei descrittori testuali presentati in precedenza per le associazioni nel formalismo ad oggetti, il numero minimo e massimo di volte che un elemento di un tipo di entità può partecipare alla relazione.

In Figura 2.15 è rappresentata una relazione N ad M tra studenti e corsi, parziale per gli studenti, e totale sui corsi, con in più il vincolo che uno studente può frequentare al più cinque corsi, ed ogni corso deve avere almeno quattro studenti e può averne al più trecento. La relazione tra corsi e corsi integrativi è univoca e totale per i corsi integrativi, mentre è multivalore e parziale per i corsi, con in più il vincolo che ad un corso possono essere collegati al più due corsi integrativi.

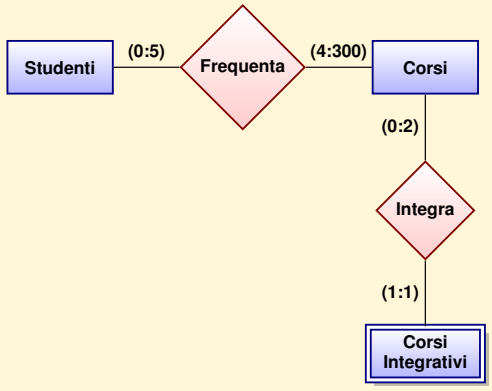


Figura 2.15: Diagramma entità-relazione

Il rettangolo doppio indica un *tipo di entità debole* (*weak entity types*), i cui elementi hanno un'esistenza che dipende da quelli del tipo entità con cui sono in relazione, similmente a quanto accade per gli oggetti composti.

Le relazioni possono essere anche non binarie e avere proprietà proprie.

Questo modello è stato successivamente esteso per trattare attributi con valori non primitivi e le gerarchie fra tipi di entità, rendendo la notazione grafica molto simile a quella di un modello ad oggetti. In Figura 2.16 è mostrato lo schema della biblioteca con un diagramma entità-relazione.

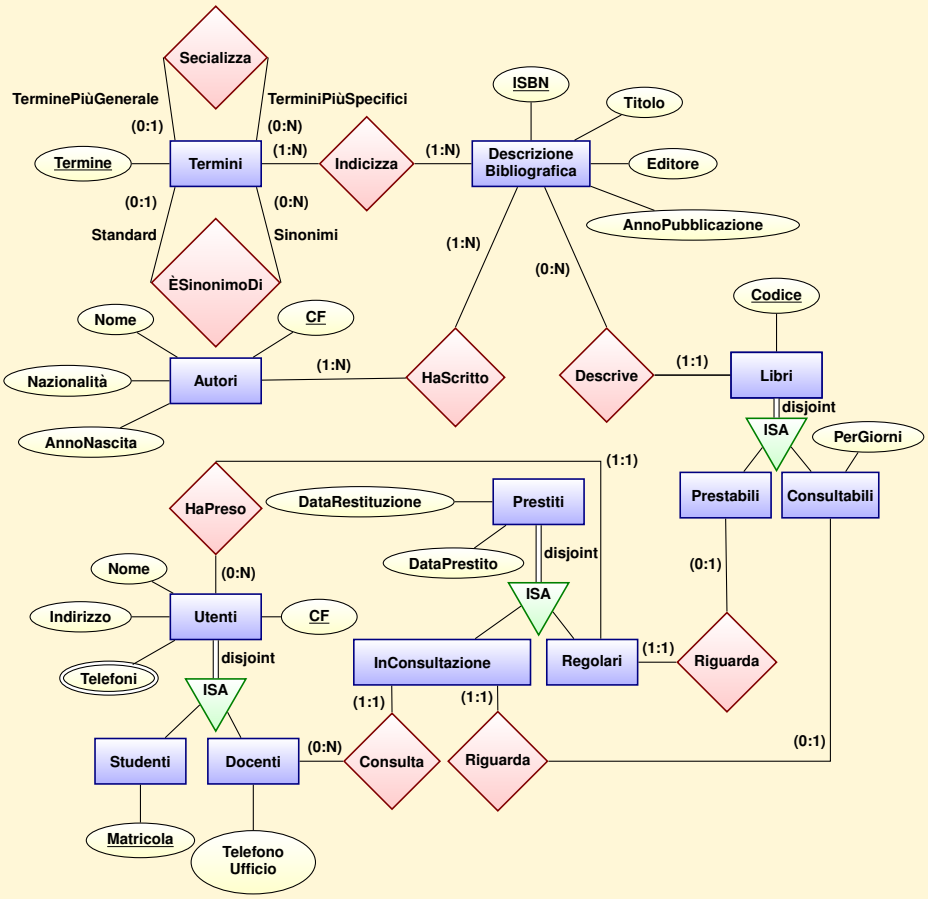


Figura 2.16: Lo schema della biblioteca con un diagramma entità-relazione

2.4.2 Il modello relazionale

Il modello relazionale dei dati è stato proposto nel 1970 ed adottato nei sistemi commerciali a partire dal 1978. È il modello dei dati usato dagli attuali sistemi commerciali.

I meccanismi per definire una base di dati con questo modello sono l'*ennupla* e la *relazione*. Un *tipo ennupla* è un insieme di coppie (attributo, tipo primitivo) ed un valore di tipo ennupla è un insieme di coppie (attributo, valore), dette anche *campi*, con gli

stessi attributi del tipo e in cui il valore di ogni attributo appartiene al corrispondente tipo primitivo. Una *relazione* è un insieme di ennuple con lo stesso tipo.

Un insieme di attributi i cui valori determinano univocamente un'ennupla di una relazione R è una *superchiave* per R; una superchiave tale che togliendo un qualunque attributo essa non sia più una superchiave è una *chiave* per R. Tra le chiavi di R ne viene scelta una come *chiave primaria*. Le associazioni tra i dati sono rappresentate attraverso opportuni attributi, chiamati *chiavi esterne*, che assumono come valori quelli della chiave primaria di un'altra relazione.

Esempio 2.9
Ad esempio, per descrivere il fatto che un esame è associato ad uno studente, nello schema della relazione Esami viene previsto un attributo Candidato che assume come valore la chiave primaria degli Studenti, cioè la Matricola. In Figura 2.17 è data una rappresentazione grafica in cui si evidenzia questo tipo di informazione e in Figura 2.18 si riporta la descrizione completa della struttura delle due relazioni.

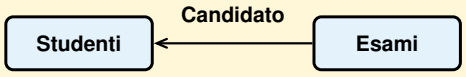


Figura 2.17: Diagramma relazionale

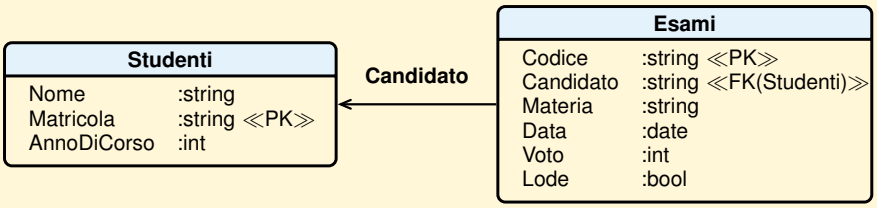


Figura 2.18: Diagramma relazionale con struttura delle relazioni

Il modello relazionale è molto intuitivo, ma l'aspetto più innovativo rispetto ai modelli preesistenti, gerarchico e reticolare, consiste nelle operazioni previste sulle relazioni: queste sono operazioni insiemistiche che restituiscono sempre altre relazioni, inoltre, fra le operazioni primitive ne è prevista una che consente di creare una nuova relazione come giunzione di due relazioni, per combinare ogni ennupla di una relazione con quelle che gli corrispondono nell'altra.

Questo modello si differenzia dal modello ad oggetti in particolare per l'assenza di metodi, di gerarchie di inclusione e di ereditarietà, per il fatto che gli attributi di un'ennupla devono avere un tipo primitivo, e infine perché non esiste un concetto simile all'identità del modello ad oggetti, per cui due ennuple con gli stessi valori degli attributi coincidono e non possono quindi modellare due entità diverse. D'altra parte, questo modello ha una semantica insiemistica estremamente semplice, che lo rende più semplice da capire e particolarmente adatto ad un trattamento formale che

ha consentito lo sviluppo di una teoria della progettazione di basi di dati. I Capitoli 4-5 sono dedicati a questo modello dei dati.

2.5 Conclusioni

Sono stati presentati i vari aspetti da trattare nella costruzione di un modello informatico di un universo del discorso e alcuni meccanismi di astrazione che permettono di modellare alcuni di tali aspetti. In particolare, per la rappresentazione della struttura della conoscenza concreta sono stati presentati i meccanismi di astrazione dei modelli a oggetti, utili per modellare in modo naturale e diretto i fatti interessanti di situazioni complesse.

I meccanismi di astrazione visti si prestano a una semplice rappresentazione grafica, per visualizzare nelle sue linee essenziali lo schema concettuale di una base di dati, in una forma facilmente comprensibile anche da non esperti. Ciò agevola le interazioni dell'analista delle applicazioni con i committenti, specialmente nei primi stadi della progettazione, quando le interazioni sono molto frequenti per chiarire gli obiettivi del sistema informatico che va realizzato. Le rappresentazioni grafiche per un modello ad oggetti o entità-relazioni esteso non sono standard, ma ne esistono diverse ed alcuni editori grafici di ambienti CASE (*Computer Aided Software Engineering*) offrono la possibilità di passare in modo automatico da una notazione all'altra.

Esercizi

1. Discutere le differenze tra le nozioni di tipo oggetto e di classe.
2. Discutere le differenze tra le nozioni di tipo oggetto definito per ereditarietà e di sottoclasse.
3. Discutere i vincoli che si possono descrivere graficamente con il modello ad oggetti.
4. Si definisca uno schema grafico per rappresentare con il modello a oggetti tre insiemi di entità: le persone e i sottoinsiemi delle persone viventi e delle persone decedute. Dare delle proprietà interessanti per gli elementi di questi insiemi. Dire quali problemi si presenterebbero nel modellare questi insiemi se il modello non offrisse né il meccanismo dei tipi oggetti definiti per ereditarietà, né il meccanismo delle sottoclassi.
5. Si vuole automatizzare il sistema di gestione degli animali in uno zoo. Di ogni *esemplare* di animale ospitato, identificato da un codice di inventario, interessano il *genere* (ad es., zebra), la data di arrivo nello zoo, il nome proprio, il sesso, il paese di provenienza e la data di nascita. Lo zoo è diviso in *aree geografiche* di provenienza degli esemplari, con un nome, un responsabile e un insieme di *case*, ognuna destinata ad un solo genere di esemplari. Ogni casa è di un certo un tipo (ad es., recinto, tana, grotta, ecc.) e contiene un insieme di *gabbie* dove vive un solo esemplare. Ogni casa ha un addetto che pulisce ciascuna gabbia in un determinato giorno della settimana. Gli animali sono sottoposti al loro arrivo, e poi periodicamente, a visite

di un veterinario che controlla il peso degli esemplari, diagnostica un'eventuale malattia e prescrive il tipo di dieta da seguire.

Dare uno schema grafico della base di dati.

6. Una banca gestisce informazioni sui mutui dei propri clienti e le rate del piano di ammortamento. Un cliente può avere più di un mutuo. Ai clienti viene inviato periodicamente un resoconto sulle rate pagate del tipo mostrato in figura. Per le rate pagate dopo la data di scadenza è prevista una mora. Si progetti la base di dati e successivamente si modifichi lo schema per trattare anche il fatto che i clienti fanno prima una richiesta di mutuo che poi può essere concesso con un rimborso a rate secondo un certo piano di ammortamento.

RESOCONTO MUTUO

Codice mutuo: 250 **Data**: 07/01/2018
Scadenza: 01/01/2020
Ammontare: 22 000,00
Codice cliente: 2000
Nome cliente: Mario Rossi
Indirizzo: Via Roma, 1 Pisa

Numero rata	Scadenza	Ammontare	Data Versamento
1	01/07/2018	6 000,00	29/06/2018
2	01/01/2019	6 000,00	30/12/2018
3	01/07/2019	6 100,00	29/06/2019
4	01/01/2020	6 100,00	30/12/2019

7. Si vogliono trattare informazioni su attori e registi di film. Di un attore o un regista interessano il nome, che lo identifica, l'anno di nascita e la nazionalità. Un attore può essere anche un regista. Di un film interessano il titolo, l'anno di produzione, gli attori, il regista e il produttore. Due film prodotti lo stesso anno hanno titolo diverso.

   Dare uno schema grafico della base di dati.

8. Un'azienda vuole gestire le informazioni riguardanti gli impiegati, i dipartimenti e i progetti in corso.

   Di un impiegato interessano il codice, assegnato dall'azienda, che l'identifica, il nome e cognome, l'anno di nascita, il sesso e i familiari a carico, dei quali interessano il nome, il sesso, la relazione di parentela e l'anno di nascita.

   Di un dipartimento interessano il numero, che lo identifica, il nome, la città dove si trova.

   Di un progetto interessano il numero, che lo identifica, e il codice. Un progetto è gestito da un solo dipartimento.

   Gli impiegati afferiscono ad un dipartimento, che gestisce più progetti ed è diretto da un impiegato. Gli impiegati partecipano a più progetti, che si svolgono presso dipartimenti di città diverse, ad ognuno dei quali dedicano una percentuale del proprio tempo. Gli impiegati sono coordinati da un responsabile, che è un impiegato. Dei direttori e dei responsabili interessa l'anno di nomina.

Dare uno schema grafico della base di dati.

Note bibliografiche

I modelli dei dati sono trattati in tutti i libri sui sistemi per basi di dati citati nel capitolo precedente. Sulla rete si possono trovare editori grafici per schemi concettuali offerti gratuitamente.

Capitolo 3

LA PROGETTAZIONE DI BASI DI DATI

Progettare una base di dati significa definire lo schema globale dei dati, i vincoli d'integrità e le operazioni delle applicazioni, per preparare la fase successiva in cui la base di dati e le operazioni verranno realizzate.[1] La progettazione di una base di dati è un'attività complessa, per la quale sono state proposte molte metodologie ed ambienti di sviluppo. In questo capitolo viene presentato l'approccio metodologico più consolidato, soffermandosi in particolare sul problema della progettazione concettuale di basi di dati.

3.1 Introduzione

La costruzione di un qualunque sistema complesso, e in particolare di un qualunque sistema informatico, è un processo che si articola in generale in tre fasi:

1. analisi dei requisiti: in questa fase si stabilisce che cosa il committente si aspetta dal sistema;
2. progetto del sistema: in questa fase si progetta un sistema che soddisfi le esigenze del committente;
3. realizzazione del sistema progettato.

Le fasi di analisi dei requisiti e progetto del sistema sono chiamate collettivamente *progettazione* poiché il loro scopo è la produzione di un progetto del sistema da realizzare.

Il problema della progettazione di basi di dati è analogo a quello della progettazione di programmi studiato nell'area dell'ingegneria del software, ed è stato affrontato con strategie simili:

1. La definizione di una metodologia di progettazione organizzata in una sequenza di fasi operative durante le quali il progettista prende un numero limitato di decisioni

1. Un'applicazione è una parte di un sistema informatico finalizzata a soddisfare le esigenze informative di un settore dell'organizzazione. Un'applicazione interagisce con una parte della base di dati (si dice che essa possiede una "vista" della base di dati). Ad un'applicazione è associato un insieme di operazioni degli utenti (le operazioni dell'applicazione).

alla volta. Ogni fase produce una definizione della soluzione ad un diverso livello di dettaglio e prevede l'impiego di opportuni strumenti. Ad esempio, nella metodologia che verrà presentata, nella prima fase (*analisi dei requisiti*) si specificano le informazioni da trattare e il comportamento esterno del sistema; nella seconda fase (*progettazione concettuale*) si raffina questa descrizione, fornendo in particolare uno schema globale dei dati e l'architettura delle operazioni; nella terza fase (*progettazione logica*) si specializza questa descrizione per i meccanismi di astrazione di uno specifico modello dei dati, e nella quarta fase (*progettazione fisica*) vengono aggiunti ulteriori dettagli riguardanti l'organizzazione fisica dei dati.

2. La definizione di strumenti metodologici da usare in ogni fase per espletare i passi della metodologia, per valutare i risultati della fase e per passare dai risultati di una fase a quelli della successiva. Questi strumenti sono di varia natura (procedure manuali, metodi di documentazione, rappresentazioni grafiche, modulistica e strumenti automatici) e finalizzati a scopi diversi a seconda della fase in cui si usano. Essi sono utilizzati per:
   - analisi, progettazione e documentazione, al fine di aiutare il progettista e gli utenti a strutturare le informazioni eliminando ridondanze, segnalando conflitti ed incompletezze;
   - la progettazione di dettaglio, cioè per produrre un progetto di ciò che va realizzato che garantisca un uso efficiente delle risorse, assicurando i tempi di risposta desiderati dalle applicazioni;
   - la realizzazione, per produrre un codice che rispetti le specifiche definite nelle fasi precedenti, per aiutare la coordinazione tra i diversi gruppi impegnati nella realizzazione, e per facilitare la manutenzione ed il test di tale codice.

Il capitolo è strutturato come segue. Nella Sezione 3.2 si approfondisce il ruolo delle metodologie nella produzione di un sistema informatico, e si introduce una metodologia in quattro fasi. Nella Sezione 3.3 vengono introdotti alcuni strumenti formali, diagrammi di flusso dati e di stato, che vengono usati in tutte le fasi della progettazione per rappresentare gli aspetti dinamici del sistema da realizzare. Infine, nelle Sezioni 3.4 e 3.5 vengono presentate in maggior dettaglio le prime due fasi, l'analisi dei requisiti e la progettazione concettuale, della metodologia delineata nella Sezione 3.2.

3.2 Le metodologie di progettazione

Una metodologia di progettazione è una descrizione di come operare per progettare una base di dati. Una metodologia stabilisce in quali fasi dividere la progettazione, come operare per svolgere i compiti propri di ciascuna fase, e quali documenti produrre al termine di ciascuna fase per tenere traccia dei risultati della fase, delle scelte operate durante tale fase e delle motivazioni di tali scelte.

In questa sezione, dopo avere inquadrato il ruolo delle metodologie, viene presentata una metodologia a più fasi e viene discusso il ruolo della prototipazione.

3.2.1 Il ruolo delle metodologie

Per capire pienamente l'importanza di una metodologia di progettazione, si consideri che la progettazione e realizzazione di un qualunque progetto software coinvolge in generale tre figure:

1. il committente, che ha un problema e paga una ditta fornitrice perché lo risolva (il quadro non cambia quando la ditta fornitrice è in realtà una divisione diversa della stessa organizzazione a cui appartiene il committente);
2. il responsabile del progetto, che è un dirigente della ditta fornitrice che dirige il *processo di produzione del software*. Il responsabile in generale non partecipa direttamente alla progettazione, ma fissa assieme al committente gli obiettivi del progetto, alloca tempi e risorse (in particolare risorse umane) alla progettazione stessa, e verifica il buon uso delle risorse ed il rispetto dei tempi e degli obiettivi;
3. il progettista, che, assieme ad altri progettisti, realizza il progetto rispettando i tempi e gli obiettivi fissati dal responsabile, o contratta con il responsabile modifiche a tempi e obiettivi che si rivelino non realistici. In generale gruppi diversi partecipano a fasi diverse della progettazione, e persone diverse possono avvicendarsi anche all'interno di una singola fase o di un singolo compito.

In questo contesto, una metodologia dovrebbe soddisfare i seguenti obiettivi:

1. controllo costante di qualità: la metodologia dovrebbe permettere di verificare costantemente la rispondenza di tutti i progetti intermedi ai requisiti dell'utente, poiché la correzione di un errore in una fase avanzata ha un costo molto superiore rispetto alla correzione di un errore in una fase precedente;
2. supporto al progettista: la metodologia dovrebbe permettere al progettista di sapere in ogni momento come operare, pur senza limitarne in modo eccessivo la libertà di azione;
3. supporto alla gestione: la struttura della metodologia dovrebbe permettere a chi gestisce il progetto di allocare le risorse per le varie fasi riducendo al minimo gli errori di valutazione. Le attività di produzione di documenti (*documentazione*) e di revisione del lavoro dei progettisti da parte di altri, previste da ogni metodologia, dovrebbero permettere a chi gestisce il progetto di verificare costantemente il rispetto dei tempi e degli obiettivi, e dovrebbero permettere di sostituire, in ogni momento, uno o più progettisti, senza che questo pregiudichi l'andamento del progetto;
4. sovraccarico limitato: le attività di documentazione, incontri e revisione previste da ogni metodologia tolgono tempo alla progettazione; queste attività devono essere calibrate in modo tale che i benefici ottenuti superino il costo. A questo scopo, è fondamentale l'utilizzo di strumenti di supporto informatici per la metodologia (*strumenti CASE*) che permettano di automatizzare tutti i compiti ripetitivi e che facilitino la produzione e gestione della documentazione.

In conclusione, le metodologie impongono in genere al progettista, come al programmatore, un forte sovraccarico, superiore di norma al tempo che il progettista utilizza per le attività di progettazione vera e propria. Questo sovraccarico è giustificato però

dal fatto che un'organizzazione che produce sistemi informatici ha bisogno non solo di produrre tali sistemi, ma anche di gestire il processo con cui il risultato viene prodotto.

3.2.2 Le metodologie con più fasi

Per situazioni complesse è inevitabile l'adozione di una metodologia a più fasi in cui i parametri di progetto vengono considerati gradualmente e le varie fasi sono coordinate per ottenere una realizzazione soddisfacente. Questo modo di procedere è basato sul principio che applicazioni complesse si sviluppano per livelli di astrazione progressivamente più dettagliati. L'orientamento più consolidato prevede quattro fasi (vedi Figura 3.1): una di analisi, l'*analisi dei requisiti*, e tre di progettazione in senso stretto, la *progettazione concettuale*, la *progettazione logica* e la *progettazione fisica*.

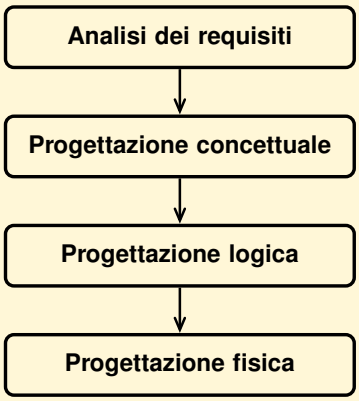


Figura 3.1: Schema di una metodologia a più fasi

- L'analisi dei requisiti serve a definire i bisogni del committente. Durante l'analisi dei requisiti si stabilisce, assieme all'organizzazione committente, quali sono le interazioni che i vari settori dell'organizzazione committente hanno con il sistema informativo dell'organizzazione, e quali sono le nuove interazioni che il commitente desidera rendere possibili. Si raccolgono inoltre i parametri quantitativi relativi alle informazioni gestite, nonché eventuali requisiti rispetto ai tempi di risposta che si richiedono al sistema. Il risultato di questa fase è un documento, la *specifica dei requisiti*, che descrive, in modo non ambiguo e comprensibile dal committente, i bisogni informativi del committente stesso.
- La progettazione concettuale serve a tradurre la specifica dei requisiti in un progetto della struttura concettuale dei dati, dei vincoli e delle operazioni su questi dati. In particolare la struttura dei dati viene descritta utilizzando un formalismo che sia il più naturale possibile, e le operazioni vengono descritte senza effettuare nessuna

scelta di tipo realizzativo, utilizzando formalismi quali quelli che saranno introdotti nella Sezione 3.3. Il risultato principale di questa fase è lo schema concettuale che descrive in maniera formale le informazioni che saranno rappresentate nella base di dati. Anche questo schema deve essere comprensibile dal committente, che lo dovrà approvare. Lo schema concettuale è espresso usando un modello dei dati ad alto livello quale il modello entità-relazioni o il modello a oggetti.

– La progettazione logica serve a tradurre lo schema concettuale nello schema logico, che è espresso nel modello dei dati del sistema scelto per la realizzazione della base di dati. Parallelamente, anche la descrizione delle operazioni viene specializzata rispetto allo schema logico. Il lavoro da svolgere in questa fase dipende molto dalle differenze tra il potere espressivo del modello concettuale e logico dei dati. Maggiori sono le differenze, maggiori saranno le trasformazioni da fare per passare dallo schema concettuale allo schema logico.
– La progettazione fisica serve a produrre lo schema fisico, che arricchisce lo schema logico con le informazioni relative all'organizzazione fisica dei dati.

Si noti che, sebbene le fasi siano concettualmente una successiva all'altra, il processo di progettazione non è lineare, perché può capitare che il progettista ritorni su decisioni prese in precedenza, quando il risultato di una fase non lo soddisfi. Questa operazione è in generale piuttosto costosa, poiché ogni modifica sul risultato di una fase si ripercuote su tutte le fasi che la seguono. Inoltre, anche se sulla scomposizione in quattro fasi c'è un generale accordo, lo stesso non succede per l'ulteriore scomposizione delle fasi in passi. Accade, infatti, che ogni specifica metodologia fissi diverse sequenze di passi per ciascuna fase, con diverse priorità, facendo a volte rientrare in una fase o in un'altra i passi che si collocano ai margini.

Nel processo di progettazione si trattano *aspetti strutturali*, *aspetti dinamici* e *aspetti quantitativi*:

– Gli aspetti strutturali riguardano la struttura della conoscenza concreta e la conoscenza astratta.
– Gli aspetti dinamici riguardano la conoscenza procedurale e la comunicazione.
– Gli aspetti quantitativi riguardano le informazioni che descrivono quantitativamente i fatti rappresentati e il modo in cui essi evolvono nel tempo. Queste informazioni sono importanti nelle fasi di progettazione logica e fisica per utilizzare al meglio il DBMS e per garantire che le applicazioni possano essere realizzate con le prestazioni desiderate. Rientrano in questa categoria sia gli aspetti quantitativi riguardanti la *struttura* della base di dati sia quelli riguardanti le *operazioni*. I primi riguardano una stima (*a*) del numero di entità di ogni tipo, (*b*) del campo di variabilità dei valori delle proprietà e di eventuali correlazioni fra i valori di proprietà delle entità dello stesso tipo, (*c*) del numero di entità coinvolte in ogni associazione. I secondi riguardano una stima della frequenza di attivazione delle operazioni che usano la base di dati e del numero di entità e di istanze di associazione coinvolte.

In Tabella 3.1 è riassunto su quali aspetti si concentrano le diverse fasi. Per ogni fase sono stati sviluppati strumenti automatici per la raccolta delle specifiche, per la loro analisi e documentazione.

	Raccolta requisiti	Progetto concettuale	Progetto logico	Progetto fisico
Aspetti strutturali	Sì	Sì	Sì	Sì
Aspetti dinamici	Sì	Sì	Sì	Sì
Aspetti quantitativi	Sì	No	Sì	Sì
Modello dei dati del DBMS	No	No	Sì	Sì
Caratteristiche del DBMS	No	No	No	Sì

Tabella 3.1: Aspetti trattati nelle varie fasi della progettazione.

3.2.3 Le metodologie con prototipazione

In generale il prodotto principale della progettazione è la specifica della struttura della base di dati e delle operazioni in un linguaggio non eseguibile. Successivamente si passa alla fase di codifica nel linguaggio del DBMS prescelto e poi si procede con le altre fasi tradizionali del processo di sviluppo del software: *validazione*, *prova*, *messa a punto* e *manutenzione* (vedi Figura 3.2a). Questo modo di procedere ha un grosso inconveniente: gli utenti possono verificare che il prodotto soddisfa le loro aspettative solo dopo che una prima realizzazione sia funzionante. Pertanto eventuali incomprensioni nella fase di raccolta dei requisiti, oppure la mancanza di chiarezza degli obiettivi da parte dei committenti, affiorano quando la realizzazione è ormai già iniziata e i costi delle modifiche per adeguarla ai bisogni reali diventano maggiori.

Per evitare questo problema è stato proposto un modo diverso di affrontare la progettazione e realizzazione delle applicazioni, che prevede la costruzione di prototipi nelle fasi iniziali della metodologia tradizionale; ad esempio, dopo l'analisi dei requisiti, per fare prototipi delle interfacce, oppure dopo la progettazione concettuale per fare un prototipo della base di dati. Un prototipo è una versione preliminare e funzionante di ciò che va realizzato, che esibisca le caratteristiche funzionali essenziali, prescindendo dai requisiti sulle prestazioni. Ovviamente, affinché un impegno di prototipazione risulti economicamente vantaggioso, il costo della realizzazione dei prototipi deve essere una piccola frazione del costo di una prima realizzazione delle applicazioni sul DBMS prescelto. Ciò è possibile usando strumenti specializzati per la produzione rapida di prototipi. Disponendo rapidamente di un prototipo, sperimentabile dagli utenti in una fase iniziale del processo di progettazione, si è in grado di verificare se il prodotto che si sta realizzando soddisfa le aspettative, prima di passare

alla sua realizzazione, e di chiarire così prima possibile quali sono i reali bisogni degli utenti.

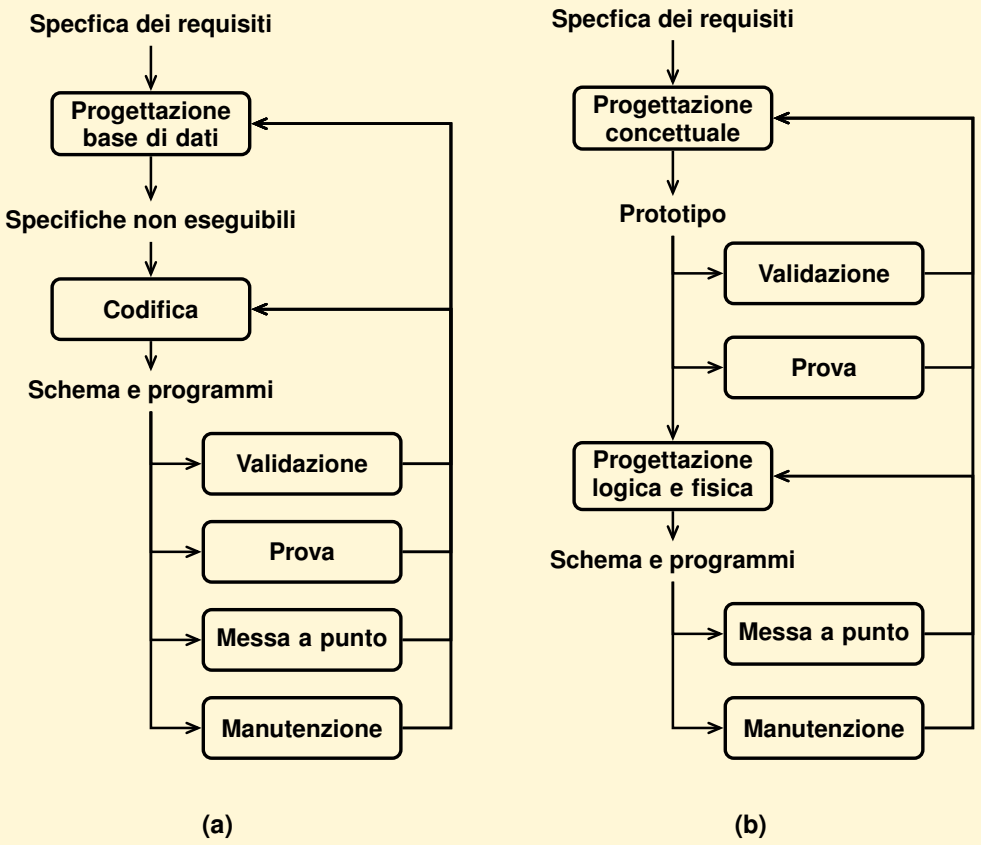


Figura 3.2: Approccio tradizionale e con prototipazione

Il prototipo della base di dati può essere del tipo "usa e getta", nel senso che non ha altro impiego nelle fasi successive della metodologia, oppure può essere trasformato in un prototipo espresso a livello di progetto logico e costituire così una specifica operazionale della base di dati e delle applicazioni, oppure ancora può essere il punto di partenza per un modo diverso di realizzare le applicazioni, come schematizzato in Figura 3.2b: il codice delle applicazioni viene ottenuto applicando al prototipo una serie di trasformazioni automatiche, o semi automatiche. Sul codice così prodotto rimarrà solo da compiere un passo di completamento per trattare tutti quegli aspetti delle applicazioni non considerati nel passo di prototipazione.

3.3 Gli strumenti formali

Le metodologie presentate adottano meccanismi di astrazione e formalismi diversi durante le varie fasi del processo di progettazione. Questi formalismi ricadono principalmente nelle seguenti categorie:

- diagrammi per la descrizione dei dati: servono a descrivere la struttura dei dati; i diagrammi entità-relazioni e la rappresentazione grafica del modello a oggetti definiti nel Capitolo 2 sono esempi tipici;
- diagrammi di flusso dati: servono a descrivere le operazioni, mostrando in particolare come ognuna possa essere divisa in sottooperazioni e qual è il flusso dei dati tra queste operazioni;
- diagrammi di stato: servono a descrivere in che modo lo stato del sistema, o meglio lo stato di una qualche componente del sistema, evolve in corrispondenza di un evento; servono a specificare sistemi che reagiscono ad eventi, quindi in particolare la parte di un'applicazione che gestisce il dialogo con l'utente.

I diagrammi di flusso dati ed i diagrammi di stato sono introdotti, senza pretese di completezza, nelle prossime sezioni.

3.3.1 I diagrammi di flusso dati

Un diagramma di flusso dati (*Data Flow Diagram, DFD*) rappresenta un sistema in cui alcuni *processi*, in cooperazione tra loro e con l'ausilio di alcuni *depositi dati* (*data store*), svolgono determinate funzioni e comunicano con alcuni agenti esterni detti *interfacce*, o *attori*. Processi, depositi dati ed interfacce sono rappresentati con le convenzioni grafiche riportate in Figura 3.3.

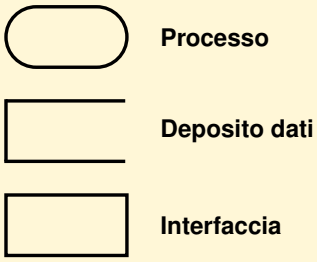


Figura 3.3: Simbologia dei diagrammi di flusso dati

Un diagramma di flusso dati rappresenta esplicitamente il flusso dei dati tra i vari processi e tra questi, i depositi e le interfacce, tuttavia *non specifica* il flusso del controllo o la temporizzazione, ovvero non specifica né il fatto che un flusso dati avviene prima di un altro, né il fatto che un flusso dati avviene solo sotto certe condizioni.

Un primo esempio di diagramma di flusso dati è riportato in Figura 3.4. Questo diagramma rappresenta il fatto che il processo "gestione esami" (rappresentato attraverso un ovale) scambia informazioni con le interfacce "studente" e "docente". Ad esempio, è presente un flusso "consegna certificato" tra la gestione esami e lo studente, ed un flusso "richiesta certificato" nella direzione opposta. Si osservi che i due flussi sono simmetrici, per cui il diagramma non specifica formalmente l'esistenza di una relazione causale o neppure temporale tra i due flussi. Il diagramma ha un'effettiva utilità solo perché si suppone che queste informazioni relative al flusso del controllo siano deducibili da chi legge il diagramma a partire dalle proprie conoscenze o da altri documenti meno astratti, quali un diagramma di stato.

Nell'analisi e progettazione degli aspetti dinamici di un sistema informativo (*analisi dinamica* o *analisi funzionale*) si parte in genere da un diagramma di flusso dati in cui l'intero sistema è rappresentato da un unico processo sul quale convergono tutte le richieste delle interfacce (gli agenti esterni), come quello rappresentato in Figura 3.4; questo particolare diagramma è chiamato il *diagramma di contesto*.

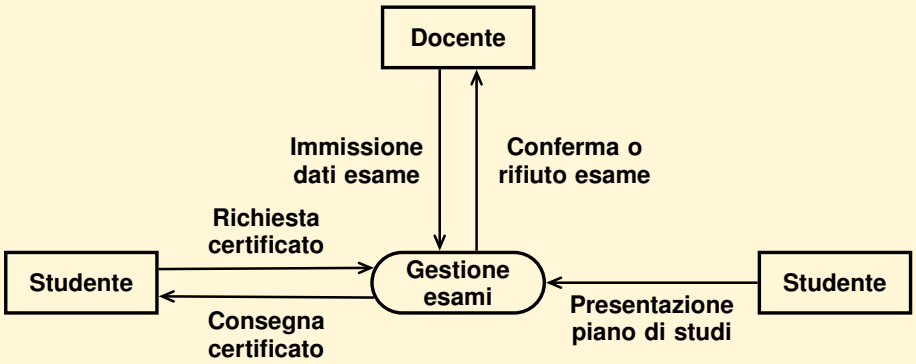


Figura 3.4: Diagramma di contesto

Il passo successivo di raffinamento è costituito in genere da un diagramma in cui ogni interazione con un'interfaccia è rappresentata da un singolo processo. A questo livello di astrazione si iniziano ad inserire i depositi dati, che sono utilizzati dai processi per memorizzare informazioni o per reperire informazioni memorizzate in precedenza; la prima operazione è rappresentata da un flusso verso il deposito, la seconda da un flusso in senso inverso (Figura 3.5).

A questo punto ogni processo può essere suddiviso, se questo risulta utile, in più sottoprocessi che comunicano attraverso flussi dati o attraverso depositi, come esemplificato in Figura 3.6 per il processo "verifica e immissione dati esami".

Il momento in cui interrompere questa suddivisione dipende dallo scopo del diagramma, ovvero dal livello di astrazione al quale si è interessati. In generale, si segue il criterio dell'*indipendenza funzionale* tra processi, che specifica che (*a*) ogni sottoprocesso deve fare una sola cosa e (*b*) che sottoprocessi diversi devono scambiare meno

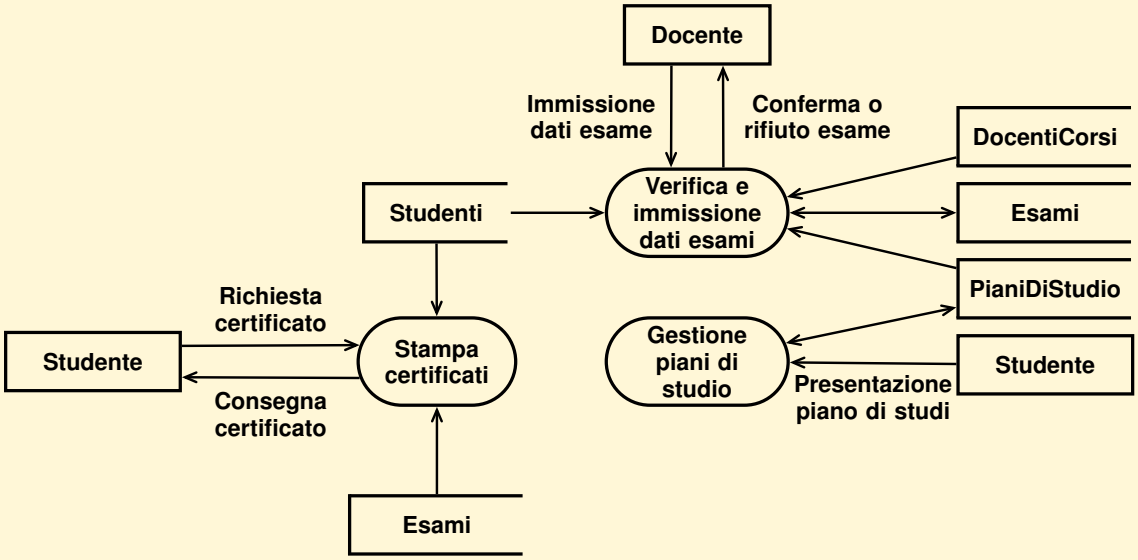


Figura 3.5: Diagramma di flusso, versione astratta

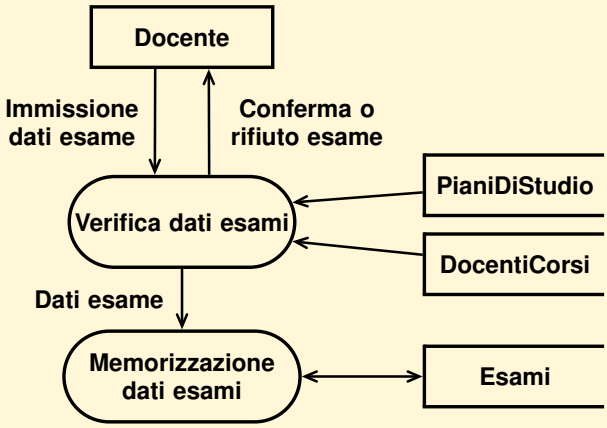


Figura 3.6: Diagramma di flusso, versione meno astratta

dati possibile. Il criterio (*a*) indica in quali casi è necessario scomporre un processo, mentre il criterio (*b*) indica in quali casi la scomposizione si debba interrompere.

L'analisi delle funzioni è strettamente collegata all'analisi dei dati, ovvero alla produzione dello schema. Alcune metodologie prevedono infatti che l'analisi delle funzioni sia eseguita per prima, poi si analizzi lo schema di ogni singolo deposito dati nominato nell'analisi funzionale, ed infine si integrino questi schemi per produrre lo

schema concettuale globale.

Esempio 3.1
Ad esempio, l'analisi separata dei cinque depositi del diagramma di Figura 3.5 potrebbe produrre i cinque sottoschemi della Figura 3.7, quattro con una sola classe ed uno di due classi, i quali possono essere poi integrati, come descritto nella sezione 3.5.3, per produrre lo schema in Figura 3.8.

In questo esempio abbiamo volutamente estremizzato gli effetti dell'analisi separata dei depositi, per rendere più chiaro il fatto che l'integrazione di questi schemi comporta alcune ristrutturazioni degli schemi stessi. Ad esempio, in entrambi gli schemi per i due depositi chiamati Esami, invece di riconoscere l'esistenza di una seconda classe Studenti cui appartiene l'attributo Matricola, questa viene considerata un attributo dell'esame. Questo modello è ragionevole solo se si considera il deposito Esami in maniera completamente isolata, ma è scorretto non appena si considerano anche gli studenti come entità di interesse.

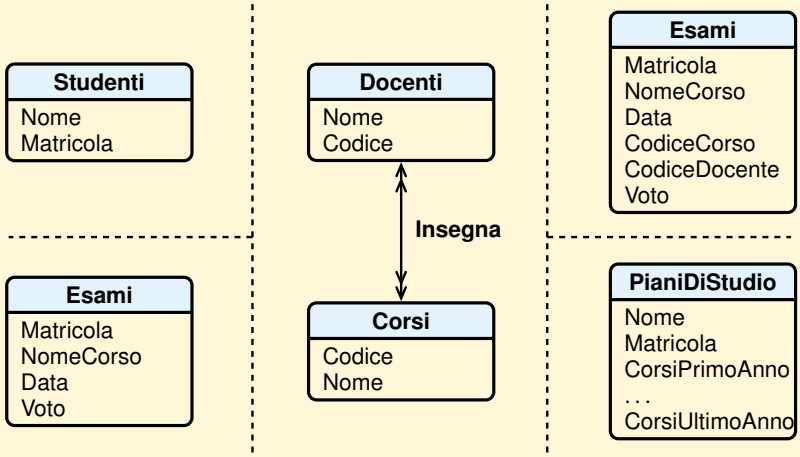


Figura 3.7: Schemi dei cinque depositi indicati nel diagramma precedente

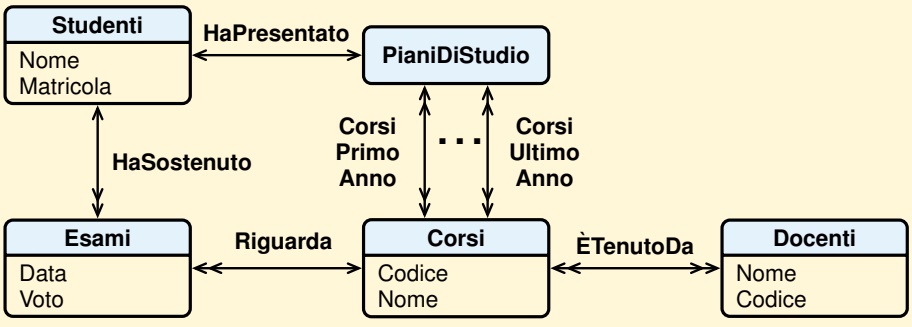


Figura 3.8: Schema prodotto integrando i cinque sottoschemi

Questa metodologia viene criticata, sulla base del fatto che l'insieme di operazioni che si richiedono ad un sistema informativo tende a cambiare nel tempo, e a produrre quindi schemi concettuali sempre diversi, ed ogni modifica dello schema concettuale ha un costo molto alto, poiché costringe a rieseguire tutte le fasi successive. Sono state quindi proposte metodologie in cui l'analisi dei dati, ovvero il disegno dello schema concettuale, viene eseguita tenendo conto per prima cosa della struttura della realtà che si vuole modellare con la base di dati, utilizzando in parallelo l'analisi funzionale per verificare di avere incluso tutte le informazioni necessarie ad implementare i depositi dati richiesti dalle operazioni.

Per maggiori dettagli sui diagrammi di flusso dati si vedano ad esempio [Yourdon, 1989; Batini et al., 1992; Marco, 1979; Rumbaugh et al., 1991].

3.3.2 I diagrammi di stato

Un diagramma di stato (*state diagram*) è un grafo etichettato che modella il modo in cui un sistema reagisce ad eventi esterni, ed è quindi particolarmente indicato per modellare il comportamento delle interfacce di un sistema informatico, ma anche per modellare in che modo un oggetto reagisce ai messaggi che riceve.

Un sistema viene modellato supponendo che esso si trovi in ogni momento in un determinato *stato*, durante il quale esso svolge un'*attività*; il sistema permane in questo stato fino a che un *evento* non provoca una transizione di stato. La differenza tra stati e transizioni è che il sistema resta in uno stato per un tempo non trascurabile, mentre la durata di una transizione di stato è trascurabile (la transizione è "istantanea"). Quindi, come verrà esemplificato tra poco, la differenza tra una transizione ed una terna <transizione, stato intermedio, transizione> non è sempre ovvia, e dipende in generale dal livello di astrazione a cui ci si pone.

Ad esempio, il diagramma in Figura 3.9 specifica che l'operazione di registrazione esami passa attraverso tre stati, uno di acquisizione dati, uno di verifica dati ed uno di memorizzazione dati. Si osservi che, mentre gli eventi "immissione completa" e "richiesta abort" sono di origine esterna, gli eventi "fallimento verifica", "dati verificati" e "dati memorizzati" sono in effetti eventi generati dall'applicazione stessa. La notazione "e/*a*" indica che il sistema reagisce all'evento "e" compiendo contemporaneamente una transizione di stato ed un'*azione istantanea* "a". La freccia con un pallino a sinistra indica lo stato iniziale del sistema.

Anche i diagrammi di stato si prestano bene alla progettazione per raffinamento, scomponendo uno stato in sottostati. In Figura 3.10 è esemplificata una notazione utilizzata a questo scopo, che permette ad esempio di associare direttamente ad un superstato (cioè ad uno stato che è stato raffinato) le transizioni che riguardano tutti i suoi sottostati (si consideri ad esempio la transizione attivata dall'evento "richiesta abort").

Nella stessa figura si mostra anche come, ad un maggior livello di dettaglio, un'azione associata ad una transizione ("segnala abort") possa essere anche considerata un'operazione eseguita dal sistema in un certo stato. Ad un livello successivo, si potrebbe effettuare un'ulteriore scomposizione anche dell'operazione "segnala abort", che probabilmente visualizza una finestra con un messaggio di errore e attende una

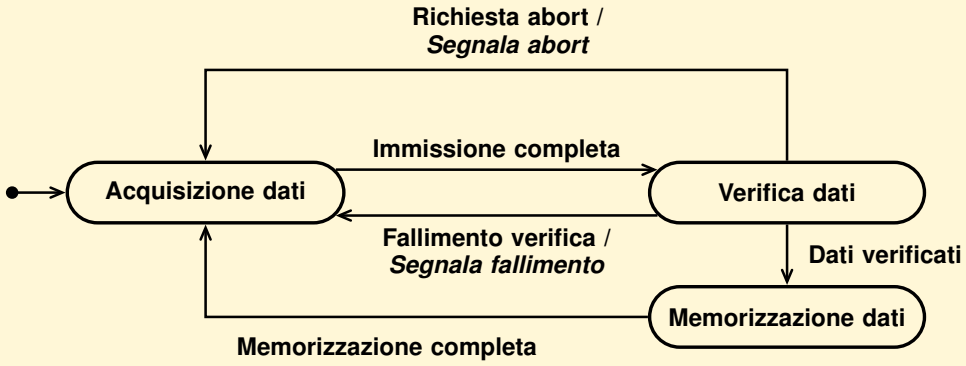


Figura 3.9: Un diagramma di stato

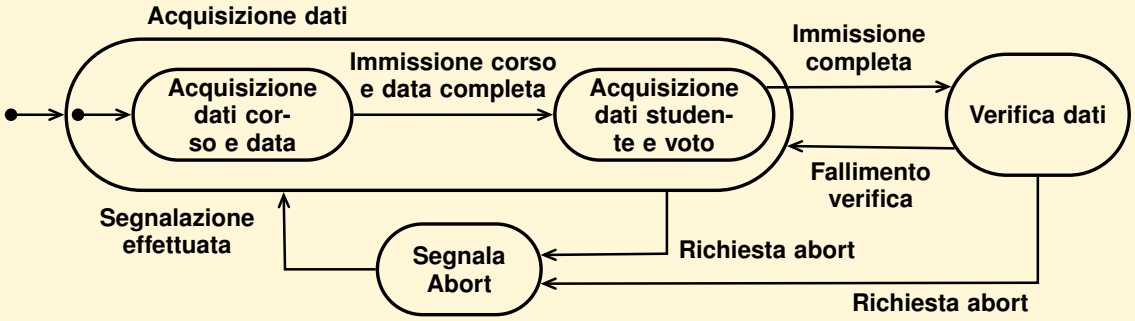


Figura 3.10: Diagramma di stato con sottostati

conferma dall'utente, mentre in parallelo memorizza l'avvenuto abort su di un archivio opportuno. Un'analisi analoga potrebbe essere eseguita per l'azione "segnala fallimento" associata all'evento "fallimento verifica".

Per i diagrammi di flusso sono stati definiti anche degli altri meccanismi di strutturazione piuttosto sofisticati, ad esempio per rappresentare il parallelismo che è concettualmente insito in molti sistemi. Per questi dettagli, e per approfondimenti, rimandiamo, ad esempio, a [Harel, 1987; Rumbaugh et al., 1991].

Riassumendo, la differenza fondamentale tra i diagrammi di stato ed i diagrammi di flusso dati consiste nel fatto che i primi rappresentano il flusso del controllo, che viene essenzialmente ignorato nei secondi, mentre questi si concentrano sul flusso e sulla memorizzazione dei dati, ignorati nei primi. I due formalismi non sono comunque del tutto ortogonali e i diagrammi di stato sono meno utilizzati, rispetto a quelli di flusso dati, nelle metodologie per la progettazione di basi di dati.

3.4 L'analisi dei requisiti

In questa sezione e nella successiva presentiamo le caratteristiche principali delle fasi di analisi dei requisiti e di progettazione concettuale, per dare dei suggerimenti su come procedere nella definizione dello schema concettuale di una base di dati. L'attenzione sarà sostanzialmente rivolta agli aspetti strutturali (progettazione dei dati) con cenni agli aspetti dinamici (progettazione delle procedure). Infine, non verranno prese in considerazione le fasi di progettazione logica e fisica, perché una trattazione di una metodologia completa esula dai fini di questo testo; per approfondire l'argomento si rinvia alla letteratura segnalata nelle note bibliografiche.

3.4.1 Scopo dell'analisi dei requisiti

L'analisi dei requisiti presuppone un adeguato studio preliminare dell'organizzazione, e delle sue finalità, che abbia valutato i seguenti aspetti:

1. gli *obiettivi* del sistema informatico da realizzare;
2. le *attività* dell'organizzazione che devono essere supportate dal sistema informatico;
3. le *unità organizzative* (settori o aree funzionali) che utilizzeranno il sistema informatico;
4. il *piano di sviluppo* del sistema informatico, ed uno *studio di fattibilità* che abbia stimato i costi e i tempi di tale piano di sviluppo, per verificarne l'effettiva convenienza.

Il risultato di questo studio preliminare riguarda pertanto scelte fondamentali dell'organizzazione e del suo modo di funzionare, presumibilmente invarianti nel medio termine, dalle quali non si può prescindere nell'affrontare l'analisi dei requisiti. Con questo studio, inoltre, vengono già individuati, ad un massimo livello di astrazione, i bisogni informativi, le procedure di interesse e le loro interazioni con i settori dell'organizzazione.

L'analisi dei requisiti raffina i risultati dello studio preliminare specificando in maniera più precisa ciò che l'organizzazione, ed i vari settori che la compongono, si aspetta dal sistema informatico in corso di progettazione, arrivando a delineare in particolare la struttura delle informazioni da trattare e le procedure che dovranno essere realizzate.

Il risultato della fase di analisi dei requisiti consiste in una serie di documenti che specificano:

- Le informazioni scambiate o condivise tra i settori, fissando il significato dei termini. Molto spesso, infatti, passando da un settore ad un altro, lo stesso termine è usato con significati diversi (omonimi) oppure termini diversi denotano la stessa cosa (sinonimi).
- La struttura della conoscenza concreta e astratta.

– La conoscenza procedurale, ponendo l'attenzione sui servizi che il sistema deve offrire ai suoi utenti per definire cosa devono fare le operazioni, piuttosto che come lo fanno, e sulle loro modalità di attivazione.
– Aspetti quantitativi riguardanti la struttura della base di dati e le modalità d'uso dei dati.
– Fatti riguardanti il grado di privatezza e di sicurezza da prevedere sui dati.

Il risultato della fase di analisi dei requisiti può essere dato in modi diversi, i quali possono anche coesistere: (*a*) informalmente, con linguaggio naturale ristretto, (*b*) con opportune tabelle integrate da rappresentazioni diagrammatiche o (*c*) con un linguaggio formale non eseguibile. Altre differenze fra le varie proposte riguardano i meccanismi di astrazione su cui si basa il linguaggio di specifica per trattare gli aspetti strutturali e dinamici.

3.4.2 Come procedere

La fase di analisi dei requisiti richiede un forte impegno e una notevole esperienza da parte del progettista. Egli acquisisce familiarità con il funzionamento dell'organizzazione utilizzando fonti diverse: interviste agli utenti, analisi di documenti esistenti (modulistica, archivi cartacei, ordini di servizio, schemi dei dati di realizzazioni già operative). Le informazioni sono di solito raccolte svolgendo due tipi di attività che in linea di principio possono procedere in parallelo, ma che spesso si intrecciano e si influenzano a vicenda: l'*analisi dei dati* e l'*analisi funzionale*.

Con l'analisi dei dati, il progettista si dedica inizialmente alla comprensione e alla specifica degli aspetti strutturali, in particolare di quei fatti da memorizzare nella base di dati, cercando di darne una descrizione indipendente da come essi vengono utilizzati dalle operazioni. Successivamente si specificano le operazioni controllando che siano eseguibili con le informazioni disponibili.

Con l'analisi funzionale il progettista specifica inizialmente le operazioni delle applicazioni per ricavare, poi, una descrizione delle informazioni necessarie per eseguirle.

Come già detto in precedenza, l'analisi dei dati e l'analisi funzionale possono essere eseguite in qualunque ordine, purché si effettui, alla fine, un controllo di coerenza tra i risultati delle due fasi; in pratica, risulta conveniente effettuare queste due analisi con un certo grado di parallelismo, tenendo presenti i risultati intermedi dell'una mentre si esegue l'altra.

Sono stati proposti vari strumenti automatici, spesso integrati in ambienti CASE, per agevolare il trattamento e la gestione memorizzazione e l'aggiornamento delle varie forme di specifiche, per segnalare incompletezze e inconsistenze, per generare come documentazione opportune tabelle riassuntive e di incroci.

Nel seguito si mostra un esempio di metodologia di analisi dei requisiti, focalizzata sull'analisi dei dati. La metodologia è descritta informalmente ed esemplificata attraverso l'analisi di un'applicazione per la gestione degli studenti di un'università.

Per ogni settore aziendale si procede con i seguenti passi:

1. Si analizza il sistema informativo esistente raccogliendo una prima versione dei requisiti, espressa in linguaggio naturale.

2. Si rivedono i requisiti espressi in linguaggio naturale, per eliminare ambiguità, imprecisioni e disuniformità linguistiche.
3. Si raggruppano le frasi relative a categorie diverse di dati, vincoli e operazioni.
4. Si costruisce un glossario dei termini (questo passo viene in realtà eseguito contemporaneamente ai due precedenti).
5. Si definisce uno schema preliminare di settore, detto *schema scheletro*, con il quale si individuano ad un primo livello di dettaglio le classi e le associazioni fra le classi più significative.
6. Si specificano le operazioni degli utenti.
7. Si verifica la completezza (tutti gli aspetti dei requisiti sono stati presi in considerazione) e consistenza (tutti i concetti usati nella specifica sono stati definiti, in particolare le operazioni fanno riferimento a dati definiti e i dati definiti sono usati dalle operazioni).

3.4.3 Un esempio di analisi dei requisiti

Immaginiamo di voler progettare una base di dati per la gestione della segreteria studenti di un Corso di Laurea in Informatica, e di effettuare l'analisi dei requisiti relativamente al settore che si occupa delle richieste di trasferimento al Corso di Laurea, occupandoci in particolare di quelle informazioni che servono alla progettazione della struttura dei dati.

Analisi del sistema informativo esistente e raccolta di una versione dei requisiti in linguaggio naturale (passo 1)

L'analisi del sistema informativo esistente permette al progettista di acquisire familiarità con i compiti svolti nei settori interessati all'automazione delle applicazioni, interagendo con gli utenti dei diversi livelli gerarchici per comprendere le loro esigenze e per far comprendere come potrà essere influenzato il loro lavoro. Il coinvolgimento degli utenti è importante per avere la loro collaborazione nella raccolta dei requisiti, perché imprecisioni in questa fase avranno notevoli ripercussioni sul costo complessivo della realizzazione.

Nel nostro esempio, intervistando il personale, si è ottenuta la seguente descrizione delle informazioni da gestire e dei servizi da prevedere.

Questa segreteria si occupa della gestione di alcune pratiche relative agli studenti del Corso di Laurea in Informatica. In particolare, noi gestiamo le richieste di trasferimento di studenti che provengono da altri Corsi di Laurea. Quando uno studente desidera trasferirsi da noi, presenta domanda al Dipartimento di appartenenza, che la trasmette al nostra Dipartimento, che poi la trasmette a questa segreteria. La segreteria apre una pratica e trasmette la domanda alla commissione Pratiche Studenti del Consiglio del Corso di Studio (CCS) che prepara una bozza di delibera, nella quale si specificano quali esami vengono riconosciuti allo studente, eventualmente con un colloquio integrativo. Nel convalidare gli esami, si tiene conto del modo in cui esami analoghi sono stati convalidati in passato. Normalmente, cerchiamo di trattare nello stesso modo esami con lo stesso nome fatti nello stesso Dipartimento, ma possono esserci

delle eccezioni quando i contenuti sono diversi. Per ciò che riguarda le informazioni da trattare, una domanda di trasferimento ha un numero di protocollo, e contiene il nome e il recapito dello studente che la presenta, il Corso di Laurea di provenienza, la data di presentazione, e l'elenco degli esami esterni superati. Il numero di protocollo è assegnato da noi ed è diverso da pratica a pratica. Per ogni domanda di trasferimento viene creata una pratica di trasferimento, che ha un proprio numero d'ordine e che contiene la domanda di trasferimento ed eventuali nostre annotazioni, e conterrà poi la delibera relativa. Una bozza di delibera specifica come sono convalidati gli esami dello studente, e può contenere delle annotazioni. Una delibera approvata contiene anche il numero e la data del verbale del CCS che ha approvato tale delibera. Un esame da convalidare è caratterizzato da un'Università, un Dipartimento, un Corso di Laurea, un Nome, un Anno Accademico. L'Università, il Dipartimento e il Corso di Laurea dove l'esame è stato superato non coincidono necessariamente con l'Università di provenienza, il Dipartimento di provenienza e il Corso di Laurea di provenienza della domanda a cui l'esame esterno appartiene. Gli esami interni hanno un nome e un codice univoco. Quando un esame esterno è convalidato per un esame interno, la convalida può essere "piena" o "previo colloquio". Per alcuni esami esterni esiste una "convalida tipica". Ad esempio, la convalida tipica dell'esame di Fisica I a Ingegneria Elettronica, Dipartimento di Ingegneria, è "Fisica Generale". La convalida tipica sarebbe utile per la preparazione automatica della bozza di delibera. La convalida tipica può essere "previo colloquio". Non sempre un esame dato da uno studente è convalidato secondo la sua convalida tipica.

Siamo interessati ad un sistema che permetta di memorizzare tutte le informazioni relative alle pratiche di trasferimento che vengono via via sbrigate. Questo sistema dovrebbe essere anche in grado di preparare una bozza di verbale a partire dall'elenco degli esami allegato ad una domanda di trasferimento, lasciando però alla segreteria la possibilità di intervenire per modificare i riconoscimenti proposti dal sistema.

Per ottenere la specifica dei requisiti, questa descrizione va rielaborata per eliminare le informazioni non rilevanti, aggiungere quelle mancanti, eliminare sinonimie (termini diversi che indicano la stessa cosa), omonimie (termini uguali che indicano cose diverse) ed ambiguità. Inoltre, è bene che la specifica dei requisiti sia scritta utilizzando frasi con una struttura il più possibile elementare ed omogenea, come nell'esempio che segue.

Revisione dei requisiti e raggruppamento delle frasi (passi 2 e 3)
La revisione dei requisiti produce la seguente specifica.

Frasi di carattere generale *Si vogliono gestire informazioni relative a domande di trasferimento ed alla corrispondenza tra corsi esterni e corsi interni.*

Frasi relative alle domande di trasferimento *Di una domanda di trasferimento interessano: il numero di protocollo, che la identifica, il nome e il recapito dello studente che la presenta, l'Università di provenienza, il Dipartimento di provenienza, il Corso di Laurea di provenienza, la data di presentazione, l'elenco dei corsi esterni per i quali è stato superato l'esame.*

Frasi relative alle pratiche di trasferimento *Di una pratica di trasferimento interessano: la domanda di trasferimento cui si riferisce, il numero progressivo che la identifica, le eventuali annotazioni e l'eventuale delibera relativa.*

Frasi relative alle delibere *Di una delibera interessa: la pratica relativa, l'insieme di convalide di esami, le eventuali annotazioni, la data ed il numero del verbale del CCS che ha approvato il trasferimento (che può mancare se si tratta solo di una bozza).*

Frasi relative ai corsi esterni *Di un corso esterno interessano: il nome del corso, l'Anno Accademico, l'Università, il Dipartimento e il Corso di Laurea dove l'esame relativo al corso è stato superato. L'Università, il Dipartimento e il Corso di Laurea dove l'esame è stato superato non coincidono necessariamente con l'Università di provenienza, il Dipartimento di provenienza e il Corso di Laurea di provenienza della domanda di trasferimento a cui il corso esterno appartiene.*

Frasi relative ai corsi interni *Di un corso interno interessano: il nome e il numero di crediti.*

Frasi relative ad una convalida di un esame *Di una convalida di un esame interessano: il corso esterno, il corso interno e sapere se l'esame è stato convalidato "previo colloquio". Il corso esterno di una convalida è detto "convalidato per" il corso interno.*

Frasi relative ad una convalida tipica *Di una convalida tipica interessano: il corso esterno, il corso interno e se l'esame viene convalidato "previo colloquio".*

La convalida tipica non vincola le convalide degli esami, ma può essere utilizzata per la preparazione automatica di una prima versione della bozza di delibera a partire da una domanda di trasferimento.

Frasi relative alle operazioni *Il servizio che il sistema deve offrire agli utenti prevede le seguenti operazioni, per le quali andranno definite opportune modalità di interazione da concordare con gli interessati (tra parentesi è riportata una stima della frequenza d'uso):*

1. *Inserimento di corsi interni (3 volte all'anno).*
2. *Immissione di una domanda di trasferimento (70 volte all'anno).*
3. *Generazione di una bozza di delibera a partire dall'elenco dei corsi esterni superati allegato ad una domanda (70 volte all'anno).*
4. *Correzione manuale di una bozza di delibera (30 volte all'anno).*
5. *Trasformazione di una bozza di delibera in delibera approvata (70 volte all'anno).*

Costruzione del glossario (passo 4)

Come già detto, il glossario viene in realtà costruito durante la fase di revisione dei requisiti. In particolare, ogni volta che si sceglie, in un gruppo di sinonimi, quello che

verrà utilizzato nella specifica dei requisiti, questa scelta deve essere riportata subito nel glossario, come esemplificato sotto per i termini "Riconoscimento di esame" e "Convalida di esame".

Si riportano, a titolo di esempio, le definizioni di alcuni dei termini usati nella specifica dei requisiti:

CCS Consiglio di Corso di Studio

Bozza di delibera Versione preliminare di una delibera relativa alla domanda di trasferimento di uno studente. Specifica un insieme di convalide per i corsi esterni superati dallo stesso studente. Può essere modificata. Diventa una "delibera approvata", immutabile, dopo che è stata approvata dal CCS. *Sinonimi*: Bozza di verbale.

Bozza di verbale *Usa*: Bozza di delibera.

Convalida di esame Parte di una delibera; stabilisce che l'esame esterno X è convalidato per il superamento dell'esame di un corso interno Y, eventualmente previo il superamento di un colloquio. *Sinonimi*: Riconoscimento di esame.

Corso esterno Corso attivato presso un'istituzione universitaria o assimilata, ma non presso il Corso di Laurea in Informatica dell'Università in questione; ad esempio: il corso di "Analisi I" attivato dal Corso di Laurea in Ingegneria Elettronica, Dipartimento di Ingegneria, Università di Padova, nell'Anno Accademico 2004/05.

Corso esterno superato Un corso esterno è stato superato da uno studente quando lo studente ha superato con successo l'esame relativo a tale corso.

Corso interno Corso attivato presso il Corso di Laurea in Informatica dell'Università in questione.

Crediti Unità di misura della dimensione di un corso, in relazione all'impegno necessario ad uno studente medio per la preparazione al corso. L'impegno totale dei corsi da superare in un anno è, di norma, di 60 crediti.

Domanda di trasferimento Domanda con la quale uno studente iscritto in altro Corso di Laurea chiede il trasferimento a questo Corso di Laurea, chiedendo che gli vengano convalidati alcuni esami che ha superato (esami esterni).

Esame Il processo attraverso cui si verifica che lo studente abbia appreso i contenuti di un corso. Quando la verifica dà esito positivo, l'esame è *superato*.

Esame esterno Esame relativo ad un corso esterno, superato prima di fare la domanda di trasferimento. Non è stato necessariamente superato presso il Corso di Laurea al quale lo studente era iscritto nel momento in cui ha presentato la domanda di trasferimento.

Riconoscimento di esame *Usa*: Convalida di esame.

Definizione dello schema scheletro (passo 5)

Lo schema scheletro di settore può essere definito a diversi livelli di dettaglio. Alcune metodologie più recenti orientate agli oggetti prevedono di terminare la specifica dei requisiti con uno schema dei dati completo, usando un formalismo come quello suggerito nel capitolo precedente. Altre metodologie suggeriscono invece di limitarsi in questa fase ad uno schema dei fatti essenziali, da completare poi nella fase di pro-

gettazione concettuale, e altre ancora non prevedono uno schema dei dati nella fase di analisi dei requisiti. Si preferisce non entrare nel merito di cosa convenga definire in uno schema scheletro e nello sviluppo dell'esempio si decide di considerare come fatti essenziali le classi e le associazioni ritenute più importanti, ma si lascia al progettista l'eventuale decisione di anticipare alcuni dei passi di raffinamento suggeriti più avanti per aggiungere altri dettagli. Si tenga comunque presente che una visione panoramica di alto livello della struttura della base di dati consente di anticipare con i committenti una verifica sulla corretta interpretazione degli aspetti essenziali dei requisiti, prima di procedere con i passi di raffinamento e di completamento previsti nella fase di progettazione concettuale. Lo schema scheletro si definisce, pertanto, procedendo come segue (tenendo conto che ciascuno di questi passi può richiedere di modificare alcune scelte fatte nei passi precedenti):

1. Identifica le classi.
2. Descrivi le associazioni fra le classi.
3. Individua le sottoclassi.

Identificare le classi

Si produce una lista preliminare delle classi di oggetti che interessa modellare e si assegna ad ognuna di esse un nome appropriato. Questo elenco iniziale ha un grado di completezza e di significatività che dipende dal grado di comprensione della realtà osservata e, in generale, sarà soggetto a modifiche mano a mano che si procede. Nella scelta iniziale delle classi si tengono presenti le entità di cui si vogliono ricordare alcuni fatti, senza nessuna pretesa di minimalità. Eventuali ridondanze verranno eliminate successivamente.

Supponiamo che le classi individuate in questa fase siano DomandeTrasferimento, PraticheTrasferimento, Delibere, ConvalideTipiche, CorsiEsterni, CorsiInterni.

Descrivere le associazioni fra le classi

Si individuano le possibili associazioni fra le classi finora definite e le loro proprietà strutturali, arrivando ad un risultato come quello esemplificato in Figura 3.11.
L'analisi delle associazioni può indurre ad eliminare una classe che può essere rappresentata da un'associazione, o ad aggiungere una nuova classe per rappresentare un'associazione, in particolare se interessa rappresentare alcuni attributi di quell'associazione.

Ad esempio, la classe delle convalide tipiche contiene solo delle coppie (corso esterno, corso interno), con un attributo booleano che indica se è richiesto un colloquio. In questa fase si può decidere di rappresentare tale informazione attraverso un'associazione tra corsi esterni e corsi interni.

Viceversa, si consideri l'associazione ternaria tra Delibere, CorsiEsterni e CorsiInterni, che stabilisce, per ogni delibera, quali specifiche convalide sono state effettuate. In questa fase si può decidere di rappresentare tale associazione con una nuova classe ConvalideEsami, che conterrà quindi un elemento per ogni "convalida specifica", quale ad esempio "l'esame di Analisi I superato l'A.A. 2004/05 ad Ingegneria Civile da Mario Rossi è convalidato per Analisi I interna".

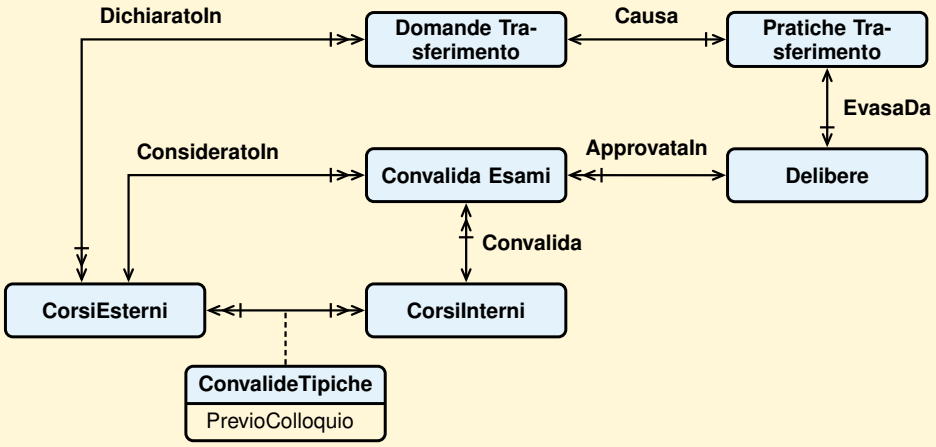


Figura 3.11: Prima versione dello schema

Individuare le sottoclassi

Per definire le sottoclassi si esaminano tutte le classi già definite per capire (*a*) se può essere utile definirne di nuove per caratterizzare particolari sottoinsiemi di alcune classi, (*b*) se esistono classi che sono un sottoinsieme di altre e quindi possono essere ridefinite, e (*c*) se esistono oggetti di classi che possono assumere nel tempo stati significativi per l'applicazione, caratterizzati da attributi propri (ad esempio, gli stati di "bozza" e di "delibera approvata" di una delibera), e quindi suggeriscono l'opportunità di specializzare le relative classi per distinguere gli oggetti in base allo stato in cui si trovano.

Ad esempio, in questa fase possiamo individuare le sottoclassi BozzeDiDelibera e DelibereApprovate che distinguono le delibere sulla base del loro stato di approvazione (Figura 3.12).

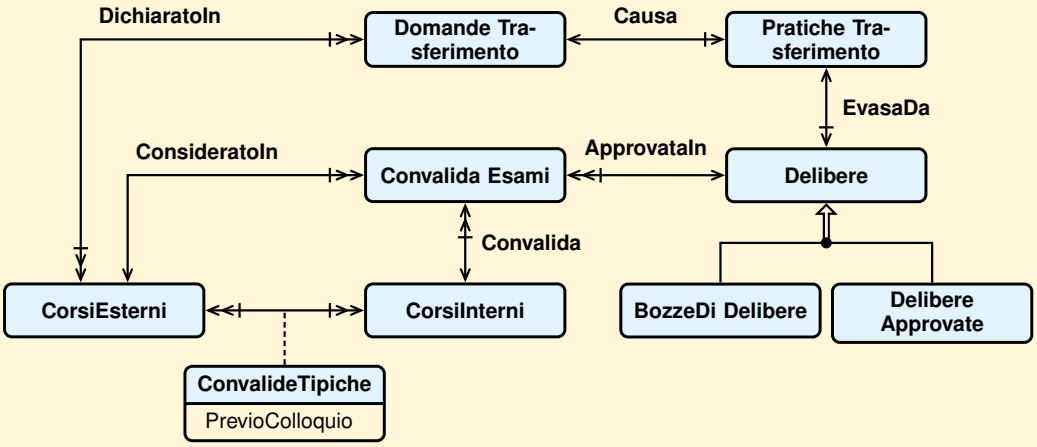


Figura 3.12: Schema con sottoclassi

Specifica delle operazioni (passo 6)

In Figura 3.13 è mostrata la specifica dell'operazione degli utenti ImmissioneDomandaDiTrasferimento usando il formalismo proposto nel capitolo precedente. Si noti come in questa fase di un'operazione si specifica sostanzialmente l'interazione con l'utente e i depositi dei dati interessati, volendo usare la terminologia dei diagrammi di flusso dati.

Operazione	ImmissioneDomandaDiTrasferimento
Scopo	Immissione dei dati di una domanda di trasferimento
Argomenti	NomeStudente :string, RecapitoStudente :string, UniversitaDiProvenienza :string, DipartimentoDiProvenienza :string, CorsoDiLaureaDiProvenienza :string, ElencoEsamiEsterniSuperati :seq CorsoEsterno;
Risultato	(OperazioneEseguita; Errore);
Errori	
Usa	
Modifica	DomandeDiTrasferimento, CorsiEsterni, DomandeDiTrasferimento;
Prima	
Poi	

Figura 3.13: Esempio di specifica preliminare di operazione.

3.5 La progettazione concettuale

3.5.1 Scopo della progettazione concettuale

Scopo della progettazione concettuale è tradurre il risultato dell'analisi dei requisiti settoriali in una descrizione *formale* ed *integrata* degli aspetti strutturali e dinamici del sistema informatico studiato. Mentre l'analisi dei requisiti analizza cosa si aspettano i *singoli* settori dal sistema, in questa fase l'attenzione è su come disegnare una base di dati ed un insieme di operazioni che garantiscano per *tutti* i settori le funzionalità desiderate. Il risultato della progettazione concettuale è lo schema o progetto concettuale, scritto in un linguaggio formale, indipendente dal DBMS che verrà usato nella realizzazione, con costrutti ad alto livello, adatti a descrivere in modo naturale il *significato* di ciò che si sta modellando.

Il passaggio dalla specifica dei requisiti al progetto concettuale avviene quindi attraverso un processo di formalizzazione e raffinamento dei requisiti settoriali ed attraverso la loro integrazione in un progetto globale. Inoltre, in questa fase gli aspetti quantitativi vengono riformulati con riferimento allo schema concettuale prodotto.

Il progetto concettuale gioca un ruolo fondamentale nel processo di sviluppo del sistema informatico, principalmente per le seguenti ragioni:

- è utilizzato dal progettista per ragionare ad un giusto livello di astrazione sulle scelte fatte e per produrre un modello soddisfacente della realtà osservata;
- è utilizzato dalle diverse figure professionali coinvolte nella progettazione e nella codifica delle applicazioni come specifica formale e dettagliata di ciò che va realizzato;
- è un preciso riferimento a cui ricondursi per modificare il sistema realizzato o per sviluppare nuove applicazioni quando il sistema è già operativo;
- è utilizzabile nelle discussioni con i committenti non informatici che devono approvare le scelte di progetto, usando opportune rappresentazioni grafiche degli aspetti principali;
- è utilizzabile come prototipo che può essere validato dagli utenti finali dopo esperimenti su dati campione, se il linguaggio in cui è espresso è eseguibile.

Un buon progetto concettuale è un obiettivo irrinunciabile all'aumentare della complessità del sistema da realizzare, poiché il processo di sviluppo richiede un tale impegno di risorse da rendere difficilmente praticabile una riprogettazione del sistema in caso di funzionamento non soddisfacente. Le principali proprietà che caratterizzano un buon progetto concettuale sono le seguenti:

Completezza concettuale Vanno specificati tutti gli aspetti rilevanti delle applicazioni, in modo dettagliato, per consentirne una realizzazione corretta.

Indipendenza dalle applicazioni Lo schema concettuale non deve essere fatto in funzione dell'efficienza di particolari applicazioni, ma con l'obiettivo di una efficace modellazione. Considerazioni sull'efficienza verranno fatte nelle fasi successive.

Indipendenza dal DBMS Nello schema concettuale si prescinde da aspetti specifici del DBMS impiegato nella realizzazione.

Sono stati proposti modi diversi per descrivere i vari aspetti di un progetto concettuale: rappresentazioni grafiche, linguaggi formali non eseguibili oppure linguaggi formali eseguibili. Per quanto riguarda gli aspetti strutturali, le soluzioni più comuni sono basate su estensioni del *modello entità-relazione* e su un *modello a oggetti*.

3.5.2 Come procedere

Per produrre lo schema concettuale di una base di dati è necessario raffinare la definizione degli schemi scheletro settoriali definiti nella specifica dei requisiti e produrre uno schema globale. La produzione di uno schema globale può procedere per *particolarizzazione* o per *integrazione*.

Nell'approccio per *particolarizzazione*, in un primo passo si definiscono i dati comuni a tutti i settori, successivamente si particolarizzano le definizioni prodotte tenendo conto delle esigenze specifiche di ciascun settore. Il vantaggio di questo modo di procedere sta nel fatto che, una volta uniformata l'interpretazione dei dati comuni, si ha una maggiore stabilità del progetto e si può procedere anche in tempi successivi alla definizione dei dati delle diverse classi di utenti, senza dover ritornare sulle definizioni già date. Le difficoltà sorgono durante il lavoro preliminare effettuato per individuare le informazioni in comune. Questo approccio è usato nelle situazioni più semplici o quando è stato già possibile fondere i requisiti nella fase precedente.

Nell'approccio per *integrazione*, in un primo passo si definisce per ogni settore uno schema parziale delle informazioni, eventualmente per aggregazioni successive dei sottoschemi associati ad ogni applicazione. Nel passo successivo si provvede ad analizzare ed integrare questi schemi parziali in un unico schema concettuale che soddisfi i requisiti di tutti i settori. Il secondo passo è particolarmente critico dovendo individuare e risolvere i conflitti espliciti ed impliciti che sorgono per una diversa modellazione degli stessi fatti in schemi di settori diversi. Per conflitti espliciti si intendono quelli rilevabili da un'analisi degli schemi parziali: ad esempio un fatto descritto come proprietà in uno schema e come entità in un altro, oppure un'associazione modellata con una cardinalità e vincolo di dipendenza diversi. I conflitti impliciti sono invece quelli più difficili da rilevare poiché riguardano le omonimie e le sinonimie. Questo approccio è usato nei casi più complessi, quando esistono molti utenti e applicazioni, ed è l'unico possibile in quelle metodologie in cui l'analisi dei dati viene effettuata a partire dai risultati dell'analisi funzionale, integrando i sottoschemi che rappresentano le informazioni usate da ciascuna applicazione.

In entrambi i casi, il lavoro del progettista dipende da come sono stati specificati i requisiti. Se si ha una rappresentazione in linguaggio naturale, la specifica dello schema concettuale dipenderà molto dall'intuizione e dall'esperienza del progettista. Se si ha invece una rappresentazione formale, il processo richiederà meno sforzi interpretativi. Una volta generata la descrizione globale dei fatti riconducibili a dati, lo schema si completa in ogni dettaglio con la specifica degli aspetti procedurali, dinamici e quantitativi.

Vediamo più in dettaglio come si procede per la produzione del progetto concettuale nell'approccio per integrazione. La progettazione richiede i seguenti passi:

1. definisci gli schemi di settore;
2. integra gli schemi di settore;
3. ristruttura eventualmente lo schema finale;
4. definisci l'architettura delle operazioni degli utenti e i metodi degli oggetti;
5. controlla la completezza delle operazioni degli utenti e di base.

3.5.3 I passi della progettazione concettuale

Vediamo ora come si può procedere per effettuare i singoli passi della progettazione concettuale.

Il primo passo verrà esemplificato con riferimento ai requisiti specificati nella sezione precedente. Poiché in tale esempio non sono previsti schemi di settore da integrare, il passo di integrazione sarà trattato invece con un esempio diverso.

1. Definizione di uno schema di settore

Le indicazioni che verranno date in questa sezione sono utilizzabili sia nell'approccio per particolarizzazione sia nel disegnare i singoli schemi di settore quando si segue l'approccio per integrazione.

Lo schema scheletro specificato nella fase di analisi dei requisiti si raffina procedendo con i seguenti passi:

1. definisci la struttura degli elementi delle classi;
2. individua le generalizzazioni;
3. tratta le dipendenze funzionali;
4. completa le definizioni delle associazioni;
5. completa le definizioni delle classi.

1. Definire la struttura degli elementi delle classi

Per ogni classe si elencano le proprietà descrittive interessanti dei suoi elementi, specificando, per ognuna di esse, il nome e il tipo. In questa fase le proprietà non vengono riportate nello schema perché non è ancora quello definitivo.

In questo passo va prestata molta attenzione alla possibilità che i valori di alcune proprietà siano più significativi come oggetti a sé stanti, e quindi convenga introdurre nuove classi, o viceversa al fatto che talune entità possano essere rappresentate come semplici attributi di altre. Inoltre, è frequente il caso in cui, tentando di elencare le proprietà di un oggetto, si scopre che la classe non era ben definita ed occorre rifarsi al significato di ciò che si sta descrivendo per decidere come procedere.

In questo caso, potremmo ad esempio scoprire che, poiché interessa mantenere il recapito di ogni studente che fa domanda di trasferimento, potrebbe essere opportuno aggiungere una classe degli Studenti, e considerare Università, Dipartimento e CorsoDiLaureaDiProvenienza come attributi degli studenti (Figura 3.14).

I tipi degli elementi delle classi possono essere definiti come in Figura 3.15.

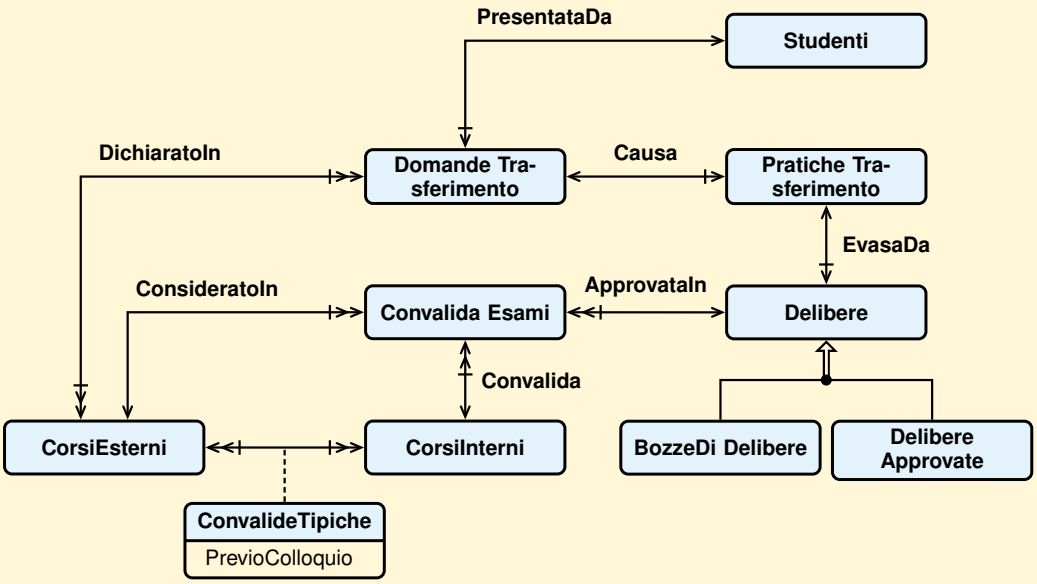


Figura 3.14: Schema arricchito con gli studenti

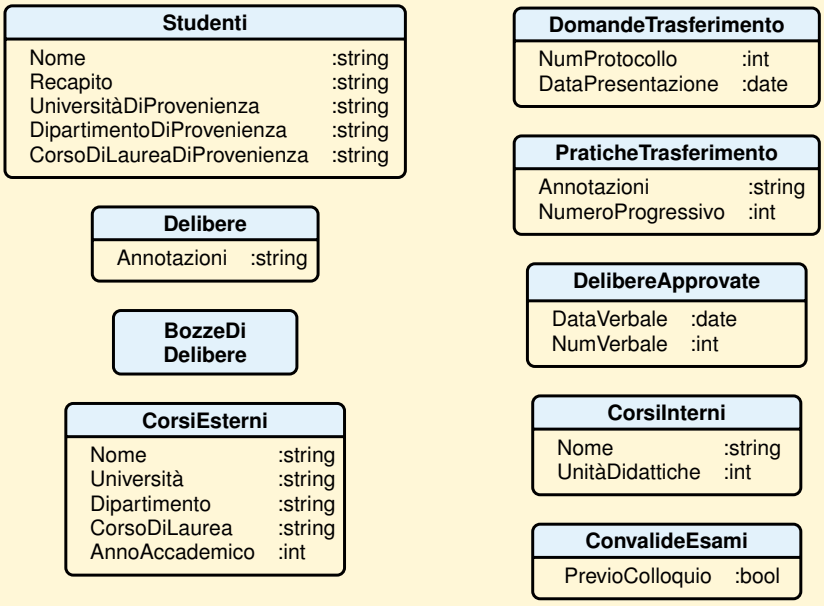


Figura 3.15: Classi con la struttura dei tipi

2. Individuare le generalizzazioni

In questo passo si fissa l'attenzione su quelle classi di oggetti con proprietà che hanno lo stesso significato. Se può essere utile, si definisce una nuova classe dalla quale si possono ridefinire le altre per specializzazione.

Ad esempio, nel nostro caso possiamo scoprire che è utile definire una superclasse comune Corsi da cui i corsi esterni ed interni ereditano alcune proprietà (Figura 3.16). Convalide e domande di trasferimento possono ora essere più correttamente collegate a Corsi, anziché a CorsiEsterni, dato che uno studente che chiede il trasferimento potrebbe avere superato alcuni degli esami internamente, ad esempio se si era trasferito nel passato nella direzione inversa.

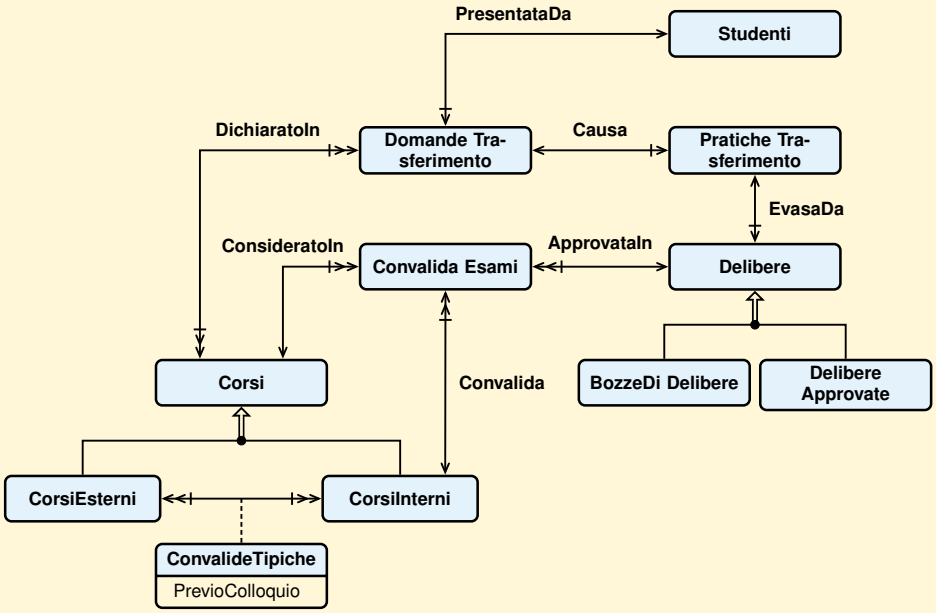


Figura 3.16: Schema dopo la generalizzazione

Si ridefiniscono le proprietà di Corsi, CorsiInterni e CorsiEsterni.

3. Trattare le dipendenze funzionali

Siano X ed Y due insiemi di attributi; diremo che X determina funzionalmente Y, o che Y è determinato funzionalmente da X, se in ogni possibile estensione della classe due elementi distinti che hanno lo stesso valore per gli attributi di X (il *determinante*) hanno anche lo stesso valore per gli attributi di Y (il *determinato*). Quando si riscontra una dipendenza funzionale in cui il determinante non è una chiave, occorre decidere se le proprietà che formano la dipendenza (determinante e determinato) non si possano meglio vedere come le proprietà degli elementi di una classe a sé stante. In caso

affermativo, si crea la nuova classe e si sostituiscono tutte le proprietà della dipendenza funzionale con un'associazione fra la vecchia e nuova classe. Nel caso, invece, che si decida di non generare una nuova classe bisognerà ricordarsi di trattare questo fatto come ogni altro vincolo d'integrità.[2]

Ad esempio, nel nostro caso potremmo stabilire che gli attributi CorsoDiLaurea e Università della classe dei corsi esterni determinano l'attributo Dipartimento, nell'ipotesi che non esistano in nessuna università due corsi di laurea con lo stesso nome in due dipartimenti diversi. Questa osservazione ci potrebbe spingere a raggruppare le tre proprietà in una nuova classe CorsiDiLaurea associata alla classe degli corsi esterni (Figura 3.17), oppure potremmo mantenere lo schema già scelto e trattare questo fatto come un vincolo di integrità.

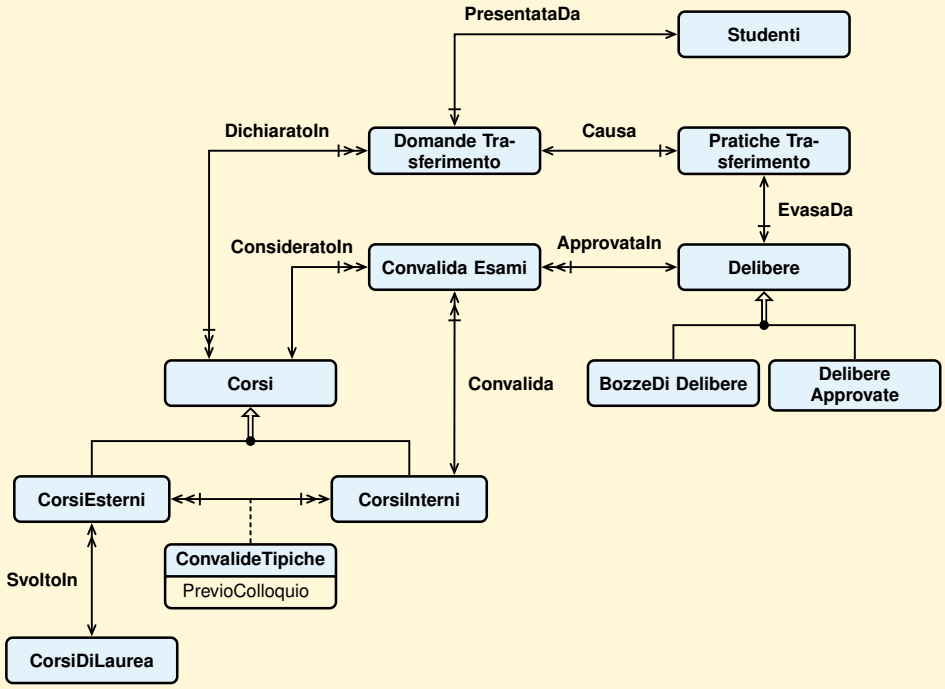


Figura 3.17: Schema con la classe CorsiDiLaurea

4. Completare la definizione delle associazioni

Se nello schema grafico le associazioni non sono state già definite, si completa la loro definizione specificandone il nome, le proprietà strutturali e gli attributi.

2. Questi aspetti saranno trattati diffusamente nel capitolo dedicato alla progettazione relazionale.

5. Completare la definizione delle classi
Si completa la definizione delle classi specificando per ognuna:

1. quali proprietà sono costanti e quali sono modificabili;
2. quali proprietà sono memorizzate e quali calcolate, eventualmente a partire da altre, specificando la regola del calcolo;
3. le chiavi, ovvero gli insiemi minimali di attributi che individuano univocamente un elemento della classe;
4. eventuali altri vincoli di integrità sugli elementi della classe;
5. i tipi ausiliari utilizzati.

È importante stabilire quali proprietà siano modificabili per prevedere poi opportuni metodi per cambiare il loro valore, eventualmente rispettando certi vincoli d'integrità.

Nei vincoli di integrità useremo una notazione per cui, se esiste un'associazione binaria R tra la classe a cui appartiene a ed una classe B, allora a.R denota l'insieme di elementi della classe B associati ad a, o l'unico elemento di B associato ad a se l'associazione è univoca. Se X è un insieme di elementi, X.R è l'unione degli insiemi a.R, per ogni elemento $a \in X$.

Nel caso di vincoli di chiave, può accadere che gli elementi di una classe siano univocamente determinati a partire dall'elemento associato in un'altra classe e da un insieme di attributi; ad esempio, un corso esterno è identificato dal corso di laurea associato, dal nome e dall'anno accademico. Questo vincolo sarà espresso come un vincolo di chiave, facendo uso della notazione $\ll$K$\gg$(attributi o elementi) come nella classe CorsiEsterni dell'esempio.

Il risultato di questo passo è mostrato in Figura 3.18.

2. Integrazione di schemi di settore
Una volta definiti gli schemi di settore, si procede con i seguenti passi alla loro integrazione per produrre lo schema globale:

1. Risolvi i conflitti di nome, di tipo e di vincoli d'integrità.
2. Fondi gli schemi.
3. Analizza le proprietà interschema.

Poiché nell'esempio sviluppato finora non sono previsti schemi di settore da integrare, per illustrare le idee che sono alla base del processo di integrazione, si considera l'esempio in Figura 3.19, tratto da [Batini et al., 1987], dove sono riportati due possibili schemi da integrare.

Risoluzione dei conflitti di nome, di tipo e di vincoli d'integrità
Si supponga di aver verificato che:

– il significato di Argomenti nel primo schema sia lo stesso di Descrittori nel secondo;
– Documenti nel secondo schema sia un concetto più astratto di Libri nel primo schema, perché esso include libri, atti, giornali, monografie ecc.

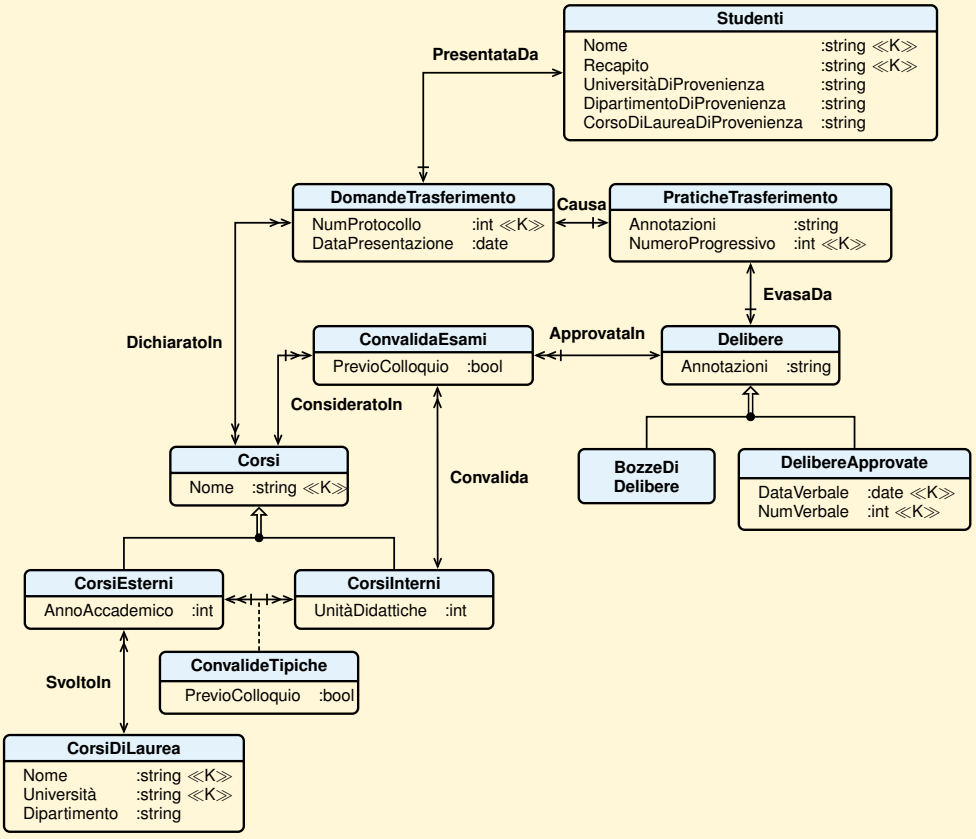


Figura 3.18: Schema finale

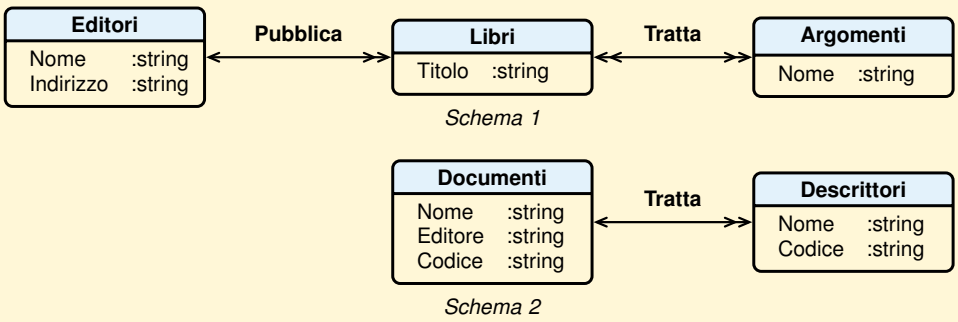


Figura 3.19: Due schemi da integrare

Per integrare i due schemi, poiché Argomenti e Descrittori rappresentano lo stesso concetto, i due nomi vanno unificati: scegliamo, ad esempio, il nome Argomenti. Un'altra differenza fra i due schemi è che lo stesso concetto Editore è rappresentato nel primo schema come entità, mentre nel secondo come attributo. Per uniformare le due rappresentazioni, decidiamo di trattare Editori come entità anche nel secondo schema, assegnandogli l'attributo Nome (vedi Figura 3.20).

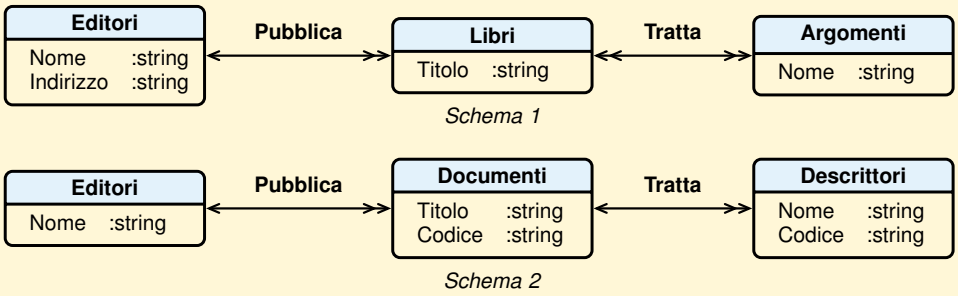


Figura 3.20: Schemi uniformati

Fusione degli schemi di settore

Una volta risolti i conflitti di rappresentazione, gli schemi di settore possono essere sovrapposti, producendo la rappresentazione in Figura 3.21.

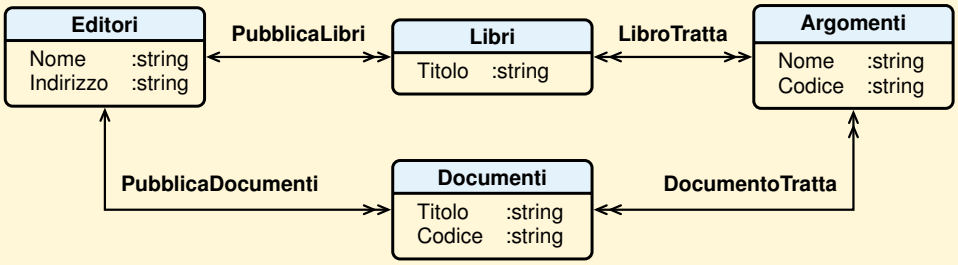


Figura 3.21: Schema integrato

Analisi delle proprietà interschema

Lo schema integrato viene esaminato per eventuali arricchimenti, dovuti ad esempio al fatto che si scoprono relazioni fra concetti provenienti da schemi diversi (proprietà interschema). Un esempio è la relazione di sottoinsieme tra i concetti Libri e Documenti, che, aggiunta, porta allo schema in Figura 3.22.

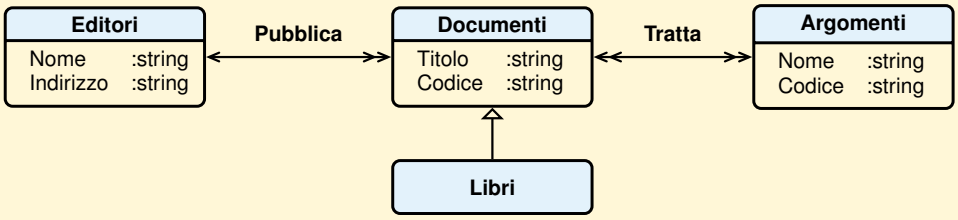


Figura 3.22: Arricchimento dello schema integrato con una proprietà interschema

Questo esempio di integrazione di schemi, nonostante la sua semplicità, evidenzia i problemi di base che nascono nel processo di integrazione, dovuti alle seguenti ragioni:

1. Differenza di percezione dei fatti rappresentati negli schemi iniziali. Nel processo di progettazione, gruppi differenti di utenti o progettisti possono percepire lo stesso concetto in modo diverso, nel senso che ad esso possono essere associati nomi e proprietà differenti. Nell'esempio visto, allo stesso concetto erano stati dati due nomi differenti (Argomenti e Descrittori).
2. Impiego di costrutti diversi per rappresentare le stesse informazioni. Ad esempio, in Figura 3.19 l'informazione sugli Editori è stata modellata con un attributo in uno schema e con un'associazione nell'altro.
3. Diversità di interpretazione di concetti comuni. Stessi concetti sono modellati con significati diversi, ad esempio un'associazione fra due classi è descritta con proprietà strutturali diverse.

3. Ristrutturazioni dello schema finale

Una volta ottenuta una prima versione dello schema finale, che rappresenta tutte le informazioni di interesse di eventuali settori aziendali diversi, si valuta l'opportunità di possibili ristrutturazioni per migliorare la leggibilità dello schema o per eliminare ridondanze, come quelle dovute a cammini distinti fra due classi che modellano associazioni equivalenti.

4. Definizione dell'architettura delle operazioni degli utenti e dei metodi degli oggetti

In questo passo si completa la specifica delle operazioni, con riferimento allo schema globale della base di dati.

Una volta definite le operazioni, si verifica che siano state previste tutte le operazioni di base necessarie. Risultano di solito utili matrici di controllo come quella che prevede una riga per ogni operazione e una colonna per ogni classe, con elementi che contengono nessuno, uno o più simboli del tipo:

– U se l'operazione utilizza la classe;

– C se l'operazione crea un elemento della classe;
– M se l'operazione modifica un elemento della classe;
– D se l'operazione distrugge un elemento della classe.

Colonne di classi prive di alcuni di questi simboli sono indizi utili per stabilire la non completezza della specifica.

Per maggiori dettagli su questa fase si rinvia ai testi citati nelle note bibliografiche.

3.6 Riepilogo della metodologia di progettazione

1. **Analisi dei requisiti** (da effettuare per ogni settore aziendale)
   - 1.1 Analizza il sistema informativo esistente e raccogli una prima versione dei requisiti, espressa in linguaggio naturale.
   - 1.2 Rivedi i requisiti espressi in linguaggio naturale per eliminare ambiguità, imprecisioni e disuniformità linguistiche.
   - 1.3 Raggruppa le frasi relative a categorie diverse di dati, vincoli e operazioni.
   - 1.4 Costruisci un glossario dei termini.
   - 1.5 Definisci uno schema preliminare di settore.
     - 1.5.1 Identifica le classi.
     - 1.5.2 Descrivi le associazioni fra le classi.
     - 1.5.3 Individua le sottoclassi.
   - 1.6 Specifica le operazioni degli utenti.
   - 1.7 Verifica la completezza e consistenza della specifica.
2. **Progettazione concettuale**
   - 2.1 Definisci gli schemi di settore.
     - 2.1.1 Definisci la struttura degli elementi delle classi.
     - 2.1.2 Individua le generalizzazioni.
     - 2.1.3 Tratta le dipendenze funzionali.
     - 2.1.4 Completa le definizioni delle associazioni.
     - 2.1.5 Completa le definizioni delle classi.
   - 2.2 Integra gli schemi di settore.
     - 2.2.1 Risolvi i conflitti di nome, di tipo e di vincoli d'integrità.
     - 2.2.2 Fondi gli schemi.
     - 2.2.3 Analizza le proprietà interschema.
   - 2.3 Ristruttura eventualmente lo schema finale.
   - 2.4 Definisci l'architettura delle operazioni degli utenti e i metodi degli oggetti.
   - 2.5 Controlla la completezza delle operazioni degli utenti e di base.

3.7 Conclusioni

Sono state presentate le caratteristiche generali delle metodologie di progettazione di basi di dati e in particolare della fase di progettazione concettuale, che è la fase più critica del processo perché produce la rappresentazione delle informazioni da trattare per soddisfare i bisogni informativi degli utenti. La progettazione concettuale avviene con metodi poco formali e richiede quindi al progettista molta competenza, intuizione ed esperienza. Una volta trovato lo schema concettuale della base di dati, la progettazione logica e fisica si avvale di metodi formalizzati che rendono necessari opportuni strumenti automatici di ausilio. L'attenzione è verso ambienti che integrino un insieme di strumenti per costituire un "banco di lavoro" che agevoli l'arduo compito del progettista in casi complessi in modo da consentirgli di concentrarsi sugli aspetti più creativi del processo, rinviando alla macchina i compiti di gestione e produzione della documentazione, di controllo sulle attività svolte ed eventualmente di addestramento.

Ambienti di questa natura sono chiamati CASE, termine coniato inizialmente per indicare semplici strumenti di analisi e documentazione. Attualmente il termine è usato per indicare un insieme di strumenti e metodi per un approccio alla gestione, allo sviluppo e alla manutenzione di un progetto software basato su un insieme di attività ben definite, coordinate e ripetibili con rappresentazioni, regole di progettazione e standard di qualità. Fra gli strumenti tipici di un ambiente CASE si ricordano i seguenti:

- strumenti di supporto all'analisi del sistema informativo;
- strumenti per la progettazione delle applicazioni con funzionalità di diagrammazione, di controllo di consistenza fra le varie rappresentazioni del progetto e di stampa;
- strumenti di supporto per il processo di produzione del codice;
- strumenti per la generazione del codice.

Esercizi

1. **Condomìni**
   Si supponga di dover memorizzare in una base di dati le informazioni di interesse per un amministratore di condomìni. Di un condominio interessano il codice che lo identifica, l'indirizzo e il numero del conto corrente dove vengono fatti i versamenti per le spese sostenute. Un condominio si compone di un certo numero di appartamenti, di ognuno dei quali interessano il codice che lo identifica, il numero dei vani, la superficie.

   Gli appartamenti possono essere locati ad un inquilino, del quale interessano il nome, il codice fiscale, i numeri di telefono e il saldo, cioè la somma che l'inquilino deve all'amministrazione condominiale per le spese sostenute. Alcuni appartamenti locati possono essere stati disdetti, ed in questo caso interessa la data della disdetta.

Un appartamento può avere più proprietari, e un proprietario può possedere più appartamenti. Di ogni proprietario interessano il nome, il codice fiscale, l'indirizzo, i numeri di telefono e il saldo, cioè la somma che il proprietario deve all'amministrazione condominiale per le spese sostenute.

Le spese riguardano i condomìni e di esse interessano il codice di identificazione, la natura (luce, pulizia, ascensore ecc.), la data e l'importo. Fra le spese si distinguono quelle straordinarie, a carico dei proprietari, e quelle ordinarie, a carico degli inquilini. Le spese ordinarie vengono pagate in un'unica rata, mentre le spese straordinarie possono essere pagate in più rate e di ognuna di esse occorre ricordare la data e l'importo.

Progettare lo schema della base di dati e dare la specifica delle operazioni per l'immissione dei dati.

2. **Società Mega S.p.A.**

   Si vogliono gestire informazioni riguardanti gli impiegati, le loro competenze, i progetti a cui partecipano e i dipartimenti a cui appartengono.

   Ogni impiegato ha una matricola che lo identifica, assegnata dalla società. Di ogni impiegato interessano il nome, la data di nascita e la data di assunzione. Se un impiegato è coniugato con un altro dipendente della stessa società, interessano la data del matrimonio e il coniuge. Ogni impiegato ha una qualifica (ad esempio, segretaria, impiegato, programmatore, analista, progettista ecc.). Dei laureati e delle segretarie interessano altre informazioni. Dei laureati interessa il tipo di laurea e delle segretarie le lingue straniere conosciute.

   La società è organizzata in dipartimenti, identificati da un nome e da un numero di telefono. Un impiegato afferisce ad un solo dipartimento. Ogni dipartimento si approvvigiona presso vari fornitori e un fornitore può rifornire più dipartimenti. Di ogni fornitore interessano il nome e l'indirizzo. Interessano, inoltre, la data e il fornitore dell'ultimo acquisto fatto da un dipartimento.

   La società lavora a diversi progetti, ciascuno dei quali è localizzato in una città. Più impiegati partecipano ad un progetto e un impiegato può partecipare a più progetti, ma tutti localizzati sulla stessa città. Di ogni città con un progetto in corso interessano la sua popolazione e la regione. Un impiegato può avere più competenze, ma usarne solo alcune per un particolare progetto. Un impiegato usa ogni sua competenza in almeno un progetto. Ad ogni competenza è assegnato un codice unico e una descrizione. I progetti in corso sono identificati da un numero ed interessa una stima del loro costo.

   Progettare lo schema della base di dati.

3. **Anagrafe**

   Si vogliono trattare informazioni sulle persone che vivono o sono decedute in un comune italiano.

   Di una persona interessano: nome, cognome, codice fiscale, data di nascita (giorno, mese, anno), sesso (m, f) madre, e padre.

   Una persona può essere vivente o deceduta. Di una persona vivente interessano: numero di cellulare, stato civile (celibe (nubile), coniugato(a), vedovo(a), separato(a), divorziato(a)), e i familiari conviventi. Le persone di un nucleo familiare

condividono lo stesso indirizzo (via, numero, cap, località), telefono e comune di residenza.
Di una persona deceduta interessano: data del decesso, età al momento del decesso, comune del decesso, comune dove è stata seppellita.
Di un matrimonio interessano: data del matrimonio, le due persone che si sono unite in matrimonio e il comune dove è stato celebrato.
Di un comune interessano: nome, se capoluogo di provincia, prefisso telefonico, gli abitanti, il numero degli abitanti, il numero delle persone seppellite e decedute.
Vanno previste le seguenti operazioni; si usi questa lista per verificare che il modello della base di dati contenga tutti i dati necessari alla loro realizzazione.

a) creazione di una persona;
b) nascita di un figlio;
c) matrimonio;
d) antenati viventi di una persona;
e) nome e cognome dei genitori di una persona;
f) nome, cognome e relazione di parentela dei familiari conviventi di una persona;
g) figli viventi e figli conviventi di una persona;
h) cambio di residenza di una persona e dei suoi familiari conviventi;
i) una persona e i suoi conviventi vanno a vivere con un'altra persona;
j) una persona va a vivere da sola;
k) decesso di una persona.

4. **Ufficio della motorizzazione**
Si vogliono modellare i seguenti fatti di interesse di un ipotetico Ufficio della motorizzazione.

Produttori di automobili C'è un certo numero di produttori di automobili, ciascuno identificato da un nome (FIAT, FORD ecc.). I dati di nuovi produttori possono essere immessi in ogni momento, se il produttore ha l'autorizzazione ad iniziare l'attività commerciale.
L'autorizzazione non può essere ritirata e non più di cinque produttori possono essere in attività contemporaneamente. Un produttore è considerato attivo finché possiede automobili registrate come prodotte da lui e non ancora vendute; nel momento in cui un produttore non possiede auto, il suo permesso di operare può essere sospeso. I dati di un produttore vengono eliminati solo quando viene eliminata la storia di tutte le auto da lui prodotte.

Automobili Un'automobile è caratterizzata da un modello, dall'anno di produzione, da un numero di serie assegnatogli dal produttore, unico fra le automobili da lui prodotte. I dati di un'automobile vengono immessi all'atto della sua registrazione presso l'Ufficio della Motorizzazione. Al momento della registrazione, all'automobile viene assegnato un numero, unico per ciascuna automobile e non

modificabile, e la data di registrazione. Il produttore viene registrato come primo proprietario.
Un'automobile può essere registrata in qualsiasi giorno dell'anno in cui è stata costruita, ma può essere registrata solo entro il 31 gennaio se costruita l'anno precedente. Nel caso di distruzione, viene registrata la data di distruzione, e da questo momento l'automobile non può essere più trasferita. Infine, la storia di un'automobile va conservata per due anni dopo la sua distruzione.

Modelli di automobile Ogni automobile è caratterizzata da un modello (Panda, Uno, Escort ecc.). Le automobili di ciascun modello sono prodotte dallo stesso produttore, il quale è libero di introdurre nuovi modelli sul mercato in qualsiasi momento. Il nome di ciascun modello è unico fra tutti i modelli registrati. Le automobili di uno stesso modello hanno lo stesso consumo di benzina. Un modello ha una potenza di almeno 6 cavalli e una cilindrata compresa fra 400 e 3.000 cc.
I dati su un modello vanno conservati finché esiste nella base di dati un'automobile di tale modello. Le automobili di un certo modello non possono essere registrate se tale modello non è ancora noto all'Ufficio della motorizzazione.

Rivenditori I rivenditori sono preposti alla distribuzione di automobili nuove, o usate, ai privati. Di un rivenditore interessano il nome, l'indirizzo, il telefono e l'eventuale numero del fax. Nuovi rivenditori possono sorgere in ogni momento, ma la loro attività commerciale può iniziare solo se hanno ricevuto il permesso dagli uffici competenti. Un rivenditore può trattare automobili nuove di al più tre produttori diversi.
Ogni rivenditore è considerato operante finché possiede automobili; in caso contrario può richiedere la sospensione del permesso di operare. I dati di un rivenditore non operante vengono eliminati solo se questo non è stato proprietario di un'auto di cui si conserva la storia.

Privati I privati sono persone proprietarie di una o più automobili già registrate. Di un privato interessano il nome, l'indirizzo e il telefono. I dati dei privati vengono immessi con l'acquisto della prima automobile, ed eliminati solo se essi non sono stati proprietari di un'automobile di cui si conserva la storia.

Trasferimenti di proprietà In ogni momento un'automobile può essere posseduta: dal suo produttore (automobile invenduta), da un rivenditore, oppure da un gruppo di privati.
All'atto del trasferimento della proprietà di un'automobile vengono registrate le seguenti informazioni: un codice che identifica il trasferimento, la data di trasferimento, l'automobile trasferita, il vecchio e il nuovo proprietario.
Vi sono norme che vincolano il trasferimento di un'automobile:

– un'automobile distrutta non può essere trasferita;

– un'automobile può essere venduta da un produttore solo ad un rivenditore, e un produttore non può acquistare automobili;
– un'automobile può essere venduta da un rivenditore solo a privati o gruppi di privati.

I dati su un trasferimento possono essere eliminati solo quando l'automobile cessa di essere di interesse per l'Ufficio della Motorizzazione.
Si usino le seguenti operazioni previste sul sistema per verificare che possono essere realizzate.

– Registrazione di una nuova auto.
– Distruzione di un'auto.
– Registrazione della vendita di un'auto.
– Eliminazione della storia delle auto distrutte da almeno due anni.
– Registrazione di un nuovo produttore in attesa del permesso di operare.
– Autorizzazione di un produttore ad operare.
– Sospensione delle attività di un produttore.
– Eliminazione di un produttore non operante.
– Registrazione di un nuovo modello.
– Registrazione di un nuovo rivenditore.
– Sospensione delle attività di un rivenditore.
– Eliminazione di un rivenditore non operante.

5. **Organizzazione di una conferenza**
Si vogliono trattare i dati riguardanti le *Working Conferences* dell'IFIP (*International Federation for Information Processing*).
L'IFIP Ã¨ una organizzazione federativa che raggruppa 48 associazioni nazionali e accademie scientifiche che operano nell'area della tecnologia dell'informazione. Ogni membro ha un nome e il paese di appartenenza (ad es. *Associazione Italiana per l'Informatica ed il Calcolo Automatico*, Italia). Le attivitÃ dell'IFIP sono coordinate da 13 *Technical Committees (TC)*, a loro volta divisi in oltre 100 *Working Groups (WG)* ai quali appartengono oltre 3.500 esperti (professionisti ICT e ricercatori di tutto il mondo). Ogni *Technical Committee* copre un aspetto particolare dell'informatica e delle discipline correlate (ad es. *TC 1: Foundations of Computer Science*: *WG 1.1 Continuous Algorithms and Complexity*, *WG 1.2 Descriptional Complexity*, ecc.).
I *Working Groups* organizzano conferenze alle quali possono partecipare solo esperti membri dei *Working Groups* e *Technical Committees* che hanno ricevuto un invito.
Di un esperto interessano il codice, che lo identifica, il nome, la nazione, l'ente di appartenenza, l'email e il telefono.
Di una conferenza interessano la sigla, che la identifica, il titolo, l'anno, il luogo, le date di inizio e fine, la soglia minima e massima dei partecipanti ai lavori per garantirsi la copertura dei costi e per non superare le capacità ricettive delle strutture.
Delle sessioni di una conferenza interessano il titolo, la sala, la data e l'ora di inizio

e fine.
La conferenza è organizzata da tre comitati di esperti:

(a) il *Comitato Organizzatore* per curare gli aspetti finanziari, logistici, gli inviti e la pubblicità,
(b) il *Comitato di Programma* per curare gli aspetti scientifici della conferenza;
(c) *Comitato dei Revisori*, nominato dal *Comitato di Programma*, per esaminare gli articoli sottoposti alla conferenza, e decidere quali articoli accettare, non superando il numero massimo prestabilito.

È previsto un *Chairman* per ogni comitato e un *General Chairman* per la conferenza. Tutti i comitati lavorano utilizzando dati comuni che vanno raccolti ed elaborati in modo consistente.

Procedure da automatizzare L'applicazione da realizzare ha lo scopo di agevolare le attività del *Comitato di Programma* e del *Comitato Organizzatore*, pensati come due settori aziendali che operano utilizzando dati in comune. I comitati devono svolgere le seguenti attività.

Comitato di Programma

- Preparare la lista degli esperti a cui sollecitare la presentazione di un articolo.
- Registrare le lettere d'intenti, cioè le risposte date da coloro che intendono inviare un lavoro. Ogni esperto invia al più una lettera che verrà presa in considerazione solo se perviene entro la data prestabilita. I lavori con nessun autore fra gli esperti che hanno risposto con una lettera d'intenti avranno una priorità bassa, se il numero complessivo dei lavori da esaminare supera il massimo prestabilito.

Comitato Organizzatore

- Preparare la lista degli esperti da invitare alla conferenza. Nella lista devono essere inclusi: i membri di tutti i *Technical Committees* e *Working Groups* interessati; i membri del *Comitato dei Revisori* e tutti coloro che hanno proposto un articolo. Occorre evitare di mandare alla stessa persona più di un invito.
- Registrare le adesioni alla conferenza pervenute entro la data prefissata.
- Generare la lista finale dei partecipanti alla conferenza. Se le adesioni superano il numero massimo prestabilito, verrà data priorità, nell'ordine, ai membri dei *Technical Committees*, ai membri dei *Working Groups* e del *Comitato dei Revisori*, agli autori degli articoli ammessi alla conferenza, agli autori degli articoli rifiutati.
- Registrare gli articoli proposti per la conferenza e pervenuti entro una data prefissata.
- Distribuire i lavori fra i membri del *Comitato dei Revisori*. Ogni lavoro sarà revisionato da 3 a 5 revisori, a seconda del numero complessivo dei lavori pervenuti.

– Raccogliere i pareri dei revisori e selezionare il numero prefissato di lavori da presentare alla conferenza.
– Raggruppare i lavori selezionati in sessioni e scegliere un *Chairman* per ogni sessione fra i membri del *Comitato di Programma*.

Note bibliografiche

Metodologie a più fasi per la progettazione di basi di dati sono descritte in [Atzeni et al., 2002], [Connolly and Begg, 2000], [Batini et al., 1992], [Ceri, 1983], [Teorey, 1999].

Molte delle metodologie attuali, quali *Rational Unified Process* (*RUP*), si basano sul modello ad oggetti e sul linguaggio *Unified Modeling Language* (*UML*) [Connolly and Begg, 2000], [Maciaszek, 2002].

Per alcuni tipici esempi di strumenti CASE per basi di dati, si vedano sulla rete le descrizioni di ER/Studio (Embarcadero), ERWin (Computer Associates), Oracle Designer (Oracle), Power Designer (Sybase).

Capitolo 4

IL MODELLO RELAZIONALE

Il modello dei dati relazionale, proposto da Codd nel 1970, è il modello dei dati che ha avuto finora la più completa trattazione teorica. La sua semplicità, la natura degli operatori che offre e la teoria delle basi di dati che ha consentito di sviluppare l'hanno reso molto popolare sia in ambienti scientifici che applicativi. Inizialmente ci fu molto scetticismo sulla possibilità di realizzare sistemi commerciali basati su questo modello dei dati, competitivo con i sistemi gerarchici e reticolari che costituivano lo standard del periodo. Ma ormai, dopo lo sviluppo dei primi prototipi nella seconda metà degli anni '70, è diventato il prodotto di punta di molti costruttori di DBMS, per ogni tipo di elaboratore e in particolare per i calcolatori personali, e oggi i sistemi relazionali dominano il mercato. Rispetto al modello ad oggetti, il modello relazionale è molto meno espressivo, per cui molti dei sistemi attuali offrono delle estensioni "ad oggetti", in qualche senso, del modello. In questo capitolo verrà descritto il modello nella sua forma pura.

4.1 Il modello dei dati

4.1.1 La relazione

Il modello dei dati relazionale si basa sul concetto matematico di relazione n-aria, definita come sottoinsieme del prodotto cartesiano $D_1 \times D_2 \times \ldots D_n$ di n insiemi di valori di tipo elementare, detti *domini*, ovvero come un insieme di ennuple ordinate di valori $\langle d_1, d_2, \ldots, d_n \rangle$, con $d_i \in D_i$, per ogni $i = 1, 2, \ldots, n$. Le componenti di un'ennupla sono identificate dalla loro posizione. In informatica, per agevolare l'uso del modello dei dati, la nozione di prodotto cartesiano si estende a quella di prodotto cartesiano etichettato associando un'etichetta distinta ad ogni dominio del prodotto $D_1 \times D_2 \times \ldots D_n$, in modo da identificare le componenti di un'ennupla con le etichette anziché con le posizioni, come accade per i *record* dei linguaggi di programmazione. Più precisamente vale la seguente definizione:

Definizione 4.1
Uno *schema di relazione* $R : \{T\}$ è una coppia formata da un *nome di relazione* R e da un *tipo relazione* definito come segue:

- int, real, boolean e string sono tipi primitivi;

- se $T_1, \ldots, T_n$ sono tipi primitivi, e $A_1, \ldots, A_n$ sono etichette distinte, dette *attributi*, allora $(A_1 : T_1, \ldots, A_n : T_n)$ è un *tipo ennupla di grado n*. Due tipi ennupla sono uguali se hanno uguale il grado, gli attributi e il tipo degli attributi con lo stesso nome. L'ordine degli attributi non è significativo;
- se T è un tipo ennupla, allora $\{T\}$ è un *tipo insieme di ennuple* o *tipo relazione*. Due tipi relazione sono uguali se hanno lo stesso tipo ennupla.

Definizione 4.2

Uno *schema relazionale* è costituito da un insieme di *schemi di relazione* $R_i : \{T_i\}$, $i = 1, \ldots, k$ e da un insieme di vincoli di integrità relativi a tali schemi.[1]

Le definizioni precedenti caratterizzano l'aspetto intensionale del modello dei dati, mentre la definizione che segue ne caratterizza l'aspetto estensionale.

Definizione 4.3

Un'ennupla $t = (A_1 := V_1, \ldots, A_n := V_n)$ di tipo $T = (A_1 : T_1, \ldots, A_n : T_n)$ è un insieme di coppie (A_i, V_i) con V_i di tipo T_i. Due ennuple sono uguali se hanno lo stesso insieme di coppie (A_i, V_i). Un'*istanza* dello schema $R_i : \{T_i\}$ o *relazione* è un insieme finito di ennuple $\{t_1, t_2, \ldots, t_k\}$, con t_i di tipo T_i. La *cardinalità* di una relazione è il numero delle sue ennuple. Un'istanza di uno schema relazionale è formata da un'istanza di ciascuno dei suoi schemi di relazione.

Se $(A_1, \ldots, A_n)$ sono gli attributi del tipo ennupla T, si dice che una relazione di tipo $\{T\}$ ha attributi $(A_1, \ldots, A_n)$. Inoltre, per uno schema di relazione, invece della notazione $R : \{(A_1 : T_1, \ldots, A_n : T_n)\}$, si userà la notazione $R(A_1 : T_1, \ldots, A_n : T_n)$, che si abbrevierà ulteriormente in $R(A_1, \ldots, A_n)$, quando non interessa evidenziare il tipo degli attributi. In questo caso si assume che attributi uguali in schemi di relazione diversi abbiano lo stesso tipo.

È tradizione visualizzare una relazione come una tabella bidimensionale, con le colonne identificate dagli attributi, e con le righe contenenti solo i valori delle ennuple, nell'ordine indicato dall'intestazione delle colonne, come mostrato nell'esempio:

Studenti

Nome	Matricola	Provincia	Nascita
Isaia	71523	Pisa	1998
Rossi	67459	Lucca	1999
Bianchi	79856	Livorno	1998
Bonini	75649	Pisa	1999

1. Non esiste una definizione univoca di quali vincoli di integrità possono fare parte di uno schema relazionale; questo aspetto sarà comunque ripreso in seguito.

Esami

Materia	Candidato*	Data	Voto
BD	71523	12/01/2018	28
BD	67459	15/09/2019	30
IA	79856	25/10/2020	30
BD	75649	27/06/2018	25
IS	71523	10/10/2019	18

Si userà, inoltre, la notazione $\mathtt{dom}(A_i)$ per riferirsi all'insieme dei possibili valori che può assumere un attributo A_i in tutte le possibili istanze degli schemi di relazione in cui compare. Per riferirsi, invece, all'insieme dei valori che assume l'attributo A_i in un'estensione della base di dati, si userà la notazione $\mathtt{adom}(A_i)$, con $\mathtt{adom}(A_i) \subseteq \mathtt{dom}(A_i)$. Nella base di dati dell'esempio, $\mathtt{dom}(\text{Voto}) = \{18, 19, \ldots, 30\}$, mentre $\mathtt{adom}(\text{Voto}) = \{18, 25, 28, 30\}$.

4.1.2 I vincoli d'integrità

Come già specificato, uno schema relazionale è costituito da un insieme di schemi di relazione e da un insieme di vincoli d'integrità sui possibili valori delle estensioni delle relazioni. I due vincoli più importanti, che verranno definiti in questa sezione, sono quelli che specificano quali attributi sono *chiavi*, e quali attributi sono *chiavi esterne*.

I vincoli d'integrità sono strettamente legati alla nozione di *istanza valida* di uno schema di relazione e di uno schema di base di dati.

La nozione di "istanza valida" è una nozione *semantica*, ovvero legata alla realtà rappresentata dai dati: sono *valide* le istanze che rispettano la struttura della realtà. I vincoli di integrità sono lo strumento che il modellista usa per catturare al meglio possibile questa nozione.

I valori nulli

Come vedremo nel Capitolo 6, i sistemi relazionali permettono di specificare per ciascun attributo di una relazione se tale attributo può assumere il *valore nullo* (null), come avviene di norma, o se per tale attributo vale il vincolo di non assumere il valore nullo (vincolo not null).

L'esigenza di ammettere che il dominio di definizione di un attributo debba essere esteso con un particolare valore nullo scaturisce dal fatto che, quando si aggiunge un'ennupla ad una relazione, per varie ragioni può capitare che non sia possibile specificare il valore di un attributo. In un rapporto dell'ANSI (*American National Standard Institute*) vengono elencati 14 motivi per cui questo può accadere, tre dei quali sono più caratteristici e si illustrano con riferimento ad una relazione contenente dati sui professori visitatori di un'università:

ProfessoriVisitatori (Cognome, Nazionalità, CodiceFiscale)

Quando arriva un nuovo professore, può capitare di non conoscere il valore del codice fiscale perché: (*a*) il codice fiscale non è previsto per professori provenienti da certi paesi (*inapplicable attribute*), (*b*) il codice fiscale è previsto, ma non si conosce (*unknown attribute*), (*c*) non è noto se il professore ha oppure no un codice fiscale (*no information*).

Mentre, come abbiamo detto, i sistemi relazionali prevedono il valore null, nella descrizione teorica del modello relazionale questi valori sono evitati. Per questo motivo, nel presente capitolo essi saranno considerati solo nella progettazione logica relazionale, e verranno ripresi nei capitoli 6 e 7 sul linguaggio SQL.

Le chiavi

Definizione 4.4
Una *superchiave* di uno schema di relazione è un insieme X di attributi tale che, in ogni istanza valida dello schema di relazione, i valori degli attributi in X individuano univocamente un'ennupla, ovvero tale che in nessuna istanza valida dello schema di relazione possono esistere due ennuple diverse che coincidono su tutti gli attributi in X.

Definizione 4.5
Una *chiave* di uno schema di relazione è una superchiave *minimale*, nel senso che se si elimina un attributo, i rimanenti non formano più una superchiave. Un attributo che appartiene ad una chiave è chiamato attributo *primo*.[2]

Definizione 4.6
La *chiave primaria* di uno schema di relazione è una delle chiavi, e di solito viene preferita quella con il minor numero di attributi.

Le chiavi esterne

Definizione 4.7
Un insieme di attributi $A_1, \ldots, A_n$ di uno schema di relazione R è una *chiave esterna*, che riferisce una chiave primaria $B_1, \ldots, B_n$ di uno schema di relazione S, se, in ogni istanza valida della base di dati, per ogni ennupla t_r dell'istanza di R esiste un'ennupla t_s "riferita da t_r", ovvero tale che, per ogni $i \in 1..n$, $t_r.A_i = t_s.B_i$, dove $t.A$ indica il valore dell'attributo A dell'ennupla t.

Attraverso il valore di una chiave esterna, un'ennupla di una relazione viene associata a quell'ennupla della relazione riferita che ha il valore della chiave primaria uguale al valore della chiave esterna. Ad esempio, l'attributo Candidato della relazione Esami è

2. In alcuni testi la chiave viene detta *chiave candidata*.

una chiave esterna che riferisce la chiave primaria Matricola della relazione Studenti, e infatti vale che $adom(\text{Candidato}) \subseteq adom(\text{Matricola})$.

Il vincolo sulla chiave esterna non era previsto nella prima definizione del modello relazionale, ma è molto utile ed è previsto in quasi tutti i sistemi relazionali commerciali. Altri vincoli più generali sulle possibili estensioni di una base di dati relazionale potrebbero essere espressi con un opportuno linguaggio, ma qui non si prendono in considerazione. Le chiavi esterne rappresentano il modo standard di rappresentare le associazioni nel modello relazionale. Questo meccanismo permette di rappresentare direttamente la direzione unaria delle associazioni binarie uno a molti (che sono chiamate anche associazioni *gerarchiche*), e quindi anche delle associazioni uno ad uno; vedremo in seguito come possa essere usato per rappresentare associazioni molti a molti, non binarie o con attributi.

Negli schemi di esempio si evidenzieranno gli attributi della chiave primaria di uno schema di relazione sottolineandoli, mentre si userà l'asterisco per indicare gli attributi chiave esterna.

4.1.3 Una rappresentazione grafica di schemi relazionali

Come per il modello ad oggetti, anche per il modello relazionale è possibile definire un formalismo grafico in cui si rappresentano solo gli schemi di relazione e le loro associazioni, o più precisamente le chiavi esterne. In questo formalismo, una relazione è rappresentata da un rettangolo che ne contiene il nome. La presenza di una chiave esterna in R che riferisce la chiave primaria di S è rappresentata da una freccia che va da R ad S. La freccia sarà tagliata, in maniera analoga all'indicazione di parzialità del modello concettuale, quando la chiave esterna può assumere il valore null. Quando sia utile, la freccia può essere etichettata con il nome degli attributi che formano la chiave esterna, e ulteriori attributi della relazione possono essere rappresentati come si è potuto vedere nel Capitolo 2. Nei diagrammi dove sono specificati gli attributi, verrà aggiunta l'etichetta ≪PK≫ alle chiavi primarie, ≪CK≫ alle altre chiavi, e ≪FK≫(*Relazione*) alle chiavi esterne. Riportiamo, a titolo di esempio, in Figura 4.1 una rappresentazione grafica della base di dati con le relazioni Studenti e Esami.

4.1.4 Operatori

Il modello relazionale è stato il primo ad introdurre la possibilità di operare su insiemi di dati con operatori insiemistici, mentre nei modelli precedenti lo stile di programmazione era basato su operatori di scansione simili agli operatori *open, read, write, next, eof, close* con i quali i linguaggi imperativi manipolano i *file*.

I linguaggi per interrogare una base di dati relazionale permettono di definire la relazione risultato a partire dalle relazioni che compongono la base di dati. Essi si possono dividere in tre famiglie, che verranno esemplificate più avanti:

1. *linguaggi algebrici*: un'interrogazione è definita da un'espressione con operatori su relazioni che producono come risultato altre relazioni;

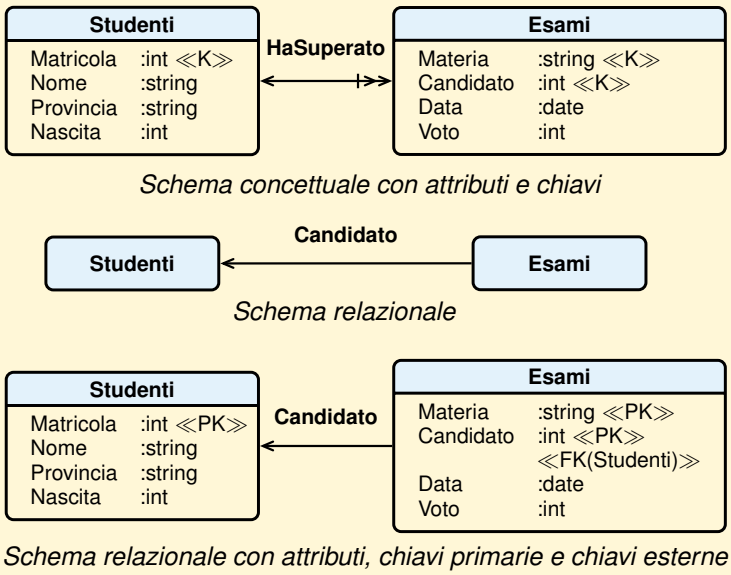


Figura 4.1: La rappresentazione grafica di uno schema relazionale

2. *linguaggi basati sul calcolo dei predicati*: un'interrogazione è definita da una formula del calcolo dei predicati del primo ordine;
3. *linguaggi logici*: sono linguaggi ispirati dal linguaggio logico Prolog in cui un'interrogazione è definita da un insieme di formule del calcolo dei predicati del primo ordine, mutuamente ricorsive ma con una struttura molto rigida (formule *Horn*).

Mentre una formula del calcolo si limita a specificare quali ennuple fanno parte del risultato, un'espressione algebrica stabilisce quali operazioni applicare per produrre il risultato. In genere, in un sistema relazionale il programmatore definisce il risultato della propria interrogazione fornendo una formula del calcolo, ed è il sistema che sceglie l'espressione algebrica equivalente da eseguire. A questo scopo il sistema per prima cosa trasforma la formula in un'espressione algebrica equivalente e poi applica a questa espressione una serie di *riscritture*, ovvero di trasformazioni che non ne modificano il risultato ma ne rendono la valutazione più efficiente. Vedremo, nella Sezione 4.3.3 un esempio di tali trasformazioni.

L'*algebra relazionale* e il *calcolo relazionale di ennuple* sono esempi di linguaggi delle prime due famiglie e furono proposti da Codd come esempi di linguaggi equivalenti e con le capacità minime di calcolo su relazioni. Egli, inoltre, introdusse il concetto di *completezza* di un linguaggio relazionale, intendendo con ciò che si possa formulare in esso un'espressione equivalente (cioè che valuti lo stesso risultato) ad una qualunque espressione dell'algebra relazionale, cioè una espressione formata da variabili e costanti di relazioni e da operatori dell'algebra. In realtà i linguaggi esistenti per sistemi relazionali offrono anche altri operatori che li rendono più che "completi". Ad esempio, quasi tutti offrono le operazioni aritmetiche e operazioni come la media,

somma, minimo o massimo, per agire su insiemi di valori per ottenere una singola quantità. I linguaggi logici sono anche più espressivi perché riescono ad esprimere interrogazioni "ricorsive", per calcolare ad esempio la chiusura transitiva di una relazione, ma pagano questa espressività con una maggiore difficoltà nell'ottimizzazione delle interrogazioni.

4.2 Progettazione logica relazionale

Il modello relazionale non si presta bene, per la sua povertà espressiva, alla progettazione concettuale; in questa fase si preferisce utilizzare il modello ad oggetti o il modello entità-relazioni. Al termine di questa fase poi, se si decide di realizzare la base di dati utilizzando un sistema relazionale, si trasforma lo schema concettuale prodotto in uno schema logico relazionale.

La trasformazione di uno schema ad oggetti in uno schema relazionale avviene eseguendo i seguenti passi:

PASSO 1: Rappresentazione delle associazioni uno a molti e uno ad uno.
PASSO 2: Rappresentazione delle associazioni molti a molti.
PASSO 3: Rappresentazione delle gerarchie fra classi.
PASSO 4: Definizione delle chiavi primarie.
PASSO 5: Rappresentazione degli attributi multivalore.
PASSO 6: Appiattimento degli attributi composti.

Questa trasformazione dovrebbe essere guidata dai seguenti obiettivi:

- rappresentare le stesse informazioni;
- minimizzare la ridondanza;
- produrre uno schema comprensibile, per facilitare la scrittura e manutenzione delle applicazioni.

In teoria, durante la trasformazione non si dovrebbe tenere conto di considerazioni relative all'efficienza delle applicazioni, poiché questi aspetti dovrebbero riguardare la fase di progettazione fisica (che in questo testo non viene presa in considerazione).

In generale nella conversione occorre duplicare certe informazioni e non si possono sempre rappresentare direttamente tutti i vincoli imposti dai meccanismi del modello ad oggetti. Per garantire la coerenza dei dati duplicati, e il rispetto dei vincoli non esprimibili nel modello relazionale, occorrerà quindi definire opportunamente le operazioni che modificano la base di dati.

Nel seguito verrà descritto un metodo per trasformare uno schema descritto con il modello ad oggetti in uno equivalente descritto con il modello relazionale, prendendo in considerazione solo gli aspetti riconducibili a dati (aspetti strutturali). Nel prossimo capitolo sulla teoria relazionale dei dati vedremo invece un altro approccio alla progettazione di uno schema relazionale, basato su un metodo formale che parte da una descrizione dei fatti da trattare data in modo completamente diverso. Alcuni

risultati di questa teoria consentono però di giustificare anche la ragionevolezza delle regole di trasformazione presentate in questo capitolo.

Il metodo che definiremo applica i primi tre passi alla rappresentazione grafica di uno schema ad oggetti, producendo, come risultati intermedi, dei disegni che contengono sempre meno costrutti ad oggetti, fino ad arrivare ad uno schema totalmente relazionale. Gli attributi delle classi vengono quindi presi in considerazione solo per gli ultimi tre passi. Il metodo sarà esemplificato applicandolo alo schema studiato nel Capitolo 2, rappresentato in Figura 4.2.

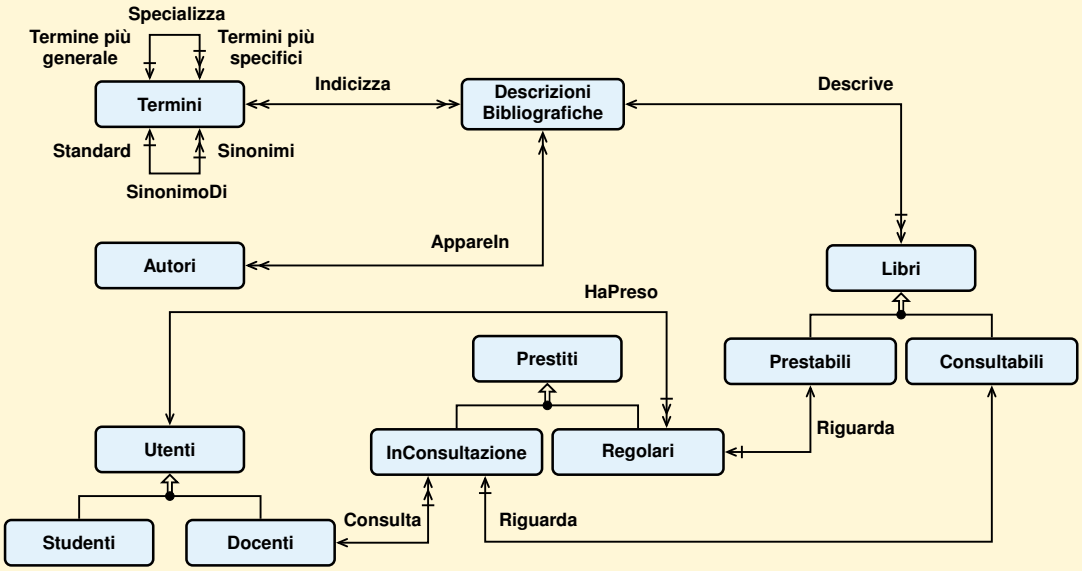


Figura 4.2: Lo schema ad oggetti da trasformare di Figura 2.7

4.2.1 PASSO 1: Rappresentazione delle associazioni uno a molti e uno ad uno

Come abbiamo già visto, le associazioni uno a molti si rappresentano aggiungendo agli attributi della relazione rispetto a cui l'associazione è univoca e totale una chiave esterna che riferisce l'altra relazione. Ad esempio, l'associazione tra Studenti e Esami, essendo univoca e totale rispetto a Esami si rappresenta aggiungendo a Esami una chiave esterna Candidato. Se l'associazione ha degli attributi, questi si aggiungono alla relazione in cui è presente la chiave esterna.

Se invece l'associazione nella direzione univoca è parziale, come capita comunemente nelle relazioni ricorsive di una classe, si possono scegliere due diverse rappresentazioni nello schema logico relazionale, come si mostra in Figura 4.3. nel caso della classe Termini. Nella prima si aggiunge allo schema una nuova relazione che contiene due chiavi esterne che riferiscono le due relazioni coinvolte dall'associazione, e gli eventuali attributi dell'associazione; questa relazione contiene un'ennupla per ogni istanza dell'associazione. La chiave primaria di questa relazione è la chiave esterna

che riferisce la relazione rispetto a cui l'associazione è multipla. Nella seconda soluzione, invece, si procede come nel caso precedente, aggiungendo una chiave esterna con possibili valori nulli alla relazione rispetto a cui l'associazione è univoca.

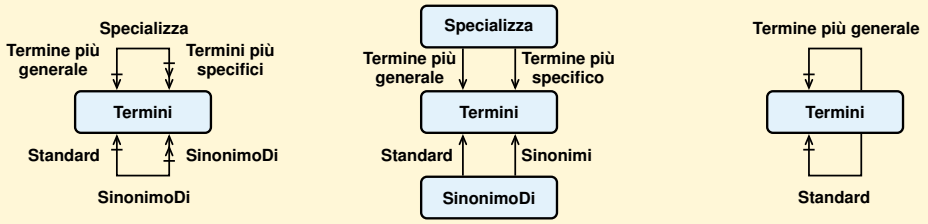


Figura 4.3: Esempio di trasformazione di associazioni ricorsive

Per le associazioni uno a uno si procede in maniera analoga, considerandole un caso particolare di associazioni molti a uno. Si consideri che quando una associazione ha sia la diretta che l'inversa, e si aggiunge una relazione con due chiavi esterne, una qualunque delle due può essere scelta come chiave primaria.

Graficamente, questo significa operare le sostituzioni specificate in Figura 4.4 su tutte le associazioni uno a molti e uno a uno presenti nello schema, supponendo che in ogni relazione sia definita la chiave primaria.

Ad esempio, applicando questo passo di trasformazione alla Figura 4.2, si ottiene il disegno in Figura 4.6.

4.2.2 PASSO 2: Rappresentazione di associazioni molti a molti

Un'associazione molti a molti tra due classi si rappresenta aggiungendo allo schema una nuova relazione che contiene due chiavi esterne che riferiscono le due relazioni coinvolte; la chiave primaria di questa relazione è costituita dall'insieme di tutti i suoi attributi. Questa relazione contiene un'ennupla per ogni istanza dell'associazione. Se l'associazione ha degli attributi, questi attributi vengono aggiunti alla nuova relazione, e non vanno a far parte della chiave della nuova relazione.

Graficamente, questo significa operare la sostituzione specificata in Figura 4.5 su tutte le associazioni molti a molti nello schema, che non cambia in presenza di indicazioni di parzialità nei primi due casi.

Ad esempio, applicando questa trasformazione alla Figura 4.6, si ottiene il disegno in Figura 4.7, dove la relazione Indicizza rappresenta l'associazione molti a molti tra Termini e DescrizioniBibliografiche.

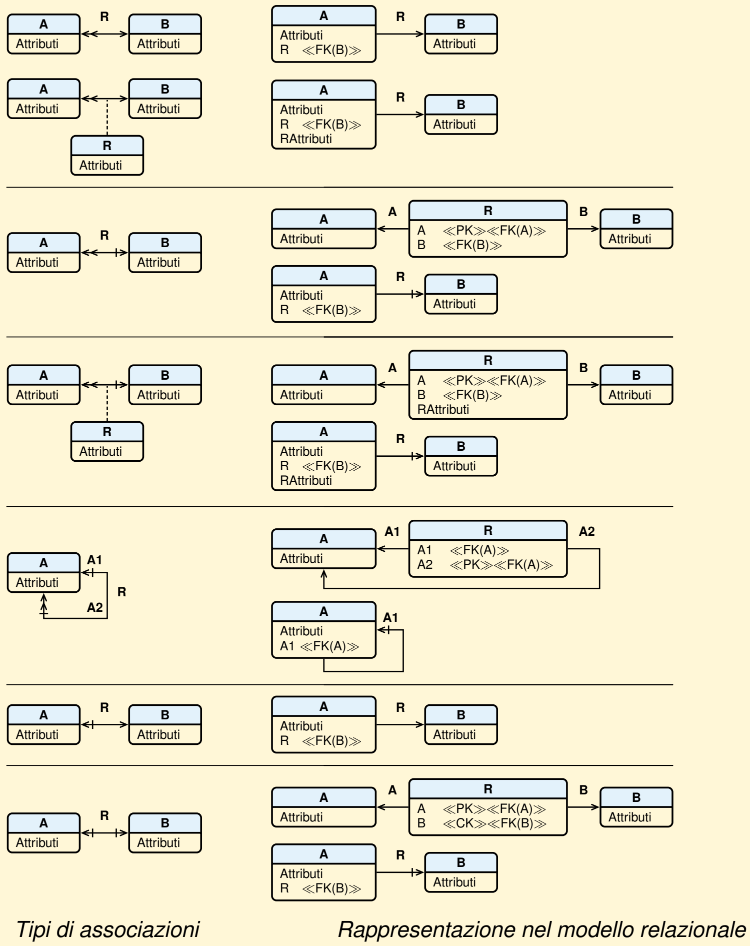


Figura 4.4: Regole per il **PASSO 1** di trasformazione

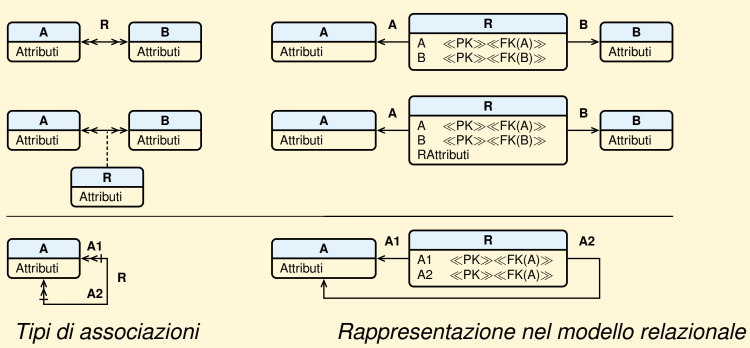


Figura 4.5: Regole per il **PASSO 2** di trasformazione

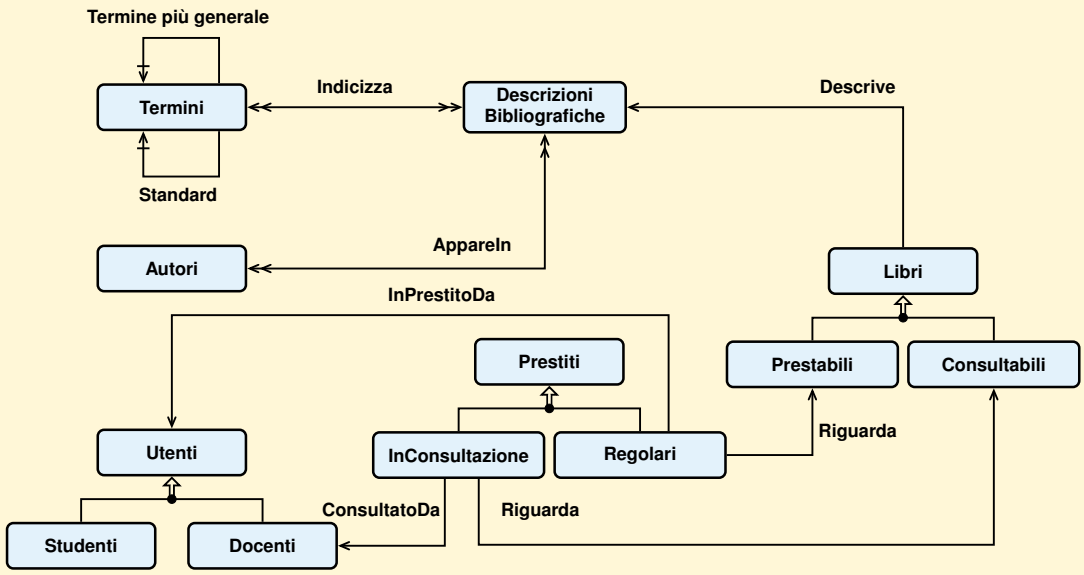


Figura 4.6: Effetto del **PASSO 1** di trasformazione

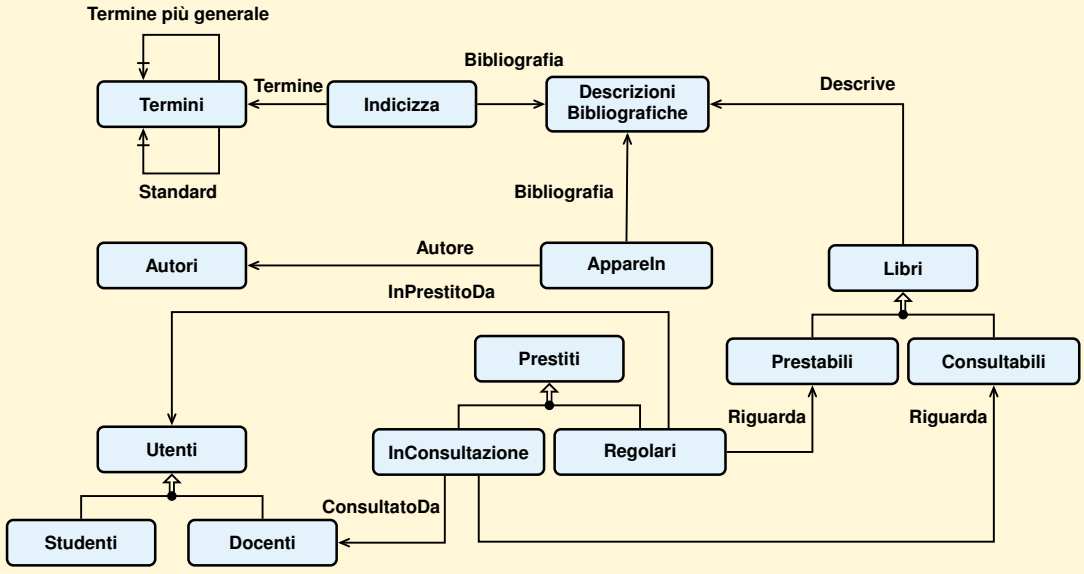


Figura 4.7: Effetto del **PASSO 2** di trasformazione

4.2.3 PASSO 3: Rappresentazione delle gerarchie fra classi

Sia data una classe A con due sottoclassi B e C, tali che i tipi associati alle tre classi abbiano, rispettivamente, attributi (X_A), $(X_A X_B)$ e $(X_A X_C)$, e sia K_A la chiave primaria di A.[3]

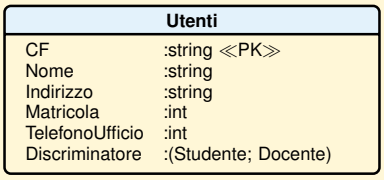
Nel modello relazionale vi sono almeno tre modi diversi di rappresentare questa situazione:

1. *relazione unica*: si definisce un'unica relazione con attributi (X_A, X_B, X_C, Discriminatore) che raccoglie tutti gli elementi delle tre classi; gli attributi X_B, X_C possono assumere il valore null e l'attributo Discriminatore serve a indicare la classe a cui appartiene l'elemento;
2. *partizionamento verticale*: si definiscono tre relazioni $R_A(X_A)$, $R_B(K_A, X_B)$ e $R_C(K_A, X_C)$. R_A contiene tutti gli elementi della classe A, anche se stanno in qualche sottoclasse, mentre R_B ed R_C contengono solo quegli attributi, degli elementi di B e di C, che non sono in X_A (attributi *propri* delle sottoclassi), ed una chiave esterna K_A che permette di ritrovare in R_A il valore degli altri attributi. K_A è anche la chiave primaria di R_B ed R_C;
3. *partizionamento orizzontale*: si definiscono tre relazioni $R_A(X_A)$, $R_B(X_A,X_B)$, $R_C(X_A, X_C)$. R_A contiene solo gli elementi della classe A che non stanno in nessuna delle sottoclassi, mentre R_B ed R_C contengono tutti gli elementi di B e di C; se le sottoclassi costituiscono una copertura, la relazione $R_A(X_A)$ non viene definita perché sarebbe sempre vuota.

Ad esempio, supponiamo di avere una classe Utenti con gli attributi CF (la chiave), Nome e Indirizzo, e con due sottoclassi di tipo partizione: Studenti, con un attributo Matricola, e Docenti, con attributo TelefonoUfficio.

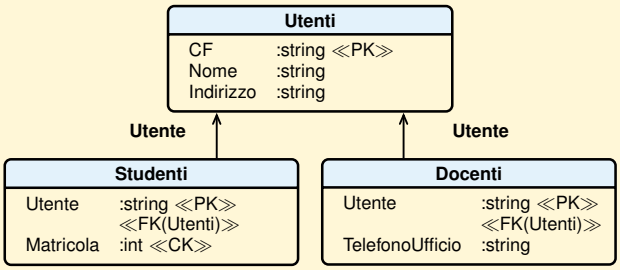
Le tre tecniche precedenti darebbero i seguenti risultati:

1. *relazione unica*: si definisce la relazione



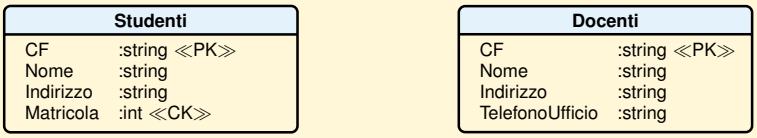
   che raccoglie tutti gli utenti; gli attributi Matricola e TelefonoUfficio possono assumere il valore null, ed il Discriminatore indica se un utente è uno studente o un docente;
2. *partizionamento verticale*: si definiscono tre relazioni

3. Per semplicità, trascuriamo il problema dell'eventuale ridefinizione del tipo degli attributi nelle sottoclassi.



Utenti contiene i dati comuni a tutti gli utenti, mentre le altre due relazioni contengono gli attributi propri delle sottoclassi, nonché il codice fiscale, per risalire al nome e all'indirizzo;

3. *partizionamento orizzontale*: trattandosi di sottoclassi che soddisfano il vincolo di copertura si definiscono solo due relazioni che contengono separatamente tutte le informazioni relative a Studenti e Docenti.



La tecnica da usare si può scegliere come segue.

Se gli attributi propri delle sottoclassi sono pochi si sceglie la relazione unica, che è la tecnica più semplice. Altrimenti, si sceglie tra le altre due tecniche tenendo presente che:

- il partizionamento orizzontale divide gli elementi della superclasse in più relazioni diverse, per cui non è possibile mantenere un vincolo referenziale verso la superclasse stessa; in conclusione, questa tecnica non si usa se nello schema relazionale grafico c'è una freccia che entra nella superclasse;
- il partizionamento verticale rende più complessa la ricostituzione di tutte le informazioni relative ad un elemento della sottoclasse, mentre il partizionamento orizzontale rende più complessa l'operazione di visita a tutti gli elementi della superclasse, per cui è necessario valutare l'importanza relativa di queste due operazioni;
- il partizionamento orizzontale è preferibile in presenza di un vincolo di copertura perché ne può garantire il mantenimento; il partizionamento orizzontale funziona al meglio in presenza di un vincolo di disgiunzione, perché, quando un oggetto appartiene a più sottoclassi, questa rappresentazione replica la chiave e gli attributi della superclasse in tutte le relazioni a cui l'oggetto appartiene, e diviene complesso mantenere la consistenza di queste informazioni replicate.

La Figura 4.8 rappresenta l'esecuzione grafica delle tre possibili trasformazioni, mostrando anche l'effetto dell'operazione sulle associazioni che escono ed entrano nei tre rettangoli.

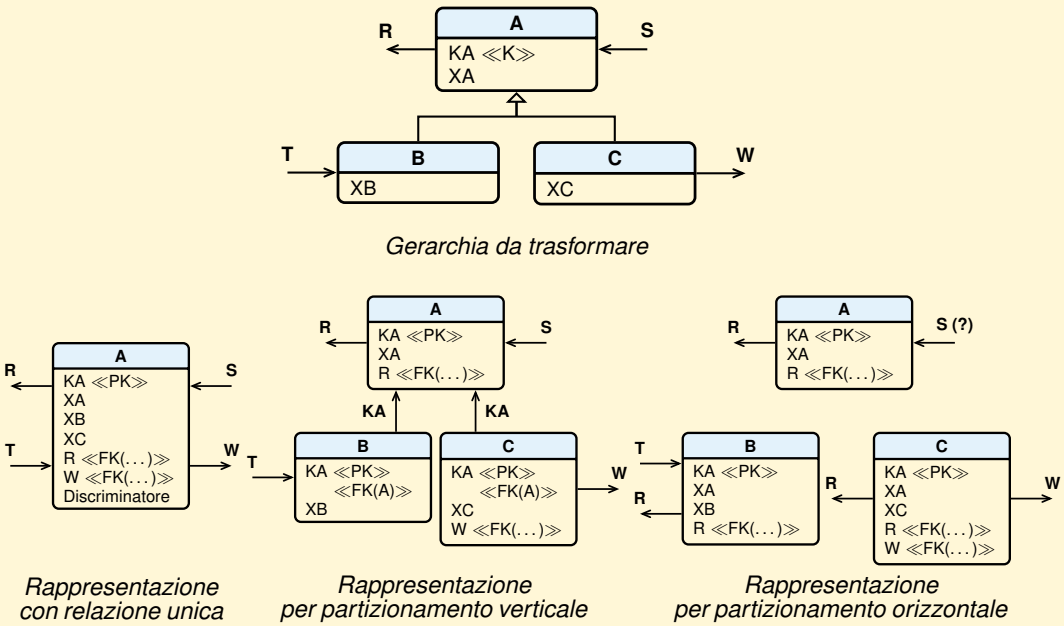


Figura 4.8: Regole per il **PASSO 3** di trasformazione

Applicando questa trasformazione alla Figura 4.7, si ha la Figura 4.9. Per evitare l'uso di attributi con valore null le sottoclassi immediate delle classi Libri e Utenti sono state rappresentate sfruttando il partizionamento verticale, mentre le sottoclassi di Prestiti sono state rappresentate con il partizionamento orizzontale (senza la superclasse perché le due sottoclassi sono una copertura).

4.2.4 PASSO 4: Definizione delle chiavi primarie

Il quarto passo della trasformazione è il primo che non può essere eseguito sullo schema grafico ma va eseguito a partire da un'analisi degli attributi.

In questa fase si deve definire, per ognuna delle relazioni dello schema, un insieme di attributi che funga da chiave primaria.

Per prima cosa, si considerano le relazioni che corrispondono a classi dello schema originale che erano prive di superclassi (classi *radice*). La chiave primaria è di norma un attributo artificiale, tipicamente un numero progressivo assegnato dal sistema. È possibile utilizzare un attributo presente nella classe, purché l'attributo sia univoco, totale e costante. Nei nostri esempi, per semplicità, useremo questo approccio.

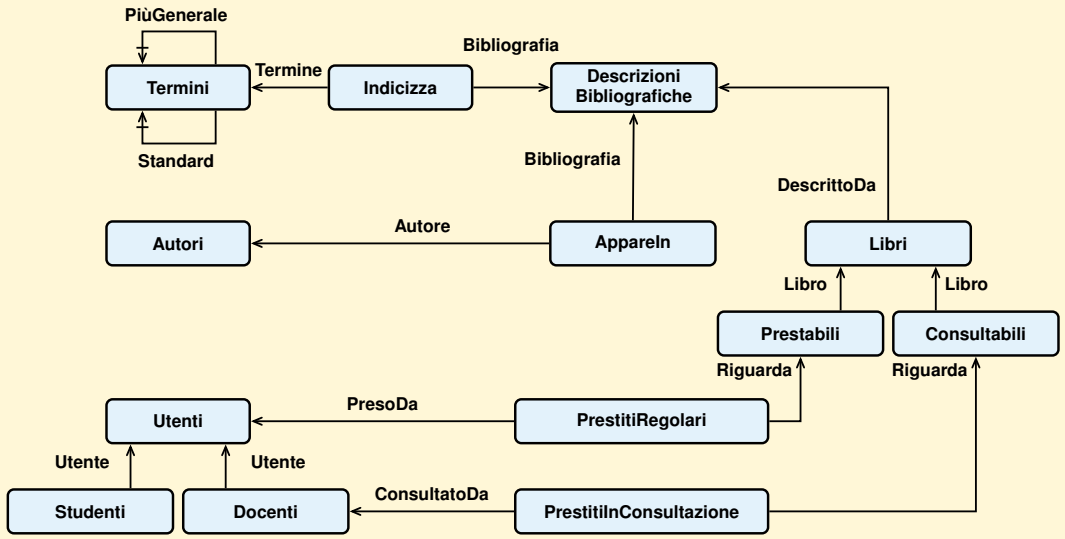


Figura 4.9: Effetto del **PASSO 3** di trasformazione

Poi, per ogni relazione dello schema che corrisponde ad una sottoclasse dello schema originario, la chiave primaria sarà la stessa della superclasse. Infine, per le relazioni che corrispondono ad associazioni molti a molti nello schema originario, la chiave primaria sarà costituita dalla concatenazione delle chiavi esterne.

L'inserimento di una chiave primaria in ogni collezione trasforma effettivamente tutte le classi in relazioni in senso matematico, ovvero in collezioni dove non possono esistere due elementi diversi che coincidono su tutti gli attributi. Tuttavia, per soddisfare il modello relazionale, è necessario eliminare da queste relazioni gli attributi multivalore e strutturati.

In Figura 4.10 è mostrato la versione dello schema di esempio con gli attributi, le chiavi primarie e le chiavi esterne.

4.2.5 PASSO 5: Rappresentazione degli attributi multivalore

Una proprietà multivalore di una classe C si rappresenta eliminando il corrispondente attributo da C e creando una nuova relazione N con una *chiave di due attributi*: una chiave esterna che fa riferimento alla chiave primaria di C ed un attributo che corrisponde all'attributo multivalore da trasformare. Un oggetto di C con chiave primaria k ed in cui l'attributo assume valore $a_1, \ldots, a_n$ si rappresenta poi inserendo nella nuova relazione N le coppie $(k, a_1), \ldots, (k, a_n)$.

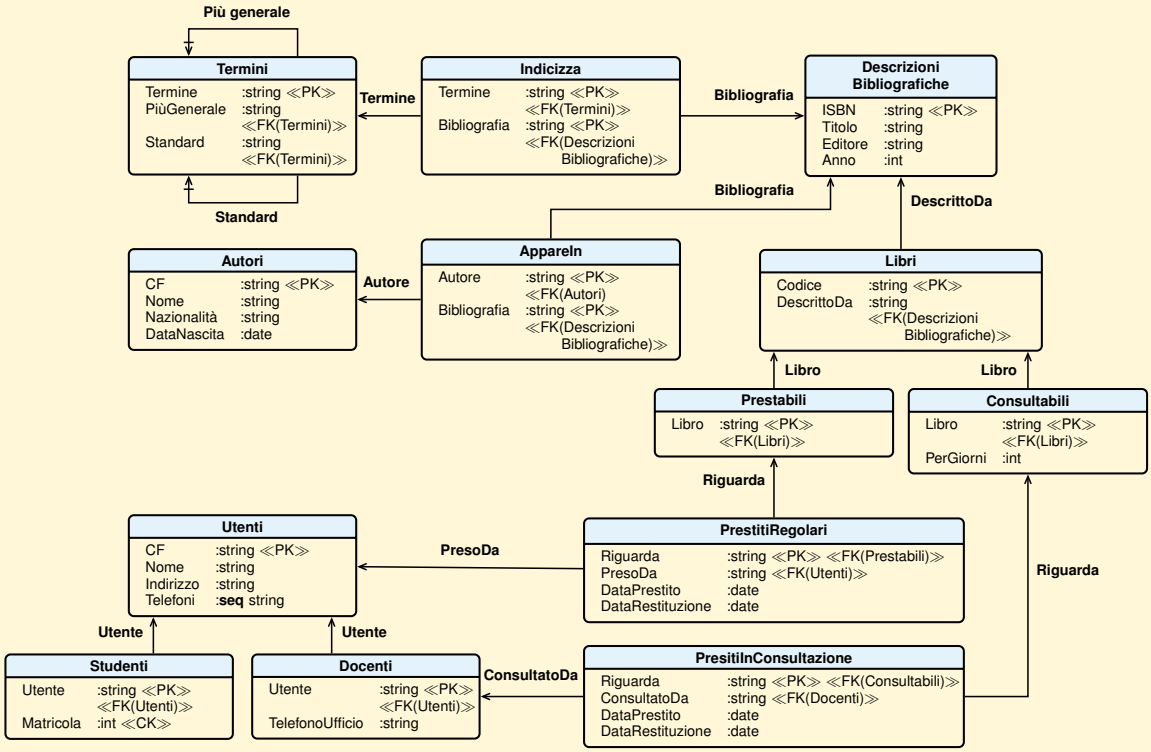
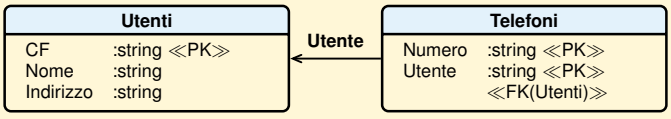


Figura 4.10: Lo schema dell'esempio con gli attributi

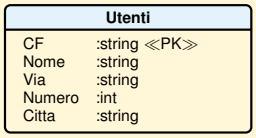
Ad esempio, si consideri la classe Utenti con l'attributo multivalore Telefoni. Con la trasformazione, si ottengono le due seguenti relazioni:



4.2.6 PASSO 6: Appiattimento degli attributi composti

Se un attributo A_i di uno schema di relazione è di tipo $[A_{i1} : T_{i1}, \ldots, A_{ij} : T_{ij}]$, si sostituisce con gli attributi $A_{i1} : T_{i1}, \ldots, A_{ij} : T_{ij}$. Se A_i faceva parte della chiave primaria dello schema di relazione, si sostituisce A_i con gli attributi $A_{i1}, \ldots, A_{ij}$ nella chiave, e poi si verifica che non esista un sottoinsieme degli attributi della nuova chiave primaria che è esso stesso una chiave.

Ad esempio, se l'attributo Indirizzo di utenti invece che String fosse definito composto da tre attributi [Via :string, Numero: int, Citta :string] applicando la trasformazione alla relazione Utenti si otterrebbe:



Al termine di questo passo, lo schema in Figura 4.11 rappresenta uno schema relazionale che equivale a quello ad oggetti di partenza, se non per il fatto che alcuni vincoli sono andati perduti.

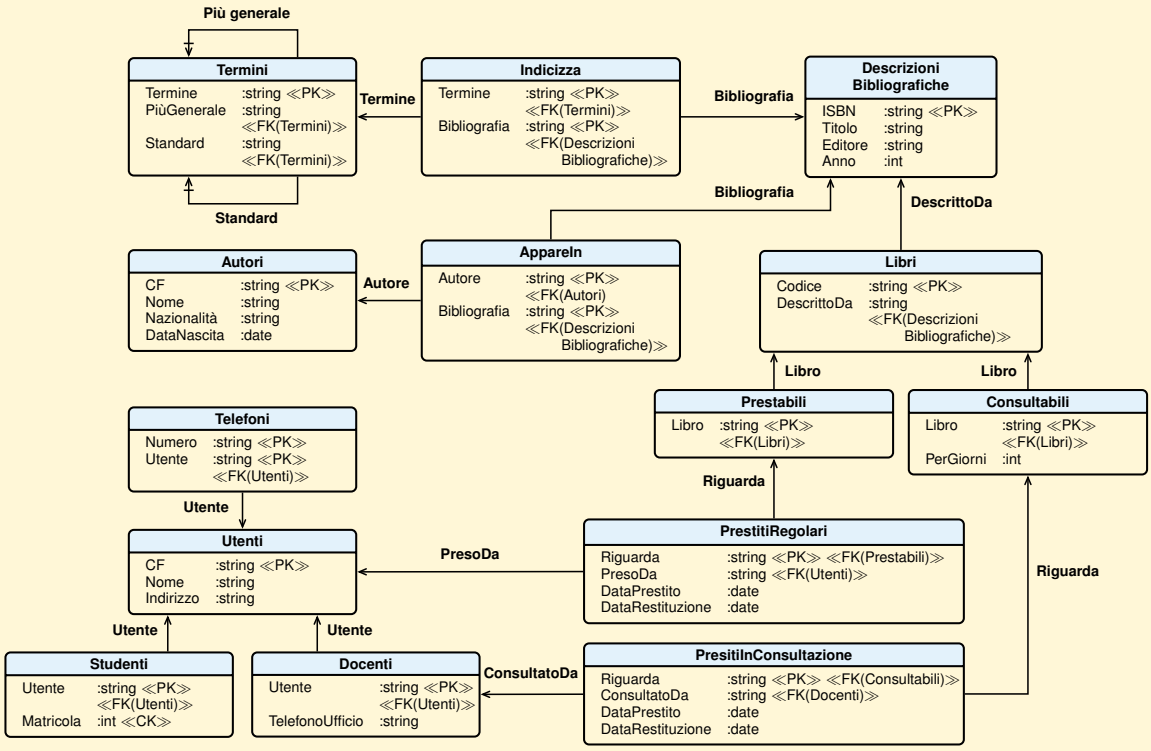


Figura 4.11: Lo schema relazionale finale per la biblioteca

4.3 Algebra relazionale

Con il termine algebra relazionale si designa in generale un insieme di operatori su relazioni che danno come risultato altre relazioni. Nel seguito si definirà prima un insieme minimo di operatori (*operatori primitivi*), poi si darà la definizione di altri operatori (*operatori derivati*), derivati dai precedenti, che sono comodi per esprimere in modo sintetico alcune operazioni complesse molto frequenti nell'uso di basi di dati relazionali. Infine, si mostreranno alcuni operatori non definibili a partire da quelli di base, ma altrettanto utili nelle applicazioni.

4.3.1 Gli operatori primitivi

Nelle definizioni che seguono useremo le seguenti notazioni:

- i simboli R, S ecc. sono nomi di relazioni; A, B, C, A_1, A_2 ecc. sono nomi di attributi e X, Y, X_1 ecc. sono insiemi di attributi;
- XY è un'abbreviazione per $X \cup Y$;
- una relazione con ennuple $t_1, t_2, \ldots, t_n$ verrà denotata con $\{t_1, t_2, \ldots, t_n\}$, e una relazione vuota verrà denotata con $\{\}$;
- i simboli A_i, A_j ecc. denotano attributi e $t_k.A_i$ denota il valore di A_i nell'ennupla t_k; se X è un sottoinsieme degli attributi di t, $t.X$ o $t[X]$ denota l'ennupla ottenuta da t considerando solo gli attributi in X;
- quando due relazioni R ed S hanno lo stesso attributo A_j, in caso di ambiguità, $R.A_j$ denota l'attributo A_j della relazione R ed $S.A_j$ denota l'attributo A_j della relazione S.

Gli operatori primitivi sono: *ridenominazione*, *unione*, *differenza*, *proiezione*, *restrizione* e *prodotto*.

Ridenominazione

Questo operatore si usa per cambiare il tipo di una relazione R.

Siano X gli attributi di R, A e B due attributi tali che $A \in X$ e $B \notin X$. R con A ridenominato B, denotata con $\rho_{A \leftarrow B}(R)$, è una relazione con attributi $X - \{A\} \cup \{B\}$ definita come segue:

$$\rho_{A \leftarrow B}(R) = \{t \mid \exists u \in R \text{ tale che } t[B] = u[A] \wedge t[C] = u[C] \text{ se } C \neq B)\}.$$

Unione

$$R \cup S = \{t \mid t \in R \vee t \in S\},$$

con R ed S relazioni dello stesso tipo. Restituisce la relazione ottenuta facendo l'unione delle ennuple di R con quelle di S. Ad esempio, se $R : \{T\}$, e $t : T$ un'ennupla non in R, $R \cup \{t\}$ sta per la relazione ottenuta aggiungendo ad R l'ennupla t.

Differenza

$R - S = \{t \mid t \in R \wedge t \notin S\}$,

con R ed S relazioni dello stesso tipo. Restituisce la relazione contenente le ennuple di R non presenti in S. Ad esempio, se $R : \{T\}$, e $t : T$ un'ennupla in R, $R - \{t\}$ sta per la relazione ottenuta eliminando l'ennupla t da R.

Proiezione

$\pi_{A_1,A_2,\ldots,A_m}(R) = \{t[A_1 A_2 \ldots A_m] \mid t \in R\}$,

con $A_1, A_2, \ldots, A_m$ attributi di R. Restituisce una relazione di tipo $\{(A_1 : T_1, A_2 : T_2, \ldots, A_m : T_m)\}$ i cui elementi sono la copia delle ennuple di R proiettate sugli attributi $A_1, A_2, \ldots, A_m$. Poiché le relazioni sono insiemi, eventuali ennuple uguali dopo la proiezione appaiono una sola volta nel risultato.

Restrizione

$\sigma_\phi(R) = \{t \mid t \in R \wedge \phi(t)\}$.

Restituisce una relazione dello stesso tipo di R i cui elementi sono la copia delle ennuple di R che soddisfano la condizione. La condizione ϕ è una formula definita come segue:

- $A_i \, \theta \, A_j$, con A_i e A_j attributi di R e θ un operatore di confronto $\{<, >, =, \neq, \leqslant, \geqslant\}$;
- $A_i \, \theta \, c$, oppure $c \, \theta \, A_i$, con θ un operatore di confronto e c una costante in $\mathrm{dom}(A_i)$;
- se ϕ e ψ sono formule, allora lo sono anche $\phi \wedge \psi$, $\phi \vee \psi$ e $\neg\psi$.

Prodotto

$R \times S = \{tu \mid t \in R \wedge u \in S\}$,

con $R(A_1 : T_1, \ldots, A_n : T_n)$ e $S(A_{n+1} : T_{n+1}, \ldots, A_{n+m} : T_{n+m})$ relazioni con attributi distinti. Restituisce una relazione del tipo $\{(A_1 : T_1, \ldots, A_n : T_n, A_{n+1} : T_{n+1}, \ldots, A_{n+m} : T_{n+m})\}$ con elementi ottenuti concatenando ogni ennupla di R con tutte quelle di S. La concatenazione tu di due ennuple t e u è un'ennupla che ha come coppie (A_i, V_i) tutte quelle di t e di u, e si indica anche con $t \circ u$. La relazione risultante ha grado uguale alla somma dei gradi degli operandi, e cardinalità uguale al prodotto delle cardinalità degli operandi.

Il nome di questo operatore è dato dalla sua somiglianza con il prodotto cartesiano di insiemi, sebbene dia come risultato un insieme di ennuple concatenate invece di un insieme di coppie di ennuple, come accadrebbe nel normale prodotto cartesiano.

Espressione dell'algebra relazionale

Un'espressione dell'algebra relazionale è definita come segue:

- una relazione R o una relazione $\{t_1, t_2, \ldots, t_n\}$ sono espressioni dell'algebra;
- se E, E_1 e E_2 sono espressioni dell'algebra, lo sono anche
  - $E_1 \cup E_2$, se E_1 e E_2 hanno lo stesso tipo;
  - $E_1 - E_2$, se E_1 e E_2 hanno lo stesso tipo;
  - $E_1 \times E_2$, se E_1 e E_2 non hanno attributi comuni;
  - $\sigma_C(E)$, se C è una condizione su attributi di E;
  - $\pi_X(E)$, se X sono attributi di E;
  - $\rho_{A \leftarrow B}(E)$, se X sono gli attributi di E, A e B sono due attributi tali che $A \in X$ e $B \notin X$.

Esempio 4.1
Siano $R(A, B, C)$, $S(A, B, C)$ e $T(A, D)$, tre schemi di relazioni con attributi definiti sui domini: $dom(A) = \{a1, a2, a3\}$, $dom(B) = \{b1, b2, b3\}$, $dom(C) = \{c1, c2\}$ e $dom(D) = \{d1, d2\}$. Si mostra il risultato degli operatori dell'algebra applicati alle tre seguenti relazioni:

R

A	B	C
a1	b1	c1
a1	b1	c2
a2	b1	c1
a3	b1	c1

S

A	B	C
a1	b1	c1
a1	b2	c2

T

A	D
a1	d1
a2	d2

$R \cup S =$

A	B	C
a1	b1	c1
a1	b1	c2
a2	b2	c2
a2	b1	c1
a3	b1	c1

$R - S =$

A	B	C
a1	b1	c2
a2	b1	c1
a3	b1	c1

$\sigma_{A='a'_1}(R) =$

A	B	C
a1	b1	c1
a1	b1	c2

$\pi_A(R) =$

A
a1
a2
a3

$T' = \rho_{A \leftarrow A'}(T) =$

A'	D
a1	d1
a2	d2

$R \times T' =$

A	B	C	A'	D
a1	b1	c1	a1	d1
a1	b1	c1	a2	d2
a1	b1	c2	a1	d1
a1	b1	c2	a2	d2
a2	b1	c1	a1	d1
a2	b1	c1	a2	d2
a3	b1	c1	a1	d1
a3	b1	c1	a2	d2

Esempio 4.2
Vediamo degli esempi di operazioni sulla base di dati con relazioni Studenti e Esami, e i loro risultati. L'attributo Matricola è chiave primaria per la relazione Studenti e l'attributo Candidato è chiave esterna per la relazione Esami ed è definito sullo stesso dominio di Matricola.

Studenti

Nome	Matricola	Provincia	Nascita
Isaia	71523	Pisa	1998
Rossi	67459	Lucca	1999
Bianchi	79856	Livorno	1998
Bonini	75649	Pisa	1999

Esami

Materia	Candidato*	Data	Voto
BD	71523	12/01/2018	28
BD	67459	15/09/2019	30
IA	79856	25/10/2020	30
BD	75649	27/06/2018	25
IS	71523	10/10/2019	18

"Trovare il nome e la matricola degli studenti della provincia di Pisa"

$\pi_{\text{Nome, Matricola}}(\sigma_{\text{Provincia = 'PI'}}(\text{Studenti}))$

Nome	Matricola
Isaia	71523
Bonini	75649

"Trovare il nome e l'anno di nascita degli studenti che hanno superato l'esame di BD con trenta"

Supponiamo di calcolare il risultato in più passi usando la seguente strategia: poiché occorrono informazioni sia dalla relazione Studenti che dalla relazione Esami, si calcola prima il prodotto delle due relazioni, producendo una relazione intermedia T

$$T = (\text{Studenti} \times \text{Esami})$$

Nome	Matricola	Provincia	Nascita	Materia	Candidato	Data	Voto
Isaia	71523	PI	1998	BD	71523	12/01/2018	28
Isaia	71523	PI	1998	BD	67459	15/09/2019	30
Isaia	71523	PI	1998	IA	79856	25/10/2020	30
Isaia	71523	PI	1998	BD	75649	27/06/2018	25
Isaia	71523	PI	1998	IS	71523	10/10/2019	18
Rossi	67459	LU	1999	BD	71523	12/01/2018	28
Rossi	67459	LU	1999	BD	67459	15/09/2019	30
Rossi	67459	LU	1999	IA	79856	25/10/2020	30
Rossi	67459	LU	1999	BD	75649	27/06/2018	25
Rossi	67459	LU	1999	IS	71523	10/10/2019	18
Bianchi	79856	LI	1998	BD	71523	12/01/2018	28
Bianchi	79856	LI	1998	BD	67459	15/09/2019	30
Bianchi	79856	LI	1998	IA	79856	25/10/2020	30
Bianchi	79856	LI	1998	BD	75649	27/06/2018	25
Bianchi	79856	LI	1998	IS	71523	10/10/2019	18
Bonini	75649	PI	1999	BD	71523	12/01/2018	28
Bonini	75649	PI	1999	BD	67459	15/09/2019	30
Bonini	75649	PI	1999	IA	79856	25/10/2020	30
Bonini	75649	PI	1999	BD	75649	27/06/2018	25
Bonini	75649	PI	1999	IS	71523	10/10/2019	18

Il prodotto di due relazioni è un'operazione che di solito non si usa da sola, né per relazioni che non descrivono fatti in associazione mediante il meccanismo della chiave esterna. Infatti, le uniche ennuple significative in T sono quelle con uguali valori degli attributi Matricola e Candidato:

$$R = \sigma_{\text{Matricola = Candidato}}(T)$$

Pertanto per concatenare ennuple in associazione, si restringe sempre il prodotto alle ennuple con valore uguale della chiave esterna e chiave primaria. Per questa ragione più avanti si introduce un operatore derivato, chiamato *giunzione*, per riferirsi alla combinazione di queste due operazioni.

Nome	Matricola	Provincia	Nascita	Materia	Candidato	Data	Voto
Isaia	71523	PI	1998	BD	71523	12/01/2018	28
Isaia	71523	PI	1998	IS	71523	10/10/2019	18
Rossi	67459	LU	1999	BD	67459	15/09/2019	30
Bianchi	79856	LI	1998	IA	79856	25/10/2020	30
Bonini	75649	PI	1999	BD	75649	27/06/2018	25

Il risultato finale è calcolato con l'espressione:

$$\pi_{\text{Nome, Nascita}}(\sigma_{\text{Materia = 'BD'} \wedge \text{Voto = 30}}(R))$$

Lo stesso risultato poteva essere calcolato direttamente con l'espressione:

$$\pi_{\text{Nome, Nascita}}(\sigma_{\text{Materia = 'BD'} \wedge \text{Voto = 30} \wedge \text{Matricola = Candidato}}(\text{Studenti} \times \text{Esami}))$$

4.3.2 Operatori derivati

Diamo la definizione di alcuni operatori che sono un'utile abbreviazione di espressioni dell'algebra usate frequentemente: *intersezione*, *divisione*, *giunzione* (*join*), *giunzione naturale* (*natural join*) e *semi-giunzione* (*semijoin*).

Intersezione

$R \cap S = \{t \mid t \in R \wedge t \in S\}$,

con R ed S relazioni dello stesso tipo. Restituisce la relazione ottenuta facendo l'intersezione delle ennuple di R e di S. Vale l'equivalenza $R \cap S \equiv R - (R - S)$.

Divisione

$R \div S = \{w \mid \forall s \in S. w \circ s \in R\} = \{w \mid \{w\} \times S \subseteq R\}$.

Siano XY gli attributi di R ed Y gli attributi di S. L'operatore divisione restituisce una relazione $W = R \div S$ (che si legge "W è R divisa per S") con attributi X ed elementi w la proiezione sugli attributi X delle ennuple di R tali che per ogni ennupla s di S, $w \circ s \in R$.

Vale l'equivalenza $R \div S \equiv \pi_X(R) - R_1$, con $R_1 = \pi_X((\pi_X(R) \times S) - R)$.

Il prodotto e la divisione sono legati dalle seguenti relazioni, valide nell'ipotesi che gli attributi di S e W siano disgiunti e la loro unione dia gli attributi di R:

$$(W \times S) \div S = W, \quad (R \div S) \times S \subseteq R$$

In più, $R \div S$ è il massimo insieme per cui vale $(R \div S) \times S \subseteq R$.

La divisione $R \div S$ serve tipicamente a rispondere a domande del tipo: trovare le ennuple di R associate a tutte le ennuple di S. Ad esempio, la seguente espressione restituisce la matricola di tutti gli studenti che hanno superato sia 'BDSI1' che 'IA':

$\pi_{\text{Candidato}}(\text{ Esami } \div \{(\text{Materia:='BDSI1'}), (\text{Materia:='IA'})\})$.

Giunzione

$$R \underset{A_i=A_j}{\bowtie} S = \{t \circ u \mid t \in R, u \in S, t.A_i = u.A_j\},$$

con $R(A_1 : T_1, \ldots, A_n : T_n)$ ed $S(A_{n+1} : T_{n+1}, \ldots, A_{n+m} : T_{n+m})$ relazioni con attributi distinti, A_i attributo di R e A_j attributo di S. Restituisce una relazione di tipo $\{(A_1 : T_1, \ldots, A_n : T_n, A_{n+1} : T_{n+1}, \ldots, A_{n+m} : T_{n+m})\}$ con elementi la copia delle ennuple del prodotto cartesiano di R ed S, con valori uguali per gli attributi A_i e A_j, ovvero $(R \underset{A_i=A_j}{\bowtie} S) \equiv \sigma_{A_i=A_j}(R \times S)$. Questo operatore è di solito chiamato *equijoin*, per distinguerlo dal θ-join in cui la restrizione non è limitata ai valori uguali degli attributi A_i e A_j, ma è consentita con valori in una relazione specificata con un qualsiasi operatore di confronto.

Giunzione naturale

$$R \bowtie S,$$

con gli attributi di uguali nomi in R ed S definiti sugli stessi domini. Sia $R(YX)$ ed $S(ZX)$ con X gli attributi in comune. $R \bowtie S$ restituisce una relazione con attributi (YXZ) tale che

$$t \in R \bowtie S \Leftrightarrow t[YX] \in R \text{ e } t[ZX] \in S.$$

Questo operatore può essere definito a partire dagli operatori primitivi nel seguente modo. Siano $A_1, \ldots, A_n$ gli attributi X comuni ad R ed S.

1. Si cambia il tipo di R ed S, in modo da rendere diversi gli attributi comuni, definendo:

   $R' = \rho_{A_1 \leftarrow RA_1, \ldots, A_n \leftarrow RA_n}(R)$ ed

   $S' = \rho_{A_1 \leftarrow SA_1, \ldots, A_n \leftarrow SA_n}(S)$.

2. Sia $T = \sigma_{RA_1=SA_1 \wedge \ldots \wedge RA_n=SA_n}(R' \times S')$.
3. La giunzione naturale di R ed S è la relazione

   $\rho_{RA_1 \leftarrow A_1, \ldots, RA_n \leftarrow A_n}(\pi_{RA_1, \ldots, RA_n, YZ}(T))$.

Si ottiene così una relazione che ha come attributi l'unione di quelli degli operandi, con le ennuple formate concatenando quelle di R ed S con valori uguali per gli attributi in comune (e togliendo naturalmente le coppie ridondanti).

La giunzione naturale, quindi, è una comoda abbreviazione dell'*equijoin* applicato a relazioni in cui l'associazione fra le ennuple è descritta con la chiave esterna e la chiave primaria costituite da attributi uguali.

Si noti che:

- se R ed S non hanno attributi comuni, $R \bowtie S \equiv R \times S$;
- se R ed S hanno lo stesso schema, $R \bowtie S \equiv R \cap S$;
- gli operatori di giunzione, consentendo di correlare ennuple di relazioni diverse, permettono la ricerca di ennuple in corrispondenza con il meccanismo delle chiavi esterne, e questi operatori sono gli unici che offrono questa possibilità.

Semi-giunzione

$$R \ltimes S = \{t \in R \mid t[X] \in \pi_X(S)\},$$

dove X sono gli attributi comuni tra R ed S. Restituisce le ennuple di R che partecipano alla giunzione naturale di R ed S.

Vale l'equivalenza $R \ltimes S \equiv \pi_{A_1,A_2,\ldots,A_m}(R \bowtie S)$, con $A_1, A_2, \ldots, A_m$ gli attributi di R.

Esempio 4.3
Si mostrano esempi di applicazione degli operatori derivati alle relazioni dell'esempio 6.1.

$R \cap S =$

A	B	C
a1	b1	c1

$R \underset{A=A'}{\bowtie} T' =$

A	B	C	A'	D
a1	b1	c1	a1	d1
a1	b1	c2	a1	d1
a2	b1	c1	a2	d2

$R \bowtie T =$

A	B	C	D
a1	b1	c1	d1
a1	b1	c2	d1
a2	b1	c1	d2

$R \ltimes T =$

A	B	C
a1	b1	c1
a1	b1	c2
a2	b1	c1

4.3.3 Proprietà algebriche degli operatori relazionali

Un'espressione dell'algebra relazionale può essere trasformata in un'altra equivalente sfruttando alcune proprietà degli operatori. Queste trasformazioni sono utili perché possono ridurre di ordini di grandezza il costo di esecuzione delle espressioni (*riscrittura algebrica*).

Si consideri la rappresentazione di un'espressione algebrica come un albero le cui foglie siano le relazioni e i nodi interni sono gli operatori dell'algebra; i figli di un nodo interno N sono gli operandi dell'operatore associato al nodo N (si veda la Figura 4.12).

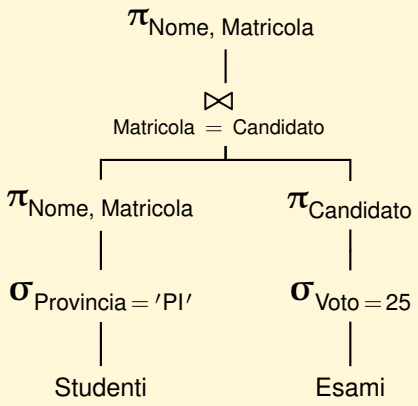


Figura 4.12: Rappresentazione ad albero di un'espressione algebrica

L'idea alla base della riscrittura algebrica è di anticipare l'applicazione degli operatori di proiezione e restrizione rispetto al prodotto e alla giunzione (spostamento verso le foglie della proiezione e restrizione), in modo da ridurre la dimensione dei risultati intermedi. Poiché la restrizione è l'operatore che in generale produce una relazione con un numero di ennuple inferiore rispetto a quello della relazione a cui viene applicato, le proprietà più utili dell'algebra relazionale sono quelle che consentono di anticipare l'operatore di restrizione. È anche utile anticipare l'operazione di proiezione rispetto al prodotto e alla giunzione, perché una proiezione riduce la dimensione delle ennuple dell'operando ed elimina eventuali ennuple uguali dal risultato.

Indicando con E un'espressione dell'algebra relazionale, esempi di regole su cui si basa la riscrittura sono:

1. *Raggruppamento di restrizioni*

   $\sigma_{\phi_X}(\sigma_{\phi_Y}(E)) = \sigma_{\phi_X \wedge \phi_Y}(E)$,

   dove σ_{ϕ_X} è l'operatore di restrizione con condizione sull'insieme di attributi X.

2. *Commutatività della restrizione e della proiezione*

   $\boldsymbol{\pi}_Y(\boldsymbol{\sigma}_{\phi_X}(E)) = \boldsymbol{\sigma}_{\phi_X}(\boldsymbol{\pi}_Y(E))$, *se* $X \subseteq Y$.

   Se la condizione interessa attributi $X \not\subseteq Y$, allora vale:

   $\boldsymbol{\pi}_Y(\sigma_{\phi_X}(E)) = \boldsymbol{\pi}_Y(\boldsymbol{\sigma}_{\phi_X}(\boldsymbol{\pi}_{XY}(E)))$.

3. *Anticipazione della restrizione rispetto al prodotto e alla giunzione*

   $\boldsymbol{\sigma}_{\phi_X}(E_1 \times E_2) = \boldsymbol{\sigma}_{\phi_X}(E_1) \times E_2$,
   $\boldsymbol{\sigma}_{\phi_X}(E_1 \bowtie E_2) = \boldsymbol{\sigma}_{\phi_X}(E_1) \bowtie E_2$,

   se X sono attributi di E_1.

   $\boldsymbol{\sigma}_{\phi_X \wedge \phi_Y}(E_1 \times E_2) = \boldsymbol{\sigma}_{\phi_X}(E_1) \times \boldsymbol{\sigma}_{\phi_Y}(E_2)$,
   $\boldsymbol{\sigma}_{\phi_X \wedge \phi_Y}(E_1 \bowtie E_2) = \boldsymbol{\sigma}_{\phi_X}(E_1) \bowtie \boldsymbol{\sigma}_{\phi_Y}(E_2)$,

   se X sono attributi di E_1 ed Y sono attributi di E_2.

   $\boldsymbol{\sigma}_{\phi_X \wedge \phi_Y \wedge \phi_J}(E_1 \times E_2) = \boldsymbol{\sigma}_{\phi_X}(E_1) \underset{\boldsymbol{\phi_J}}{\bowtie} \boldsymbol{\sigma}_{\phi_Y}(E_2)$,

   se X sono attributi di E_1, Y sono attributi di E_2 e ϕ_J è una condizione di giunzione.

4. *Raggruppamento di proiezioni*

   $\boldsymbol{\pi}_Z(\boldsymbol{\pi}_Y(E)) = \boldsymbol{\pi}_Z(E)$,

   dove $\boldsymbol{\pi}_Z$ è l'operatore di proiezione sull'insieme di attributi Z, e $Z \subseteq Y$ in espressioni ben formate.

5. *Eliminazioni di proiezioni superflue*

   $\boldsymbol{\pi}_Z(E) = E$,

   se Z sono gli attributi di E.

6. *Anticipazione della proiezione rispetto al prodotto e alla giunzione*

   $\boldsymbol{\pi}_{XY}(E_1 \times E_2) = \boldsymbol{\pi}_X(E_1) \times \boldsymbol{\pi}_Y(E_2)$,
   $\boldsymbol{\pi}_{XY}(E_1 \underset{\boldsymbol{\phi_J}}{\bowtie} E_2) = \boldsymbol{\pi}_X(E_1) \underset{\boldsymbol{\phi_J}}{\bowtie} \boldsymbol{\pi}_Y(E_2)$,

   dove X sono attributi di E_1, Y sono attributi di E_2 e ϕ_J la condizione di giunzione con attributi $J \subseteq XY$.

   $\boldsymbol{\pi}_{XY}(E_1 \underset{\boldsymbol{\phi_J}}{\bowtie} E_2) = \boldsymbol{\pi}_{XY}((\boldsymbol{\pi}_{XX_{E_1}}(E_1)) \underset{\boldsymbol{\phi_J}}{\bowtie} (\boldsymbol{\pi}_{YX_{E_2}}(E_2)))$,

   dove X sono attributi di E_1, Y sono attributi di E_2, X_{E_1} sono gli attributi di giunzione di E_1 che non sono in XY e X_{E_2} sono gli attributi di giunzione di E_2 che non sono in XY.

7. *Commutatività del prodotto e della giunzione*

   $E_1 \times E_2 = E_2 \times E_1$
   $E_1 \underset{\boldsymbol{\phi_J}}{\bowtie} E_2 = E_2 \underset{\boldsymbol{\phi_J}}{\bowtie} E_1$

8. *Associatività del prodotto e della giunzione*

   $(E_1 \bowtie E_2) \bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3)$,
   $(E_1 \underset{\phi_{J_1}}{\bowtie} E_2) \underset{\phi_{J_2} \wedge \phi_{J_3}}{\bowtie} E_3 = E_1 \underset{\phi_{J_1} \wedge \phi_{J_3}}{\bowtie} (E_2 \underset{\phi_{J_2}}{\bowtie} E_3)$,

   dove ϕ_{J_2} contiene attributi solo di E_2 e E_3. Se mancano le condizioni di giunzione, ne segue che anche il prodotto $\times$ è associativo.

9. *Commutatività degli operatori insiemistici*

   $E_1 \cup E_2 = E_2 \cup E_1$
   $E_1 \cap E_2 = E_2 \cap E_1$
   $E_1 - E_2 \neq E_2 - E_1$

10. *Associatività degli operatori insiemistici*

    $(E_1 \cup E_2) \cup E_3 = E_1 \cup (E_2 \cup E_3)$
    $(E_1 \cap E_2) \cap E_3 = E_1 \cap (E_2 \cap E_3)$

11. *Anticipazione della restrizione rispetto agli operatori insiemistici*

    $\boldsymbol{\sigma}_{\phi_X}(E_1 - E_2) = \boldsymbol{\sigma}_{\phi_X}(E_1) - \boldsymbol{\sigma}_{\phi_X}(E_2)$

    l'equivalenza vale anche sostituendo $-$ con $\cup$ o $\cap$. Mentre

    $\boldsymbol{\sigma}_{\phi_X}(E_1 - E_2) = \boldsymbol{\sigma}_{\phi_X}(E_1) - E_2$

    vale anche sostituendo $-$ con $\cap$, ma non con $\cup$.

12. *Anticipazione della proiezione rispetto agli operatori insiemistici*

    $\boldsymbol{\pi}_X(E_1 \cup E_2) = (\boldsymbol{\pi}_X(E_1)) \cup (\boldsymbol{\pi}_X(E_2))$

Un possibile algoritmo di riscrittura algebrica procede quindi secondo i seguenti passi:

1. Si anticipa l'esecuzione delle restrizioni sulle proiezioni usando la regola 2 da destra verso sinistra (in tutti gli altri casi le regole verranno usate nell'altra direzione); dato che la regola è usata in questa direzione, vale sempre la condizione $X \subseteq Y$.
2. Si raggruppano le restrizioni usando la regola 1.
3. Si anticipa l'esecuzione delle restrizioni sul prodotto e sulla giunzione usando la regola 3.
4. Si ripetono i tre passi precedenti fino a che è possibile.
5. Si eliminano le proiezioni superflue usando la regola 5.
6. Si raggruppano le proiezioni usando la regola 4.
7. Si anticipa l'esecuzione delle proiezioni rispetto al prodotto e alla giunzione usando ripetutamente la regola 2, ma solo nel caso in cui E è un prodotto o una giunzione, e la regola 6.

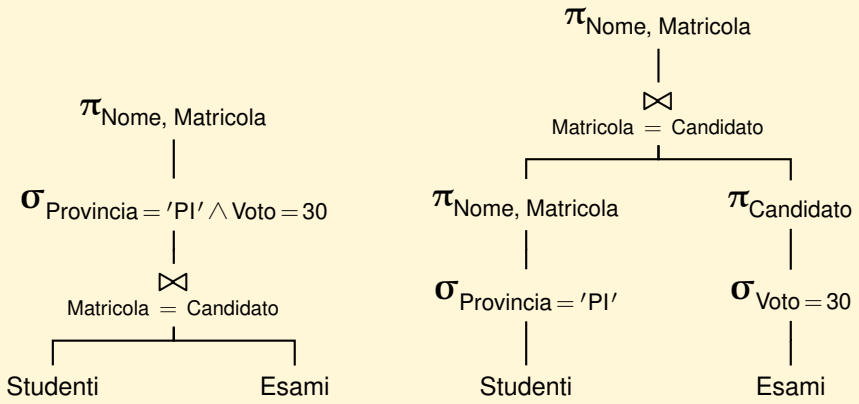
In generale, quindi, il risultato di questo algoritmo è un'espressione in cui la restrizione e la proiezione sono eseguite il più presto possibile, e la restrizione è anticipata rispetto alla proiezione. In particolare, nel caso di espressioni con una sola giunzione, l'espressione ottimizzata ha la forma:

$\pi_{X_1}(\pi_{X_2}(\sigma_{\phi_{Y_2}}(R)) \bowtie \pi_{X_3}(\sigma_{\phi_{Y_3}}(S)))$.

Ad esempio, applicando l'algoritmo all'espressione

$$\pi_{\text{Nome, Matricola}}(\sigma_{\text{Provincia = 'PI'} \wedge \text{Voto = 30}}(\text{Studenti} \underset{\text{Matricola=Candidato}}{\bowtie} \text{Esami}))$$

l'albero dell'interrogazione si trasforma come mostrato in Figura 4.13.



(a) *Albero iniziale* **(b)** *Albero trasformato*

Figura 4.13: Trasformazione di un'espressione algebrica

4.3.4 Altri operatori

Esistono interrogazioni su una base di dati relazionale che non possono essere formulate usando solo gli operatori di base dell'algebra relazionale. Per questa ragione sono stati proposti altri operatori, alcuni dei quali sono presenti in tutti i sistemi commerciali come la *proiezione generalizzata* e le *funzioni di aggregazione e raggruppamenti*.

Proiezione generalizzata

La proiezione generalizzata estende la proiezione con la possibilità di usare costanti o espressioni aritmetiche nella lista degli attributi. Per comodità si può anche assegnare un'etichetta ad un'espressione con l'operatore **AS**:

$\pi_{e_1\ \textbf{AS}\ \text{ide1}, e_2\ \textbf{AS}\ \text{ide2},\ldots, e_n\ \textbf{AS}\ \text{ideN}}(E)$,

con $e_1, \ldots, e_n$ espressioni aritmetiche, ottenute a partire da costanti e attributi di E, ed ide1, ..., ideN etichette distinte.

Per esempio, se $A_1, A_2, \ldots, A_m$ attributi di tipo intero di R, si può scrivere:

$\pi_{A_1, 2\ \textbf{AS}\ \text{due}, A_1+A_3\ \textbf{AS}\ \text{sommaA1eA3}}(R)$,

per ottenere una relazione di tipo $\{(A_1\text{: int, due: int, sommaA1eA3: int})\}$.

Funzioni di aggregazione e raggruppamenti

Le *funzioni di aggregazione* hanno come argomenti multinsiemi e ritornano come risultato un valore. Per esempio, la funzione di aggregazione *sum* ritorna la somma degli elementi, *avg* ritorna la media dei valori, *count* ritorna il numero degli elementi, *min* e *max* ritornano il minimo e il massimo valore degli elementi. Quando interessa applicare una funzione di aggregazione ad un multinsieme ignorando eventuali duplicati, il nome della funzione si estende con la stringa "*-distinct*". Per esempio, per contare gli elementi diversi di un multinsieme invece della funzione *count* si usa *count-distinct*.

Le funzioni di aggregazione si possono usare con l'operatore di proiezione generalizzato per produrre una relazione con un'unica ennupla. Per esempio, se $A_1, A_2, \ldots, A_m$ sono attributi di tipo intero della relazione R, si può scrivere:

$\pi_{\text{count}(A_1)\ \textbf{AS}\ \text{n}, \max(A_2)\ \textbf{AS}\ \text{maxA2}}(R)$,

per ottenere una relazione di tipo $\{(n : int, maxA2 : int)\}$, con un'unica ennupla i cui campi sono il numero dei valori di A_1 e il massimo valore di A_2 in R. Lo stesso risultato si ottiene scrivendo

$\pi_{\text{count}(*)\ \textbf{AS}\ \text{n}, \max(A_2)\ \textbf{AS}\ \text{maxA2}}(R)$,

dove *count*(*) ritorna il numero degli elementi di R. Nella proiezione, le funzioni di aggregazione non si possono usare insieme ad attributi diversi da costanti. Per esempio, la seguente espressione è scorretta:

$\pi_{A_1, \max(A_2)\ \textbf{AS}\ \text{maxA2}}(R)$

Per combinare nella proiezione attributi e valori di funzioni di aggregazione bisogna usare le funzioni di aggregazione con l'operatore di *raggruppamento* γ, definito come segue:

$_{A_1,\ldots,A_n}\gamma_{f_1,\ldots,f_m}(E)$,

dove $A_1, \ldots, A_n$ sono attributi di E ed $f_1, \ldots, f_m$ sono funzioni di aggregazione con argomenti che sono espressioni aritmetiche ottenute a partire da costanti e attributi di E.

L'operatore restituisce una relazione di tipo $\{(A_1 : T_1, \ldots, A_n : T_n, f_1 : T_{f_1}, \ldots, f_m : T_{f_m})\}$, con $T_{f_1}, \ldots, T_{f_m}$ i tipi dei risultati delle espressioni $f_1, \ldots, f_m$, i cui elementi sono calcolati in questo modo (Figura 4.14):

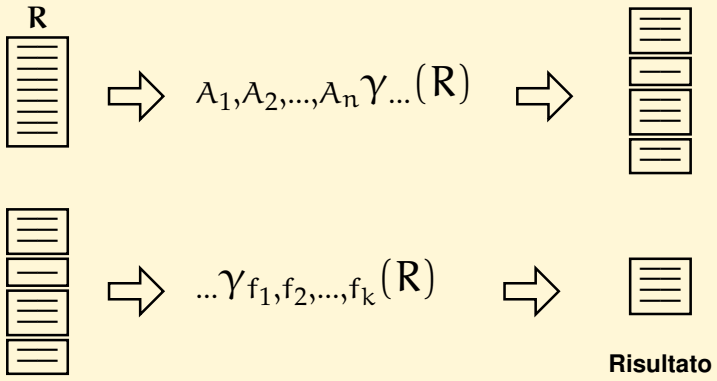


Figura 4.14: Valutazione del raggruppamento

– si partizionano le ennuple di E in un insieme di gruppi, mettendo nello stesso gruppo tutte le ennuple che coincidono su tutti gli attributi $A_1, \ldots, A_n$. Se l'insieme $\{A_1, \ldots, A_n\}$ è vuoto, gli elementi di E fanno parte di un unico gruppo;
– si calcolano le funzioni di aggregazione per ogni gruppo;
– si restituisce una relazione con un'ennupla per ogni gruppo e componenti i valori degli attributi $A_1, \ldots, A_n$ e i risultati delle espressioni $f_1, \ldots, f_m$.

Esempio 4.4
Ad esempio,

$_{\text{Materia}}\gamma_{\text{avg(Voto)}}(\text{Esami})$,

calcola la media dei valori dell'attributo Voto per ogni gruppo di esami con la stessa materia, e restituisce la relazione con attributi Materia e avg(Voto). Questa espressione, applicata alla relazione:

Esami

Materia	**Candidato***	**Data**	**Voto**
BD	71523	12/01/2018	28
BD	67459	15/09/2019	30
IA	79856	25/10/2020	30
BD	75649	27/06/2018	25
IS	71523	10/10/2019	18

dà come risultato la relazione:

Materia	avg(Voto)
BD	27.6
IA	30
IS	18

Come nella proiezione generalizzata, gli attributi del risultato si possono ridenominare usando l'operatore **AS**. Ad esempio,

$_{\text{Materia}}\gamma_{\text{avg(Voto)}}$ **AS** $_{\text{VotoMedio}}(\text{Esami})$,

dà come risultato la relazione:

Materia	VotoMedio
BD	27.6
IA	30
IS	18

Esistono altre regole di riscrittura algebrica di espressioni che usano il raggruppamento, ad esempio l'*anticipazione della proiezione rispetto al raggruppamento*:
$\sigma_\phi({}_A\gamma_F(E)) = {}_A\gamma_F(\sigma_\phi(E))$, se ϕ usa solo attributi in A.

4.4 Calcolo relazionale su ennuple

Finora una base di dati relazionale è stata pensata come un insieme di relazioni sulle quali sono previsti certi operatori che costituiscono un'algebra. Un altro punto di vista, ricco di conseguenze teoriche e pratiche, è di pensare una base di dati relazionale come l'estensione di un insieme di predicati, uno per ogni relazione, che costituiscono un'*interpretazione*, o un *modello*, di una logica del primo ordine (*model theoretic perspective*). Un'interrogazione è espressa come una formula ben formata di un linguaggio del primo ordine e il suo risultato è ottenuto cercando gli elementi del modello che, sostituiti alle variabili libere della formula, la rendono vera. Questo punto di vista consente anche di trattare vincoli d'integrità generali, espressi con formule del linguaggio del primo ordine. Una base di dati soddisfa i vincoli d'integrità se essi sono veri considerando l'estensione della base di dati come un modello.

L'uso della logica consente di guardare ad una base di dati relazionale anche da un altro punto di vista, chiamato *proof theoretic perspective*: una base di dati è vista come un insieme di formule di un linguaggio del primo ordine, piuttosto che come un modello, e le interrogazioni sono formule da dimostrare, a partire dalla base di dati come premessa. Pertanto, la valutazione di un'interrogazione diventa una dimostrazione, dalla quale si estrae il risultato cercato.

Nel seguito, si adotta il primo punto di vista e si presenta un particolare linguaggio del primo ordine per basi di dati relazionali, chiamato *calcolo relazionale su ennuple*, mentre il punto di vista *proof theoretic* verrà illustrato nella sezione successiva.

Il calcolo relazionale è un linguaggio che permette di definire il risultato di un'interrogazione come l'insieme di quelle ennuple che soddisfano una certa condizione ϕ.[4] Ad esempio, l'insieme delle matricole degli studenti che hanno superato qualcuno degli esami elencati nella relazione Materie(Materia), si può definire come:

$$\{t.\text{Matricola} \mid t \in \text{Studenti}, \exists m \in \text{Materie}.\ \exists e \in \text{Esami}.\ e.\text{Candidato} = t.\text{Matricola} \wedge e.\text{Materia} = m.\text{Materia}\}$$

La stessa interrogazione si esprime, con la notazione algebrica, come:

$$\pi_{\text{Matricola}}(\text{Studenti} \underset{\text{Matricola} = \text{Candidato}}{\bowtie} (\text{Esami} \bowtie \text{Materie}))$$

È in genere più facile scrivere ed interpretare una formula usando la notazione del calcolo, mentre la notazione algebrica è più adatta quando si desidera specificare non solo il risultato dell'interrogazione ma anche in che modo questo risultato può essere ottenuto. Per questo motivo, molti sistemi offrono ai programmatori un linguaggio basato sul calcolo, che poi traducono internamente in un formato algebrico al momento di stabilire come eseguire l'interrogazione.

Si può dire, in sintesi, che l'algebra dà la possibilità di scrivere espressioni in cui gli operatori sono applicati al risultato di altri operatori (espressioni annidate), mentre il calcolo ha una struttura piatta ma permette di esprimere condizioni più complesse. Un linguaggio che si colloca a metà tra i due stili si può ottenere in uno dei due modi seguenti:

1. aggiungendo al calcolo la possibilità di annidare il costruttore di insiemi, inserendo nella condizione ϕ predicati del tipo $t \in \{u \mid \Psi(u)\}$;
2. aggiungendo all'algebra la possibilità di avere nell'operatore di restrizione σ_ϕ condizioni ϕ che fanno uso anche di quantificatori e di predicati di appartenenza a relazioni.

Il risultato di una simile fusione tra i due stili è un linguaggio che ha sia la capacità di esprimere interrogazioni in modo annidato che la possibilità di esprimere condizioni logiche complesse, come accade, ad esempio, nel linguaggio SQL, che sarà illustrato nel Capitolo 6. Un linguaggio basato sul calcolo relazionale su ennuple è QUEL, usato dal sistema INGRES, mentre un linguaggio basato sul calcolo relazionale su domini è QBE, descritto brevemente nel Capitolo 6.

Un risultato importante della teoria relazionale consiste nel fatto che l'algebra relazionale ed il calcolo relazionale su ennuple e su domini hanno lo stesso potere espressivo, cioè permettono di scrivere le stesse interrogazioni.

4. Esiste un altro linguaggio, detto *calcolo relazionale su domini*, equivalente al *calcolo relazionale su ennuple* dal quale differisce per il fatto che le variabili della condizione ϕ assumono come valori elementi di domini invece che ennuple.

4.5 I linguaggi logici

I linguaggi relazionali logici sono linguaggi ispirati al linguaggio Prolog, nei quali un'interrogazione si esprime scrivendo un insieme di clausole Horn piatte. Una clausola Horn piatta è una formula con la seguente struttura:

$$\forall X_1, \ldots, X_n.P_1(t_{1,1}, \ldots, t_{1,n_1}), \ldots, P_m(t_{m,1}, \ldots, t_{m,n_m}) \rightarrow Q(t_{r,1}, \ldots, t_{r,n}),$$

dove ogni $t_{i,j}$ è una costante oppure una delle variabili X_i. Nel linguaggio Datalog, il più noto tra i linguaggi relazionali logici, la clausola si scrive con la seguente sintassi:

$$Q(t_{r,1}, \ldots, t_{r,n}) \text{ :- } P_1(t_{1,1}, \ldots, t_{1,n_1}), \ldots, P_m(t_{m,1}, \ldots, t_{m,n_m}),$$

dove ogni P_i è una relazione della base di dati, nel qual caso $P_i(t_{i,1}, \ldots, t_{i,n_i})$ significa $(t_{i,1}, \ldots, t_{i,n_i}) \in P_i$, oppure è un predicato definito da un'altra clausola Horn (o dalla stessa clausola, poiché questi linguaggi ammettono le definizioni ricorsive), o, ancora, è un predicato di confronto.

Questa clausola definisce una relazione Q che contiene il minimo insieme di ennuple che soddisfano la condizione specificata dalla clausola stessa.

Ad esempio, il seguente programma, formato da due clausole, definisce una relazione HannoSuperato che contiene tutti gli studenti che hanno superato una delle materie in Materie con un voto superiore al 27, oppure che hanno superato BDSI1 con 30.

```
HannoSuperato(Nome, Matricola, Provincia, Nascita)
    :– Studenti(Nome, Matricola, Provincia, Nascita),
       Esami(Materia, Candidato, Data, Voto),
       Matricola = Candidato, Voto > 27, Materie(Materia);

HannoSuperato(Nome, Matricola, Provincia, Nascita)
    :– Studenti(Nome, Matricola, Provincia, Nascita),
       Esami('BD', Candidato, Data, 30),
       Matricola = Candidato;
```

La relazione HannoSuperato può essere definita con la seguente espressione algebrica:

$$\begin{aligned}
&\text{HannoSuperato} = \\
&\quad \pi_{\text{Nome, Matricola, Provincia, Nascita}}(\\
&\qquad \sigma_{\text{Voto} > 27}(\text{Studenti} \underset{\text{Matricola=Candidato}}{\bowtie} (\text{Esami} \bowtie \text{Materie})) \\
&\qquad \cup\ \sigma_{\text{Voto = 30} \wedge \text{Materia= 'BD'}} (\text{Studenti} \underset{\text{Matricola = Candidato}}{\bowtie} \text{Esami})).
\end{aligned}$$

Generalizzando la tecnica usata in questo esempio è possibile trasformare ogni programma Datalog in un sistema di equazioni algebriche della forma

$$R_i = \pi_X(\sigma_\phi(R_{1,1} \bowtie \ldots \bowtie R_{1,n_1})) \cup \ldots \cup \pi_X(\sigma_\phi(R_{m,1} \bowtie \ldots \bowtie R_{m,n_m})).$$

Tuttavia, il linguaggio Datalog è più espressivo dell'algebra relazionale perché le clausole possono essere mutuamente ricorsive, come nell'esempio seguente, dove la relazione virtuale Antenato è definita in termini di una relazione Genitore, definita nella base di dati, ed in termini di se stessa:

```
Antenato(X,Z) :– Genitore(X,Y), Antenato(Y,Z)
Antenato(X,Z) :– Genitore(X,Z)
```

La relazione Antenato, che corrisponde alla chiusura transitiva della relazione Genitore, non è esprimibile nell'algebra e nel calcolo. D'altra parte il linguaggio Datalog puro non è in grado di esprimere gli operatori di differenza e di complemento, esprimibili nell'algebra e nel calcolo. Valgono tuttavia le seguenti equivalenze:

- il Datalog senza definizioni ricorsive ha lo stesso potere espressivo dell'algebra e del calcolo nelle loro versioni positive, ovvero prive di negazione, complemento e differenza;
- il Datalog senza definizioni ricorsive ma con l'aggiunta di opportuni operatori di negazione ha lo stesso potere espressivo dell'algebra e del calcolo;
- l'algebra ed il calcolo positivi, con l'aggiunta di opportuni operatori di punto fisso, hanno lo stesso potere espressivo del Datalog;
- l'algebra ed il calcolo, con l'aggiunta di opportuni operatori di punto fisso, hanno lo stesso potere espressivo del Datalog arricchito con un'opportuna forma di negazione.

Questo tipo di risultati permette di affermare che le differenze tra i diversi tipi di linguaggi relazionali riguardano in realtà lo stile di programmazione che questi supportano piuttosto che il loro potere espressivo.

4.6 Conclusioni

La semplicità concettuale del modello dei dati relazionale, e la completa indipendenza degli operatori da aspetti riguardanti l'organizzazione fisica dei dati, ha reso molto popolare questo approccio alla descrizione ed uso di basi di dati. Inoltre, il modello relazionale ha consentito lo sviluppo di semplici linguaggi di interrogazione, avvicinando così a questo tipo di sistemi anche utenti non esperti. Nei prossimi capitoli vedremo il linguaggio più noto: SQL.

Recentemente ci sono state diverse proposte di estensioni del modello relazionale per superare alcune limitazioni dei suoi meccanismi di astrazione. Le proposte si possono raggruppare in tre categorie.

- Proposte che estendono il modello relazionale dei dati per trattare non solo ennuple con attributi elementari (relazioni in *prima forma normale, 1NF*), ma anche ennuple con componenti strutturate o di tipo insieme, chiamate oggetti complessi (*modello relazionale annidato*). Sulle relazioni complesse sono disponibili opportuni operatori di tipo algebrico che sono una generalizzazione degli operatori dell'algebra relazionale classica [Abiteboul and Bidoit, 1984]. Del paradigma ad oggetti si adotta

quindi solo la possibilità di definire relazioni di ennuple con componenti strutturate, di tipo ennupla o relazione complessa, ma non la possibilità di condividere oggetti, di definire gerarchie o aspetti procedurali. Un'altra estensione più recente prevede anche la nozione di *identificatore di oggetto* per modellare la condivisione di elementi comuni.
- Proposte che estendono il modello dei dati con la possibilità di definire domini in modo calcolato o come tipi astratti, cioè corredati di operatori propri.
- Proposte che estendono il modello dei dati con entrambe le possibilità.

Esercizi

1. Elencare le differenze tra la nozione di ennupla nei sistemi relazionali e la nozione di oggetto nei sistemi ad oggetti.
2. Si traduca lo schema grafico ottenuto dall'Esercizio 2.6 in uno schema relazionale.
3. Si traduca lo schema grafico ottenuto dall'Esercizio 2.8 in uno schema relazionale.
4. Dimostrare le seguenti proprietà:

   a) $\boldsymbol{\sigma}_{\phi\wedge\psi}(E) = \boldsymbol{\sigma}_{\phi}(E) \cap \boldsymbol{\sigma}_{\psi}(E)$;
   b) $\boldsymbol{\sigma}_{\phi}(\boldsymbol{\sigma}_{\psi}(E)) = \boldsymbol{\sigma}_{\phi\wedge\psi}(E)$;
   c) $\boldsymbol{\sigma}_{\phi}(E_1 \cup E_2) = \boldsymbol{\sigma}_{\phi}(E_1) \cup \boldsymbol{\sigma}_{\phi}(E_2)$;
   d) $\boldsymbol{\pi}_{Y}(\boldsymbol{\pi}_{X}(E)) = \boldsymbol{\pi}_{Y}(E)$, se $Y \subseteq X$;
   e) $\boldsymbol{\pi}_{X}(\boldsymbol{\sigma}_{\phi}(E)) = \boldsymbol{\sigma}_{\phi}(\boldsymbol{\pi}_{X}(E))$, se ϕ usa solo attributi in X;
   f) $\boldsymbol{\sigma}_{\phi}(E_1 \underset{\boldsymbol{\Phi_J}}{\bowtie} E_2) = \boldsymbol{\sigma}_{\phi}(E_1) \underset{\boldsymbol{\Phi_J}}{\bowtie} E_2$, se ϕ usa solo attributi di E_1;
   g) $\boldsymbol{\sigma}_{\phi}({}_{A}\boldsymbol{\gamma}_{F}(E)) = {}_{A}\boldsymbol{\gamma}_{F}(\boldsymbol{\sigma}_{\phi}(E))$, se ϕ usa solo attributi in A.

5. Si consideri lo schema relazionale

   S(<u>M</u>, N, P, A)
   E(<u>C</u>, <u>S</u>, V)

   e l'espressione dell'algebra relazionale

   $$\boldsymbol{\sigma}_{P='c'\wedge S='s'}(\boldsymbol{\pi}_{N,P,S}(S \underset{\mathbf{M=C}}{\bowtie} E)).$$

   Si dia una rappresentazione ad albero dell'interrogazione e si mostri come si modifica l'albero dopo la riscrittura algebrica.
6. Si consideri lo schema relazionale dell'esercizio precedente e l'espressione dell'algebra relazionale

   $$\boldsymbol{\pi}_{N,S}(\boldsymbol{\sigma}_{A='a'\wedge S='s'}(\boldsymbol{\pi}_{N,M,A}(S) \underset{\mathbf{M=C}}{\bowtie} \boldsymbol{\pi}_{C,S}(E))).$$

   Si dia una rappresentazione ad albero dell'interrogazione e si mostri come si modifica l'albero dopo la riscrittura algebrica.

Note bibliografiche

Il modello e l'algebra relazionali furono introdotti in [Codd, 1970]. Una descrizione approfondita del modello relazionale è riportata in [Maier, 1983], [Ullman and Widom, 2001], [Silberschatz et al., 2002], [Atzeni and Antonellis, 1993], [Abiteboul et al., 1995]. Per un'analisi dei diversi tipi di linguaggi relazionali si veda [Ullman and Widom, 2001]. Per il Datalog si veda [Ceri et al., 1990]. Le estensioni del modello relazionale sono trattate in particolare in [Abiteboul et al., 1995]. Ogni libro di testo sulle basi di dati, comunque, dedica un ampio spazio al modello relazionale.

Capitolo 5

NORMALIZZAZIONE DI SCHEMI RELAZIONALI

Il modello dei dati relazionale adotta come unica struttura per la rappresentazione dei dati la relazione, prodotto cartesiano di domìni elementari. Questa struttura, se da una parte attira per la sua estrema semplicità, dall'altra pone problemi nell'uso in caso di situazioni complesse, in cui è necessario ricorrere a degli artifici per modellare, ad esempio, attributi composti o multivalore, associazioni molti a molti, classi incluse in altre.

In questi casi esistono in genere diverse rappresentazioni possibili della stessa situazione, per cui sorge il problema di verificare se: (*a*) queste diverse rappresentazioni sono tra di loro equivalenti; (*b*) queste rappresentazioni sono di buona qualità. In particolare, la qualità di una rappresentazione viene qui valutata come l'assenza di determinate *anomalie* che sono definite nel capitolo. La teoria della normalizzazione si occupa per l'appunto di definire criteri formali per giudicare l'equivalenza di schemi e la qualità di tali schemi, e di definire algoritmi per trasformare uno schema in un altro equivalente ma privo di anomalie.

Come discuteremo al termine del capitolo, i risultati della teoria sono di particolare interesse per un progettista che deve partire da uno schema relazionale esistente e migliorarlo. Al progettista che parte da uno schema ad oggetti privo di anomalie, la teoria sviluppata in questo capitolo dimostra che lo schema relazionale ottenuto applicando le regole del Capitolo 4 sarà anch'esso privo di anomalie.

5.1 Le anomalie

Per mostrare i problemi che si presentano nella definizione di schemi relazionali, si supponga di rappresentare con un'unica relazione i dati relativi a prestiti di una biblioteca:

Biblioteca(NomeUtente, Residenza, Telefono, NumeroLibro, Autore, Titolo, Data)

I libri, di cui esiste solo una copia, sono identificati da NumeroLibro e possono essere presi in prestito da utenti dei quali interessano il NomeUtente, che li identifica, la Residenza e il Telefono. Un utente può avere più libri in prestito contemporaneamente e di ogni prestito interessa memorizzare la data. La chiave della relazione è (NumeroLibro, NomeUtente). Un esempio di istanza della relazione è:

Biblioteca

NomeUtente	Residenza	Telefono	NumeroLibro	Autore	Titolo	DataPrestito
Rossi Carlo	Carrara	75444	XY188A	Boccaccio	Decameron	07/07/19
Paolicchi Luca	Avenza	59729	XY256B	Verga	Novelle	07/07/19
Pastine Maurizio	Dogana	66133	XY090C	Petrarca	Canzoniere	01/08/19
Paolicchi Laura	Avenza	59729	XY101A	Dante	Vita Nova	05/08/19
Paolicchi Luca	Avenza	59729	XY701B	Manzoni	Adelchi	14/01/20
Paolicchi Luca	Avenza	59729	XY008C	Moravia	La noia	17/08/20

Lo schema di relazione precedente presenta i seguenti inconvenienti o anomalie:

Ripetizione dell'informazione
Ogni volta che un utente prende in prestito un nuovo libro, vengono ripetuti la sua residenza e il suo numero telefonico; oltre allo spreco di spazio si complica l'aggiornamento della relazione, quando un utente cambia residenza o telefono.

Impossibilità di rappresentare certi fatti
Informazioni relative agli utenti della biblioteca possono essere memorizzate solo quando questi hanno un libro in prestito, non potendo lasciare l'attributo chiave NumeroLibro non specificata.

Anche la scelta di utilizzare più relazioni potrebbe portare a degli inconvenienti, come nel caso seguente, dove si è "decomposto" la relazione Biblioteca in due relazioni:

Utenti(NomeUtente, Residenza, Telefono)
Prestiti(NumeroLibro, Autore, Titolo, DataPrestito, Telefono)

In questo caso l'associazione fra utenti e prestiti è modellata attraverso il numero di telefono. I dati delle relazioni Utenti e Prestiti si ottengono per proiezione dalla relazione Biblioteca sui rispettivi attributi:

$\text{Utenti} = \pi_{\text{NomeUtente, Residenza,Telefono}}(\text{Biblioteca})$

Utenti

NomeUtente	Residenza	Telefono
Rossi Carlo	Carrara	75444
Paolicchi Luca	Avenza	59729
Pastine Maurizio	Dogana	66133
Paolicchi Laura	Avenza	59729

$\text{Prestiti} = \pi_{\text{NumeroLibro, Autore, Titolo, DataPrestito, Telefono}}(\text{Biblioteca})$

Prestiti

NumeroLibro	Autore	Titolo	DataPrestito	Telefono
XY188A	Boccaccio	Decameron	07/07/19	75444
XY256B	Verga	Novelle	07/07/19	59729
XY090C	Petrarca	Canzoniere	01/08/19	66133
XY101A	Dante	Vita Nova	05/08/19	59729
XY701B	Manzoni	Adelchi	14/01/20	59729
XY008C	Moravia	La Noia	17/08/20	59729

Questa decomposizione permette di eliminare la duplicazione dei dati, ma presenta nuovi problemi. L'esempio seguente mostra il risultato della ricerca degli utenti che hanno prestiti a Gennaio 2020:

$\pi_{\text{NomeUtente, Residenza}}(\text{Utenti} \bowtie \sigma_{\text{DataPrestito} \in [01/01/20, 31/01/20]}(\text{Prestiti}))$

NomeUtente	Residenza
Paolicchi Luca	Avenza
Paolicchi Laura	Avenza

Questa relazione è errata, perché Paolicchi Laura non ha preso in prestito un libro a Gennaio 2020. Infatti, la giunzione fra Utenti e Prestiti contiene più ennuple della relazione originale Biblioteca. Una decomposizione che presenta questa anomalia è detta *con perdita di informazione* (*lossy decomposition*). Il motivo di questa perdita è la scelta di rappresentare l'associazione fra Utenti e Prestiti usando come chiave esterna il numero di telefono, che non identifica univocamente gli utenti. Una decomposizione senza perdita di informazione è invece la seguente:

Utenti(NomeUtente, Residenza, Telefono)
Prestiti(NumeroLibro, Autore, Titolo, Data, NomeUtente*)

È lecito a questo punto chiedersi se questa decomposizione non introduca nuovi problemi o svantaggi. Ad esempio, volendo reperire la residenza dell'utente che ha in prestito un certo libro occorre fare una giunzione delle due relazioni, non necessaria nel caso della relazione unica. Per questi motivi, il problema di valutare se uno schema è "migliore" di un altro non è banale, in particolare se si vuol tener conto anche del costo delle operazioni.

Lo scopo principale della teoria della normalizzazione è quello di fornire strumenti formali per la progettazione di basi di dati che non presentino anomalie del tipo precedentemente mostrato, senza prendere in considerazione il costo delle operazioni. In particolare, nell'attività di progettazione, si parte da un qualche schema che modella la realtà di interesse e si cerca di ottenere uno schema che, in base a certi criteri, sia "migliore" di quello di partenza, ma "equivalente" ad esso, nel senso che contenga la stessa informazione. Per questo motivo si parla anche di *analisi di schemi* anziché di *progettazione di schemi*. Quindi, in sostanza, la teoria della normalizzazione si occupa dei seguenti problemi:

- definire quando due schemi sono equivalenti;
- definire criteri di bontà per schemi (cosa vuol dire che uno schema è migliore di un altro);
- trovare metodi algoritmici per ottenere da uno schema uno schema migliore ed equivalente.

Il primo problema è quello che in letteratura va sotto il nome di *problema della rappresentazione*, ossia quando e in che misura si può dire che uno schema *rappresenta* un altro. La formalizzazione del secondo problema invece ha portato, come vedremo, alla definizione di *forme normali* per schemi di relazioni, con caratteristiche desiderabili dal punto di vista della minimizzazione della ridondanza ed eliminazione delle anomalie. Per questo motivo l'attività di progettazione è anche detta *normalizzazione*, cioè riduzione a schemi in forma normale.

Prima di procedere nella descrizione della teoria della normalizzazione, è necessario discutere un assunto che ne è alla base: l'ipotesi che le informazioni da memorizzare in una base di dati siano descrivibili da uno *schema di relazione universale*.

Definizione 5.1

Lo *schema di relazione universale* U di una base di dati relazionale ha come attributi l'unione degli attributi di tutte le relazioni della base di dati.

Questa ipotesi comporta che tutti gli attributi con lo stesso nome in relazioni diverse abbiano lo *stesso* significato e quindi, in particolare, siano definiti sullo *stesso* dominio. Ad esempio, se vogliamo riferirci alle date di nascita e alle date di assunzione degli impiegati di una ditta, non possiamo usare due attributi con nome Data, neanche se sono in relazioni diverse, ma dovremo usare due nomi diversi (ad esempio, DataNascita e DataAssunzione). Questa ipotesi semplifica la trattazione della teoria perché permette di evitare operazioni di ridenominazione degli attributi, e permette di utilizzare sempre la giunzione naturale per collegare le ennuple correlate in due relazioni.

Nel seguito, verranno usate le seguenti notazioni:

- A, B, C, A_1, A_2 ecc. indicano singoli attributi.
- T, X, Y, X_1 ecc. indicano insiemi di attributi.
- XY è un'abbreviazione per $X \cup Y$, AB è un'abbreviazione per $\{A, B\}$, $A_1A_2 \ldots A_n$ è un'abbreviazione per $\{A_1, A_2, \ldots, A_n\}$, e XA è un'abbreviazione per $X \cup \{A\}$.
- $R(T)$ è uno schema di relazione, r una sua generica istanza e t è un'ennupla di r. Se $X \subseteq T$, allora $t[X]$ indica l'ennupla ottenuta da t considerando solo gli attributi in X.

5.2 Dipendenze funzionali

5.2.1 Definizione

Come mostrato nella sezione precedente, le anomalie che si incontrano in certi schemi relazionali provengono da una rappresentazione impropria dei fatti dell'universo del discorso. Per formalizzare la nozione di schema senza anomalie, occorre quindi, per prima cosa, introdurre nella teoria un modo per rappresentare formalmente informazioni sulle proprietà dei fatti che si modellano. Codd, che ha affrontato per primo questo problema, ha proposto a questo scopo un formalismo basato sulla nozione di *dipendenza fra dati* [Codd, 1970]. Il primo tipo di dipendenza che si considera, chiamato *dipendenza funzionale*, formalizza la nozione di valore di un attributo determinato funzionalmente dal valore di altri.

Definizione 5.2
Dato uno schema di relazione $R(T)$, una *dipendenza funzionale* fra due sottoinsiemi di attributi X e Y di T è un vincolo d'integrità sulle istanze della relazione, espresso nella forma $X \to Y$, con il seguente significato: un'istanza r di $R(T)$ *soddisfa* la dipendenza $X \to Y$, o $X \to Y$ *vale* in r, se per ogni coppia di ennuple t_1 e t_2 di r, se $t_1[X] = t_2[X]$ allora $t_1[Y] = t_2[Y]$, ovvero

$$X \to Y \Leftrightarrow \forall t_1, t_2 \in r, t_1[X] = t_2[X] \Rightarrow t_1[Y] = t_2[Y]^1 \tag{5.1}$$

In altre parole, una dipendenza funzionale $X \to Y$ è un vincolo di integrità che specifica che, per ogni istanza valida r di $R(T)$, $\pi_{XY}(r)$ è una *funzione* con dominio X e codominio Y.

Quando su $R(T)$ è definita la dipendenza funzionale $X \to Y$, si dice che X *determina* Y, o che Y *è determinato* da X. Se F è un insieme di dipendenze funzionali su $R(T)$, si dice che r è un'istanza *valida* di $R(T)$, rispetto ad F, se per ogni $X \to Y \in F$ vale la 5.1.

È importante notare due aspetti significativi relativi alle dipendenze funzionali:

1. Notare che gli insiemi X ed Y possono avere elementi in comune.

– esse sono definite solo all'interno di *uno* schema di relazione, e non possono esistere, quindi, fra attributi appartenenti a relazioni diverse;
– sono proprietà intensionali, legate al significato dei fatti che si rappresentano; non è quindi possibile inferirle dall'osservazione di alcune istanze della relazione.

Esempio 5.1
Nello schema:

Biblioteca(NomeUtente, Residenza, Telefono, NumeroLibro, Autore, Titolo, Data)

valgono le seguenti dipendenze funzionali:

NomeUtente $\rightarrow$ Residenza, Telefono
NumeroLibro $\rightarrow$ Autore, Titolo
NumeroLibro $\rightarrow$ NomeUtente, Data

che sono rispettate nell'esempio d'istanza visto nella sezione precedente.

In conclusione, per specificare il significato dei fatti rappresentati si usano le dipendenze funzionali, che vengono utilizzate per verificare l'eventuale presenza di anomalie nel progetto e, se questo è il caso, per normalizzare lo schema. Data la loro importanza per la progettazione di uno schema relazionale, le dipendenze funzionali si considerano facenti parte dello schema di una relazione, che d'ora in poi si indicherà, in generale, con $R\langle T, F\rangle$, con F insieme di dipendenze funzionali su T.

5.2.2 Dipendenze derivate

Dato uno schema di relazione $R\langle T, F\rangle$, è facile convincersi che le sue istanze valide soddisfano non solo le dipendenze espresse in F, ma anche altre derivabili da esse. Ad esempio, dato $R\langle T, \{X \rightarrow Y, X \rightarrow Z\}\rangle$, con $X, Y, Z \subseteq T$, e $W \subseteq X$, si può notare che in tutte le sue istanze valide valgono anche le dipendenze $X \rightarrow W$ e $X \rightarrow YZ$. Nel primo caso, infatti, se due ennuple coincidono su X coincideranno a maggior ragione su W che è un sottoinsieme di X.[2] Nel secondo caso se $e_1[X] = e_2[X]$, poiché e_1, e_2 soddisfano le dipendenze in F, si avrà che $e_1[Y] = e_2[Y]$ e $e_1[Z] = e_2[Z]$, e quindi $e_1[YZ] = e_2[YZ]$. L'implicazione di un insieme di dipendenze da altre viene definita nel modo seguente:

Definizione 5.3
Dato $R\langle T\rangle$ e dato F, diciamo che $F \models X \rightarrow Y$ (F *implica logicamente* $X \rightarrow Y$), se ogni istanza r di $R\langle T\rangle$ che soddisfa F soddisfa anche $X \rightarrow Y$.

2. Le dipendenze $X \rightarrow W$, con $W \subseteq X$, sono dette *dipendenze banali*.

In base a questa definizione, per lo schema precedente valgono le seguenti implicazioni logiche:

$\{X \to Y, X \to Z\} \models X \to YZ$

e

$\{\} \models X \to X$

La definizione precedente non fornisce un modo algoritmico per trovare le dipendenze funzionali implicate da un insieme F: per questo occorre un'*assiomatizzazione* delle dipendenze funzionali che fornisca un insieme *corretto* e *completo* di regole di inferenza, chiamate anche *assiomi*, che possono essere usate per derivare nuove dipendenze da un dato insieme di dipendenze.

Definizione 5.4
Sia RI un insieme di regole di inferenza per F. Indichiamo con $F \vdash X \to Y$ il fatto che $X \to Y$ sia derivabile da F usando RI.
L'insieme RI è *corretto* se: $F \vdash X \to Y \Rightarrow F \models X \to Y$.
L'insieme RI è *completo* se: $F \models X \to Y \Rightarrow F \vdash X \to Y$.

Il più noto insieme corretto e completo di regole di inferenza per le dipendenze funzionali è il seguente (*assiomi di Armstrong*) [Armstrong, 1974]:

Riflessività: se $Y \subseteq X$, allora $X \to Y$.
Arricchimento: se $X \to Y$ e $W \subseteq T$, allora $XW \to YW$.
Transitività: se $X \to Y$ e $Y \to Z$, allora $X \to Z$.

Possiamo adesso definire formalmente il concetto di *derivazione* di una dipendenza.

Definizione 5.5
Una *derivazione* di f da F è una sequenza finita $f_1, \ldots, f_m$ di dipendenze, dove $f_m = f$ e ogni f_i è un elemento di F oppure è ottenuta dalle precedenti dipendenze $f_1, \ldots, f_{i-1}$ della derivazione usando una regola di inferenza.

Si noti che una sottosequenza $f_1, \ldots, f_k$ di una derivazione $f_1, \ldots, f_m$ è anche una derivazione, quindi $F \vdash f_k$ per ogni $k = 1, \ldots, m$.
Alcune regole comunemente usate, derivabili a partire dagli assiomi di Armstrong, sono:

Unione: $\{X \to Y, X \to Z\} \vdash X \to YZ$.
Decomposizione: $\{X \to YZ\} \vdash X \to Y$.
Indebolimento: $\{X \to Y\} \vdash XZ \to Y$.
Identità: $\{\} \vdash X \to X$.

Dimostriamo la prima regola, lasciando al lettore la dimostrazione delle successive.

Teorema 5.1
$\{X \to Y, X \to Z\} \vdash X \to YZ$

Dimostrazione

1.	$X \to Y$	(per ipotesi)
2.	$X \to XY$	(per arricchimento da 1)
3.	$X \to Z$	(per ipotesi)
4.	$XY \to YZ$	(per arricchimento da 3)
5.	$X \to YZ$	(per transitività da 2, 4) ■

Introduciamo ora la nozione di *chiusura* di un insieme di attributi rispetto ad un insieme di dipendenze funzionali.

Definizione 5.6
Dato uno schema $R\langle T, F\rangle$ con $X \subseteq T$, la *chiusura* di X rispetto a F è data da $\{A \in T \mid F \vdash X \to A\}$, e si indica con X^+_F, o più semplicemente con X^+ se non vi sono ambiguità.

Teorema 5.2
$F \vdash X \to Y \Leftrightarrow Y \subseteq X^+$

Dimostrazione
($\Rightarrow$) Sia $Y = A_1 \dots A_k$, $A_i \in T$. Per la regola di decomposizione si ha che

$F \vdash X \to A_i, 1 \leqslant i \leqslant k$

quindi, in base alla definizione di X^+,

$A_i \in X^+, 1 \leqslant i \leqslant k$

e quindi anche $Y \subseteq X^+$.
($\Leftarrow$) Secondo la definizione di X^+,

$F \vdash X \to A_i,\ 1 \leqslant i \leqslant k$

e dalla regola di unione

$F \vdash X \to A_1, \dots, A_k$ cioè $F \vdash X \to Y$ ■

Il prossimo teorema dimostra l'equivalenza della nozione di implicazione logica ($\models$) e di quella di derivazione ($\vdash$) nel caso degli assiomi di Armstrong. Questo teorema stabilisce, in particolare, che se una dipendenza è derivabile con gli assiomi di Armstrong allora è anche implicata logicamente (correttezza degli assiomi), e viceversa se una dipendenza è implicata logicamente allora è anche derivabile dagli assiomi (completezza degli assiomi).

Teorema 5.3
Gli assiomi di Armstrong sono corretti e completi.

Dimostrazione
Supponiamo di avere un insieme di dipendenze funzionali F su T, e una dipendenza $X \to Y$.
Correttezza. Si deve dimostrare che se $F \vdash X \to Y$ allora $F \models X \to Y$. Si procede per induzione sulla lunghezza della derivazione. Sia $f_1, \ldots, f_m$ la derivazione di $X \to Y$ da F, e supponiamo che il teorema valga per tutte le derivazioni più corte. La dipendenza $f_m = X \to Y$ è un elemento di F, oppure è stata derivata usando un assioma di Armstrong. Nel primo caso è implicata logicamente in maniera banale. Supponiamo che f_m sia stata inferita con la regola riflessiva, allora $Y \subseteq X$. Quindi, f_m è implicata logicamente in maniera banale.
Supponiamo che f_m sia stata ottenuta da f_i usando la regola di arricchimento, allora f_i ha la forma $X' \to Y'$, con X' e Y' tali che $X = X'Z$ e $Y = Y'Z$ per qualche Z. Per l'ipotesi induttiva $F \models f_i$. Siano t e t' due ennuple con $t[X'Z] = t'[X'Z]$. Allora $t[X'] = t'[X']$, quindi, per f_i, $t[Y'] = t'[Y']$, per cui $t[Y'Z] = t'[Y'Z]$. Quindi f_m è implicata logicamente da F.
Infine, supponiamo che f_m sia stata ottenuta da $f_i = X \to W$ e $f_j = W \to Y$, per qualche W, usando la regola transitiva. Per l'ipotesi induttiva $F \models f_i$ e $F \models f_j$. Siano t e t' due ennuple con $t[X] = t'[X]$; per f_i, $t[W] = t'[W]$ e quindi, per f_j, $t[Y] = t'[Y]$. Dunque, $X \to Y$ è implicata logicamente da F.
Completezza. Si deve dimostrare che se $F \models X \to Y$, allora $F \vdash X \to Y$.
Si consideri una relazione r su T di due ennuple costruita come segue: $r = \{t, t'\}$ con $t[X^+] = t'[X^+]$ e $t[A] \neq t'[A]$ per ogni $A \in T - X^+$; r soddisfa F. Infatti, sia $V \to W$ una dipendenza di F. Se $V \not\subseteq X^+$, allora $t[V] \neq t'[V]$, ed r soddisfa ovviamente la dipendenza. Se invece $V \subseteq X^+$, si ha che $F \vdash X \to V$ e poiché $V \to W$, per transitività, $F \vdash X \to W$, da cui $W \subseteq X^+$ e quindi $t[W] = t'[W]$. Pertanto $V \to W$ è soddisfatta in r.
Mostriamo ora la tesi, cioè $F \models X \to Y \Rightarrow F \vdash X \to Y$. Da $F \models X \to Y$ segue che r soddisfa $X \to Y$. Poiché $X \subseteq X^+$, e $t[X^+] = t'[X^+]$, segue che $t[Y] = t'[Y]$ per cui $Y \subseteq X^+$ e $F \vdash X \to Y$ per il Teorema 5.2. ■

Il Teorema 5.3 permette di sostituire, nella trattazione precedente, tutte le occorrenze di $\models$ con $\vdash$ e viceversa; in particolare, si ha che $X^+ = \{A \in T \mid F \models X \to A\}$.

È infine interessante osservare che le regole di Armstrong sono indipendenti fra di loro (cioè nessuna di esse può essere derivata dalle altre due), e che non costituiscono l'unico insieme di regole con le proprietà di correttezza e completezza.

5.2.3 Chiusura di un insieme di dipendenze funzionali

L'esistenza di regole di inferenza per derivare nuove dipendenze funzionali da altre ci consente di definire la nozione di *chiusura* di un insieme di dipendenze funzionali.

Definizione 5.7
Dato un insieme F di dipendenze funzionali, si definisce la *chiusura* di F come $F^+ = \{X \to Y \mid F \vdash X \to Y\}$.

Un problema che si presenta spesso è quello di decidere se una dipendenza funzionale appartiene a F^+ (*problema dell'implicazione*); la sua risoluzione con l'algoritmo banale di generare F^+ applicando ad F ripetutamente ed esaustivamente gli assiomi di Armstrong ha una complessità esponenziale rispetto al numero di attributi dello schema. Infatti, se a è la dimensione di T, F^+ contiene almeno le $2^a - 1$ dipendenze funzionali banali $T \to X$, dove X è un sottoinsieme non vuoto di T.

Un metodo di complessità inferiore per risolvere il problema dell'implicazione scaturisce invece dalla seguente considerazione: per decidere se $X \to Y \in F^+$, si può controllare se $Y \subseteq X^+$. Questa riformulazione del problema è resa possibile dalla validità del Teorema 5.2.

Un semplice algoritmo per calcolare X^+ è il seguente:

Algoritmo 5.1
Chiusura lenta

input $R\langle T, F\rangle$, $X \subseteq T$
output *XPIU*
begin
  XPIU := X
  while (*XPIU* è cambiato) **do**
    for each $W \to V \in F$ **with** $W \subseteq$ *XPIU* **and** $V \not\subseteq$ *XPIU* **do**
      XPIU := *XPIU* $\cup V$
end

Esempio 5.2
Sia $R\langle T, F\rangle$ con $T = \{A, B, C, D, E, G, H, I\}$ ed $F = \{ADG \to GI, ACH \to ADG, BC \to AD, CE \to ACH\}$. Indicando con $XPIU^j$ il valore della variabile *XPIU* alla fine della j-esima ripetizione del ciclo **while**, la chiusura di BCE è calcolata come segue:

1. $XPIU^0 = BCE$;
2. $XPIU^1 = BCE \cup AD \cup ACH = ABCDEH$;
3. $XPIU^2 = ABCDEH \cup ADG = ABCDEGH$;
4. $XPIU^3 = ABCDEGH \cup GI = ABCDEGHI = T$.

A questo punto l'algoritmo termina con $(BCE)^+ = T$ e quindi BCE è una superchiave.

Teorema 5.4
L'algoritmo **Chiusura lenta** è corretto.

Dimostrazione
Terminazione. Ad ogni iterazione si aggiunge qualche attributo a $XPIU$ oppure si termina. Poiché il numero degli attributi è finito segue che l'algoritmo non può ciclare indefinitamente.
Correttezza. Per dimostrare che quando l'algoritmo termina $XPIU = X^+$, si dimostra che $XPIU \subseteq X^+$ e $X^+ \subseteq XPIU$.
Per dimostrare che $XPIU \subseteq X^+$ è sufficiente provare per induzione sul numero di iterazioni j che $XPIU^j \subseteq X^+$, per ogni j. Per $j = 0$ l'affermazione è ovvia essendo $XPIU^0 = X \subseteq X^+$ per riflessività. Per l'ipotesi induttiva, sia $XPIU^j \subseteq X^+$ e proviamo che $XPIU^{j+1} \subseteq X^+$. Sia A un attributo aggiunto alla $j+1$-esima iterazione. Per come è stato definito l'algoritmo si ha che $A \in V$, $W \to V \in F$, $W \subseteq XPIU^j$. Per l'ipotesi induttiva $W \subseteq X^+$, e dal Teorema 5.2, $X \to W$; per transitività $X \to V$ e quindi $X \to A$. Pertanto $A \in X^+$.
Dimostriamo ora che $X^+ \subseteq XPIU$. Si noti innanzitutto che se $V \subseteq W$, allora $V^+ \subseteq W^+$. Pertanto, poiché $X \subseteq XPIU$, $X^+ \subseteq XPIU^+$. Basta allora dimostrare che $XPIU^+ \subseteq XPIU$, ovvero che se $A \notin XPIU$ allora $A \notin XPIU^+$, cioè esiste un'istanza valida r di R che non soddisfa $XPIU \to A$. Si costruisce una relazione $r = \{t, t'\}$ di due ennuple distinte che dipendono dal risultato dell'algoritmo ponendo $t[A] = t'[A]$ per ogni $A \in XPIU$ e $t[B] \neq t'[B]$ per ogni $B \in T - XPIU$. Supponiamo di aver dimostrato che r soddisfa ogni dipendenza in F; si dimostra che r non soddisfa $XPIU \to A$: t e t' coincidono su $XPIU$ per costruzione, mentre non coincidono su A perché per ipotesi $A \notin XPIU$.
Rimane da provare che r soddisfa ogni dipendenza in F. Sia $W \to V$ una dipendenza di F. Se $W \not\subseteq XPIU$, allora $t[W] \neq t'[W]$, ed r soddisfa ovviamente la dipendenza. Se $W \subseteq XPIU$, si ha che $V \subseteq XPIU$ poiché l'algoritmo termina con $XPIU$. Ma $V \subseteq XPIU$ implica $t[V] = t'[V]$ e quindi r soddisfa la dipendenza. ■

Sia a il numero degli attributi di T e p il numero delle dipendenze funzionali in F. Il ciclo **while** viene eseguito al più p volte (perché ogni dipendenza funzionale dà il suo contributo alla chiusura al più una volta) e al più a volte (perché si possono aggiungere al più a attributi). Per ogni ciclo **while**, il ciclo interno viene eseguito al più p volte e ogni volta si controllano due inclusioni di insiemi. Poiché il test di contenuto tra insiemi ordinati di a elementi ha una complessità di tempo $O(a)$, si conclude che l'algoritmo **Chiusura lenta** ha complessità di tempo, nel caso peggiore, di $O(ap \min\{a, p\})$.

L'algoritmo **Chiusura lenta** ha il pregio della semplicità, ma adotta una tecnica molto inefficiente per verificare, per ogni iterazione ed ogni dipendenza, se la dipendenza può essere usata per aggiungere attributi ad X^+. Questa verifica può essere resa molto più efficiente con l'impiego di opportune strutture dati, ottenendo un algoritmo con complessità di tempo di $O(ap)$ [Beeri and Bernstein, 1979].

L'algoritmo **Chiusura veloce** ha un ruolo fondamentale nelle applicazioni della teoria della normalizzazione, perché il calcolo della chiusura di un insieme di attributi è richiesto in molti altri problemi, come vedremo nel seguito. Inoltre, con un'opportuna rappresentazione dei dati si può mostrare come l'algoritmo **Chiusura veloce** possa essere usato per risolvere altri problemi ben noti, come il *problema della raggiungibilità* in un grafo diretto $G = (V, E)$ (dato un nodo del grafo, trovare tutti gli altri nodi raggiungibili da esso). Si pone $T = V$ e $F = \{A \to B | A, B \in V \wedge (A, B) \in E\}$. Dato un nodo $v \in V$, la chiusura di $\{v\}$ contiene i nodi raggiungibili.

5.2.4 Chiavi

Usando le nozioni di dipendenza funzionale e di chiusura di un insieme di dipendenze, si possono definire in modo formale le nozioni di *superchiave*, *chiave* e *attributo primo* di uno schema.

Definizione 5.8
Dato uno schema $R\langle T, F\rangle$, un insieme di attributi $W \subseteq T$ è una *superchiave* di R se $W \to T \in F^+$.

Definizione 5.9
Dato uno schema $R\langle T, F\rangle$, un insieme di attributi $W \subseteq T$ è una *chiave* di R se W è una superchiave e non esiste un sottoinsieme stretto di W che sia una superchiave di R.

Definizione 5.10
Dato uno schema $R\langle T, F\rangle$, un attributo $A \in T$ si dice *primo* se e solo se appartiene ad almeno una chiave, altrimenti si dice *non primo*.

Dato che in uno schema ci possono essere più chiavi, di solito ne viene scelta una (generalmente con il minimo numero di attributi), la *chiave primaria*, come identificatore delle ennuple delle istanze dello schema.

In generale, il problema di trovare tutte le chiavi di uno schema richiede un algoritmo di complessità esponenziale nel caso peggiore e il problema di sapere se un attributo è primo è NP-completo [Lucchesi and Osborn, 1978].

Come trovare tutte le chiavi

Vedremo più avanti come in alcuni casi occorra stabilire se un attributo di uno schema $R\langle T, F\rangle$ sia primo e, quindi, trovare tutte le chiavi di R.

Osserviamo per prima cosa che valgono le seguenti proprietà:

1. Se un attributo A di T non appare a destra di alcuna dipendenza in F, allora A appartiene ad ogni chiave di R (si veda l'Esercizio 4).

2. Se un attributo A di T appare a destra di qualche dipendenza in F, ma non appare a sinistra di alcuna dipendenza non banale, allora A non appartiene ad alcuna chiave.

Sia X l'insieme degli attributi che non appaiono a destra di alcuna dipendenza in F. Dalla proprietà (1), segue che se $X^+ = T$, allora X è una chiave di R ed è anche l'unica possibile.

Per esempio, sia $R\langle T, F\rangle$ con $T = \{A, B, C, D, E, G\}$ ed $F = \{BC \to AD, D \to E, CG \to B\}$. C e G non appaiono a destra delle dipendenze, quindi devono far parte di ogni chiave. Poiché $CG^+ = T$, CG è l'unica chiave di R.

Se invece $X^+ \neq T$, allora occorre aggiungere a X altri attributi. Per la proprietà (2), basta aggiungere quegli attributi W di T che appaiono a destra di qualche dipendenza e a sinistra di qualche altra, uno alla volta. Ad ogni passo occorre evitare di aggiungere attributi che siano già nella chiusura di X, poiché tali attributi sono chiaramente ridondanti, oppure attributi che producono un insieme X' che contiene una chiave trovata in precedenza. Poi si calcola la chiusura di ogni X', fino a che questa non coincide con T, il che garantisce che X' sia una chiave.

Per esempio, sia $R\langle T, F\rangle$ con $T = \{A, B, C, D, E, G\}$ ed $F = \{AB \to C, BC \to AD, D \to E, CG \to B\}$.

G non appare a destra delle dipendenza e $G^+ = G$. Aggiungendo un attributo di $W = \{A, B, C, D\}$ a G si trova che

$GA^+ = GA \neq T$.
$GB^+ = GB \neq T$.
$GC^+ = T$. GC è una chiave di R.
$GD^+ = GDE \neq T$.

Si prova ad aggiungere ad GA, GB e GD un altro attributo di W, considerando solo insiemi di attributi che non contengono la chiave GC, e si trova che

$GAB^+ = T$. GAB è una chiave di R.
$GAD^+ = GADE \neq T$.
$GBD^+ = GBDE \neq T$.

Si prova infine ad aggiungere ad GAD e GBD un altro attributo di W, ma non si trovano insiemi di attributi che non contengono le chiavi GC e GAB, e quindi si conclude che non ne esistono altre.

In generale, una soluzione si trova con il seguente algoritmo che analizza tutti i "candidati", ovvero i sottinsiemi di T che potrebbero essere chiavi.

Algoritmo 5.2
Trova tutte le chiavi

input $R\langle T, F\rangle$ *output* *Chiavi* l'insieme di tutte le chiavi di $R\langle T, F\rangle$

```
begin
    NoDes := T − ∪X→A∈F A;
    SinDes := ∪X→A∈F X ∩ ∪X→A∈F A;
    Candidati := [NoDes::(SinDes)];
    Chiavi := [ ];
    while (Candidati non vuoto) do
        begin
            X::(Y) := first(Candidati);
            Candidati := rest(Candidati);
            if not some K in Chiavi with K ⊂ X
            then if X+ = T then Chiavi := Chiavi + X;
                    else begin
                            A1 ... An := Y − X+;
                            for i in 1..n do Candidati := Candidati
                                                    append [XAi::(Ai+1 ... An)]
                         end
        end
end
```

L'algoritmo memorizza in *Chiavi* le chiavi già trovate e nella lista *Candidati* i candidati ancora da analizzare.[3] Ogni candidato è un insieme di sottoinsiemi di T rappresentato in una forma compatta $X::(Y)$, che denota tutti gli insiemi formati dagli attributi X uniti ad un qualsiasi sottoinsieme degli attributi Y. Ad esempio, $AB::(CD)$ rappresenta $\{AB, ABC, ABD, ABCD\}$. Se *NoDes* sono gli attributi che non appaiono a destra di nessuna dipendenza e *SinDes* sono quelli che appaiono sia a sinistra che a destra, per le osservazioni precedenti tutte le chiavi appartengono all'insieme $NoDes::(SinDes)$, per cui inizialmente $Candidati = [NoDes::(SinDes)]$.

Gli insiemi in $X::(Y)$ sono analizzati a partire da X. Se X è chiave, allora tutti gli altri insiemi in $X::(Y)$ sono scartati. Altrimenti, poiché gli elementi di X^+ non possono apparire in una chiave che contiene X, dato $Y - X^+ = \{A_1 \dots A_n\}$, si mettono in *Candidati* i nuovi candidati $XA_1::(A_2, \dots, A_n)$, $XA_2::(A_3, \dots, A_n)$, $\dots$, $XA_n::()$, i quali coprono $(X::(A_1 \dots A_n)) - \{X\}$.

Il test $X^+ = T$ assicura che X è superchiave. Per essere certi che X sia anche chiave, si controlla che X non contenga chiavi già trovate in precedenza. Le chiavi trovate in seguito invece non potranno mai essere contenute strettamente in X perché tutti i

3. Su una lista $L = [x_1, \dots, x_n]$ si usano i seguenti operatori: first(L) che ritorna il primo elemento di una lista non vuota L; rest(L) che ritorna la lista non vuota L priva del primo elemento; $L + x_i$ che ritorna una lista i cui primi elementi sono quelli di L e l'ultimo x_i; L_1 append L_2 che ritorna una lista i cui primi elementi sono quelli di L_1 ed i successivi quelli di L_2.

candidati analizzati dopo X avranno lunghezza maggiore o uguale. Questa invariante è assicurata mantenendo *Candidati* ordinata per lunghezze crescenti, inserendo i nuovi candidati sempre in coda alla lista, con l'operatore **append**.

Esemplifichiamo l'applicazione dell'algoritmo su $R\langle T, F\rangle$ con $T = \{A, B, C, D, E\}$ ed $F = \{AC \to B,\ C \to D,\ D \to E,\ ABD \to C,\ B \to E\}$, mostrando l'effetto su *Chiavi* e *Candidati* di ciascuna esecuzione del ciclo **while**, con le variabili inizializzate a

$$NoDes = A;\ Chiavi = [\,];\ SinDes = BCD;\ Candidati = [A::(BCD)]$$

1. X::(Y) := first(*Candidati*) $= A::(BCD)$;
   Candidati := rest(*Candidati*) = [ ]
   $X^+ = A^+ = A \ \Rightarrow\ A$ non chiave
   $Y - X^+ = BCD - A = BCD$ e quindi
   $Candidati := [\,] + AB::(CD) + AC::(D) + AD::()$
2. X::(Y) := first(*Candidati*) $= AB::(CD)$
   Candidati := rest(*Candidati*) = $[AC::(D), AD::()]$
   $X^+ = AB^+ = ABE \ \Rightarrow\ AB$ non chiave
   $Y - X^+ = CD - ABE = CD$ e quindi
   $Candidati := [AC::(D), AD::()] + ABC::(D) + ABD::()$
3. X::(Y) := first(*Candidati*) $= AC::(D)$
   Candidati := rest(*Candidati*) = $[AD::(), ABC::(D), ABD::()]$
   $X^+ = AC^+ = ACBDE \ \Rightarrow\ AC$ chiave
   $Chiavi := [AC]$;
4. X::(Y) := first(*Candidati*) $= AD::()$
   Candidati := rest(*Candidati*) = $[ABC::(D), ABD::()]$
   $X^+ = AD^+ = ADE \ \Rightarrow\ AD$ non chiave
   $Y = ()$, per cui non si genera nessun nuovo candidato.
5. X::(Y) := first(*Candidati*) $= ABC::(D)$
   Candidati := rest(*Candidati*) = $[ABD::()][\,]$
   esiste $AC \in Chiavi$ con $AC \subset ABC$, per cui $ABC::(D)$ è scartato
6. X::(Y) := first(*Candidati*) $= ABD::()$
   Candidati := rest(*Candidati*) = $[\,]$
   $X^+ = ABD^+ = ABDEC \ \Rightarrow\ ABD$ chiave
   $Chiavi := [AC, ABD]$;

   Candidati è vuoto, per cui l'algoritmo si ferma e restituisce le due chiavi trovate.

5.2.5 Copertura di un insieme di dipendenze

Per operare su insiemi di dipendenze, è comodo portarli in una forma "minima". Per arrivare ad una definizione formale di minimalità, si introduce per prima cosa il concetto di *copertura*.

Definizione 5.11

Due insiemi di dipendenze F e G sugli attributi T di una relazione R sono *equivalenti*, ($F \equiv G$), se e solo se $F^+ = G^+$. Se $F \equiv G$ allora F è una *copertura* di G e viceversa.

La relazione di equivalenza fra insiemi di dipendenze permette di stabilire quando due schemi di relazione rappresentano gli stessi fatti: basta controllare se gli attributi sono gli stessi e le dipendenze equivalenti. Per controllare l'equivalenza delle dipendenze è sufficiente controllare che tutte le dipendenze di F appartengano a G^+ e che tutte quelle di G appartengano a F^+.

La definizione seguente ci fornisce il tipo di copertura di insiemi di dipendenze funzionali che cerchiamo.

Definizione 5.12

Sia F un insieme di dipendenze funzionali.

1. Dato $X \to Y \in F$, X contiene un *attributo estraneo* A se e solo se $(F - \{X \to Y\}) \cup \{X - \{A\} \to Y\} \equiv F$, ovvero se e solo se $X - \{A\} \to Y \in F^+$.
2. $X \to Y$ è una *dipendenza ridondante* se e solo se $(F - \{X \to Y\}) \equiv F$, ovvero se e solo se $X \to Y \in (F - \{X \to Y\})^+$.
3. F è una *copertura canonica* se e solo se:
   a) ogni parte destra di una dipendenza ha un unico attributo;
   b) le dipendenze non contengono attributi estranei;
   c) non esistono dipendenze ridondanti.

La terminologia qui adottata è quella di [Maier, 1983]; in [Ullman and Widom, 2001] questo insieme viene chiamato insieme minimale.

Le dipendenze che non contengono attributi estranei, e il cui determinato è un unico attributo, sono dette *dipendenze elementari*.

Il seguente teorema, infine, ci permetterà d'ora in poi di utilizzare, quando ci serviranno, insiemi di dipendenze che sono coperture canoniche.

Teorema 5.5

Per ogni insieme di dipendenze F esiste una copertura canonica.

Questo teorema verrà dimostrato fornendo un algoritmo per calcolare la copertura canonica di un qualsiasi insieme di dipendenze. Assumeremo che le dipendenze abbiano un unico attributo nella parte destra; se questo non fosse il caso è semplice trasformarle (ad esempio, $A \to BC$ diventa $A \to B$ e $A \to C$).

Algoritmo 5.3

Calcolo della copertura canonica

input	F insieme di dipendenze funzionali
output	G copertura canonica di F

```
begin
    G := F;
    for each X → Y ∈ G
        begin
        Z := X;
        for each A ∈ X do
            if Y ⊆ [Z − {A}]⁺_F
            then Z := Z − {A};
        G = (G − {X → Y}) ∪ {Z → Y}
        end ;
    for each f ∈ G do
        if f ∈ (G − {f})⁺ then G = G − {f} end
```

Questo algoritmo prima elimina gli attributi estranei dalle dipendenze, quindi elimina le dipendenze ridondanti. In entrambi i casi viene utilizzato il calcolo della chiusura di un insieme di attributi, con uno degli algoritmi già discussi. Esso ha complessità $O(a^2p^2)$, perché il test $Y \subseteq [X - A]^+_F$ ha complessità $O(ap)$. Un algoritmo più complicato con un costo medio inferiore è stato dato da [Diederich and Milton, 1988].

Si osservi che la verifica "$Y \subseteq [Z - \{A\}]^+_F$" dell'estraneità dell'attributo A va fatta rispetto all'insieme F che contiene la dipendenza sotto esame, poiché tale dipendenza potrebbe essere utilizzata nella dimostrazione del fatto che $Z - \{A\} \rightarrow Y$. Viceversa, la verifica della ridondanza "$f \in (G - \{f\})^+$" va fatta senza considerare f, perché altrimenti sarebbe banalmente verificata.

È importante eliminare prima gli attributi estranei e dopo le dipendenze ridondanti, come dimostra il seguente esempio: $F = \{BA \rightarrow D, B \rightarrow A, D \rightarrow A\}$. La dipendenza $B \rightarrow A$ non è riconosciuta come ridondante finché l'attributo estraneo A in $BA \rightarrow D$ non viene eliminato.

Notare che il Teorema 5.5 afferma l'esistenza, ma non l'univocità di una copertura canonica. Il seguente esempio mostra che in generale F può avere più coperture canoniche.

Esempio 5.3
Per $F = \{AB \rightarrow C, A \rightarrow B, B \rightarrow A\}$, sia $\{A \rightarrow C, A \rightarrow B, B \rightarrow A\}$ che $\{B \rightarrow C, A \rightarrow B, B \rightarrow A\}$ sono coperture canoniche.

In [Maier, 1983] sono presentati altri tipi di coperture e i relativi algoritmi di calcolo.

Definizione 5.13
Un insieme F di dipendenze funzionali è *minimo* se ha meno dipendenze di ogni insieme a lui equivalente.

Esempio 5.4
L'insieme $G = \{A \to BC, B \to A, AD \to E, BD \to I\}$ non è ridondante ma nemmeno minimo poiché $F = \{A \to BC, B \to A, AD \to EI\}$ è equivalente a G ma ha meno dipendenze. F è una copertura minima di G.

Definizione 5.14
Un insieme F di dipendenze funzionali è *ottimo* se ha meno simboli di attributi di ogni insieme a lui equivalente.

Esempio 5.5
L'insieme $F = \{EC \to D, AB \to E, E \to AB\}$ è una copertura ottima per $G = \{ABC \to D, AB \to E, E \to AB\}$. Notare che G è ridotto e minimo ma non ottimo.

5.3 Decomposizione di schemi

Come discusso inizialmente, l'approccio da seguire per eliminare anomalie da uno schema mal definito, è quello di decomporlo in schemi più piccoli che godono di particolari proprietà (*forme normali*), ma sono in qualche senso equivalenti allo schema originale. Il concetto di decomposizione viene definito come segue.

Definizione 5.15
Dato uno schema $R(T)$, $\rho = \{R_1(T_1), \ldots, R_k(T_k)\}$ è una *decomposizione* di R se e solo se $\cup_i T_i = T$.

Si noti che la definizione non richiede che gli schemi R_i siano disgiunti.

Per ciò che riguarda l'equivalenza tra lo schema originario e la sua decomposizione, si richiede in genere che questa soddisfi due condizioni, indipendenti fra di loro: *preservi i dati* (*decomposizione senza perdite o lossless join*) e *preservi le dipendenze*.

5.3.1 Decomposizioni che preservano i dati

Vediamo con un esempio cosa vuol dire che una decomposizione preservi i dati, e perché tale proprietà sia importante.

Esempio 5.6
Si consideri la relazione con schema $R(P, T, C)$ che memorizza fatti relativi ai proprietari (P) di case (C), che possono avere più telefoni (T) intestati al loro proprietario. Supponiamo che valgano le dipendenze $T \to C$ e $C \to P$:

R

P	T	C
p1	t1	c1
p1	t2	c2
p1	t3	c2

Supponiamo che si decida di decomporre lo schema e la relazione come segue:

$R_1 = \pi_{P,T}(R) =$

P	T
p1	t1
p1	t2
p1	t3

$R_2 = \pi_{P,C}(R) =$

P	C
p1	c1
p1	c2

Per ricostruire la relazione di partenza bisogna ricombinare le due relazioni della decomposizione effettuando una giunzione naturale:

$R_1 \bowtie R_2 =$

P	T	C
p1	t1	c1
p1	t1	c2
p1	t2	c1
p1	t2	c2
p1	t3	c1
p1	t3	c2

Il risultato ottenuto non coincide però con la tabella iniziale: ha più ennuple e viola le dipendenze.

L'esempio ha mostrato come nella decomposizione di una relazione e successiva giunzione si possa perdere parte dell'informazione. Quando ciò si verifica si dice che la decomposizione *non preserva i dati*.

Definizione 5.16
$\rho = \{R_1(T_1), \ldots, R_k(T_k)\}$ è una *decomposizione di* $R\langle T, F\rangle$ *che preserva i dati* se e solo se per ogni relazione r che soddisfa $R\langle T, F\rangle$:

$$r = \pi_{T_1}(r) \bowtie \ldots \bowtie \pi_{T_k}(r)$$

La precedente definizione asserisce che, per una decomposizione che preserva i dati, ogni istanza valida r della relazione di partenza deve essere uguale alla giunzione naturale della sua proiezione sui T_i, mentre, per una decomposizione qualunque, ne è solo un sottoinsieme:

Teorema 5.6
Se $\rho = \{R_1(T_1), \ldots, R_k(T_k)\}$ è una (qualunque) decomposizione di $R\langle T, F\rangle$, allora per ogni istanza r di $R\langle T\rangle$ si ha:

$$r \subseteq \pi_{T_1}(r) \bowtie \ldots \bowtie \pi_{T_k}(r)$$

Dimostrazione
Poiché $\cup_i T_i = T$, anche la relazione $\pi_{T_1}(r) \bowtie \ldots \bowtie \pi_{T_k}(r)$ è definita su T. Data un'ennupla $t \in r$, $t[T_i] \in \pi_{T_i}(r)$ e dalla definizione di giunzione naturale segue che $t \in \pi_{T_1}(r) \bowtie \ldots \bowtie \pi_{T_k}(r)$.

Questo teorema chiarisce che cosa sia una perdita di informazione: proiettando una relazione sui sottoschemi e poi facendo la giunzione si ottengono più ennuple di quante ce ne fossero nella relazione originaria.

Le definizione precedente è quantificata sull'insieme generalmente infinito di tutte le istanze valide di una relazione, e quindi non è utilizzabile direttamente per controllare la proprietà di preservazione dei dati di una decomposizione. Per questo, ci viene in aiuto il seguente teorema.

Teorema 5.7
Sia $\rho = \{R_1(T_1), R_2(T_2)\}$ una decomposizione di $R\langle T, F\rangle$. ρ è una decomposizione che preserva i dati se e solo se $T_1 \cap T_2 \to T_1 \in F^+$ oppure $T_1 \cap T_2 \to T_2 \in F^+$.

Dimostrazione
($\Leftarrow$) Supponiamo che $T_1 \cap T_2 \to T_1 \in F^+$. Sia r un'istanza valida di $R\langle T, F\rangle$ e $s = \pi_{T_1}(r) \bowtie \pi_{T_2}(r)$. Sia $t \in s$, bisogna dimostrare che $t \in r$.
Per come è stata definita s, esistono due ennuple u e v in r tali che $u[T_1] = t[T_1]$, $v[T_2] = t[T_2]$ e $u[T_1 \cap T_2] = v[T_1 \cap T_2] = t[T_1 \cap T_2]$. Poiché $T_1 \cap T_2 \to T_1 \in F^+$, ne segue che $u[T_1] = v[T_1]$ e quindi $t = v$.
Il caso $T_1 \cap T_2 \to T_2 \in F^+$ è perfettamente analogo.
($\Rightarrow$) Supponiamo che per ogni istanza valida r di $R\langle T, F\rangle$, $r = \pi_{T_1}(r) \bowtie \pi_{T_2}(r)$; bisogna dimostrare che $T_1 \cap T_2 \to T_1 \in F^+$ oppure $T_1 \cap T_2 \to T_2 \in F^+$. Procederemo per assurdo, supponendo che nessuna delle due dipendenze funzionali sia implicata da F.
Sia $W = (T_1 \cap T_2)^+$, $Y_1 = T_1 - W$ e $Y_2 = T_2 - W$. Y_1 e Y_2 sono non vuoti per ipotesi, e W, Y_1 e Y_2 sono una partizione di T.
Per ogni $A_i \in T$, $1 \leqslant i \leqslant k$, consideriamo due valori $a_i, a_i' \in \mathrm{dom}(A_i)$, con $a_i \neq a_i'$. Costruiamo la relazione r con attributi WY_1Y_2 formata dalle due ennuple:

$e_1[A_i] = a_i, 1 \leqslant i \leqslant k$

$$e_2[A_i] = \begin{cases} a_i & \text{se } A_i \in W \\ a_i' & \text{se} A_i \in Y_1Y_2 \end{cases}$$

r soddisfa ogni dipendenza $V \to Z \in F$. Infatti, se $V \not\subseteq W$, allora $e_1[V] \neq e_2[V]$, ed r soddisfa ovviamente la dipendenza. Se $V \subseteq W$, si ha che $(T_1 \cap T_2) \to V$,

quindi $(T_1 \cap T_2) \rightarrow Z$ per transitività, quindi $Z \subseteq W$, per cui $e_1[Z] = e_2[Z]$, e quindi r soddisfa la dipendenza. Inoltre, poiché Y_1 e Y_2 non sono vuoti, $\pi_{T_1}(r)$ e $\pi_{T_2}(r)$ contengono due ennuple e la loro giunzione naturale ne contiene quattro, più di quelle di r:

$\pi_{T_1}(r) \bowtie \pi_{T_2}(r) \neq r$

contraddicendo l'ipotesi. ■

Questo risultato è stato generalizzato con un algoritmo, che non sarà qui presentato, che controlla se una decomposizione con un numero qualsiasi di schemi di relazione preserva i dati (si veda, ad esempio, [Atzeni and Antonellis, 1993]).

5.3.2 Decomposizioni che preservano le dipendenze

L'altra proprietà interessante di una decomposizione è che preservi le dipendenze. Intuitivamente, questo significa che l'unione delle dipendenze dei sottoschemi è "equivalente" alle dipendenze dello schema originario.

Riprendiamo l'esempio precedente per mostrare cosa vuol dire che una decomposizione preservi le dipendenze e perché anche questa proprietà sia importante.

Esempio 5.7
Supponiamo di decomporre la relazione con schema $R(P, T, C)$ e dipendenze $T \rightarrow C$ e $C \rightarrow P$ come segue:

$R_1 = \pi_{P,T}(R) =$

P	T
p1	t1
p1	t2
p1	t3

$R_2 = \pi_{T,C}(R) =$

T	C
t1	c1
t2	c2
t3	c2

La decomposizione preserva i dati, per il teorema visto, ma non conserva la dipendenza $C \rightarrow P$, perché gli attributi C e P sono finiti in relazioni diverse. La conseguenza negativa di questo fatto è questa: supponiamo che si voglia inserire nella base di dati il fatto che la casa c1 ha un nuovo telefono t4 intestato alla persona p2. Se l'operazione si fa nella relazione con schema $R(P, T, C)$, l'operazione verrebbe proibita perché si viola il vincolo $C \rightarrow P$, ma se l'operazione si fa con le relazioni con schemi $R_1(P, T)$ ed $R_2(T, C)$ essa andrebbe a buon fine perché non viola le dipendenze dei singoli schemi, a meno di non fare i controlli sulla loro giunzione perdendo i benefici della decomposizione.

Per le ragioni che vedremo più avanti, una decomposizione di $R(P, T, C)$ che preservi dati e dipendenze è $R_1(T, C)$ ed $R_2(C, P)$.

Per definire formalmente cosa vuol dire che una decomposizione preservi le dipendenze, ci serve la nozione di *proiezione di un insieme di dipendenze*:

Definizione 5.17
Dato $R\langle T, F\rangle$, e $T_i \subseteq T$, la *proiezione* di F su T_i è:

$$\pi_{T_i}(F) = \{X \to Y \in F^+ \mid X, Y \subseteq T_i\}$$

È importante notare che in questa definizione la proiezione viene costruita considerando le dipendenze presenti in F^+, non quelle in F.

Esempio 5.8
Si consideri lo schema $R\langle T, F\rangle$, con $T = \{ABC\}, F = \{A \to B, B \to C, C \to A\}$. Allora $\pi_{AB}(F) = \{A \to B, B \to A\}$ e $\pi_{AC}(F) = \{A \to C, C \to A\}$.

Vediamo un algoritmo banale di complessità esponenziale per il calcolo di $\pi_{T_i}(F)$.

Algoritmo 5.4
Calcolo della proiezione di un insieme di dipendenze

input $R\langle T, F\rangle$ e $T_i \subseteq T$
output Una copertura della *proiezione* di F su T_i
begin
  for each $Y \subset T_i$ **do**
    begin
      $Z := Y_F^+ - Y$;
      ritorna $Y \to (Z \cap T_i)$
    end
end

La nozione di proiezione di un insieme di dipendenze ci permette di formalizzare l'equivalenza fra le dipendenze dello schema originale e quelle degli schemi decomposti:

Definizione 5.18
$\rho = \{R_1(T_1), \ldots, R_k(T_k)\}$ è una *decomposizione di* $R\langle T, F\rangle$ *che preserva le dipendenze* se e solo se $\cup\pi_{T_i}(F) \equiv F$.

Esempio 5.9
Si consideri lo schema di relazione $R\langle T, F\rangle$, con $T = \{ABC\}, F = \{A \to B, B \to C, C \to A\}$. Potrebbe sembrare che la decomposizione $\rho = \{R_1(A, B), R_2(B, C)\}$ non preservi le dipendenze perché A e C non appaiono insieme in uno schema della decomposizione. Invece $\pi_{AB}(F) = \{A \to B, B \to A\}$ e $\pi_{BC}(F) = \{B \to C, C \to B\}$ e $\pi_{AB}(F) \cup \pi_{BC}(F) \vdash C \to A$.

Potrebbe sembrare che il controllo che una decomposizione preservi le dipendenze, ovvero che ogni dipendenza $X \to Y \in F$ appartenga a G^+, con $G = \cup\pi_{T_i}(F)$, richieda un algoritmo di complessità di tempo esponenziale in quanto occorre calcolare le proiezioni di F sui T_i. In [Ullman, 1983] viene invece presentato un algoritmo di complessità di tempo polinomiale per risolvere il problema. L'idea su cui si basa l'algoritmo è di stabilire se $X \to Y \in G^+$ trovando la chiusura X^+_G di X rispetto a G senza calcolare G in maniera esplicita:

Algoritmo 5.5
Chiusura X^+_G

input $\rho = \{R_1(T_1), \ldots, R_k(T_k)\}$ decomposizione di $R\langle T, F\rangle$, $X \subseteq T$
output X^+_G, con $G = \cup\pi_{T_i}(F)$
begin
  $X^+_G := X$;
  while (X^+_G è cambiato) **do**
    for each $i := 1$ **to** k **do**
      $X^+_G := X^+_G \cup ((X^+_G \cap T_i)^+_F \cap T_i)$
end

dove la chiusura $(X^+_G \cap T_i)^+_F$ viene calcolata usando l'algoritmo visto in precedenza. Ad ogni iterazione il passo $X^+_G := X^+_G \cup ((X^+_G \cap T_i)^+_F \cap T_i)$ aggiunge a X^+_G gli attributi A_i tali che $(X^+_G \cap T_i) \to A_i \in \pi_{T_i}(F)$.

Si dimostra che il seguente algoritmo determina correttamente se una decomposizione preservi le dipendenze.

Algoritmo 5.6
Controllo se una decomposizione conserva le dipendenze

input $\rho = \{R_1(T_1), \ldots, R_k(T_k)\}$ decomposizione di $R\langle T, F\rangle$
output "Sì" se ρ preserva le dipendenze di R
begin
  for each $X \to Y \in F$ **do**
    if $Y \not\subseteq X^+_G$ **then** Termina con "No";
  Termina con "Sì"
end

Esempio 5.10
Si consideri lo schema di relazione $R\langle T, F\rangle$, con $T = \{ABCD\}, F = \{A \to B, B \to C, C \to D, D \to A\}$ e quindi tale che in F^+ ogni attributo determina tutti gli altri. Vogliamo controllare se la decomposizione $\rho = \{R_1(AB), R_2(BC), R_3(CD)\}$ preserva le dipendenze di R. Intuitivamente potrebbe sembrare che ciò non sia vero perché la dipendenza $D \to A$ non viene proiettata su nessuna relazione R_i.

Dobbiamo però considerare che in realtà è F^+ che viene proiettata sugli R_i, ed F^+ contiene anche le dipendenze $\{D \to C, C \to B, B \to A\}$, che rimangono nelle proiezioni e che, una volta rimesse insieme, implicano logicamente $D \to A$.

Verifichiamo come l'algoritmo stabilisca correttamente che $D \to A \in G^+$, con $G = \boldsymbol{\pi}_{AB}(F) \cup \boldsymbol{\pi}_{BC}(F) \cup \boldsymbol{\pi}_{CD}(F)$, ovvero che $A \in D^+_G$.

Iniziamo il calcolo di D^+_G ponendo $D^+_G := \{D\}$. Il corpo del ciclo eseguito per R_1 non modifica D^+_G perché $\{D\} \cup (\{\{D\} \cap \{A, B\})^+_F \cap \{A, B\})$ è ancora uguale a $\{D\}$. In maniera analoga il caso per R_2, mentre per R_3 abbiamo:

$$\begin{aligned} D^+_G &= \{D\} \cup (\{\{D\} \cap \{C, D\})^+_F \cap \{C, D\}) \\ &= \{D\} \cup (\{D\}^+_F \cap \{C, D\}) \\ &= \{D\} \cup (\{A, B, C, D\} \cap \{C, D\}) \\ &= \{C, D\} \end{aligned}$$

Analogamente, nel passo successivo, eseguendo il corpo del ciclo per R_2 su $D^+_G = \{C, D\}$ produce $D^+_G = \{B, C, D\}$ ed infine, nel terzo passo, otteniamo $D^+_G = \{A, B, C, D\}$, da cui risulta vero che $A \in D^+_G$.

È importante osservare che le due proprietà delle decomposizioni, di preservare i dati e le dipendenze, sono del tutto indipendenti: esistono cioè delle decomposizioni che preservano i dati ma non le dipendenze, e viceversa. Il seguente risultato collega tuttavia le due cose, e fornisce una condizione *sufficiente* (anche se non necessaria) per stabilire che una decomposizione che preserva le dipendenze preserva anche i dati.

Teorema 5.8

Sia $\rho = \{R_i\langle T_i, F_i\rangle\}$ una decomposizione di $R\langle T, F\rangle$ che preserva le dipendenze e tale che T_j, per qualche j, è una superchiave per $R\langle T, F\rangle$. Allora ρ preserva i dati.

Dimostrazione

Supponiamo, senza perdita di generalità, che T_1 sia la superchiave. Dobbiamo dimostrare che, per ogni r che soddisfa $R\langle T, F\rangle$,

$$\boldsymbol{\pi}_{T_1}(r) \bowtie \boldsymbol{\pi}_{T_2}(r) \ldots \bowtie \boldsymbol{\pi}_{T_n}(r) \subseteq r$$

Sia $e \in \boldsymbol{\pi}_{T_1}(r) \bowtie \pi_{T_2}(r) \ldots \bowtie \boldsymbol{\pi}_{T_n}(r)$ Per definizione di giunzione esistono $e_i \in \boldsymbol{\pi}_{T_i}(r)$ tali che, per i in $1, \ldots, n$, $e_i[T_i] = e[T_i]$. Per definizione di proiezione esistono $e'_i \in r$ tali che, per i in $1, \ldots, n$, $e'_i[T_i] = e_i[T_i]$, da cui $e'_i[T_i] = e[T_i]$. Vogliamo ora dimostrare che in realtà $e = e'_1$; la tesi segue, poiché $e'_1 \in r$. A tale scopo basta dimostrare che la relazione s, con schema $R(T)$, composta solo da e e da e'_1, soddisfa F; questo implicherebbe $e = e'_1$, poiché le due ennuple coincidono sulla superchiave T_1.

Poiché la scomposizione preserva le dipendenze, è sufficiente dimostrare che s soddisfa $\boldsymbol{\pi}_{T_i}(F)$ per ogni i. Si osservi che, per ogni dipendenza $X \to Y$ e per ogni $T' \subseteq T$ che includa sia X che Y, una relazione $r : R\langle T, F\rangle$ soddisfa la dipendenza se e solo se la soddisfa $\boldsymbol{\pi}_{T'}(r)$. Quindi anche s soddisfa $\cup_i \boldsymbol{\pi}_{T_i}(F)$ se e solo se $\pi_{T_i}(s)$, ovvero $\{e[T_i], e'_1[T_i]\}$, soddisfa $\pi_{T_i}(F)$ per ogni i. $\{e[T_i], e'_1[T_i]\}$, soddisfa $\boldsymbol{\pi}_{T_i}(F)$ poiché $\{e[T_i], e'_1[T_i]\} \subseteq \boldsymbol{\pi}_{T_i}(r)$: $e[T_i] = e'_i[T_i]$, ed e'_1 ed e'_i stanno entrambi in r. ■

Nel seguito, discutendo gli algoritmi di decomposizioni di schemi in forme normali, avremo come obiettivo l'ottenimento di decomposizioni con entrambe le proprietà. Vedremo, però, che questo non sarà sempre possibile, e che in alcuni casi otterremo degli schemi non completamente equivalenti a quelli di partenza.

5.4 Forme normali

Con l'aiuto dei concetti introdotti, siamo ora in grado di affrontare l'obiettivo principale della normalizzazione: il passaggio da schemi "anomali" a schemi "ben fatti". Questa teoria, nata dai lavori di Codd, insieme al modello relazionale, si è via via affinata, sia negli strumenti che negli obiettivi perseguiti. Durante il suo sviluppo, sono state individuate diverse categorie di "anomalie" dovute a cattiva progettazione, e questo ha portato alla definizione di diverse *forme normali*, intese come proprietà che devono essere soddisfatte dalle dipendenze fra attributi di schemi "ben fatti".

Tra le forme normali proposte alcune hanno solo un interesse storico, come la *prima forma normale* e la *seconda forma normale*. La prima forma normale richiede che i valori di tutti i domìni di una relazione siano atomici, proprietà oggi considerata un vincolo implicito del modello dei dati. Recentemente, sono stati introdotti dei modelli relazionali che non soddisfano il vincolo dell'atomicità dei domini, estendendo sia i linguaggi relazionali che la teoria, per trattare anche gli operatori sui valori dei domini (*modelli relazionali estesi*, *modelli relazionali non in prima forma normale*, *modelli relazionali a oggetti*). Questi nuovi modelli relazionali prevedono che il valore di un attributo in una ennupla possa essere un valore elementare, un'ennupla, oppure una relazione.

Le forme normali più interessanti sono la *forma normale di Boyce-Codd (BCNF)* e la *terza forma normale (3NF)*, meno restrittiva della BCNF, che vedremo in questo ordine anche se la BCNF è stata proposta successivamente alla 3NF.

5.4.1 Forma Normale di Boyce-Codd

L'idea su cui si basa la nozione di forma normale di Boyce-Codd è che una dipendenza funzionale $X \to A$ (in cui X non contiene attributi estranei) indica che, nel dominio che si modella, esiste una collezione C di entità che sono univocamente identificate da X. Ad esempio, facendo riferimento allo schema

Biblioteca(NomeUtente, Residenza, Telefono, NumeroLibro, Autore, Titolo, Data)

presentato all'inizio del capitolo, la dipendenza NomeUtente $\rightarrow$ Residenza, Telefono, indica l'esistenza nel dominio modellato di entità identificate da un NomeUtente e con proprietà Residenza e Telefono. Quindi, se una relazione ben disegnata contiene X, vi sono solo due possibilità:

1. se la relazione modella la collezione C, allora X deve essere una chiave per tale relazione;
2. se la relazione non modella la collezione C, allora X deve apparire solo come chiave esterna, per cui nessuno degli altri attributi delle entità della collezione deve apparire nella relazione.

Ne segue che, in ogni relazione, o X è una chiave (caso 1), oppure non deve apparire nessuna $A \notin X$ tale che $X \rightarrow A$ (caso 2).

Più precisamente, vale la seguente definizione.

Definizione 5.19
Uno schema $R\langle T, F\rangle$ è in BCNF se e solo se per ogni dipendenza funzionale non banale $X \rightarrow Y \in F^+$, X è una superchiave.

In seguito, le dipendenze $X \rightarrow Y$ non banali tali che X non è una superchiave saranno chiamate *dipendenze anomale*.

Questa definizione indica chiaramente come il fatto di essere in BCNF dipenda solo dalla chiusura F^+ delle dipendenze, e non dalla specifica copertura F che si è scelta, ma non è utilizzabile in pratica, perché la generazione di tutto F^+ è un'operazione troppo costosa. Il teorema seguente fornisce invece un criterio per stabilire se uno schema di relazione sia in BCNF con un algoritmo di complessità polinomiale.

Teorema 5.9
Uno schema $R\langle T, F\rangle$ è in BCNF se e solo se per ogni dipendenza funzionale non banale $X \rightarrow Y \in F$, X è una superchiave.

Corollario 5.1
Uno schema $R\langle T, F\rangle$, con F copertura canonica, è in BCNF se e solo se per ogni dipendenza funzionale elementare $X \rightarrow A \in F$, X è una superchiave (ovvero è una chiave).

Da questo risultato scaturisce che un algoritmo per controllare se uno schema di relazione è in BCNF ha complessità $O(ap^2)$.

Esempio 5.11
Si consideri il seguente schema di relazione:

Magazzini$\langle\{$Articolo, Magazzino, Quantità, Indirizzo$\}$,

$$\{\text{Articolo Magazzino} \rightarrow \text{Quantità},\ \text{Magazzino} \rightarrow \text{Indirizzo}\}\ \rangle$$

che descrive gli articoli presenti in un certo numero di magazzini, insieme con la quantità disponibile. Lo schema non è in forma normale perché l'attributo Indirizzo dipende dall'attributo Magazzino, che non è una chiave per la relazione, come si può verificare dalla seguente istanza valida.

Magazzini

Articolo	Magazzino	Quantità	Indirizzo
Flauto	Roma	10	Via Cavour, 7
Oboe	Roma	5	Via Cavour, 7
Arpa	Torino	1	Via Mazzini, 11

Se ne può dedurre che la relazione mescola informazioni relative a quelle entità che sono identificate dall'attributo Magazzino con informazioni relative ad altre entità (gli articoli), per cui lo schema è mal disegnato. Più concretamente, dalla violazione della forma normale scaturiscono le seguenti anomalie:

1. ridondanza:
   a) inserendo un nuovo articolo in un magazzino già presente occorre replicare l'indirizzo del magazzino;
   b) la modifica di indirizzo di un magazzino deve essere riportata in tutte le ennuple dove è presente quel magazzino, altrimenti si ha una inconsistenza.

   Questa ridondanza scaturisce direttamente dalla presenza di dipendenze anomale, in modo del tutto indipendente da ogni altro aspetto del mondo modellato;
2. difficoltà a gestire l'esistenza di magazzini senza merce:
   a) per inserire un nuovo magazzino è necessario che vi sia almeno un articolo;
   b) eliminando l'ultimo esemplare di un certo articolo (come "arpa" nel magazzino di "Torino"), se l'articolo era l'unico presente va perduto anche l'indirizzo del magazzino.

   Queste anomalie non scaturiscono direttamente dalle dipendenze anomale, ma piuttosto dall'esistenza nel mondo modellato di una nozione di "magazzino" che è indipendente da quella delle merci immagazzinate.

5.4.2 Normalizzazione di schemi in BCNF

La BCNF fornisce un criterio per stabilire in modo formale se uno schema è privo di anomalie. In caso contrario esistono algoritmi, detti di *normalizzazione*, per trasformare opportunamente lo schema.

Gli algoritmi di normalizzazione di schemi in BCNF sono detti di *analisi* (dal greco, *scomposizione*) in quanto partono da uno schema contenente tutti gli attributi e lo dividono fino a che non si ottiene uno schema in BCNF. Più precisamente, se $R\langle XAZ, F\rangle$ non è in BCNF a causa della dipendenza anomala $X \to A$, allora R si decompone nei due schemi $R_1(X, A)$ ed $R_2(X, Z)$, si calcolano le proiezioni delle dipendenze di R su R_1 e R_2 e si ripete il procedimento. La decomposizione trovata non è l'unica possibile ma dipende dall'ordine in cui si prendono in considerazione le dipendenze anomale dello schema.

Algoritmo 5.7
Decomposizione per forma normale di Boyce-Codd

input $R_1\langle T_1, F_1\rangle$, con F copertura canonica (non necessario, ma utile)
output $\rho = \{R_1, \ldots, R_m\}$ decomposizione di R in BCNF che preserva i dati

begin
$\rho := \{R_1\langle T_1, F_1\rangle\}$; $n := 1$;
while esiste $R_i\langle T_i, F_i\rangle \in \rho$ non in BCNF per la dipendenza $X \to A$ **do**
begin
$n := n + 1$;
$T' := X^+$;
$F' := \pi_{T'}(F_i)$;
$T'' := T_i - (T' - X)$;
$F'' := \pi_{T''}(F_i)$;
$\rho := \rho - R_i\langle T_i, F_i\rangle + \{R_i\langle T', F'\rangle, R_n\langle T'', F''\rangle\}$
end ;
end ;

L'algoritmo ha complessità esponenziale, a causa del calcolo della proiezione delle dipendenze funzionali, e produce una decomposizione che preserva i dati per la seguente proprietà delle decomposizioni.

Teorema 5.10
Sia $\rho = \{R_1, \ldots, R_m\}$ una decomposizione di $R\langle T, F\rangle$ che preserva i dati e sia $\sigma = \{S_1, S_2\}$ una decomposizione di R_1 che preserva i dati rispetto a $\pi_{T_1}(F)$.
Allora anche la decomposizione $\rho = \{S_1, S_2, R_2, \ldots, R_m\}$ preserva i dati rispetto a F.

Purtroppo non esiste alcuna garanzia che la decomposizione generata, oltre a preservare i dati, preservi anche le dipendenze, come mostra l'esempio seguente.

Esempio 5.12
Partiamo dalla relazione:

Telefoni⟨{Prefisso, Numero, Località},
{Prefisso, Numero → Località
Località → Prefisso} ⟩

Inizialmente, $\rho = \{\text{Telefoni}\}$. Nella prima iterazione viene scoperto che la dipendenza Località → Prefisso viola la BCNF, e quindi si rimpiazza ρ con $R_1 = \langle\{\text{Località, Prefisso}\}, \{\text{Località} \rightarrow \text{Prefisso}\}\rangle$ e con $R_2 = \langle\{\text{Numero, Località}\}\rangle$.

Nella seconda iterazione, poiché le relazioni ottenute sono in BCNF, l'algoritmo termina.

La scomposizione prodotta preserva i dati, ma la dipendenza Prefisso, Numero → Località va perduta. In termini concreti, supponiamo di avere assegnato, per sbaglio, lo stesso numero 504050 a due abbonati che risiedono in due diverse località Pisa e Pontedera con lo stesso prefisso 050. Questo errore verrebbe prevenuto da un sistema che gestisse la dipendenza funzionale Prefisso, Numero → Località, ma non viola nessuno dei vincoli relativi allo schema decomposto.

Questo esempio mostra anche che non possono esistere algoritmi di decomposizione in BCNF che preservano sempre le dipendenze. Infatti, una decomposizione in BCNF dello schema Telefoni non può mantenere tutti e tre gli attributi nella stessa relazione, per cui la dipendenza Prefisso, Numero → Località non appare nella proiezione di F sugli schemi della decomposizione. In queste condizioni, dato che né Prefisso né Numero, presi separatamente, determinano la località, la chiusura di Prefisso, Numero secondo le dipendenze proiettate non potrà contenere la località.

L'algoritmo di analisi presentato ha complessità esponenziale, a causa del costo dell'operazione di proiezione delle dipendenze. Sono stati proposti anche algoritmi polinomiali ([Tsou and Fischer, 1982]), ma non sono utilizzati nella pratica poiché producono schemi poco naturali ed eccessivamente decomposti.

Per essere certi di poter sempre ottenere uno schema normalizzato che preservi dati e dipendenze occorre abbandonare la BCNF e limitarsi ad una forma normale meno restrittiva, la *terza forma normale*.

5.4.3 Terza forma normale

La terza forma normale (3NF) è un criterio per valutare la qualità di uno schema che è meno restrittivo della BCNF, e che gode della seguente proprietà: ogni schema $R\langle T, F\rangle$ ammette una decomposizione che preserva i dati, preserva le dipendenze, ed è in 3NF; una tale decomposizione può essere ottenuta in tempo polinomiale. Lo svantaggio della 3NF sta nel fatto che, essendo meno restrittiva della BCNF, accetta anche schemi che presentano delle anomalie.

Come nel caso della BCNF, sono state date diverse definizioni equivalenti di 3NF. Una semplice definizione è:

Definizione 5.20
Uno schema $R\langle T, F\rangle$ è in 3NF se e solo se, per ogni dipendenza funzionale non banale $X \to A \in F^+$, allora X è una superchiave oppure A è primo.

Come si vede da questa definizione, se una relazione è in forma normale di Boyce-Codd allora è anche in terza forma normale.

Questa definizione non è utilizzabile in pratica per verificare se uno schema sia in 3NF a causa delle dimensioni esponenziali di F^+. Esiste un teorema, analogo a quello visto per la BCNF, che indica la possibilità di effettuare il test in modo molto più efficiente, anche se non in tempo polinomiale.

Teorema 5.11
Uno schema $R\langle T, F\rangle$ è in 3NF se e solo se, per ogni dipendenza funzionale $X \to A_1, \ldots, A_n \in F$, e per ogni $i \in \{1 \ldots n\}$, $A_i \in X$ oppure X è una superchiave oppure A_i è primo.

Corollario 5.2
Uno schema $R\langle T, F\rangle$, con F copertura canonica, è in 3NF se e solo se, per ogni dipendenza funzionale $X \to A \in F$, allora X è una superchiave (ovvero una chiave) oppure A è primo.

In ogni caso, per stabilire se uno schema è in 3NF occorre conoscere gli attributi primi. Per questo motivo, dati gli analoghi risultati per la ricerca delle chiavi, vale la seguente:

Proposizione 5.1
Il problema di decidere se uno schema di relazione è in 3NF è NP-completo.

La terza forma normale, dato che ammette la presenza di dipendenze anomale, non elimina tutte le anomalie, come mostra l'esempio seguente.

Esempio 5.13
Consideriamo il seguente schema di relazione:

Telefoni⟨{Prefisso, Numero, Località, NomeAbbonato, Via},
{Prefisso, Numero → Località,
Prefisso, Numero → NomeAbbonato,
Prefisso, Numero → Via,
Località → Prefisso}⟩

che descrive gli abbonati al telefono. Le chiavi della relazione sono (Prefisso, Numero) e (Località, Numero). La relazione è in 3NF, ma non in BCNF, come si può facilmente verificare, e infatti esiste una ridondanza dovuta al fatto che ogni volta

che si inserisce un nuovo numero telefonico di una certa località bisogna ripetere l'informazione sul prefisso.

5.4.4 Normalizzazione di schemi in 3NF

L'interesse per la terza forma normale deriva dal fatto che esiste un algoritmo semplice ed efficiente che permette di decomporre qualunque schema in un insieme di schemi di relazione in 3NF preservando dati e dipendenze. L'algoritmo si basa sulla seguente idea: dato un insieme di attributi T ed una copertura canonica G, si partiziona G in gruppi G_i tali che tutte le dipendenze in ogni G_i hanno la stessa parte sinistra. Quindi, da ogni G_i, si definisce uno schema di relazione composto da tutti gli attributi che vi appaiono, la cui chiave, detta *chiave sintetizzata*, è la parte sinistra comune.

Algoritmo 5.8

Algoritmo intuitivo ***per la sintesi di schemi in 3NF***

input Un insieme R di attributi e un insieme F di dipendenze su R.
output Una decomposizione $\rho = \{S_i\}_{i=1..n}$ di R tale che ρ preservi dati e dipendenze e ogni S_i sia in 3NF, rispetto alle proiezioni di F su S_i.

begin
{Passo 1} Trova una copertura canonica G di F e poni $\rho = \{\}$.
{Passo 2} Sostituisci in G ogni insieme $X \to A_1, \ldots, X \to A_h$ di dipendenze con lo stesso determinante, con la dipendenza $X \to A_1 \cdots A_h$.
{Passo 3} Per ogni dipendenza $X \to Y$ in G, metti uno schema con attributi XY in ρ.
{Passo 4} Elimina da ρ ogni schema che sia contenuto in un altro schema di ρ.
{Passo 5} Se la decomposizione non contiene alcuno schema i cui attributi costituiscano una superchiave per R, aggiungi ad essa lo schema con attributi W, con W una chiave di R.
end

L'algoritmo è chiamato di *sintesi* perché opera raggruppando attributi per formare schemi. Vale la seguente proprietà.

Teorema 5.12

Dato uno schema $R\langle T, F\rangle$, l'algoritmo elementare produce una decomposizione $\rho = \{S_i\}_{i=1..n}$ di R che preserva dati e dipendenze.

Dimostrazione

L'algoritmo produce una decomposizione perché tutti gli attributi di T appartengono allo schema generato; in particolare, se ci sono attributi che non partecipano a nessuna dipendenza, questi fanno parte di ogni chiave di R, per cui compaiono nella relazione aggiunta al passo 5. La decomposizione preserva le dipendenze poiché per ogni dipendenza $X \to A_i$ in G esiste uno schema che contiene XA_i. Infine, la decomposizione preserva i dati poiché, grazie al passo 5, soddisfa la condizione espressa dal Teorema 5.8. ■

La complessità dell'algoritmo dipende da quella del calcolo della copertura canonica, pari a $O(a^2p^2)$.

Esempio 5.14
Si considerino gli attributi Prefisso, Numero e Località e le dipendenze Prefisso, Numero $\to$ Località e Località $\to$ Prefisso. Le dipendenze sono una copertura canonica e l'algoritmo elementare produce la decomposizione con schema {Prefisso, Numero, Località}. Notare che la decomposizione è in 3NF ma non in BCNF.

Esempio 5.15
Applicando solo i primi quattro passi dell'algoritmo elementare si produrrebbero decomposizioni che preservano le dipendenze ma non i dati. Si considerino gli attributi A, B, C e D e le dipendenze $A \to B$ e $C \to D$. Con i primi quattro passi si produce la decomposizione con schemi $R_1(A, B)$ e $R_2(C, D)$ che non preserva i dati. Applicando il passo 5 si aggiunge alla decomposizione anche lo schema $R_3(A, C)$ e la decomposizione preserva anche i dati.

■ **Teorema 5.13**
Dato uno schema $R\langle T, F\rangle$, l'algoritmo elementare produce una decomposizione $\rho = \{S_i\}_{i=1..n}$ di R in schemi in 3NF.

Dimostrazione
Supponiamo, per assurdo, che nella decomposizione ci sia uno schema S non in 3NF, con attributi $XA_1 \cdots A_h$, dove X è la "chiave sintetizzata". X è una chiave di S: X implica tutti gli altri attributi perché $X \to A_i$ appartiene a G per ogni A_i, e non contiene attributi ridondanti perché altrimenti G non sarebbe una copertura canonica.
Poiché S non è in 3NF, esiste una dipendenza $V \to C$ in $(\pi_S(G))^+$, e quindi in G^+, in cui V non è una superchiave di S, C non è primo in S e $C \notin V$.
Poiché X è una chiave di S, si ha che $C \notin X$ e quindi $C = A_i$ per qualche i. Si prendono in esame due casi, a seconda che $V \subset X$ oppure no.
Se $V \subset X$, allora l'esistenza in G della dipendenza $X \to A_i$ è in contraddizione col fatto che G è una copertura canonica.

Se $V \not\subset X$, allora $V = YW$ per qualche $Y \subset X$ e $W \subseteq A_1 \cdots A_{i-1}A_{i+1} \cdots A_h$, con $W \neq \emptyset$. Sappiamo che la dipendenza $X \to A_i$ appartiene alla copertura canonica; raggiungiamo ora l'assurdo dimostrando invece che essa è ridondante e quindi può essere derivata dall'insieme $G' = G - \{X \to A_i\}$. A questo scopo mostriamo che $G' \vdash X \to YW$ e $G' \vdash YW \to A_i$.
$G' \vdash X \to YW$, segue dal fatto che $Y \subseteq X$, mentre W contiene solo attributi A_j con $j \neq i$, e le dipendenze $X \to A_j$ $(j \neq i)$ appartengono a G'.
$G' \vdash YW \to A_i$: poiché $G \vdash X \to S$ mentre $G \not\vdash YW \to S$ ($YW = V$ non è una superchiave per S), si ha che $X \not\subseteq V^+_G$; quindi $V^+_{G'} = V^+_G$, poiché nel calcolo della chiusura di V in G la dipendenza $X \to A_i$ non interviene. Avendo supposto $G \vdash YW \to A_i$, ciò implica che $G' \vdash YW \to A_i$. ■

L'algoritmo di sintesi produce una decomposizione $\rho = \{S_i\}_{i=1..n}$ di R tale che ρ preserva dati e dipendenze e ogni S_i è in 3NF, rispetto alle proiezioni di F su S_i, ma non fornisce per ogni S_i una copertura delle proiezioni di F su S_i.

Esempio 5.16
Sia $R = \{ABCD\}$ e $F = \{AB \to C, C \to D, D \to B\}$.
Le dipendenze sono una copertura canonica e l'algoritmo intuitivo produce la decomposizione con schemi:

$R_1(ABC)$ con chiave sintetizzata AB;
$R_2(CD)$ con chiave sintetizzata C;
$R_3(DB)$ con chiave sintetizzata D.

La proiezione di F su R_2 è $\{C \to D\}$, su R_3 è $\{D \to B\}$, su R_1 è $\{AB \to C, C \to B\}$ e non $\{AB \to C\}$.

In [Bernstein, 1976] è dato un altro algoritmo di sintesi di complessità $O(a^2p^2)$ che produce una decomposizione con il minor numero possibile di schemi. Infatti, l'algoritmo intuitivo non ha questa proprietà, come mostrato nell'esempio che segue.

Esempio 5.17
Sia $R = \{ABCDEH\}$ e $F = \{EH \to A, EH \to D, CD \to E, CD \to H, AE \to B, BH \to C, C \to A\}$.
Le dipendenze sono una copertura canonica e l'algoritmo intuitivo produce la decomposizione con schemi:

$R_1(EHAD)$ con chiave sintetizzata EH;
$R_2(CDEH)$ con chiave sintetizzata CD;
$R_3(AEB)$ con chiave sintetizzata AE;
$R_4(BHC)$ con chiave sintetizzata BH;
$R_5(CA)$ con chiave sintetizzata C.

La decomposizione è senza perdite di dati perché le chiavi di R sono EH, CD e BDH, con EH e CD chiavi sintetizzate di R_1 e R_2.

L'algoritmo dato in [Bernstein, 1976] produce invece una decomposizione con uno schema in meno:

$R_1(EHCD)$ con chiavi sintetizzate EH e CD;
$R_2(AEB)$ con chiave sintetizzata AE;
$R_3(BHC)$ con chiave sintetizzata BH;
$R_4(CA)$ con chiave sintetizzata C.

L'idea usata per ridurre il numero degli schemi della decomposizione è di controllare dopo il passo 3 se esistono gruppi con determinanti X e Y equivalenti (cioè se vale $X \to Y$ e $Y \to X$); in tal caso si fondono i due gruppi in uno solo. Tuttavia non basta aggiungere solo questo passo all'algoritmo intuitivo per ottenere un algoritmo corretto, come mostrato nel seguito.

Alla fine del passo 3 si hanno i seguenti gruppi di dipendenze:

$G_1 = \{EH \to A, EH \to D\}$;
$G_2 = \{CD \to E, CD \to H\}$;
$G_3 = \{AE \to B\}$;
$G_4 = \{BH \to C\}$;
$G_5 = \{C \to A\}$.

I determinanti EH e CD sono equivalenti, infatti:

da $EH \to A$ e $AE \to B$ si deduce $EH \to B$;
da $EH \to B$ e $BH \to C$ si deduce $EH \to C$;
da $EH \to C$ e $EH \to D$ si deduce $EH \to CD$.

Pertanto si fondono G_1 e G_2 ottenendo i gruppi:

$G_{12} = \{EH \to A, EH \to D, CD \to E, CD \to H\}$;
$G_3 = \{AE \to B\}$;
$G_4 = \{BH \to C\}$;
$G_5 = \{C \to A\}$.

Se si definissero gli schemi della decomposizione a partire da questi gruppi si otterrebbe:

$R_1(EHACD)$ con chiavi sintetizzate EH e CD;
$R_2(AEB)$ con chiave sintetizzata AE;
$R_3(BHC)$ con chiave sintetizzata BH;
$R_4(CA)$ con chiave sintetizzata C.

con R_1 non in forma normale per la dipendenza $C \to A$.
Per risolvere questo problema, viene previsto un altro passo che ha l'effetto di rimuovere l'attributo che provoca l'anomalia (nel nostro esempio A da R_1); que-

sto attributo si trova sicuramente in un altro schema della decomposizione e si dimostra che la sua rimozione non altera le proprietà dell'algoritmo.

In conclusione, dovendo scegliere tra la 3NF e la BCNF, la strategia di controllare prima se esiste una decomposizione in BCNF che preservi le dipendenze e in caso negativo ripiegare sulla 3NF non può essere usata, perché richiede un algoritmo di complessità esponenziale. È compito del progettista quindi scegliere in anticipo fra i due tipi di normalizzazione: la 3NF, con un algoritmo di complessità polinomiale che produce decomposizioni che preservano i dati e le dipendenze, ma con la possibilità di ottenere schemi che contengono ancora delle anomalie, oppure la BCNF, con un algoritmo di complessità esponenziale (quello polinomiale ha scarso interesse pratico) che produce decomposizioni che non preservano necessariamente le dipendenze.

5.5 Dipendenze multivalore

La BCNF risolve tutte le anomalie connesse alle dipendenze funzionali. In una base di dati relazionali, d'altra parte, possono essere presenti altre anomalie non imputabili alle dipendenze funzionali. Consideriamo il seguente esempio:

Esempio 5.18
Consideriamo la relazione Impiegati(Nome, Figlio, Stipendio) che contiene informazioni relative ai figli e alla storia degli stipendi percepiti di un impiegato.

Una possibile estensione è la seguente:

Impiegati

Nome	Figlio	Stipendio
Bragazzi	Maurizio	100
Bragazzi	Maurizio	120
Bragazzi	Maurizio	140
Bragazzi	Marcello	100
Bragazzi	Marcello	120
Bragazzi	Marcello	140
Fantini	Maria	160
Fantini	Maria	180
Fantini	Maria	190

La relazione dell'esempio precedente è in BCNF, infatti non vi sono dipendenze funzionali non banali, ma è presente una notevole ridondanza. Ci si può accorgere facilmente che una ridondanza di questo tipo si verifica tutte le volte che in una relazione si rappresentano proprietà multivalore indipendenti, come le proprietà figli a carico e storia dello stipendio degli impiegati.

Notare che le anomalie non dipendono esclusivamente dal fatto che esista una proprietà multivalore, ma dal fatto che una tale proprietà coesiste nello schema con altre proprietà semplici o multivalore indipendenti. Ad esempio, decomponendo lo schema in due sottoschemi in modo da modellare separatamente le proprietà multivalori indipendenti (come è stato fatto nell'algoritmo di trasformazione di uno schema ad oggetti in relazionale), si avrebbe una base di dati priva di anomalie:

StipendiImpiegati

Nome	Stipendio
Bragazzi	100
Bragazzi	120
Bragazzi	140
Fantini	160
Fantini	180
Fantini	190

FigliImpiegati

Nome	Figlio
Bragazzi	Maurizio
Bragazzi	Marcello
Fantini	Maria

Notare anche che se si cambia la semantica dell'attributo Stipendio, immaginando che non sia più lo stipendio percepito da un impiegato, ma quello percepito attualmente da ogni figlio, allora nello schema sarebbe modellata un'unica proprietà multivalore costituita da un insieme di coppie (Figlio, Stipendio), e non più due indipendenti, e lo schema sarebbe di nuovo privo di anomalie:

Impiegati

Nome	Figlio	Stipendio
Bragazzi	Maurizio	100
Bragazzi	Marcello	120
Fantini	Maria	180

Per evitare le anomalie dovute alla coesistenza di proprietà multivalore indipendenti la teoria della normalizzazione vista finora è stata estesa nel modo seguente:

1. È stata definita una nuova dipendenza fra i dati, detta *dipendenza multivalore* (*MVD*).
2. È stata estesa alle dipendenze multivalore la nozione di dipendenza derivata.
3. È stato dato un nuovo insieme di regole di inferenza corretto e completo per dipendenze funzionali e multivalore.
4. È stata estesa la nozione di decomposizione che preserva dati e dipendenze.
5. È stata data una nuova definizione di forma normale, la *quarta forma normale* (*4NF*), che generalizza la forma normale BCNF.
6. È stato dato un algoritmo di normalizzazione di schemi non in quarta forma normale che generalizza quello visto per BCNF, ed ha le stesse proprietà di conservare i dati ma non le dipendenze.

5.6 La denormalizzazione

L'eliminazione di ogni dipendenza anomala da uno schema va perseguita per produrre un buon schema concettuale ed un buon schema logico della realtà modellata.

Quando poi si passa ad effettuare la progettazione fisica, e quindi si prendono in considerazione i problemi legati all'efficienza dell'esecuzione delle applicazioni, può essere opportuno riintrodurre nello schema fisico qualche dipendenza anomala. Si immagini, ad esempio, una situazione in cui si deve gestire un archivio di studenti e di esami da loro superati, in cui gli aggiornamenti siano estremamente rari, vi sia a disposizione una grande quantità di spazio disco, e in cui si desideri effettuare la giunzione di uno studente e i suoi esami con la massima efficienza. In questo caso, è opportuno memorizzare le informazioni relative a studenti ed esami in un'unica relazione (*denormalizzazione*), evitando così la necessità di effettuare giunzioni, anche se a prezzo di un maggior costo delle operazioni di modifica, e di una maggiore occupazione di memoria.

È tuttavia di grande importanza non anticipare queste considerazioni dalla fase di progettazione fisica a quella di progettazione logica. Questo, non solo perché è importante che lo schema logico sia della massima qualità, ma anche perché, quando si produce lo schema fisico, ci possono essere dei mezzi migliori per ottenere lo stesso effetto. In particolare, quasi tutti i sistemi per la gestione di basi di dati permettono di memorizzare due diverse relazioni in modo *aggregato* (*clustered*), senza modificare lo schema logico, ma solo indicando la volontà di aggregare le due relazioni all'interno dello schema fisico. La memorizzazione aggregata significa, nel nostro esempio, memorizzare ogni studente accanto a tutti gli esami da lui superati. Questa modalità di memorizzazione rende estremamente efficiente l'esecuzione della giunzione, a scapito dell'efficienza di altre operazioni (ad esempio, la scansione di tutti gli studenti), senza appesantire troppo l'occupazione di memoria.

La denormalizzazione è molto comune quando i dati sono gestiti usando non un DBMS ma un sistema di archiviazione oppure un foglio elettronico (*spreadsheet*). In questi casi, la denormalizzazione si utilizza non solo per ragioni di efficienza, ma anche per velocizzare la produzione delle applicazioni, evitando la codifica di algoritmi di giunzione.

5.7 Uso della teoria della normalizzazione

L'uso delle nozioni definite in questo capitolo da parte di un progettista dipende dallo scopo che egli sta cercando di perseguire, distinguendo in particolare tra:

1. traduzione relazionale di un progetto concettuale ad oggetti;
2. analisi di una base di dati, o archivio, già esistente;
3. progettazione secondo l'approccio della relazione universale.

Nel primo caso, il progettista ha imparato, da questo capitolo, che, durante la progettazione concettuale (anche ad oggetti), le dipendenze anomale devono essere guardate con sospetto, chiedendosi se una dipendenza anomala $X \to Y$ non nasconda in realtà

un'entità con attributi XY che non è stata individuata. La teoria sviluppata in questo capitolo gli permette anche di sapere che l'eliminazione totale delle dipendenze anomale può, in certi casi, comportare la perdita di alcune di esse. I risultati contenuti nel capitolo possono essere inoltre usati per dimostrare che, se si traduce uno schema ad oggetti privo di dipendenze anomale usando le regole definite nel Capitolo 4, si ottiene uno schema relazionale in BCNF.

La teoria qui sviluppata è molto più utile in una situazione in cui il progettista deve analizzare una base di dati, o un insieme di archivi, già esistente, per verificare la bontà dello schema. In questo caso, la tecnica di verificare, relazione per relazione o archivio per archivio, l'esistenza di dipendenze funzionali anomale è di decomporre lo schema secondo tali dipendenze, e di semplice applicazione e porta in genere a buoni risultati.

Chi, infine, decida di progettare un sistema seguendo l'approccio della relazione universale può sfruttare a pieno la teoria sviluppata in questo capitolo, ma deve tenere conto del fatto che questo approccio ha un potere espressivo minore rispetto all'utilizzo di un modello ad oggetti per la progettazione concettuale, e del fatto che uno schema realistico non può essere modellato senza utilizzare, accanto alle dipendenze funzionali, almeno le dipendenze multivalore.

5.8 Conclusioni

In questo capitolo sono stati presentati i principali risultati della teoria della normalizzazione. L'obiettivo di questa teoria è la costruzione di strumenti formali per la definizione di schemi di relazioni che non presentino anomalie. Per raggiungere questo obiettivo sono stati introdotti alcuni tipi di vincoli, le dipendenze fra i dati, che esprimono formalmente aspetti significativi dell'universo del discorso. Questi vincoli sono stati utilizzati per descrivere proprietà di schemi di relazioni senza anomalie (schemi normalizzati). Sono stati descritti alcuni algoritmi che permettono di trasformare schemi non normalizzati in schemi normalizzati, mantenendo, per quanto possibile, il significato originario del modello (decomposizioni che preservano i dati e decomposizioni che preservano le dipendenze). Sono stati inoltre discussi alcuni risultati negativi di tale teoria, dovuti sia alla intrinseca complessità computazionale di alcuni algoritmi, che è esponenziale, sia al fatto che è in generale impossibile decomporre uno schema in forma normale di Boyce-Codd preservando nello stesso tempo i dati e le dipendenze.

La teoria della normalizzazione non si limita a considerare le dipendenze funzionali e multivalore, come è stato fatto in questo capitolo, e ha studiato altri tipi di dipendenze fra i dati (ad esempio, le dipendenze di giunzione,*join dependencies*, le dipendenze multivalore racchiuse, *embedded multivalued dependencies*), altre forme normali (ad esempio, 5NF e DKNF) ed altri interessanti problemi. Per approfondire questa tematica si rinvia ai testi specifici sull'argomento.

Esercizi

1. Usando gli assiomi di Armstrong si dimostri che:
   a) se $X \to YZ$, allora $X \to Y$ e $X \to Z$;
   b) se $X \to Y$ e $WY \to Z$, allora $XW \to Z$;
   c) se $X \to YZ$ e $Z \to BW$, allora $X \to YBW$.
2. Dimostrare che se uno schema $R\langle T, F\rangle$ ha solo due attributi A e B, e le possibili istanze di R non hanno i valori tutti uguali di A, o di B, allora è in BCNF.
3. Sia $R(A, B, C, D)$ uno schema relazionale con dipendenze $F = \{A \to B, C \to B, D \to ABC, AC \to D\}$. Trovare un insieme di dipendenze G in forma canonica equivalente a F.
4. Sia $R\langle T, F\rangle$ uno schema relazionale. Dimostrare che se un attributo $A \in T$ non appare a destra di una dipendenza in F, allora A appartiene ad ogni chiave di R.
5. Sia $R\langle T, F\rangle$ uno schema relazionale. Dimostrare che se un attributo A di T appare a destra di qualche dipendenza in F, ma non appare a sinistra di alcuna dipendenza non banale, allora A non appartiene ad alcuna chiave.
6. Sia $R(A, B, C, D, E)$ uno schema di relazione con le dipendenze funzionali

$$F = \{AB \to CDE, AC \to BDE, B \to C, C \to B, C \to D, B \to E\}$$

   Si richiede di:
   a) portare F in forma canonica;
   b) determinare le possibili chiavi;
   c) mostrare che lo schema non è in terza forma normale;
   d) portare lo schema in terza forma normale.
7. Mostrare (*a*) che se uno schema relazionale è in BCNF allora è anche in 3NF e (*b*) che se uno schema non è in 3NF allora non è neanche in BCNF.
8. Si consideri l'insieme di attributi $T = ABCDEGH$ e l'insieme di dipendenze funzionali $F = \{AB \to C, AC \to B, AD \to E, B \to D, BC \to A, E \to G\}$. Per ognuno dei seguenti insiemi di attributi X, (*a*) trovare una copertura della proiezione di F su X, e (*b*) dire qual è la forma normale più forte soddisfatta da X:
   a) $X = ABC$;
   b) $X = ABCD$;
   c) $X = ABCEG$;
   d) $X = ABCH$;
   e) $X = ABCDEGH$.
9. Si consideri la relazione con schema $R(A, B, C, D)$ e si supponga che l'unica istanza possibile di R sia:

A	B	C	D
a1	b1	c1	d1
a1	b1	c2	d2
a2	b1	c1	d3
a2	b1	c3	d4

Si trovi una copertura canonica delle dipendenze funzionali soddisfatte da R.
Se lo schema non è in BCNF, si applichi l'algoritmo di decomposizione per trovare uno decomposizione in BCNF e dire se conserva le dipendenze.

10. Si consideri il seguente schema relazionale:

    Impiegati(Nome, Livello, Stipendio)

    per il quale valgono le seguenti dipendenze funzionali
    Nome $\rightarrow$ Livello, Stipendio
    Livello $\rightarrow$ Stipendio

    a) Lo schema è in 3NF o in BCNF?
    b) Se lo schema non è in 3NF, si applichi l'algoritmo di sintesi per trovare una decomposizione che preservi dati e dipendenze.
    c) Se lo schema non è in BCNF, si applichi l'algoritmo di decomposizione per trovare uno decomposizione in BCNF e dire se conserva le dipendenze.

11. Si supponga che per archiviare dati sull'inventario delle apparecchiature di un'azienda sia stata usata una tabella con la seguente struttura:

    Inventario(NInventario, Modello, Descrizione, NSerie, Costo, Responsabile, Telefono)

NInventario	Modello	Descrizione	NSerie	Costo	Responsabile	Telefono
...	...	...	...	...	...	...
111	SUN3	Stazione Sun	ajk0785	25 000	Caio	576
112	PB180	Notebook Mac	a908m	6000	Tizio	587
113	SUN3	Stazione Sun	ajp8907	27 000	Tizio	587
...	...	...	...	...	...	...

    Il numero di inventario identifica un'apparecchiatura. Un'apparecchiatura ha un costo, un modello e un numero di serie. Apparecchiature dello stesso modello possono avere costi differenti, perché acquistate in momenti diversi, ma hanno la stessa descrizione. Il numero di serie è una caratteristica dell'apparecchiatura e due diverse apparecchiature dello stesso modello hanno numero di serie diverso. Ogni apparecchiatura ha un responsabile, che può avere più apparecchiature, ma un unico numero di telefono. I responsabili sono identificati dal cognome.

Definire le dipendenze funzionali e dire se lo schema proposto presenta anomalie, giustificando la risposta.
Trasformare la rappresentazione in uno schema relazionale in 3NF.

12. Una palestra ospita diversi corsi appartenenti a diverse tipologie (aerobica, danza moderna, ...). Ogni corso ha una sigla, che lo identifica, un insegnante e alcuni allievi. Un insegnante offre in generale più corsi, anche con diverse tipologie, e anche un allievo può essere iscritto a più corsi. Di ogni insegnante interessano il nome (che lo identifica) e l'indirizzo. Di ogni allievo interessano il nome (che lo identifica) e il numero di telefono. Per ogni allievo interessa sapere, per ogni corso che frequenta, quanto ha già versato finora. La palestra gestisce attualmente i dati con un foglio elettronico con tante colonne quanti sono i fatti elementari da trattare. Si chiede di:

    a) Definire le dipendenze funzionali.
    b) Dare una copertura canonica delle dipendenze in tale schema ed elencare le chiavi.
    c) Applicare allo schema l'algoritmo di sintesi per portarlo in 3NF, e dire se lo schema così ottenuto è anche in BCNF.
    d) Applicare allo schema l'algoritmo di decomposizione per portarlo in BCNF, e dire se tale decomposizione preserva le dipendenze.

13. Quali di questi test ammettono algoritmi di complessità polinomiale:

    a) Dato lo schema di relazione $R\langle T, F\rangle$, $A \in T$, A è primo?
    b) Dati due insiemi di dipendenze F e G, $F \equiv G$?
    c) Dato lo schema di relazione $R\langle T, F\rangle$, e $X \subseteq T$, X è una superchiave?
    d) Dato lo schema di relazione $R\langle T, F\rangle$, e $X \subseteq T$, X è una chiave?

14. Dato lo schema $R\langle T, F\rangle$, discutere la complessità dei seguenti problemi:

    a) Trovare una chiave di R.
    b) Trovare tutte le chiavi di R.
    c) $F - \{X \to Y\} \equiv F$.

15. Discutere la complessità dei seguenti test:

    a) *Test 3NF*: dato lo schema di relazione $R\langle T, F\rangle$, R è in 3NF rispetto ad F?
    b) *Test BCNF*: dato lo schema di relazione $R\langle T, F\rangle$, R è in BCNF rispetto ad F?
    c) *Test BCNF di sottoschema*: dato lo schema di relazione $R\langle T, F\rangle$, dato $X \subseteq T$, X è in BCNF rispetto alla proiezione di F su X?
    d) *Test copertura canonica*: dato lo schema di relazione $R\langle T, F\rangle$, F è in forma canonica?

16. Si supponga che una dipendenza funzionale $X \to Y$ sia soddisfatta da un'istanza di relazione r. Sia $s \subseteq r$ (quindi s è una relazione con gli stessi attributi di r). s soddisfa $X \to Y$? Se sì, dire perché, altrimenti dare un controesempio.

17. Si supponga che una dipendenza funzionale $X \to Y$ sia soddisfatta da due istanze di relazione r ed s con gli stessi attributi. $r \cap s$ soddisfa $X \to Y$? $r \cup s$ soddisfa $X \to Y$? Se sì, dire perché, altrimenti dare un controesempio.
18. Sia $R\langle T, F\rangle$ uno schema relazionale, con F una copertura canonica. Si dimostri che se lo schema ha una sola chiave ed è in 3NF allora è anche in BCNF. (Suggerimento: si inizi con lo scrivere la definizione di 3NF e di BCNF, e si dia un nome, ad esempio Y, all'insieme di attributi che forma la sola chiave di $R\langle T, F\rangle$. Si ragioni per assurdo, supponendo che $R\langle T, F\rangle$ sia in 3NF ma non in BCNF).
19. Dimostrare il Teorema 5.9.
20. Dimostrare il Teorema 5.11.

Note bibliografiche

Un'introduzione alla teoria della normalizzazione è riportata in ogni libro sulle basi di dati. Una discussione approfondita della teoria può essere trovata in volumi specifici sull'argomento [Atzeni and Antonellis, 1993], [Maier, 1983], [Mannila and Räihä, 1992], [Abiteboul et al., 1995].

Capitolo 6

SQL PER L'USO INTERATTIVO DI BASI DI DATI

Il linguaggio SQL (*Structured Query Language*), sviluppato nel 1973 da ricercatori dell'IBM per il sistema relazionale System/R[1], è il linguaggio universale per la definizione e l'uso delle basi di dati relazionali.

La versione originaria si è differenziata in una serie di dialetti agli inizi degli anni '80, quindi il linguaggio è stato sottoposto ad una standardizzazione da parte dei comitati X/OPEN, ANSI e ISO. Dopo il primo standard ANSI e ISO del 1984 (SQL-84), rivisto poi nel 1989 per introdurre principalmente il vincolo d'integrità referenziale (SQL-89), si è giunti allo standard di fatto oggi molto diffuso proposto dal comitato X/OPEN per i sistemi operativi UNIX, mentre i comitati ANSI e ISO hanno proseguito il loro lavoro, con una serie di estensioni significative a SQL-89. Il primo risultato è lo standard SQL-92 (o SQL2), concluso nel 1992, e l'ultimo è SQL:2003 per trattare basi di dati relazionali ad oggetti.

Attualmente, lo standard SQL-92 è seguito solo ai livelli più semplici (in pratica nel DML) dai principali DBMS relazionali (come DB2, Oracle, e Microsoft SQL Server), mentre per aspetti relativi al DDL, anche significativi, vi sono differenze importanti.

Lo standard prevede tre diversi livelli di linguaggio, di complessità crescente: *Entry SQL*, *Intermediate SQL* e *Full SQL*. Inoltre, ogni sistema che aderisce allo standard deve fornire almeno i seguenti modi di usare SQL (detti anche *binding styles*):

- *Direct* SQL, per l'uso interattivo.
- *Embedded* SQL, per l'uso con linguaggi di programmazione per lo sviluppo di applicazioni.

La trattazione di SQL verrà suddivisa in tre parti.

- In questo capitolo vengono trattati gli aspetti del linguaggio per ricerche e modifiche fatte interattivamente (*DML*), chiamate generalmente interrogazioni o *query*.
- Nel prossimo capitolo verranno mostrati gli aspetti del linguaggio per la definizione dello schema e l'amministrazione della base di dati (*DDL*).

1. Il linguaggio si chiamava SEQUEL, acronimo delle parole "Structured English QUEry Language."

– Nel Capitolo 8 verrà discusso l'uso di SQL all'interno di linguaggi di programmazione.

6.1 Algebra relazionale su multinsiemi

Nella terminologia SQL una relazione è pensata come una *tabella* con tante colonne quanti sono gli attributi delle ennuple (dette anche *record*) e tante righe quante sono le ennuple della relazione. L'ordine degli attributi è *significativo*. Un componente di un'ennupla è detto *campo*.

Si noti che in generale una tabella nel linguaggio SQL non è un insieme, come visto nel capitolo precedente quando si è discusso il modello relazionale, ma un multinsieme.[2]

Come l'algebra relazionale, il linguaggio SQL prevede solo operatori su tabelle e una query esprime in forma dichiarativa un'espressione di operatori dell'algebra relazionale estesa su multinsiemi come segue:

– ***Proiezione con duplicati***: $\pi^{b}_{X}(O)$

  con X attributi di O. Il risultato è un multinsieme.

– ***Eliminazione di duplicati***: $\delta(O)$

  Il risultato è un insieme.

– ***Ordinamento***: $\tau_X(O)$

  con X attributi di O. Il risultato è un *multinsieme ordinato*, valore che non appartiene al dominio dell'algebra su multinsiemi e quindi τ può essere usato *solo* come radice di un albero logico.

– ***Unione, intersezione e differenza***: $O_1 \cup^{b} O_2$, $O_1 \cap^{b} O_2$, $O_1 -^{b} O_2$

  Se un elemento t appare n volte in O_1 e m volte in O_2, allora

  – t appare $n + m$ volte nel multinsieme $O_1 \cup^{b} O_2$:

  $$\{1,1,2,3\} \cup^{b} \{2,2,3,4\} = \{1,1,2,3,2,2,3,4\}$$

  – t appare $min(m,n)$ volte nel multinsieme $O_1 \cap^{b} O_2$:

  $$\{1,1,2,3\} \cap^{b} \{2,2,3,4\} = \{2,3\}$$

  – t appare $max(0, n-m)$ volte nel multinsieme $O_1 -^{b} O_2$:

  $$\{1,1,2,3\} -^{b} \{1,2,3,4\} = \{1\}$$

2. In effetti una tabella è un insieme solo se fra i suoi attributi vi è una chiave.

Alcune proprietà dell'algebra relazionale su insiemi non valgono nel caso dell'algebra relazionale su multinsiemi. Per esempio, la proprietà distributiva dell'unione e dell'intersezione su insiemi non vale nel caso di multinsiemi:

$$\{1\} \cap^{\mathbf{b}} (\{1\} \cup^{\mathbf{b}} \{1\}) = \{1\} \cap^{\mathbf{b}} \{1,1\} = \{1\}$$

mentre

$$(\{1\} \cap^{\mathbf{b}} \{1\}) \cup^{\mathbf{b}} (\{1\} \cap^{\mathbf{b}} \{1\}) = \{1\} \cup^{\mathbf{b}} \{1\} = \{1,1\}$$

Gli altri operatori dell'algebra relazionale su insiemi (selezione, raggruppamento, prodotto e giunzione) si generalizzano in modo ovvio al caso di multinsiemi.

6.2 Valori nulli

I sistemi commerciali prevedono l'uso di uno speciale valore, **NULL**, che fa parte di ogni tipo di dato e quindi assegnabile ad ogni attributo, a meno che non si imponga il vincolo d'integrità **NOT NULL** nella sua definizione, come vedremo nel prossimo capitolo.

Nel linguaggio SQL tutti gli operatori sono stati definiti in modo da poter operare anche in presenza di valori nulli e, quindi, la loro semantica presenta alcune complicazioni che ne rendono più difficile la comprensione.

Per questo motivo la presentazione degli operatori SQL per la ricerca dei dati è organizzata in due parti.

Nella prima parte, la prossima sezione, essi verranno presentati assumendo di operare su tabelle senza valori nulli e con operatori che non producono valori nulli.

Nella seconda parte, invece, descriveremo quali modifiche agli operatori descritti nella prima parte e quali nuovi operatori sono introdotti a causa di questi valori, insieme ad alcune indicazioni su come utilizzarli.

6.3 Operatori per la ricerca di dati

I comandi per la definizione delle tabelle verranno mostrati nel prossimo capitolo. In seguito negli esempi di interrogazioni faremo riferimento allo schema della base di dati di Figura 6.1.

Il comando principale dell'SQL è **SELECT** che, in una forma semplificata, ha la seguente struttura:

SELECT [**DISTINCT**] *Attributi*
FROM *GiunzioneDiTabelle*
[**WHERE** *Condizione*];

dove

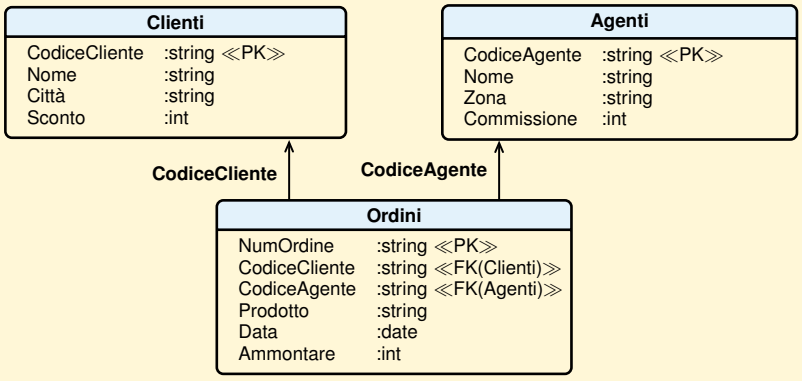


Figura 6.1: Rappresentazione grafica dello schema esempio

Attributi ::= $*$ | *Attributo* { , *Attributo* }
GiunzioneDiTabelle ::= *Tabella* [*Ide*] { **JOIN** *Tabella* [*Ide*] **ON** *CondizioneDiGiunzione* }

Il significato del comando **SELECT** può essere dato con le seguenti equivalenze:

SELECT $*$
FROM R_1 **JOIN** R_2 **ON** C_2
... **JOIN** R_n **ON** C_n
WHERE C; $\equiv \sigma_C (R_1 \underset{C_2}{\bowtie} R_2 \cdots \underset{C_n}{\bowtie} R_n)$

SELECT **DISTINCT** $A_1, \ldots, A_n$
FROM R_1 **JOIN** R_2 **ON** C_2
... **JOIN** R_n **ON** C_n
WHERE C ; $\equiv \delta(\pi^{\mathbf{b}}_{A_1,\ldots,A_n}(\sigma_C (R_1 \underset{C_2}{\bowtie} R_2 \cdots \underset{C_n}{\bowtie} R_n)))$

Il comando **SELECT** è quindi una combinazione di una giunzione, una restrizione e di una proiezione:

- con *Attributi* si specificano gli attributi del risultato; "$*$" sta per tutti gli attributi e **DISTINCT** è un'opzione per escludere i duplicati dal risultato;
- con *GiunzioneDiTabelle* si specificano le tabelle che intervengono nella giunzione con le relative condizioni di giunzione;
- con *Condizione* si specifica la condizione che deve essere soddisfatta dalle ennuple del risultato.

Le ricerche più semplici si fanno usando solo alcune clausole, come negli esempi seguenti.

- Singola tabella:

```
SELECT  *
FROM    Agenti;
```

– Restrizione:

```
SELECT  *
FROM    Ordini
WHERE   Ammontare > 1000;
```

– Proiezione:

```
SELECT  DISTINCT Nome, Città
FROM    Clienti;
```

– Giunzione:

```
SELECT  *
FROM    Clienti
        JOIN Ordini ON Clienti.CodiceCliente = Ordini.CodiceCliente;
```

Si noti che:

1. l'operatore **SELECT** può restituire un multinsieme se non viene specificata l'opzione **DISTINCT**, come nell'esempio seguente, dove il risultato è una tabella con righe duplicate se almeno un agente ha fatto più ordini per lo stesso ammontare:

   ```
   SELECT  CodiceAgente, Ammontare
   FROM    Ordini;
   ```

   Per non avere duplicati nel risultato occorre porre **SELECT DISTINCT**.
2. Per evitare ambiguità quando si opera sulla giunzione di tabelle con gli stessi attributi, si usa la notazione con il punto Tabella.Attributo, come nel seguente esempio, dove è usata uniformemente per trovare i codici dei clienti e l'ammontare degli ordini fatti:

   ```
   SELECT  Clienti.CodiceCliente, Ordini.Ammontare
   FROM    Clienti
           JOIN Ordini ON Clienti.CodiceCliente = Ordini.CodiceCliente;
   ```

   La condizione che segue la parola chiave **ON** è detta *condizione di giunzione*. Si noti che una sintassi datata in SQL prevedeva di elencare le tabelle separate da virgola nella clausola **FROM** e di mettere le condizioni di giunzioni in **AND** nella clausola **WHERE**. La virgola rappresenta il prodotto, che nella sintassi moderna viene indicato con **CROSS JOIN**. Vedremo più avanti questo e altri operatori per vari tipi di giunzione.
3. Una notazione alternativa alla precedente prevede l'associazione di identificatori alle tabelle, da usare per specificare gli attributi nella notazione Identificatore.Attributo. Questi identificatori sono detti *variabili di correlazione*, *pseudonimi* o *alias*. Così l'esempio precedente può essere scritto come:

```
SELECT   C.CodiceCliente, O.Ammontare
FROM     Clienti C
         JOIN Ordini O ON C.CodiceCliente = O.CodiceCliente;
```

Questa seconda notazione è indispensabile nei casi in cui si debba fare la giunzione di una tabella con se stessa, come nel seguente esempio:

```
SELECT   A2.Nome
FROM     Agenti A1
         JOIN Agenti A2 ON A2.Zona = A1.Zona
WHERE    A1.CodiceAgente = 'A01';
```

che restituisce i nomi degli agenti della stessa zona dell'agente con codice 'A01'.

Vediamo adesso in dettaglio le varie componenti del comando.

6.3.1 La clausola SELECT

Il risultato di un'espressione **SELECT** *Attributi* **FROM** ... è una tabella (a cui è possibile dare un nome con il comando **CREATE TABLE**, come vedremo nel prossimo capitolo), i cui nomi di colonna sono quelli indicati in *Attributi*. La sintassi completa della clausola *Attributi* è la seguente:[3]

Attributi ::= $*$ | *Espr* [[**AS**] *NuovoNome*] {, *Espr* [[**AS**] *NuovoNome*] }
Espr ::= [*Ide*.]*Attributo* |
Costante |
"(" *Espr* ")" |
[−] *Espr* [ρ *Espr*] |
(**SUM** | **COUNT** | **AVG** | **MAX** | **MIN**)
"(" [**DISTINCT**] [*Ide*.]*Attributo* ")" |
COUNT "(" $*$ ")"

ρ ::= (+ | − | $*$ | /)

NuovoNome ::= *Identificatore*

[**AS**] *NuovoNome* si usa per cambiare il nome delle colonne del risultato, come in:

```
SELECT   CodiceCliente AS Codice, Nome AS NomeECognome
FROM     Clienti;
```

che restituisce una tabella ottenuta proiettando Clienti su CodiceCliente e Nome e ridenominando le due colonne con Codice e NomeECognome rispettivamente.

3. In realtà i sistemi commerciali prevedono espressioni con un numero maggiore di operatori.

Con il comando **SELECT** si possono calcolare tabelle le cui colonne non corrispondono a colonne delle tabelle selezionate, ma sono ottenute come espressioni a partire da attributi e costanti.

Ad esempio:

```
SELECT   CodiceAgente, Nome,
         Commissione / 100 AS Commissione,
         'Italia' AS Paese
FROM     Agenti;
```

restituisce una tabella con tante righe quanti sono gli agenti, e con quattro colonne, due ottenute per proiezione dalla tabella Agenti, una con valori calcolati da un'altra colonna, ed una formata da valori costanti.

6.3.2 La clausola FROM

Come già detto, nella parte che segue il **FROM** vi può essere una tabella, su cui fare una restrizione o una proiezione, oppure una serie di tabelle combinate attraverso un operatore di giunzione sul cui risultato viene fatto una restrizione o una proiezione.

Al posto di una tabella si può usare un'espressione **SELECT**, ma questa possibilità non verrà considerata per semplicità. Un'espressione **SELECT** può invece apparire nella condizione, come vedremo, e viene detta *SottoSelect* o **SELECT** *annidata*. Quando daremo la sintassi completa del **SELECT**, si vedrà che in una *SottoSelect* non si possono usare tutte le opzioni previste per una **SELECT**, ma per ora queste limitazioni si possono ignorare.

Nello standard SQL-92 sono stati introdotti vari tipi di giunzione, comprese le giunzioni esterne, oltre al prodotto. La sintassi dell'argomento *TabelleDiGiunzione* della clausola **FROM** diventa così la seguente:

TabelleDiGiunzione ::= Tabella [*Ide*]
| "(" *TabelleDiGiunzione* ")"
| Tabella [*Ide*] { *Giunzione Tabella* [*Ide*]
[**USING** "(" *Attributo* {, *Attributo* } ")" | **ON** *Condizione*] }

Giunzione ::= , | [**CROSS** | **NATURAL**] **JOIN**

dove le clausole **USING** e **ON** si possono usare solo con l'operatore **JOIN**. Il significato è il seguente:

1. **CROSS JOIN** e "," corrispondono all'operatore di prodotto cartesiano; nella notazione moderna si preferisce l'uso dell'operatore **CROSS JOIN**;
2. **NATURAL JOIN** è la giunzione naturale;
3. **JOIN** ... **USING** è la giunzione sui valori uguali degli attributi specificati e presenti in entrambe le tabelle, in generale un sottoinsieme degli attributi con lo stesso nome presenti in entrambe le tabelle;

4. **JOIN** ... **ON** è la giunzione sui valori degli attributi che soddisfano una generica condizione;

Vediamo degli esempi d'uso dei nuovi operatori:

– *Giunzione naturale*: per trovare i codici dei clienti e l'ammontare degli ordini fatti, si pone:

```
SELECT  Clienti.CodiceCliente, Ordini.Ammontare
FROM    Clienti
        NATURAL JOIN Ordini;
```

in alternativa, si può usare la forma **USING**:

```
SELECT  Clienti.CodiceCliente, Ordini.Ammontare
FROM    Clienti
        JOIN Ordini USING (CodiceCliente);
```

– *Giunzione*: per trovare gli ordini degli agenti, si pone:

```
SELECT  Agenti.CodiceAgente, Ordini.Ammontare
FROM    Agenti
        JOIN Ordini ON Agenti.CodiceAgente = Ordini.CodiceAgente;
```

6.3.3 La clausola WHERE

Come già detto, nella parte che segue il **WHERE** viene specificata la condizione che deve essere soddisfatta dalle ennuple da elaborare.

Una condizione di ricerca è definita ricorsivamente come segue:

Condizione ::= *Predicato* |
"(" *Condizione* ")" |
NOT *Condizione* |
Condizione (**AND** | **OR**) *Condizione*

con **AND**, **OR** e **NOT** i classici connettivi logici per esprimere condizioni complesse.

SQL prevede anche i seguenti predicati:

1. *Espr* θ (*Espr* | "(" *SottoSelect* ")")

   con θ un operatore di confronto dell'insieme $\{=, <>, >, >=, <, <=\}$. La *SottoSelect* a destra è ammessa solo se la tabella risultante contiene un solo valore. Ad esempio, per trovare il codice degli agenti con commissione uguale a quella dell'agente con codice 'A01', evitando che il suo codice appaia nel risultato, si pone:

```
SELECT   CodiceAgente
FROM     Agenti
WHERE    NOT (CodiceAgente = 'A01')
         AND Commissione = (
             SELECT   Commissione
             FROM     Agenti
             WHERE    CodiceAgente = 'A01');
```

2. *Espr*$_1$ **BETWEEN** *Espr*$_2$ **AND** *Espr*$_3$

   Il predicato è equivalente alla condizione:

   (*Espr*$_2$ $<=$ *Espr*$_1$ **AND** *Espr*$_1$ $<=$ *Espr*$_3$)

3. *Attributo* **LIKE** *Stringa*

   È un predicato per confrontare stringhe. *Stringa* può contenere i seguenti caratteri speciali:

   a) "_" che sta per qualsiasi carattere;
   b) "%" che sta per una qualsiasi sequenza di caratteri, anche vuota.

4. **EXISTS** "(" *SottoSelect* ")"

   Il predicato **EXISTS** è vero se la *SottoSelect* ritorna una relazione con almeno un'ennupla, falso se la relazione restituita è la relazione vuota.
   Ad esempio, per trovare i nomi dei clienti che hanno fatto acquisti dall'agente con codice 'A01', si pone:

```
SELECT   C.Nome
FROM     Clienti C
WHERE    EXISTS (
             SELECT   *
             FROM     Ordini O
                      JOIN Agenti A ON O.CodiceAgente = A.CodiceAgente
             WHERE    O.CodiceCliente = C.CodiceCliente
                      AND A.CodiceAgente = 'A01');
```

   Si noti che in una *SottoSelect* si può usare la variabile di correlazione della **SELECT** più esterna, ma non si può fare il contrario.
   Il predicato **EXISTS** è bene usarlo quando è strettamente necessario e non in espressioni che possono usare la giunzione, sfruttando la seguente equivalenza:

```
SELECT   R1.A1,..., R1.An
FROM     R1
WHERE    [Condizione C1 su R1 AND ]
         EXISTS ( SELECT   *
                  FROM     R2
                  WHERE    Condizione C2 su R2 e R1 [AND Condizione C3 su R2] );
```

è equivalente a[4]:

```
SELECT   DISTINCT R1.A1,..., R1.An
FROM     R1 JOIN R2 ON Condizione C2 su R1 e R2
WHERE    [Condizione C1 su R1 AND]
         [Condizione C3 su R2];
```

Ad esempio, per la query precedente si può scrivere:

```
SELECT   DISTINCT C.Nome
FROM     Clienti C
         JOIN Ordini O ON O.CodiceCliente = C.CodiceCliente
         JOIN Agenti A ON O.CodiceAgente = A.CodiceAgente
WHERE    A.CodiceAgente = 'A01');
```

Questa seconda formulazione è preferibile perché in generale i sistemi relazionali prevedono un *ottimizzatore* delle interrogazioni che ha più possibilità di ottimizzare l'esecuzione di una giunzione che un'espressione con *SottoSelect*. Solo in alcuni sistemi invece l'ottimizzatore è anche in grado di riconoscere che l'interrogazione con il predicato **EXISTS** può essere trasformata innanzitutto nell'espressione con la giunzione prima di procedere con gli altri passi di ottimizzazione.

5. *Espr* **IN** ("(" *Valore* {, *Valore* } ")" | "(" *SottoSelect* ")")

È un predicato per controllare se un valore di *Espr* appartiene ad un insieme definito per enumerazione o come risultato di una *SottoSelect*. *Espr* è un valore numerico o una stringa. Il predicato vale **TRUE** se il valore è fra quelli elencati o quelli prodotti dalla *SottoSelect* e falso altrimenti. Ad esempio, per trovare i codici degli agenti della zona di Pisa, Firenze o Siena si pone:

```
SELECT   CodiceAgente
FROM     Agenti
WHERE    Zona IN ('Pisa', 'Firenze', 'Siena');
```

Si noti che per questo operatore, come per **BETWEEN**, **LIKE** e **EXISTS** la negazione, oltre che con la sintassi **NOT** (*Predicato*), si ottiene anche premettendo **NOT** al nome dell'operatore. Ad esempio per trovare i codici di tutti gli agenti tranne quelli della zona di Pisa e Firenze si pone:

```
SELECT   CodiceAgente
FROM     Agenti
WHERE    Zona NOT IN ('Pisa', 'Firenze');
```

4. L'equivalenza vale sole se R1.A1,..., R1.An sono una superchiave per R1.

Nel caso dell'uso con una *SottoSelect*, il predicato **IN** è equivalente ad **EXISTS** come si può vedere dal seguente esempio, in cui si cercano i nomi dei clienti con ordini per più di 1000 euro:

```
SELECT   Nome
FROM     Clienti
WHERE    CodiceCliente IN (
             SELECT   CodiceCliente
             FROM     Ordini
             WHERE    Ammontare > 1000);
```

equivalente a:

```
SELECT   Nome
FROM     Clienti C
WHERE    EXISTS (
             SELECT   *
             FROM     Ordini O
             WHERE    Ammontare > 1000
                      AND C.CodiceCliente = O.CodiceCliente);
```

Una trasformazione equivalente con **NOT EXISTS** vale per il **NOT IN**.
Per le considerazioni precedenti su **EXISTS** anche in questo caso è bene riscrivere l'interrogazione usando una giunzione.

6. *Espr* θ(**ALL** | **ANY**) "(" *SottoSelect* ")"

Il predicato "θ **ALL**" (con θ un operatore di confronto), è vero se il primo operando sta nella relazione specificata con ogni elemento del secondo operando, che deve essere un insieme risultato di una **SELECT**. Se l'insieme è vuoto il predicato è vero.
Il predicato "θ **ANY**", è vero se il primo operando sta nella relazione specificata con almeno un elemento del secondo operando. Se l'insieme è vuoto, il predicato è falso.
Si noti che "*Espr* **IN** (*SottoSelect*)" equivale a "*Espr* =**ANY** (*SottoSelect*)", mentre "*Espr* **NOT IN** (*SottoSelect*)" equivale a "*Espr* <>**ALL** (*SottoSelect*)".
Ad esempio, per trovare il codice degli agenti la cui commissione supera quella di ogni agente della zona di Pisa, si pone:

```
SELECT   CodiceAgente
FROM     Agenti
WHERE    Commissione >ALL(
             SELECT   Commissione
             FROM     Agenti
             WHERE    Zona = 'Pisa');
```

Condizioni con quantificatore esistenziale

Immaginiamo di voler trovare i nomi dei clienti che hanno fatto *almeno* un ordine per più di 1000 euro. Se esistesse in SQL il quantificatore esistenziale, come previsto nell'SQL:2003, estensione dell'SQL per sistemi relazionali ad oggetti, e definito come segue:

FOR SOME x **IN** S [**WHERE** C1(x)] : C2(x)

che è vero se esiste almeno un elemento di S [**WHERE** C1(x)] che soddisfa C2(x), si porrebbe:

```
SELECT  Nome
FROM    Clienti C
WHERE   FOR SOME O IN Ordini WHERE O.CodiceCliente = C.CodiceCliente :
              Ammontare > 1000;
```

Questo tipo di interrogazione si può esprimere in SQL in vari modi, vediamo quelli più comuni.

– si usa una giunzione:

```
SELECT  DISTINCT Nome
FROM    Clienti C
        JOIN Ordini O ON O.CodiceCliente = C.CodiceCliente
WHERE   Ammontare > 1000;
```

– si usa il predicato **EXISTS**:

```
SELECT  Nome
FROM    Clienti C
WHERE   EXISTS (
            SELECT  *
            FROM    Ordini O
            WHERE   O.CodiceCliente = C.CodiceCliente
                    AND Ammontare > 1000);
```

Condizioni con quantificatore universale

Immaginiamo di voler trovare il codice dei clienti che hanno fatto ordini da *tutti* gli agenti della zona Pisa. Questo tipo di interrogazioni sono fra quelle più difficili da scrivere in SQL e si mostra un metodo per farlo, anche se il risultato finale non è l'unico possibile. Se esistesse in SQL il quantificatore universale definito come segue:

FOR ALL x **IN** S [**WHERE** C1(x)] : C2(x)

che è vero se ogni elemento di S [**WHERE** C1(x)] soddisfa C2(x), l'interrogazione si potrebbe esprimere nella forma "trovare il codice dei clienti C tali che per ogni agente A della zona Pisa esiste un ordine che riguarda C e A" ponendo:

```
SELECT  CodiceCliente
FROM    Clienti C
WHERE   FOR ALL A IN Agenti WHERE Zona = 'Pisa' :
            FOR SOME O IN Ordini WHERE A.CodiceAgente = O.CodiceAgente :
                O.CodiceCliente = C.CodiceCliente;
```

Purtroppo in SQL non esiste l'operatore **FOR ALL**, e i predicati del tipo =**ALL** o =**ANY** non consentono in generale di formulare interrogazioni di questo tipo.

Per risolvere il problema, una soluzione può essere trovata ricordando che vale la tautologia:

$$\forall x \in P.Q(x) \Leftrightarrow \neg\exists x \in P.\neg Q(x)$$

da cui discende:

$$\forall x \in A(\exists y \in B(x).p(x,y)) \Leftrightarrow \neg\exists x \in A\neg(\exists y \in B(x).p(x,y))$$

In parole, la relazione significa che sono equivalenti le due affermazioni: (1) per ogni x esiste un y tale che $p(x,y)$ è vero e (2) non esiste un x tale che non esiste un y con $p(x,y)$ vero.

Pertanto, l'interrogazione iniziale si può formulare in modo equivalente con una doppia negazione usando il quantificatore esistenziale: trovare il codice dei clienti C per i quali non esiste un agente A della zona Pisa per il quale non esiste un ordine che riguarda C e A:

```
SELECT  CodiceCliente
FROM    Clienti C
WHERE   NOT FOR SOME A IN Agenti WHERE Zona = 'Pisa' :
            NOT FOR SOME O IN Ordini WHERE A.CodiceAgente = O.CodiceAgente :
                O.CodiceCliente = C.CodiceCliente;
```

A questo punto l'interrogazione si può esprimere in SQL usando il predicato **EXISTS**:

```
SELECT  C.CodiceCliente
FROM    Clienti C
WHERE   NOT EXISTS (
            SELECT  *
            FROM    Agenti A
            WHERE   A.Zona = 'Pisa'
                    AND NOT EXISTS (
                        SELECT  *
                        FROM    Ordini O
                        WHERE   A.CodiceAgente = O.CodiceAgente
                                AND O.CodiceCliente = C.CodiceCliente));
```

Si noti che se non ci sono agenti della zona Pisa, l'interrogazione ritorna i codici di tutti i clienti, perché la quantificazione universale **FOR ALL** x **IN** S [**WHERE** C1(x)]: C2(x)

è vera se l'insieme S è vuoto. Per escludere questo caso occorre completare l'interrogazione con una quantificazione esistenziale come segue:

```
SELECT  C.CodiceCliente
FROM    Clienti C
WHERE   NOT EXISTS(
            SELECT  *
            FROM    Agenti A
            WHERE   A.Zona = 'Pisa'
                    AND NOT EXISTS (
                        SELECT  *
                        FROM    Ordini O
                        WHERE   A.CodiceAgente = O.CodiceAgente
                                AND O.CodiceCliente = C.CodiceCliente))
        AND EXISTS (
            SELECT  *
            FROM    Agenti A
            WHERE   A.Zona = 'Pisa');
```

6.3.4 Clausola di ordinamento

ORDER BY *Attributo* [**DESC**] {, *Attributo* [**DESC**]}

Questa specifica va posta dopo l'eventuale **WHERE**. Il risultato del **SELECT** viene ordinato in base al valore di certi attributi, in ordine discendente specificando **DESC**, altrimenti in ordine ascendente. Ad esempio:

```
SELECT      CodiceAgente, Commissione
FROM        Agenti
WHERE       Zona = 'Pisa'
ORDER BY    Commissione DESC;
```

6.3.5 Funzioni di aggregazione

Nella clausola **SELECT** si possono usare anche le funzioni predefinite **MAX**, **MIN**, **COUNT**, **AVG**, **SUM**, dette *funzioni di aggregazione* o *funzioni statistiche*. Esse operano su un insieme di valori restituendo rispettivamente il massimo valore, il minimo, o il numero dei valori, il valore medio e la somma (solo per valori numerici). Per semplicità, assumeremo che i valori siano quelli di una colonna, ma in generale possono essere calcolati con un'espressione aritmetica da quelli di più colonne.

COUNT(**DISTINCT** *Attributo*) restituisce il numero dei valori diversi di una colonna, mentre **COUNT**(∗) restituisce il numero di ennuple del risultato.

```
SELECT   MIN(Ammontare), MAX(Ammontare), AVG(Ammontare)
FROM     Ordini
WHERE    Data = '01032019';
```

restituisce il valore minimo, massimo e medio dell'ammontare degli ordini effettuati in data 1/3/2019, mentre

```
SELECT   COUNT(*)
FROM     Ordini
WHERE    CodiceAgente = 'A01';
```

restituisce il numero degli ordini dell'agente con codice 'A01'.

Quando si usano le funzioni di aggregazione nella **SELECT**, il risultato deve essere un'unica ennupla di valori e quindi si possono usare più funzioni di aggregazione, ma non si possono usare funzioni di aggregazione e attributi, se non nel caso descritto nella prossima sezione. Ad esempio, è scorretto scrivere:

```
SELECT   NumOrdine, MIN(Ammontare), MAX(Ammontare)
FROM     Ordini
WHERE    Data = '01032019';
```

6.3.6 Operatore di raggruppamento

Il comando **SELECT** con il **GROUP BY** ha la seguente struttura:

```
SELECT      DISTINCT S_A, S_FA
FROM        T
WHERE       W_C
GROUP BY    G_A
HAVING      H_C
ORDER BY    O_A;
```

dove

- S_A sono gli attributi e S_{FA} sono le funzioni di aggregazione della clausola **SELECT**;
- T sono le tabelle della clausola **FROM**;
- W_C è la condizione della clausola **WHERE**;
- G_A sono gli attributi di raggruppamento della clausola **GROUP BY**, con $S_A \subseteq G_A$;
- H_C è la condizione della clausola **HAVING** con le funzioni di aggregazione H_{FA};
- O_A sono gli attributi di ordinamento della clausola **ORDER BY**;
- le clausole **DISTINCT**, **WHERE**, **HAVING** e **ORDER BY** sono opzionali.

In Figura 6.2 è mostrato il significato del comando **SELECT** con il **GROUP BY**, e l'uso di due tabelle R e S in giunzione, con un albero logico dell'algebra relazionale su multinsiemi.

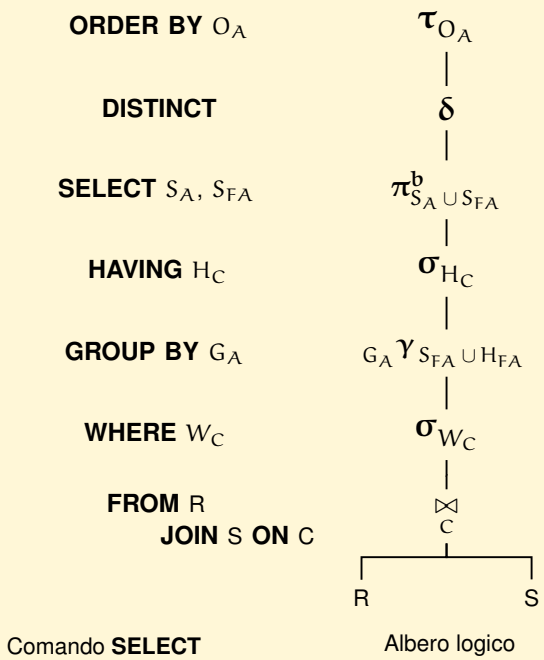


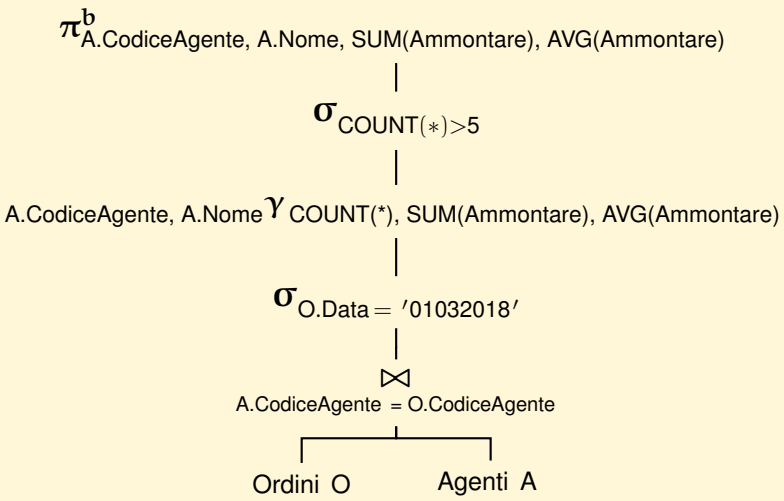
Figura 6.2: Significato del comando **SELECT** con il **GROUP BY**

Ad esempio, per trovare il codice e nome degli agenti con più di cinque ordini in data 1/3/2019 e, degli ordini fatti, il totale e la media dell'ammontare, si pone:

```
SELECT    A.CodiceAgente, A.Nome,
          SUM(Ammontare), AVG(Ammontare)
FROM      Ordini O
          JOIN Agenti A ON A.CodiceAgente = O.CodiceAgente
WHERE     O.Data = '01032019'
GROUP BY  O.CodiceAgente, A.Nome
HAVING    COUNT(*) > 5;
```

Si osservi che due agenti con lo stesso codice hanno anche lo stesso nome, per cui raggruppare rispetto a CodiceAgente, Nome equivale a raggruppare rispetto al solo codice. Tuttavia, se si raggruppasse solo rispetto al codice, il sistema non permetterebbe poi di selezionare il campo Nome.

Il significato del comando **SELECT** è dato dal seguente albero logico:



6.3.7 Operatori insiemistici

UNION, **INTERSECT** ed **EXCEPT** sono equivalenti agli operatori insiemistici dell'algebra relazionale $\cup$, $-$ e $\cap$, che ritornano *insiemi* anche se gli argomenti contengono duplicati.

Gli argomenti sono espressioni **SELECT** con risultato tabelle che hanno lo stesso numero di attributi, anche con nomi diversi, ma con tipi compatibili (tutti i tipi stringhe sono compatibili fra loro, così come tutti i tipi numerici).

Il risultato ha gli attributi della prima relazione.

Per esempio, date due **SELECT** che restituiscono due relazioni con tipo $(A_1 : T_1, A_2 : T_2)$ e $(B_1 : T_1, B_2 : T_2)$ rispettivamente, il risultato è una relazione di tipo $(A_1 : T_1, A_2 : T_2)$ e ennuple:

- per **UNION** quelli che sono solo in R o solo in S; se una ennupla è sia in R che in S l'unione ne contiene solo una perché un insieme non contiene duplicati;
- per **EXCEPT** quelle che sono in R ma non in S;
- per **INTERSECT** quelle sono sia in R che in S.

Un operatore insiemistico seguito dalla parola chiave **ALL** (ad esempio **UNION ALL**) non elimina i duplicati dal risultato come i corrispondenti operatori $\cup^{b}$, $-^{b}$ e $\cap^{b}$ dell'algebra su multinsiemi definiti in Sezione 6.1.

Vediamo alcuni esempi:

```
SELECT   CodiceAgente
FROM     Agenti
    EXCEPT
SELECT   CodiceAgente
FROM     Ordini;
```

Si noti che il secondo **SELECT** può restituire un multiinsieme se uno o più agenti hanno prodotto più ordini. In ogni caso il risultato è un insieme, dato che si usa la versione della differenza insiemistica senza duplicati.

La seguente interrogazione restituisce i nomi dei clienti e degli agenti.

```
SELECT   Nome
FROM     Clienti
    UNION ALL
SELECT   Nome
FROM     Agenti;
```

In questo caso sono possibili duplicati, ad esempio se più clienti o agenti hanno lo stesso nome, oppure se un cliente ha un nome uguale a quello di un agente.

6.3.8 Sintassi completa del SELECT

Come riepilogo, diamo la sintassi del **SELECT** per come è stato presentato finora:

Select ::= *SelectEspr*
[**ORDER BY** *Attributo* [**DESC**] {, *Attributo* [**DESC**]}]

SelectEspr ::= *BloccoSelect* |
SelectEspr OperatoreInsiemistico SelectEspr |
"(" SelectEspr ")"

dove *BloccoSelect* è definita come:

BloccoSelect ::=
SELECT [**DISTINCT**]
(∗ | *Espr* [[**AS**] *NuovoNome*] {, *Espr* [[**AS**] *NuovoNome*] })
FROM *Tabella* [*Ide*] { **JOIN** *Tabella* [*Ide*] **ON** *Condizione* }
[**WHERE** *Condizione*]
[**GROUP BY** *Attributo* {, *Attributo*} [**HAVING** *Condizione*]]

Si noti che in un *BloccoSelect* non si può usare **ORDER BY**.

Una condizione ha la seguente sintassi:

Condizione ::= *Predicato* |
"(" Condizione ")" |

NOT *Condizione* |
Condizione (**AND** | **OR**) *Condizione*

Predicato ::= *Espr* [**NOT**] **IN** “(” *SottoSelect* “)” |
Espr [**NOT**] **IN** “(” *Valore* {,*Valore*} “)” |
Espr θ (*Espr* | “(” *SelectEspr* “)”) |
Espr **IS** [**NOT**] NULL |
Espr θ(**ANY** | **ALL**) “(” *SelectEspr* “)” |
[**NOT**] **EXISTS**“(” *SelectEspr* “)” |
Espr [**NOT**] **BETWEEN** *Espr* **AND** *Espr* |
Espr [**NOT**] **LIKE** *Stringa*

θ ::= $<$ | $<=$ | $=$ | $<>$ | $>$ | $>=$

Un’espressione *Espr* è definita dalla seguente grammatica:

Espr ::= [*Ide.*]*Attributo* |
Costante |
“(” *Espr* “)” |
[−] *Espr* [ρ *Espr*] |
(**SUM** | **COUNT** | **AVG** | **MAX** | **MIN**)
“(” [**DISTINCT**] [*Ide.*]*Attributo* “)” |
COUNT “(” $*$ “)”

ρ ::= ($+$ | $-$ | $*$ | $/$)

Infine, una tabella è data dalla seguente grammatica:

Tabella ::= *Ide* |
Tabella Giunzione Tabella
[**USING** "(" *Attributo* { , *Attributo* } ")" | **ON** *Condizione*]

Giunzione ::= "," | [(**CROSS** | **NATURAL**)] **JOIN**

OperatoreInsiemistico ::=
(**UNION** [**ALL**] | **INTERSECT** [**ALL**] | **EXCEPT** [**ALL**])

Si noti che la sintassi permette di scrivere delle **SELECT** che non sono ammesse nel linguaggio. Un'interrogazione valida deve infatti soddisfare almeno i seguenti vincoli:

- le funzioni di aggregazione non possono essere usate nei predicati della clausola **WHERE**;
- in un'interrogazione senza **GROUP BY**, tutti gli attributi nella clausola **SELECT** sono argomento di una funzione di aggregazione oppure nessuna funzione di aggregazione può apparire nella clausola;
- ricordiamo infine che in un'interrogazione con **GROUP BY**, tutti gli attributi che appaiono nelle clausole **HAVING** e **SELECT**, non come argomento di una funzione di aggregazione, sono attributi di raggruppamento.

6.4 Il valore NULL

In SQL i valori nulli sono rappresentati dal valore speciale **NULL** che può essere il valore di un attributo di qualunque tipo, a meno che non sia presente il vincolo **NOT NULL** nella definizione dell'attributo.

La presenza di questo valore complica il linguaggio SQL, e limita alcune importanti ottimizzazioni. In questa sezione vediamo le principali conseguenze della presenza del valore **NULL**, rimandando ad un manuale del linguaggio per approfondimenti sull'argomento.

In seguito negli esempi di interrogazioni faremo riferimento allo schema della base di dati di Figura 6.3 in cui l'attributo Supervisore di Agenti rappresenta una relazione gerarchica su Agenti con diretta univoca e parziale. Si noti che, come descritto nel Capitolo 4, la freccia che rappresenta il collegamento attraverso la chiave esterna ha un taglio per indicare che l'attributo può avere valore **NULL** .

6.4.1 Operazioni con valori NULL

Operatori aritmetici e di confronto su dati elementari

Gli operatori aritmetici e di confronto sui dati elementari quando applicati ad almeno un operando nullo non provocano errore, ma restituiscono il valore **NULL**. Si noti che questo è vero anche per l'operatore di uguaglianza, quindi anche se si confronta **NULL** con sé stesso il risultato è **NULL**.

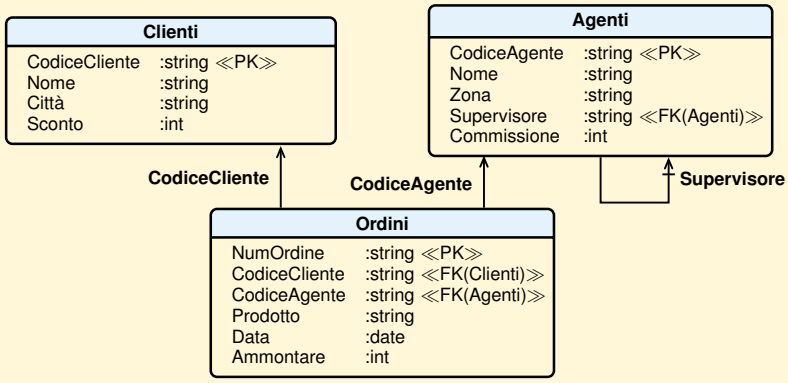


Figura 6.3: Rappresentazione grafica dello schema esempio con valori nulli

Si noti che nello standard, in questo caso seguito solo da alcuni DBMS, gli operatori di confronto in presenza di valore **NULL** producono in realtà lo speciale valore booleano **UNKNOWN**, che può essere considerato come un tipo particolare di valore nullo. Infatti, quando tale valore è il risultato di una espressione usata nella clausola **SELECT**, viene convertito a **NULL**. Per questo motivo, e dato che questo valore non è presente in tutti i sistemi, in questo testo lo consideriamo equivalente al valore **NULL**.

Operatori booleani

Gli operatori booleani seguono la regola precedente con le seguenti eccezioni:

- L'operatore **OR** quando ha un operando **TRUE** restituisce sempre **TRUE**;
- **AND** con un operando **FALSE** restituisce **FALSE**;
- gli operatori logici che accettano un insieme come operando, come gli operatori **IN**, "θ **ALL**" e "θ **ANY**", se l'insieme contiene almeno un valore **NULL**, hanno un comportamento analogo a quello di **AND** e **OR**: se l'operatore può restituire un risultato certo, come **IN** o "θ **ANY**" quando trovano un valore che soddisfa la condizione restituiscono **TRUE**, altrimenti restituiscono il valore **NULL** (o, come già detto, **UNKNOWN** per i sistemi che lo prevedono).

In ogni caso la cosa importante da notare è che ai fini del recupero delle informazioni, cioè quano la condizione è usata nelle clausole **WHERE** and **HAVING**, il valore **NULL** è equivalente al valore **FALSE** (e quindi i dati non vengono recuperati).

Operatori di aggregazione

Le funzioni di aggregazione **MAX**(A), **MIN**(A), **AVG**(A), **SUM**(A) e **COUNT**(A) ignorano i valori **NULL**. Se una colonna A contiene solo valori **NULL** le funzioni **MAX**(A), **MIN**(A), **AVG**(A), **SUM**(A) resituiscono **NULL**, mentre **COUNT**(A) vale zero.

COUNT(**DISTINCT** A) restituisce il numero dei valori diversi di una colonna, esclusi i valori **NULL**, mentre **COUNT**(∗) restituisce il numero di ennuple del risultato. Si noti quindi che in presenza di valori **NULL**, **AVG**(A) è uguale a **SUM**(A)/**COUNT**(A) ma è diverso da **SUM**(A)/**COUNT**(∗).

GROUP BY e operatori insiemistici

Se negli attributi di raggruppamento esistono valori **NULL**, questi sono considerati indistinti ai fini del raggruppamento e quindi costituiscono un unico gruppo.

Analogamente, gli operatori insiemistici che eliminano i duplicati eliminano anche le eventuali duplicazioni del valore **NULL**.

Operatori specifici per i valori NULL

1. *Espr* **IS NULL**

   per controllare se un'espressione ha valore **NULL**, ritorna sempre **TRUE** oppure **FALSE**. Ad esempio, assumendo che alcuni agenti non hanno supervisore, quindi che per loro l'attributo Supervisore è **NULL**, per trovare i loro codici si pone:

   ```
   SELECT  CodiceAgente AS CodiceSupervisoreAltoLivello
   FROM    Agenti
   WHERE   Supervisore IS NULL;
   ```

2. $Espr_1$ **IS DISTINCT FROM** $Expr_2$

   Questo predicato, diversamente da <>, ritorna **FALSE** non solo quando i due valori sono uguali, ma anche quando sono entrambi **NULL**. Se i due valori sono diversi, o uno solo dei due è **NULL**, ritorna **TRUE**[5].

   Si possono usare **IS NOT NULL**, e **IS NOT DISTINCT FROM** come abbreviazioni della forma negativa.

3. **COALESCE** "(" $Espr_1, \ldots Espr_n$ ")" .

   Viene usato per trasformare un valore **NULL** in un valore non nullo. Valuta le espressioni in sequenza, da sinistra verso destra. Viene restituito il primo valore trovato diverso da **NULL**. L'operatore ritorna **NULL** se tutte le espressioni hanno valore **NULL**.

Giunzioni esterne

Oltre alle forme di giunzione riportate precedentemente, sono disponibili anche le giunzioni **esterne** (**outer join**), che possono restituire ennuple con valori **NULL**.

5. Questo operatore, anche se presente nello standard, non è implementato da tutti i DBMS commerciali.

La sintassi completa della giunzione diventa la seguente:

Giunzione ::= "," | [**CROSS** | **NATURAL**] [[**LEFT** | **RIGHT** | **FULL**] [**OUTER**]] **JOIN**

In particolare, solo per **NATURAL JOIN** e **JOIN**, se è presente anche una delle specifiche **LEFT**, **RIGHT** o **FULL**, allora viene effettuata la giunzione esterna come segue:

a) con **LEFT** si aggiungono al risultato della giunzione le ennuple della tabella a sinistra che non fanno parte della giunzione, con valori **NULL** per i campi della tabella a destra:

SELECT $*$
FROM R **NATURAL LEFT JOIN** S; $\equiv R \overleftarrow{\bowtie} S$

b) con **RIGHT** si aggiungono al risultato della giunzione le ennuple della tabella a destra che non fanno parte della giunzione, con valori **NULL** per i campi della tabella a sinistra:

SELECT $*$
FROM R **NATURAL RIGHT JOIN** S; $\equiv R \overrightarrow{\bowtie} S$

c) con **FULL** si combinano gli effetti di **LEFT** e **RIGHT** aggiungendo al risultato della giunzione le ennuple delle due tabelle che non fanno parte della giunzione, estese opportunamente con valori **NULL**.

SELECT $*$
FROM R **NATURAL FULL JOIN** S; $\equiv R \overleftrightarrow{\bowtie} S$

Esempio 6.1
Siano $R(A, B, C)$ e $S(A, D)$, due schemi di relazioni con attributi definiti sui domini: $dom(A) = \{a1, a2, a3\}$, $dom(B) = \{b1, b2, b3\}$, $dom(C) = \{c1, c2\}$ e $dom(D) = \{d1, d2, d4\}$. Si mostra il risultato della *giunzione naturale, esterna, esterna destra* ed *esterna sinistra* con le relazioni R e S:

R

A	B	C
a1	b1	c1
a1	b1	c2
a2	b1	c1
a3	b1	c1

S

A	D
a1	d1
a2	d2
a4	d4

$R \bowtie S =$

A	B	C	D
a1	b1	c1	d1
a1	b1	c2	d1
a2	b1	c1	d2

$R \overleftrightarrow{\bowtie} S =$

A	B	C	D
a1	b1	c1	d1
a1	b1	c2	d1
a2	b1	c1	d2
a3	b1	c1	null
a4	null	null	d4

$R \overleftarrow{\bowtie} S =$

A	B	C	D
a1	b1	c1	d1
a1	b1	c2	d1
a2	b1	c1	d2
a3	b1	c1	null

$R \overrightarrow{\bowtie} S =$

A	B	C	D
a1	b1	c1	d1
a1	b1	c2	d1
a2	b1	c1	d2
a4	null	null	d4

6.4.2 Uso dei valori NULL

I valori **NULL** sono stati oggetto di critiche e di opinioni contrastanti sia negli aspetti teorici, che in quelli pratici, in particolare per come gli operatori SQL li trattano, in alcuni casi in maniera non omogenea, e per le complicazioni che introducono nell'ottimizzazione delle query.

Di seguito diamo alcune indicazioni sul loro uso, ricordando che il loro trattamento può differire nei sistemi commerciali che adottano il linguaggio SQL.

Ridurre l'uso dei valori NULL

Esistono varie interpretazioni di **NULL**: ad esempio come valore presente nella realtà ma non conosciuto nella base di dati, oppure come valore non presente (e quindi, in pratica, non un valore).

Data la natura problematica del valore **NULL**, alcuni suggeriscono di limitarne l'uso. Ad esempio, nella traduzione di uno schema concettuale in uno schema relazionale, in presenza di un'associazione con diretta parziale, al posto del valore **NULL** possiamo usare una tabella ausiliaria. Ad esempio, se alcuni agenti, ma non tutti, hanno un supervisore, possiamo definire, al posto della chiave esterna Supervisore nella tabella Agenti, una tabella ausiliaria che contiene una ennupla per ogni agente che ha un supervisore, con le due chiavi esterne verso l'agente e il suo supervisore. Analogamente può essere fatto per gli attributi opzionali: se non tutti gli agenti hanno il numero di telefono, potremmo definire, al posto dell'attributo, una tabella ausiliaria dei telefoni con la chiave esterna per l'agente e il numero di telefono. Ovviamente questo approccio richiede la definizione di molte tabelle ausiliare, e la corrispondente esecuzione di un gran numero di giunzioni per la ricostruzione dell'intera entità.

Escludere i valori NULL dalle operazioni di ricerca

Se dobbiamo scrivere delle query per per alcuni attributi che possono avere valore **NULL**, si usino opportunamente i predicati **IS NULL** e la sua negazione per escludere le ennuple con valori **NULL** dalla query.

Ad esempio, se la relazione Agenti ha l'attributo Supervisore che può avere valori **NULL** e vogliamo trovare tutti gli agenti con un codice minore di quello del proprio supervisore, si scriva:

```
SELECT   *
FROM     Agenti
WHERE    Supervisore IS NOT NULL AND Codice < Supervisore;
```

Convertire i valori NULL in valori significativi

Talvolta si devono produrre dei dati di riepilogo, che devono essere usati da un utente finale. In questi casi si può usare l'operatore **COALESCE** nella clausola **SELECT** per tradurre gli eventuali valori **NULL**, anche generati da giunzioni esterne, in valori significativi.

Ad esempio, per produrre una lista completa di tutti gli agenti, specificando il nome del supervisore, quando presente, si può scrivere:

```
SELECT   A1.Nome, COALESCE (A2.Nome, 'NON HA SUPERVISORE')
FROM     Agenti A1
         LEFT JOIN Agenti A2 ON A1.Supervisore = A2.CodiceAgente
WHERE    A1.Zona = 'Pisa';
```

Si noti che il **LEFT JOIN** produce un risultato per tutti gli agenti, anche per quelli che hanno il valore Supervisore uguale a **NULL**.

6.5 Operatori per la modifica dei dati

I comandi per modificare i dati sono i seguenti:

```
INSERT INTO Tabella [ "(" Attributo {, Attributo } ")" ]
    VALUES "(" Valore {,Valore}")" ;
```

per effettuare l'inserzione di un'ennupla in una tabella. La lista valori deve rispettare l'ordine degli attributi o, in loro assenza, l'ordine specificato nella definizione della tabella. Ad esempio, per inserire un nuovo cliente:

```
INSERT INTO Clienti
    VALUES ('A03', 'Rossi Mario', 'Roma', 10);
```

La modifica si effettua con il comando **UPDATE**:

```
UPDATE   Tabella
   SET   Attributo = Espr {, Attributo = Espr}
   WHERE Condizione;
```

Con un solo comando **UPDATE** si possono modificare uno o più attributi di un insieme di ennuple che soddisfano una condizione. Si noti che l'ennupla è la più piccola unità di inserzione e l'attributo la più piccola unità di aggiornamento.

Ad esempio, se vogliamo assegnare a tutti gli agenti con supervisore 's1' il nuovo supervisore 's2', possiamo scrivere:

UPDATE Agenti
  SET Supervisore = ‘s2’
  WHERE Supervisore = ‘s1’;

Infine, il comando di cancellazione è il seguente:

DELETE FROM *Tabella*
  WHERE *Condizione*;

Come **UPDATE**, il comando **DELETE** può coinvolgere una o più ennuple della tabella come determinato dalla clausola **WHERE**.

Ad esempio, se vogliamo cancellare tutti gli ordini precedenti al 2018, per cui l’agente corrispondente non esiste più nella relativa tabella, possiamo scrivere:

DELETE FROM Ordini
  WHERE Data $<$ ‘01012018‘;

6.6 Il potere espressivo di SQL

Come già accennato, il linguaggio SQL non ha la potenza computazionale delle Macchine di Turing, e quindi non permette di scrivere espressioni equivalenti a tutte le funzioni calcolabili. Una tale limitazione non è un grosso problema pratico perché le interrogazioni che non si possono esprimere non sono molto comuni. Quando si presentano occorre usare l’SQL all’interno di un linguaggio di programmazione “Turing-equivalente”.

La lista che segue mostra alcuni esempi di interrogazioni non esprimibili in SQL.

1. Le funzioni di aggregazione di solito disponibili non sono tutte quelle interessanti: se si vuole calcolare ad esempio la *moda*, o la *varianza*, di una colonna di valori, come altre importanti funzioni di tipo statistico, siamo impossibilitati a farlo. Infatti, se queste operazioni non sono fornite come primitive (né potrebbero esserlo tutte), non è possibile esprimerle come semplici espressioni sugli attributi (richiedono infatti uno o più “cicli” di calcolo sui valori di una colonna).
2. Le funzioni di aggregazione non si possono applicare ad altre funzioni. Ad esempio, non si può calcolare il totale di tutte le medie dell’ammontare degli ordini degli agenti, o il massimo di tutti i loro totali di vendita.[6]
3. La possibilità di creare “report” con la clausola **GROUP BY** è molto limitata rispetto a quelle offerte da un linguaggio di tipo generale. Ad esempio, non possiamo creare una tabella che contenga per certi intervalli di ammontare di ordini il totale delle vendite relative ad ogni intervallo.

6. Per risolvere questi problemi si può ricorrere all’espediente di definire tabelle derivate, come sarà mostrato nel prossimo capitolo.

4. Le condizioni permesse impediscono alcuni tipi di interrogazioni. Ad esempio, supponiamo di avere le seguenti tabelle per trattare documenti e le parole chiavi che ne descrivono il contenuto:

   Documenti(CodiceDocumento, TitoloDocumento, Autore)
   ParoleChiave(CodiceDocumento, ParolaChiave)

   È semplice trovare i titoli dei documenti con una certa parola chiave oppure che contengono tutte le parole chiave di un certo insieme K, ma non è possibile scrivere un'interrogazione per trovare tutti i documenti che contengono almeno k delle chiavi di K (problema tipico di *information retrieval*). Ad esempio, per trovare i documenti che contengono quattro chiavi di un insieme di sei chiavi occorre invece formulare $(6 \times 5)/2 = 15$ interrogazioni.
5. La *chiusura transitiva* di una relazione binaria rappresentata in forma di tabella non può essere calcolata (ma alcune varianti di SQL prevedono un operatore apposito). Ad esempio, supponiamo che non solo gli agenti ma anche i loro supervisori possano avere un supervisore. Con gli operatori visti non è possibile trovare tutti i supervisori, di qualsiasi livello, di un agente.

Come si vede da questi esempi, quindi, il linguaggio SQL non è sufficiente per esprimere interrogazioni di qualsiasi natura, ma è stato definito prevedendo gli operatori più utili per le interrogazioni più comuni.

Diverso è il caso del suo uso in un linguaggio di programmazione, come vedremo nel capitolo sullo sviluppo di applicazioni: lì è usato come un insieme di operatori "primitivi" (anche se non certo elementari) per operare sulla base di dati, mentre il linguaggio di programmazione utilizza i risultati delle interrogazioni per esprimere computazioni qualsiasi.

6.7 QBE: un esempio di linguaggio basato sulla grafica

Il linguaggio QBE (*Query by Example*), sviluppato alla IBM negli anni '70, è un esempio di linguaggio per il calcolo relazionale di domini, basato su un'interfaccia grafica. Le interrogazioni sono formulate usando la seguente rappresentazione grafica delle tabelle:

Clienti	CodiceCliente	Nome	Città	Sconto

Ordini	NumOrdine	CodiceCliente	CodiceAgente	Prodotto	Data	Ammontare

Questo modo di visualizzare le tabelle è quello originariamente previsto per i terminali a caratteri. Con i moderni sistemi provvisti di interfacce grafiche le cose cambiano molto e un esempio molto noto è Microsoft Access.

Per formulare un'interrogazione, l'utente riempie le righe con esempi di valori che desidera nel risultato e con variabili che assumono valori nei domini di certe colonne. Per distinguere costanti da variabili, queste ultime iniziano con il carattere "_". I campi che si vogliono vedere nel risultato sono quelli in cui si pongono i caratteri "P.". Un valore può essere preceduto da un operatore di confronto "$>$" o "$\geqslant$". Tutte le condizioni espresse nella stessa riga sono in **AND** e quelle su righe diverse sono in **OR**.

Ad esempio, per trovare il codice e la commissione degli agenti di Pisa, lo scheletro della tabella si riempie nel modo seguente:

Agenti	CodiceAgente	Nome	Zona	Supervisore	Commissione
	P._x		Pisa		P._y

A differenza dell'SQL, QBE restituisce sempre insiemi, mentre per mantenere nel risultato ennuple uguali si usa "P. all".

Condizioni più complesse si danno separatamente in una *condition box*, senza usare l'operatore di negazione "$\neg$", che va eliminato quindi usando le classiche equivalenze:

$\neg\neg C_1 \equiv C_1$
$\neg(C_1 \wedge C_2) \equiv \neg C_1 \vee \neg C_2$
$\neg(C_1 \vee C_2) \equiv \neg C_1 \wedge \neg C_2$

Si "spinge" poi la negazione verso l'interno di qualunque condizione, fino ad accostarla alle condizioni atomiche, che a loro volta sono in grado di assorbirla, grazie alle equivalenze:

$\neg x = y \equiv x \neq y$
$\neg x \neq y \equiv x = y$
$\neg x < y \equiv x \geqslant y$ ecc.

Ad esempio, per trovare il nome dei clienti che hanno fatto ordini con ammontare superiore a 10 000 euro, si pone:

Clienti	CodiceCliente	Nome	Città	Sconto
	_x	P.		

Ordini	NumOrdine	CodiceCliente	CodiceAgente	Prodotto	Data	Ammontare
		_y				$\geqslant$10 000

Conditions
_x = _y

6.8 Conclusioni

Sono state presentate le caratteristiche del linguaggio SQL per il recupero dei dati. SQL è ormai uno standard ed è offerto da tutti i sistemi, anche se continuano ad esistere differenze fra le versioni dei sistemi commerciali più diffusi. Molti sistemi prevedono interfacce grafiche per facilitarne l'uso interattivo, nello stile di Access, un prodotto della Microsoft molto popolare in ambiente Windows.

Nel prossimo capitolo si vedranno gli altri comandi dell'SQL per definire e amministrare basi di dati e poi come si possa usare in un linguaggio di programmazione per sviluppare applicazioni.

Per fare pratica con l'SQL è possibile utilizzare il sistema didattico gratuito multipiattaforma **JRS** (Java Relational System)[7]. Oltre a permettere di eseguire comandi SQL, l'applicazione permette di vedere gli alberi logici e fisici e i risultati delle fasi di ottimizzazione delle interrogazioni.

Esercizi

1. Dare un'espressione **SELECT** per stabilire se i valori di A in una relazione con schema R(A, B, C) siano tutti diversi. L'espressione deve essere diversa da **SELECT** A **FROM** R.
2. Si ricorda che il predicato **IN** è equivalente a =**ANY**. Spiegare perché il predicato **NOT IN** non è equivalente a <>**ANY** ma a <>**ALL**.
3. È importante sapere quando una **SELECT** ritorna una tabella con righe diverse per evitare di usare inutilmente la clausola **DISTINCT**, che comporta un costo addizionale per l'esecuzione dell'interrogazione (perché?).

   a) È vero che con una **SELECT** con una tabella nella parte **FROM**, e senza **GROUP BY**, non vi saranno righe duplicate nel risultato se gli attributi della parte **SELECT** sono una superchiave della tabella?

   b) È vero che con una **SELECT** con più tabelle nella parte **FROM**, e senza **GROUP BY**, non vi saranno righe duplicate nel risultato se gli attributi della parte **SELECT** sono una superchiave di ogni tabella?

   c) È vero che nessuna interrogazione con un **GROUP BY** può avere duplicati nel risultato?

7. **JRS** è stato sviluppato in Java per il supporto allo studio dei Sistemi di Gestione di Basi di Dati Relazionali presso il Dipartimento di Informatica dell'Università di Pisa dagli autori del presente testo e dagli studenti Lorenzo Brandimarte, Leonardo Candela, Giovanna Colucci, Patrizia Dedato, Stefano Fantechi, Stefano Dinelli, Martina Filippeschi, Simone Marchi, Cinzia Partigliani, Marco Sbaffi e Ciro Valisena. Può essere scaricato, insieme con la documentazione, dal sito del libro, http://fondamentidibasididati.it

Per ogni caso dare un esempio di interrogazione che ritorna duplicati se la condizione non è verificata.

4. Si consideri lo schema relazionale

   Studenti(Matricola, Nome, Provincia, AnnoNascita)
   Esami(Materia, MatricolaStudente, Voto, NumAppello, Anno)

   Formulare in SQL le seguenti interrogazioni:

   a) Trovare il nome degli studenti di Pisa che hanno superato l'esame di Programmazione con 30.
   b) Trovare il nome degli studenti che hanno superato cinque esami.
   c) Trovare per ogni studente di Pisa il numero degli esami superati, il voto massimo, minimo e medio.
   d) Trovare per ogni materia il numero degli esami fatti al primo appello, il voto massimo, minimo e medio.

5. Usando la base di dati relazionale ottenuta dall'Esercizio 2.8, formulare in SQL le seguenti interrogazioni:

   a) Trovare il nome e l'anno di nascita dei figli dell'impiegato con codice 350.
   b) Trovare il nome e codice degli impiegati e il nome del dipartimento dove lavorano.
   c) Trovare il nome degli impiegati, il nome e l'anno di nascita dei figli maschi a carico.
   d) Trovare, per ogni progetto in corso a Pisa, il numero e il nome del progetto, il nome del dipartimento dove si svolge, il cognome del direttore del dipartimento.
   e) Trovare il nome dei dipartimenti con almeno un impiegato con persone a carico.
   f) Trovare il numero degli impiegati del dipartimento di Informatica.
   g) Trovare, per ogni progetto al quale lavorano più di due impiegati, il nome, il numero e il numero degli impiegati che vi lavorano.
   h) Trovare per ogni dipartimento il nome, il numero degli impiegati e la media del loro anno di nascita.
   i) Trovare i nomi dei supervisori e dei loro dipendenti.
   j) Trovare il nome degli impiegati e il nome del dipartimento in cui lavorano.
   k) Trovare il nome degli impiegati senza familiari a carico.
   l) Trovare i progetti cui partecipa il sig. Rossi come impiegato o come direttore del dipartimento che gestisce il progetto.
   m) Trovare il nome degli impiegati che hanno i familiari a carico dello stesso sesso.
   n) Trovare il nome degli impiegati che hanno tutti i familiari a carico dello stesso sesso dell'impiegato.
   o) Trovare il nome degli impiegati che lavorano almeno a tutti i progetti dell'impiegato con codice 300.
   p) Trovare il nome del dipartimento e il numero di impiegati nati dopo il 1950, per i dipartimenti con più di due impiegati.

Note bibliografiche

Il linguaggio SQL è trattato in ogni libro sulle basi di dati. Per un'analisi più approfondita esistono numerosi testi specifici come [van der Lans, 2001], che viene venduto insieme ad un CD-ROM con una versione di un sistema relazionale. Per lo standard SQL-92, e per un'introduzione all'SQL:2003, si veda [Kifer et al., 2005].

Capitolo 7

SQL PER DEFINIRE E AMMINISTRARE BASI DI DATI

In questo capitolo vengono presentati i comandi SQL-92 per definire e amministrare basi di dati. Il materiale è organizzato presentando nell'ordine i comandi per:

- definire una base di dati e la struttura logica delle sue tabelle, memorizzate o calcolate;
- definire i vincoli d'integrità sui valori ammissibili degli attributi di un'ennupla, di ennuple diverse della stessa tabella (*vincoli intrarelazionali*) o di tabelle diverse (*vincoli interrelazionali*);
- definire aspetti procedurali nello schema della base di dati, sia sotto forma di *procedure memorizzate* che di *trigger*;
- definire aspetti fisici riguardanti criteri di memorizzazione dei dati e tipi di strutture di accesso per rendere più rapide certe operazioni su tabelle di grandi dimensioni. Mentre gli aspetti precedenti fanno parte del cosiddetto *schema logico*, gli aspetti fisici fanno parte del cosiddetto *schema fisico*;
- adeguare la base di dati a nuove esigenze che si manifestano durante il funzionamento a regime (*evoluzione dello schema*);
- limitare i dati accessibili e le modalità d'uso agli utenti autorizzati.

Poiché non tutti questi aspetti sono trattati dallo standard SQL-92, per alcuni di essi si presenteranno le soluzioni presenti in alcuni sistemi commerciali.

7.1 Definizione della struttura di una base di dati

Un DBMS permette la creazione di più schemi all'interno di un'unica base di dati. Uno schema, in questa terminologia, corrisponde ad un insieme di tabelle, memorizzate e calcolate, procedure, trigger, e agli altri "oggetti" che possono essere presenti in una base di dati, ed è associato ad un utente *proprietario*, che stabilisce la la disponibilità degli oggetti in esso contenuti ad altri utenti.

Negli esempi che seguono si useranno i comandi SQL-92 nella forma testuale, ma si tenga presente che di solito i sistemi commerciali prevedono anche strumenti di tipo grafico interattivi, facili da usare ma non standard. Questi strumenti generano poi i

comandi SQL che verranno eseguiti dal sistema. Come base di dati di riferimento si userà quella vista nel capitolo precedente, che per comodità si riporta in Figura 7.1.

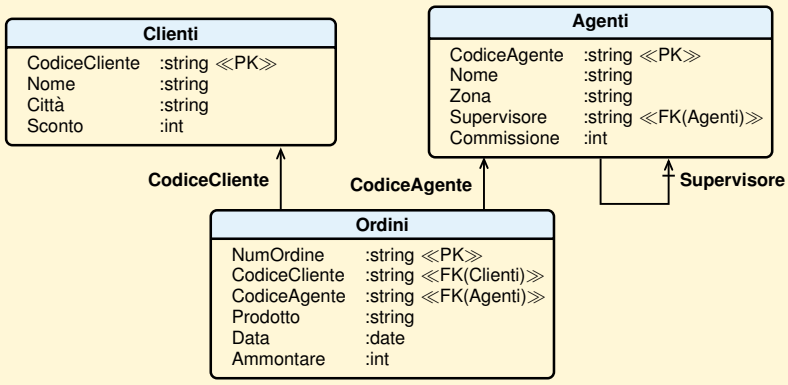


Figura 7.1: Rappresentazione grafica dello schema esempio

Premettiamo, infine, una caratteristica importante dei sistemi relazionali: lo schema di una base di dati si costruisce incrementalmente con opportuni comandi (ad esempio **CREATE TABLE**) dati interattivamente, o attraverso uno strumento grafico, e tutte le definizioni sono memorizzate nella "metabase di dati", o *catalogo* del sistema, la cui struttura verrà discussa più avanti. Non esiste, quindi, in generale, un "testo" memorizzato da qualche parte con tutte le definizioni che costituiscono lo schema; è però possibile usare opportuni programmi SQL che producono dinamicamente un testo di questo tipo interrogando il catalogo e stampando i risultati.

7.1.1 Base di dati

Lo schema di una base di dati viene creata con il comando:

CREATE SCHEMA *Nome* **AUTHORIZATION** *Utente*
Definizioni

dove *Nome* è il nome dello schema della base di dati che viene creata, *Utente* è il nome dell'utente proprietario, e *Definizioni* sono i comandi per la creazione degli elementi della base di dati (tabelle, viste, indici ecc.).

Come detto precedentemente, questi elementi possono essere creati o modificati in un qualunque momento successivo, durante una *sessione* d'uso dello schema della base di dati, con gli opportuni comandi.

Uno schema di base di dati può essere rimosso con il comando:

DROP SCHEMA *Nome* [**RESTRICT** | **CASCADE**]

Con la forma **RESTRICT** l'operazione fallisce se vi sono ancora dati, mentre con la forma **CASCADE** si rimuovono automaticamente tutti i dati in essa presenti.[1]

7.1.2 Tabelle

La forma base del comando di creazione di una tabella è la seguente:

CREATE TABLE Nome
"(" *Attributo Tipo* [*Vincolo* {, *Vincolo* }]
{, *Attributo Tipo* [*Vincolo* {, *Vincolo* }] }
[, *VincoloDiTabella* {, *VincoloDiTabella* }] ")"

I vincoli saranno discussi in dettaglio nella sezione 7.2. I tipi più comuni per i valori degli attributi sono:

- **CHAR**(n) per stringhe di caratteri di lunghezza fissa *n*;
- **VARCHAR**(n) per stringhe di caratteri di lunghezza variabile di al massimo *n* caratteri;
- **INTEGER** per interi con la dimensione uguale alla parola di memoria standard dell'elaboratore;
- **REAL** per numeri reali con dimensione uguale alla parola di memoria standard dell'elaboratore;
- **NUMBER**(p,s) per numeri con p cifre, di cui s decimali;
- **FLOAT**(p) per numeri binari in virgola mobile, con almeno p cifre significative;
- **DATE** per valori che rappresentano istanti di tempo (in alcuni sistemi, come Oracle), oppure solo date (e quindi insieme ad un tipo **TIME** per indicare ora, minuti e secondi).

Vediamo un esempio di definizione di una tabella, per ora senza vincoli:

```
CREATE  TABLE Clienti (
        CodiceCliente   CHAR(3),
        Nome            CHAR(30),
        Città           CHAR(30),
        Sconto          INTEGER);
```

Una tabella si può anche creare inserendovi immediatamente un insieme di ennuple ottenute come risultato di una espressione SQL:

```
CREATE  TABLE Nome
    AS  EspressioneSelect;
```

1. Una regola generale del linguaggio è che per ogni comando **CREATE** *Elemento* vi è un corrispondente comando di cancellazione **DROP** *Elemento*.

Ad esempio, i seguenti comandi creano una tabella che rappresenta un archivio storico contenente informazioni sugli ordini precedenti al 2003, togliendoli dalla tabella corrente:

```
CREATE   TABLE  ArchivioStoricoOrdini
    AS   SELECT     *
         FROM       Ordini
         WHERE      Data < '01012003';

DELETE   FROM       Ordini
         WHERE      Data < '01012003';
```

Una tabella può essere eliminata con il comando:

```
DROP TABLE Nome;
```

7.1.3 Tabelle virtuali

Oltre alla costruzione di tabelle normali, contenenti dati, (*tabelle di base*), si possono definire delle *tabelle virtuali* con il comando:

```
CREATE   VIEW NomeVista [ "(" Attributo {, Attributo} ")" ]
    AS   EspressioneSelect
```

Una tabella virtuale, o *vista* (*view*), è il risultato di un'espressione SQL a partire da altre tabelle, sia base che virtuali.

Il contenuto di una tabella virtuale *non è fisicamente* memorizzato nella base di dati, ma solo l'espressione SQL che la definisce viene memorizzata nel catalogo del sistema. In generale, il contenuto viene calcolato ogni volta che la tabella virtuale viene usata come se fosse una normale tabella in un'interrogazione, tranne in quei casi in cui l'ottimizzatore del sistema sia in grado di stabilire che l'interrogazione possa essere "riscritta" combinandola con l'espressione SQL che definisce la tabella virtuale.

Ad esempio, la seguente vista:

```
CREATE   VIEW       OrdiniPerAgente(CodiceAgente, TotaleOrdini)
    AS   SELECT     CodiceAgente, SUM(Ammontare)
         FROM       Ordini
         GROUP BY   CodiceAgente;
```

descrive una tabella virtuale formata, per ogni agente che ha fatto qualche ordine, dal codice dell'agente e dal totale dei suoi ordini. Un'interrogazione come:

```
SELECT    A.Nome, TotaleOrdini
FROM      Agenti A, OrdiniPerAgente O
WHERE     A.CodiceAgente = O.CodiceAgente
          AND TotaleOrdini > 1000;
```

può essere riscritta come:

```
SELECT      A.Nome, SUM(Ammontare) AS TotaleOrdini
FROM        Agenti A, Ordini O
WHERE       A.CodiceAgente = O.CodiceAgente
GROUP BY    A.CodiceAgente, A.Nome
HAVING      SUM(Ammontare) > 1000;
```

Una tabella virtuale può comparire in una espressione **SELECT** esattamente come una tabella base, mentre le operazioni di modifica su una tabella virtuale sono soggette a restrizioni, perché non sempre sono riconducibili a modifiche sulle tabelle base usate per definirla. In particolare, le modifiche sono ammesse solo quando la tabella virtuale è definita con un'espressione che soddisfa le seguenti condizioni:

- la clausola **SELECT** non ha l'opzione **DISTINCT** e i valori degli attributi della tabella virtuale non sono calcolati;
- la clausola **FROM** riguarda una sola tabella base o virtuale, a sua volta modificabile, ovvero sono escluse tabelle virtuali ottenute per giunzione;
- la clausola **WHERE** non contiene *SottoSelect*;
- non sono presenti gli operatori **GROUP BY** e **HAVING**;
- le colonne definite nelle tabelle di base con il vincolo **NOT NULL** devono far parte della tabella virtuale.

Ad esempio, la vista precedente non può essere usata per modifiche, dato che contiene un attributo calcolato (TotaleOrdini).

Con l'introduzione delle tabelle virtuali occorre rivedere in generale il comando di cancellazione delle tabelle, sia base che virtuali. Infatti, se una tabella è utilizzata nella definizione di un'altra virtuale, al momento della sua cancellazione si può specificare quale azione intraprendere:

DROP (*Tabella* | *Vista*) [**RESTRICT** | **CASCADE**]

In entrambi i casi, se viene specificato **RESTRICT**, la tabella o la vista non vengono rimosse se sono utilizzate in altre viste, mentre **CASCADE** provoca la rimozione automatica di tutte le viste che utilizzano la tabella o la vista cancellata.

Uso delle tabelle virtuali

Le tabelle virtuali sono utili per diverse ragioni:

1. per semplificare certi tipi di interrogazioni complesse, o addirittura non esprimibili in SQL su tabelle base. Ad esempio, come già discusso nel capitolo precedente, se vogliamo trovare il numero medio degli agenti per zona, sarebbe un errore scrivere qualcosa del genere:

   ```
   SELECT      AVG(COUNT(*))
   FROM        Agenti
   GROUP BY    Zona;
   ```

   Possiamo però formulare l'interrogazione con l'aiuto di una tabella virtuale:

```
CREATE   VIEW       AgentiPerZona (Zona, NumeroAgenti)
    AS   SELECT     Zona, COUNT(*)
         FROM       Agenti
         GROUP BY   Zona;

SELECT   AVG(NumeroAgenti)
FROM     AgentiPerZona;

DROP     AgentiPerZona;
```

2. per nascondere alle applicazioni alcune modifiche dell'organizzazione logica dei dati (*indipendenza logica*). Ad esempio, supponiamo che per ogni zona vi sia un supervisore, che sia egli stesso un agente, ma privo di supervisore. La memorizzazione dei dati potrebbero essere cambiata nel modo seguente:

   a) aggiungere una tabella contenente le zone e i relativi supervisori:

```
CREATE   TABLE      SupervisoriPerZone
    AS   SELECT     DISTINCT Zona, Supervisore
         FROM       Agenti
         WHERE      Supervisore IS NOT NULL;
```

   b) togliere l'attributo Supervisore dalla tabella degli agenti:

```
CREATE   TABLE      NuoviAgenti
    AS   SELECT     CodiceAgente, Nome, Zona, Commissione
         FROM       Agenti;
```

   c) cancellare la "vecchia" tabella degli agenti:

```
DROP     Agenti;
```

   d) costruire una vista che ricostruisce la "vecchia" situazione di agenti con l'attributo Supervisore:

```
CREATE   VIEW       Agenti(CodiceAgente, Nome, Zona, Supervisore,
                           Commissione)
    AS   SELECT     CodiceAgente, Nome, Z.Zona, Supervisore , Commissione
         FROM       NuoviAgenti A, SupervisoriPerZone Z
         WHERE      A.Zona = Z.Zona AND CodiceAgente <> Supervisore
           UNION ALL
         SELECT     CodiceAgente, Nome, Z.Zona, NULL AS Supervisore,
                    Commissione
         FROM       NuoviAgenti A, SupervisoriPerZone Z
         WHERE      A.Zona = Z.Zona AND CodiceAgente = Supervisore;
```

3. possono essere usate per dare visioni diverse degli *stessi* dati (viste utente). Ad esempio, se vogliamo aggregare gli ordini per agente, per applicazioni di tipo statistico, possiamo fornire la tabella virtuale creata all'inizio di questa sezione.

7.2 Vincoli d'integrità

I vincoli che possono essere espressi in uno schema relazionale riguardano i valori ammissibili degli attributi di un'ennupla, o di ennuple diverse della stessa tabella (*vincoli intrarelazionali*) o di tabelle diverse (*vincoli interrelazionali*).

Per impedirne la violazione, i vincoli d'integrità vengono controllati dal sistema quando si effettua una modifica della base di dati (operazioni **INSERT**, **DELETE** e **UPDATE**) che li coinvolge. Nel caso di violazione di un vincolo, l'azione eseguita dal sistema dipende da come è stato dichiarato il vincolo. In assenza di direttive esplicite, il sistema interrompe l'operazione annullando la transazione corrente, e segnala il fatto, come accade ad esempio quando si cerca di inserire una ennupla che viola il vincolo di chiave primaria. Nel caso invece che sia stata specificata un'azione da compiere per portare la base di dati in uno stato corretto (come è possibile fare ad esempio per il vincolo di chiave esterna), il sistema si comporta di conseguenza.

1. Vincoli su attributi di un'ennupla.
   a) Si può specificare che un attributo non possa avere il valore **NULL** con il vincolo **NOT NULL**. Questo vincolo è implicito se l'attributo fa parte della chiave primaria.
   b) Con la sintassi **CHECK** *Condizione* è possibile specificare con una condizione i valori ammissibili dell'attributo. Ad esempio, se vogliamo richiedere che l'ammontare minimo di un ordine sia 100, si pone

      Ammontare **INTEGER NOT NULL CHECK** (Ammontare >= 100)

      oppure se vogliamo imporre che un attributo Sesso abbia solo valori M o F, si pone

      Sesso **CHAR**(1) **NOT NULL CHECK** (Sesso **IN** ('M', 'F'))

   c) Si può fare in modo che il sistema assegni un valore di default all'attributo quando viene inserita una ennupla con **DEFAULT** (*Costante* | **NULL**).
   d) È possibile definire vincoli fra i valori di attributi diversi di una stessa ennupla; nella condizione non possono essere coinvolte altre tabelle:

      CHECK Condizione

2. Vincoli intrarelazionali.
   a) Il vincolo **UNIQUE** richiede che non vi siano duplicati nei valori dell'attributo (cioè l'attributo è una chiave). Si noti che la chiave primaria *deve* invece essere dichiarata con il seguente vincolo di **PRIMARY KEY**.

b) È possibile definire una chiave primaria, anche formata da più attributi, assegnandole un nome:

PRIMARY KEY [*NomeChiave*] "(" *Attributo* {, *Attributo*} ")"

Gli attributi delle chiavi devono essere dichiarati **NOT NULL**, e non vi può essere più di un vincolo di tipo **PRIMARY KEY** in una tabella. Una chiave primaria è obbligatoria quando si devono introdurre vincoli d'integrità referenziali.

c) È possibile definire chiavi formate da più attributi:

UNIQUE "(" *Attributo* {, *Attributo*} ")"

3. Vincoli interrelazionali. È possibile definire il vincolo d'integrità referenziale su chiavi esterne, con la seguente sintassi:

FOREIGN KEY [*NomeChiaveEsterna*] "(" *Attributo* {, *Attributo*} ")"
REFERENCES *TabellaReferenziata*
[**ON DELETE** (**NO ACTION** | **CASCADE** | **SET NULL**)]

dove *TabellaReferenziata* è una tabella per la quale è stata definita una chiave primaria (con l'opzione **PRIMARY KEY**) il cui tipo è uguale a quello degli attributi specificati.
Gli attributi di una chiave esterna, al contrario di quelli di una chiave, possono avere il valore **NULL**.
Il vincolo d'integrità referenziale su chiavi esterne ha i seguenti effetti:

- se si cerca di inserire una ennupla nella tabella con il valore della chiave esterna che non corrisponde ad un valore della chiave primaria in *TabellaReferenziata,* il vincolo viene violato e l'operazione fallisce;
- se si cerca di cancellare una ennupla di *TabellaReferenziata* la cui chiave primaria è il valore di qualche chiave esterna nella tabella (violando il vincolo), si opera in base alla specifica del vincolo:
  a) **ON DELETE NO ACTION** richiede il rifiuto dell'operazione e il fallimento della transazione. L'assenza della specifica **ON DELETE** è equivalente a questa opzione;
  b) **ON DELETE CASCADE** richiede la cancellazione delle ennuple che hanno il valore della chiave esterna uguale a quello della chiave primaria delle ennuple cancellate;
  c) **ON DELETE SET NULL** richiede di assegnare il valore nullo agli attributi della chiave esterna.
- se si cerca di modificare una ennupla assegnando agli attributi della chiave esterna un valore che non corrisponde ad un valore della chiave primaria in *TabellaReferenziata,* oppure se si modifica il valore di una chiave primaria in *TabellaReferenziata,* di solito nei sistemi commerciali l'operazione viene rifiutata, mentre nei sistemi che seguono il livello intermedio dell'SQL-92, con l'opzione

ON UPDATE è prevista la possibilità di scelta dell'azione da intraprendere come nel caso delle cancellazioni.

Esempio 7.1 Diamo lo schema SQL completo per la base di dati di Figura 6.1:

```
CREATE   TABLE Clienti (
         CodiceCliente   CHAR(3)      NOT NULL,
         Nome            CHAR(30)     NOT NULL,
         Città           CHAR(30)     NOT NULL,
         Sconto          INTEGER      NOT NULL DEFAULT 0
                         CHECK(Sconto>=0 AND Sconto<100),
         PRIMARY KEY     pk_Clienti (CodiceCliente) );

CREATE   TABLE Agenti (
         CodiceAgente    CHAR(3)      NOT NULL,
         Nome            CHAR(30)     NOT NULL,
         Zona            CHAR(8)      NOT NULL,
         Supervisore     CHAR(3),
         Commissione     INTEGER,
         PRIMARY KEY     pk_Agenti (CodiceAgente),
         FOREIGN KEY     fk_SupervisoreAgente (Supervisore)
                         REFERENCES Agenti
                         ON DELETE SET NULL,
         CHECK           (Supervisore <> CodiceAgente
                          OR Supervisore IS NULL) );

CREATE   TABLE Ordini (
         NumOrdine       CHAR(3)      NOT NULL,
         CodiceCliente   CHAR(3)      NOT NULL,
         CodiceAgente    CHAR(3)      NOT NULL,
         Data            CHAR(8)      NOT NULL,
         Prodotto        CHAR(3)      NOT NULL,
         Ammontare       INTEGER      NOT NULL CHECK(Ammontare > 100),
         PRIMARY KEY     pk_Ordini (NumOrdine),
         FOREIGN KEY     fk_ClienteOrdine (CodiceCliente)
                         REFERENCES Clienti
                         ON DELETE NO ACTION,
         FOREIGN KEY     fk_AgenteOrdine (CodiceAgente)
                         REFERENCES Agenti
                         ON DELETE NO ACTION);
```

```
CREATE   VIEW       OrdiniPerAgente(CodiceAgente, TotaleOrdini)
      AS SELECT     CodiceAgente, SUM(Ammontare)
         FROM       Ordini
         GROUP BY   CodiceAgente;

CREATE   VIEW       AgentiConOrdini
      AS SELECT     A.CodiceAgente AS CodiceAgente, Nome, Zona,
                    Supervisore, Commissione, TotaleOrdini
         FROM       OrdiniPerAgente O, Agenti A
         WHERE      A.CodiceAgente = O.CodiceAgente;
```

Si noti la definizione dei vincoli e delle tabelle virtuali. Nella seconda, AgentiConOrdini, si utilizza la prima (OrdiniPerAgente), e non si dichiarano gli attributi della tabella, perché vengono presi quelli specificati nel **SELECT**.

7.3 Aspetti procedurali

Nei sistemi relazionali commerciali si possono trattare nello schema aspetti procedurali definendo *procedure* o *trigger*.

Le procedure memorizzate nella base di dati sono programmi che vengono eseguiti dal DBMS su esplicita richiesta delle applicazioni o degli utenti. I *trigger*, invece, sono anch'essi delle procedure memorizzate nella base di dati, che però vengono attivate automaticamente dal DBMS quando si fanno determinate operazioni sulle tabelle.

7.3.1 Procedure memorizzate

Le procedure memorizzate nella base di dati sono state previste dall'SQL-92 come procedure con un nome, parametri e un corpo costituito da un unico comando SQL, raggruppabili attraverso un meccanismo di *moduli*. Normalmente, però, i sistemi commerciali prevedono un linguaggio più ricco per definirle, come il linguaggio PL/SQL del sistema Oracle, che verrà presentato nel prossimo capitolo, insieme a esempi di procedure, e il Transact/SQL del sistema Sybase.

Le procedure memorizzate nella base di dati sono utili per diverse ragioni:

1. Consentono di condividere fra le applicazioni del codice di interesse generale. In questo modo si semplificano le applicazioni e quindi la loro manutenzione. Inoltre, essendo le procedure gestite in modo centralizzato, una loro modifica non richiede di essere riportata in tutte le applicazioni che ne fanno uso.
2. Consentono di garantire che certe operazioni sulla base di dati abbiano la stessa semantica per ogni applicazione che ne fa uso.
3. Consentono di controllare in modo centralizzato certi vincoli d'integrità non esprimibili nella definizione delle tabelle.

4. Consentono di ridurre il traffico sulla rete dovuto ad applicazioni remote: il programma cliente, invece di interagire con il DBMS servente spedendo un'interrogazione per volta, spedisce semplicemente una chiamata di una procedura memorizzata, che viene eseguita dal servente, e riceve alla fine solo il risultato finale.
5. Consentono di garantire la sicurezza dei dati perché a certi utenti si può permettere di usare solo certe procedure per ottenere dei dati, ma non di accedere direttamente alle tabelle dalle quali le procedure estraggono i dati restituiti.

L'uso delle procedure memorizzate nella base di dati pone però nuovi problemi nella progettazione di basi di dati, e le metodologie disponibili non aiutano in genere a definire ed organizzare tali procedure. Il loro uso pone, inoltre, nuovi problemi di carattere organizzativo poiché richiede una nuova figura professionale, l'amministratore delle procedure, che può essere diverso dal tradizionale amministratore dei dati.

7.3.2 Trigger

I *trigger* sono presenti in tutti i sistemi commerciali più sofisticati, sebbene non siano stati previsti dallo standard SQL-92.

Un *trigger* specifica un'azione da attivare automaticamente al verificarsi di un'operazione di modifica su una tabella (**INSERT**, **UPDATE**, **DELETE**).

Come esempio, riportiamo la forma base del comando per creare *trigger* in PL/SQL:

CREATE **TRIGGER**, *NomeTrigger*
TipoTrigger
(*TipoOperazione* {**OR** *TipoOperazione*})
[**OF** *Attributo*] **ON** *NomeTabella*
[**FOR EACH ROW**]
[**WHEN** "(" *Condizione* ")"]
Programma

TipoTrigger := (**BEFORE** | **AFTER**)
TipoOperazione := (**SELECT** | **DELETE** | **INSERT** | **UPDATE**)

Nella definizione si specificano le seguenti informazioni:

- il tipo di *trigger* (**BEFORE**, **AFTER**) per stabilire quando il *trigger* debba essere attivato, se prima o dopo l'operazione;
- il tipo di operazione che attiva il *trigger* e su quale tabella (ed eventualmente su quale attributo);
- un'eventuale clausola **FOR EACH ROW** per stabilire quante volte il *trigger* debba essere attivato, se una volta oppure tante volte quante sono le righe della tabella interessate dall'operazione;
- un'eventuale ulteriore condizione che deve essere vera perché il codice del *trigger* venga eseguito;
- il codice da eseguire.

Esempio 7.2
Supponiamo che nella base di dati dell'Esempio 7.1 gli ordini siano fatture da pagare e che si voglia imporre il vincolo che non si accettano nuovi ordini da clienti con uno scoperto maggiore di 10.000:

```
CREATE TRIGGER ControlloFido
BEFORE INSERT ON Ordini
DECLARE
    DaPagare NUMBER;
BEGIN
    SELECT SUM(Ammontare) INTO DaPagare
    FROM   Ordini
    WHERE  CodiceCliente = :new.CodiceCliente;
    IF DaPagare >= 10000 − :new.Ammontare
    THEN
        RAISE_APPLICATION_ERROR(-2061, 'fido superato');
    END IF;
END;
```

Nel corpo di un *trigger*, :new sta per il valore della ennupla da inserire o il valore modificato, mentre :old per il valore precedente. Quindi, in questo caso :new.CodiceCliente è il valore del campo CodiceCliente dell'ennupla da inserire.

Come ulteriore esempio, mostriamo l'uso di un *trigger* per mantenere una tabella memorizzata, ma aggiornata automaticamente in funzione di un'altra tabella, senza l'uso di viste.

Esempio 7.3
Il seguente programma crea una nuova tabella di coppie (agente, totale ordini effettuati), e un *trigger* che automaticamente, da questo momento in poi, manterrà aggiornata la nuova tabella.

```
CREATE TABLE Totali(CodiceAgente CHAR(3), TotaleOrdini INTEGER);

CREATE TRIGGER esempioTrig
AFTER INSERT ON Ordini
FOR EACH ROW
DECLARE
    esiste NUMBER;
BEGIN
    /* Si controlla se l'agente dell'ordine e' gia' presente
        nella tabella dei totali */
```

```
    SELECT COUNT(*) INTO esiste
    FROM   Totali
    WHERE  CodiceAgente = :new.CodiceAgente;
    IF esiste = 0 /* L'agente non e' presente, deve essere inserita una nuova riga */
    THEN
        INSERT INTO Totali
        VALUES(:new.CodiceAgente, :new.Ammontare);
    ELSE
        UPDATE Totali
        SET TotaleOrdini = TotaleOrdini + :new.Ammontare
        WHERE CodiceAgente = :new.CodiceAgente;
END;
```

Si noti che :new.CodiceAgente rappresenta il valore del campo CodiceAgente della riga di Ordini inserita.

Se vogliamo anche cancellare la riga della tabella Totali quando l'agente corrispondente viene cancellato, è sufficiente il seguente *trigger*:

```
CREATE TRIGGER esempio2Trig
AFTER DELETE ON Agenti
FOR EACH ROW BEGIN
    DELETE Totali WHERE CodiceAgente = :old.CodiceAgente;
END;
```

Uso dei trigger

A seconda del loro uso, possiamo dividere i *trigger* in *passivi* ed *attivi*. Un *trigger* è passivo se serve a provocare un fallimento della transazione corrente sotto certe condizioni, mentre un *trigger* è attivo quando, in corrispondenza di certi eventi, modifica lo stato della base di dati (così, negli esempi precedenti, il primo è passivo mentre gli ultimi due sono attivi).

I *trigger* sono utili per diverse ragioni:

1. *Trigger* passivi:
   a) per definire vincoli d'integrità non esprimibili nel modello dei dati usati (ad esempio vincoli dinamici);
   b) per fare controlli sulle operazioni ammissibili degli utenti basati sui valori dei parametri di comandi SQL. Ad esempio, si possono inserire certi dati solo se il codice del dipartimento è quello dell'utente che esegue l'operazione.
2. *Trigger* attivi:
   a) per definire le cosiddette *regole degli affari* (*business rules*), ovvero le azioni da eseguire per garantire la corretta evoluzione del sistema informativo. Ad esempio,

le azioni per la manutenzione automatica del magazzino, spedizione automatica di ordini e solleciti, passaggio di documenti fra utenti diversi ecc.;

b) per memorizzare eventi sulla base di dati per ragioni di controllo (*auditing and logging*). Ad esempio, in un'opportuna tabella si registrano dati sulle operazioni eseguite su tabelle riservate. In verità si memorizzano solo gli effetti di transazioni terminate normalmente perché di solito se una transazione termina prematuramente anche gli effetti dei *trigger* vengono annullati;

c) per propagare su altre tabelle gli effetti di certe operazioni su tabelle (base o calcolate);

d) per mantenere allineati eventuali dati duplicati quando si modifica uno di essi;

e) per mantenere certi vincoli d'integrità modificando la base di dati in modo opportuno; ad esempio, quando una ennupla viene cancellata, il vincolo referenziale può essere mantenuto in maniera attiva cancellando tutte le ennuple che fanno riferimento alla ennupla cancellata.

Vantaggi e problemi nell'uso dei trigger

Poiché i *trigger* sono memorizzati nella base di dati e la loro attivazione è sotto il controllo del DBMS, si hanno i seguenti vantaggi:

1. si semplifica la codifica delle applicazioni che non devono preoccuparsi dei controlli effettuati dai *trigger*;
2. i *trigger* sono definiti e amministrati centralmente e quindi gli utenti che usano la base di dati, in modo interattivo o per mezzo di applicazione, non possono evitare i controlli da essi garantiti.

Un problema che si presenta nell'uso dei *trigger* è che i sistemi commerciali adottano una diversa semantica per la loro attivazione e prevedono una diversa modalità di interazione fra i meccanismi di attivazione dei *trigger* e l'esecuzione della transazione che li attiva [Fraternali and Tanca, 1995]. In particolare, i punti principali che differenziano i vari sistemi sono:

– *Granularità*. Se la modifica riguarda un insieme di ennuple (come è possibile per i comandi **UPDATE**, **INSERT** e **DELETE**), in alcuni sistemi il *trigger* viene eseguito una sola volta per il comando (*trigger di comando*), mentre in altri viene eseguito tante volte quante sono le ennuple modificate dal comando (*trigger di riga*). Ad esempio, in Oracle è possibile scegliere fra i due casi: con la specifica esplicita **FOR EACH ROW** si dichiara che il *trigger* deve essere attivato tante volte quante sono le righe modificate. Un approccio diverso è adottato da Sybase, dove un *trigger* è attivato *una* sola volta per comando. In questo caso si possono usare le tabelle speciali inserted e deleted che contengono le righe inserite o cancellate dal comando (una modifica è considerata una cancellazione seguita da un'inserzione). L'esempio precedente diventerebbe così:

```
CREATE TRIGGER esempio3Trig
AFTER DELETE ON Agenti
BEGIN
    DELETE Totali
    WHERE CodiceAgente IN
            (SELECT deleted.CodiceAgente FROM deleted);
END;
```

- *Risoluzione dei conflitti*. Se è possibile definire più *trigger* attivabili dallo stesso comando, in alcuni sistemi l'ordine di definizione dei *trigger* stabilisce anche l'ordine in cui vengono attivati (Oracle), in altri sistemi si può specificare in quale ordine vanno attivati, in altri ancora l'ordine è stabilito dal sistema.
- *Trigger in cascata*. Se un *trigger* T_1 provoca l'attivazione di un altro T_2 (*trigger* in "cascata", *cascade trigger*, o annidati, *nested trigger*), si possono creare dei cicli se T_2 provoca l'attivazione di T_1 (*trigger* ricorsivi). In Sybase è possibile scegliere se permettere o meno le attivazioni in cascata (e in questo caso se permettere attivazioni ricorsive), mentre in Oracle le attivazioni ricorsive sono proibite.
- *Esecuzione dei trigger*. Di solito l'azione di un *trigger* viene eseguita immediatamente come parte della transazione che lo ha attivato. Un'altra possibilità da prevedere sarebbe di poter specificare che l'azione di un *trigger* andrebbe eseguita solo alla fine della transazione che lo attiva (esecuzione differita). Questa possibilità sarebbe utile nel seguente caso:

  1. supponiamo che esista un *trigger* T che elimina un dipartimento quando questo non ha un direttore;
  2. viene eseguita una transazione che elimina un impiegato direttore di un dipartimento, al quale assegna poi un nuovo direttore.

  Con l'esecuzione immediata di T il dipartimento viene eliminato, mentre con l'esecuzione differita di T la transazione produce l'effetto voluto.
- *Interazione con le transazioni*. Un altro aspetto critico è l'interazione dell'esecuzione dell'azione di un *trigger* con l'esecuzione della transazione che lo attiva. Di solito l'azione diventa parte della transazione che viene eseguita globalmente in modo atomico: se l'azione abortisce, abortisce anche la transazione e viceversa.

  In generale questo modo di procedere è quello desiderato, ma esistono casi in cui non lo è. Un esempio è quando un *trigger* viene usato per memorizzare eventi sulla base per ragioni di controllo, visto in precedenza: se una transazione legge un dato questo evento va registrato in un'opportuna tabella, anche nel caso in cui la transazione fallisca.

  Un'altra possibilità sarebbe di poter specificare che l'azione di un *trigger* vada eseguita come una transazione indipendente da quella che lo attiva.

Oltre a queste difficoltà semantiche, i *trigger* presentano problemi per ciò che riguarda la loro progettazione e la realizzazione di applicazioni in un ambiente in cui se ne faccia uso.

Per ciò che riguarda la progettazione dei *trigger*, il problema risiede nel fatto che molte metodologie, tra cui quella descritta nei primi capitoli di questo testo, non forniscono indicazioni riguardo all'utilizzo dei *trigger*, analogamente a quanto è già stato osservato a proposito delle procedure memorizzate. Tuttavia, una corretta progettazione dei *trigger* è cruciale per evitare poi problemi nella fase di realizzazione delle applicazioni.

Per ciò che riguarda la realizzazione delle applicazioni in un ambiente in cui sui dati siano definiti dei *trigger*, possiamo citare in particolare due problemi:

1. Complessità: chi realizza un'applicazione, per conoscerne gli effetti, deve capire non solo in che modo l'applicazione modifica la base di dati in modo diretto, ma anche qual è l'effetto dei *trigger* attivati da tale applicazione. Questo aumenta in generale la complessità della progettazione delle applicazioni. La situazione è ancora più complessa quando si utilizzano *trigger* attivi, poiché questi possono provocare l'attivazione indiretta di ulteriori *trigger*. In questa situazione, il comportamento del sistema tende a diventare imprevedibile, principalmente perché è difficile capire in che ordine e in che esatto momento i diversi *trigger* vengono effettivamente eseguiti. Trattandosi però di *trigger* che modificano lo stato, il tempo di esecuzione ne influenza in generale la semantica, che tende quindi ad uscire dal controllo del programmatore. A questo proposito, molti sistemi definiscono dei meccanismi automatici per impedire l'attivazione mutuamente ricorsiva di più *trigger*, a dimostrazione di quanto siano comuni situazioni in cui il comportamento di un insieme di *trigger* travalica le intenzioni di chi li ha disegnati.

2. Rigidità: un *trigger*, in genere, costringe al rispetto di un certo vincolo adottando una certa strategia; ad esempio, potrebbe forzare il vincolo che ogni dipartimento ha un direttore cancellando i dipartimenti che ne sono privi. Se una specifica applicazione intende mantenere lo stesso vincolo con una strategia diversa, questa applicazione può finire in conflitto con il *trigger*, come esemplificato in precedenza nel caso di una applicazione che vorrebbe cancellare il vecchio direttore e fissarne uno nuovo. In questo caso l'applicazione è costretta a cercare un modo per convivere con il *trigger*, poiché non è possibile, in generale, evitarne l'attivazione.

Concludiamo questa sezione accennando alle *basi di dati attive*: sono basi di dati in cui l'uso dei trigger è notevolmente ampliato al fine di definire delle regole (dette regole ECA, *evento-condizione-azione*), in cui gli eventi sono molto più ampi di quelli classici (ad esempio eventi temporali, eventi su condizioni qualunque, eventi composti, eventi esterni ecc.), e i meccanismi di valutazione delle regole sono più sofisticati. Queste regole vengono usate in maniera estensiva per arricchire gli aspetti funzionali delle basi di dati gestite, rendendole utilizzabili per lo sviluppo di applicazioni più sofisticate di quelle normalmente disponibili. Per approfondire l'argomento, e per esempi di DBMS attivi, si rinvia ai riferimenti riportati nelle note bibliografiche.

7.4 Progettazione fisica

La progettazione fisica di una base di dati è molto critica perché comporta numerose decisioni che devono tener conto delle caratteristiche del DBMS e del tipo di uso che le applicazioni fanno della base di dati. Per il carattere non specialistico del testo, ci limiteremo a ricordare che le principali informazioni che vanno fornite dalla progettazione fisica di una base di dati relazionale sono:

- distribuzione delle tabelle sugli archivi del sistema operativo; è necessario stabilire quali archivi contengano le tabelle, fornendo in particolare la possibilità di spezzare una singola tabella su archivi diversi, e di mantenere più tabelle in un singolo archivio;
- dimensioni degli archivi e loro organizzazione; l'organizzazione di un archivio determina, ad esempio, se i record sono inseriti nella tabella in ordine di arrivo, oppure ordinate sul valore di un attributo, oppure in base al risultato dell'applicazione di una funzione *hash* al valore di un attributo. Le organizzazioni dei dati verranno presentate nel Capitolo 9;
- presenza di indici; un indice su di un attributo A di una tabella è una struttura dati che permette di trovare rapidamente una registrazione nella tabella a partire dal suo valore per l'attributo A. L'importanza degli indici nell'esecuzione delle interrogazioni verrà mostrato nel Capitolo 9.

Come per gli aspetti procedurali, le modalità di definizione degli aspetti fisici di una base di dati non sono standardizzate. Vedremo quindi, a titolo di esempio, quelle previste in Oracle per la definizione degli indici.

7.4.1 Definizione di indici

Un indice I su un attributo A di una tabella R, dal punto di vista logico, è una tabella con due attributi $I(A, \text{TID})$, con gli elementi ordinati su A e valori $(A := a_i, \text{TID} := r_j)$, dove a_i è un valore di A presente in un'ennupla di R, ed r_j è un riferimento (TID) all'ennupla di R con il valore a_i di A. Se A non è una chiave, nell'indice sono presenti tante ennuple con lo stesso valore a_i di A quante sono le ennuple di R con il valore a_i di A.

Un indice può essere anche definito su un insieme di attributi, e in questo caso il primo elemento della coppia è formato da una combinazione dei valori relativi. Di solito una tabella è memorizzata in modo seriale, mentre un indice è memorizzato con una struttura ad albero per trovare con pochi accessi, a partire da un valore v, le ennuple di R in cui il valore di A è in una relazione specificata con v.

L'esistenza degli indici non cambia il modo in cui si formulano le interrogazioni (*indipendenza fisica*), ma viene sfruttata dall'ottimizzatore del sistema per stabilire le strategie di esecuzione delle interrogazioni, in particolare se e quali indici usare.

La creazione e distruzione di indici può essere fatta in ogni momento su tabelle già esistenti: questa operazione può invalidare i risultati di ottimizzazioni precedenti.

La forma base del comando per creare indici è:

CREATE [**UNIQUE**] **INDEX** *NomeIndice*
ON *NomeTabella*
"(" *Attributo* [**ASC** | **DESC**] {, *Attributo* [**ASC** | **DESC**] } ")"
[**TABLESPACE** *NomeArchivio*]
[**STORAGE** *ParametriDiMemoria*]

L'opzione **UNIQUE** è usata per specificare che l'indice riguarda un attributo chiave, **TABLESPACE** richiede la memorizzazione dell'indice nell'archivio specificato, e **STORAGE** modifica i parametri di allocazione fisica default dell'archivio. Le opzioni **ASC** e **DESC** sono usate per scegliere l'ordinamento per valori crescenti o decrescenti di un attributo, per default crescente.

Un indice può essere eliminato con il comando **DROP INDEX** *NomeIndice*.

I sistemi costruiscono automaticamente un indice sugli attributi delle chiavi, per controllare facilmente che i loro valori siano diversi nella tabella.

Come scegliere gli indici

In generale non è semplice stabilire quali indici creare su una base di dati per migliorare le prestazioni complessive delle applicazioni. Infatti, se è vero che gli indici favoriscono le operazioni di ricerca, è anche vero che occupano memoria (di solito si stima che un indice occupa il 20% della memoria occupata dalla relazione), e vanno aggiornati ad ogni operazione **INSERT**, **UPDATE** e **DELETE**, aumentando i loro tempi di esecuzione. Altro aspetto che complica il problema è che gli indici vanno scelti conoscendo le strategie di ottimizzazione del sistema per evitare di costruire indici che non verranno mai usati.

Sono stati studiati algoritmi opportuni per scegliere gli indici ed alcuni sistemi forniscono strumenti per aiutare il progettista nel farlo [Albano, 2001]. Si tengano comunque presenti i seguenti suggerimenti almeno per evitare scelte errate:

1. Non creare indici su tabelle piccole, ovvero che occupano meno di sei pagine.
2. Non creare indici su attributi poco selettivi, ovvero che hanno pochi valori diversi come il sesso o lo stato civile.
3. Evitare indici su attributi modificati frequentemente.
4. Prevedere più di quattro indici per relazione solo se le operazioni di modifica sono rare.
5. Creare indici sulle chiavi esterne per agevolare l'esecuzione delle operazioni di giunzione.
6. Sono utili indici ordinati su attributi usati frequentemente nelle opzioni **ORDER BY**, **DISTINCT** e **GROUP BY**. Infatti, l'esecuzione di interrogazioni con queste opzioni, in assenza di indici, comporta la creazione di tabelle temporanee da ordinare.
7. Prevedere indici su attributi usati frequentemente nelle operazioni di restrizione con condizione di uguaglianza o comunque molto restrittive.

7.5 Evoluzione dello schema

Una caratteristica dei sistemi relazionali, assente nei sistemi che li hanno preceduti, è la possibilità di modificare la struttura delle tabelle anche dopo che vi sono stati inseriti dei dati.

In particolare, una tabella può essere modificata con il comando **ALTER TABLE** per:

1. aggiungere un nuovo attributo:

   ALTER TABLE *Nome* **ADD** *Attributo Tipo*

   I valori della nuova colonna saranno tutti nulli (e non sarà possibile imporre il vincolo **NOT NULL** prima di assegnare nuovi valori a tutta la colonna);
2. eliminare un attributo:

   ALTER TABLE *Nome* **DROP** *Attributo*
3. modificare la definizione di un attributo:

   ALTER TABLE *Nome* **MODIFY** *Attributo Tipo*

   Le modifiche di tipo sono soggette a certe restrizioni di compatibilità. Ad esempio, è possibile passare da **INTEGER** a **FLOAT**, o da **CHAR**(4) a **CHAR**(6), ma non da **CHAR**(4) a **INTEGER**;
4. cambiare il nome di un attributo:

   ALTER TABLE*Nome* **RENAME** *VecchioAttributo NuovoAttributo*
5. aggiungere o eliminare un vincolo d'integrità:

   ALTER TABLE *Nome Vincolo*
   ALTER TABLE *Nome* **DROP** *Vincolo*

Ad esempio, se dopo aver creato la tabella Ordini si vuole aggiungere una colonna che registra i pagamenti effettuati, assumendo che gli ordini già fatti siano stati tutti pagati, si può scrivere:

```
ALTER TABLE Ordini ADD Pagato INTEGER;

UPDATE Ordini
    SET Pagato = Ammontare
```

7.6 Utenti e Autorizzazioni

Per garantire la protezione dei dati da accessi non desiderati, i DBMS forniscono normalmente un sistema di permessi basato sui concetti di *utente*, *autorizzazione* (o *privilegio*) e *profilo*.

Gli utenti sono creati dall'amministratore della base di dati, e da questi possono ricevere le autorizzazioni di base, che gli permettono di iniziare a lavorare (collegarsi,

creare schemi, tabelle, ecc.). Se un utente crea un oggetto, come una tabella, a sua volta può autorizzare altri utenti a lavorare su di esso.

Un'autorizzazione è il permesso di eseguire una o più operazioni, e si indica con il nome dell'operazione o con un nome simbolico (ad esempio **SELECT** o **CONNECT**). Un utente ha associato un insieme di autorizzazioni, che delimitano le operazioni che può effettuare (e i dati su cui può effettuarle).

Per semplificare la gestione dell'assegnazione delle autorizzazioni, sono definiti i profili, cioè insiemi di autorizzazioni che possono essere assegnati ad un utente in maniera globale, e che definiscono le categorie tipiche di utenti di una certa base di dati.

Un utente viene creato con il seguente comando:

CREATE USER *NomeUtente* **PROFILE** *NomeProfilo*
IDENTIFIED BY *Password*
DEFAULT TABLESPACE *NomeArchivio*
TEMPORARY TABLESPACE *NomeArchivio*
ACCOUNT UNLOCK

che crea un utente assegnandogli un profilo, una password e degli spazi di lavoro. Il profilo conterrà tipicamente almeno le autorizzazioni di base (ad esempio **CONNECT**, per collegarsi alla base di dati, o **RESOURCE** per costruire tabelle e altri oggetti di base).

Oltre alle autorizzazioni associate ad un profilo, il proprietario di una risorsa può assegnare e ritirare singole autorizzazioni sulla risorsa con i seguenti comandi, dei quali si mostrano solo alcune possibilità, riferite a tabelle base e tabelle virtuali:

GRANT *Autorizzazioni*
ON *Tabella*
TO (**PUBLIC** | *Utente* {, *Utente* })
[**WITH GRANT OPTION**]

REVOKE [**GRANT OPTION FOR**] *Autorizzazioni*
ON *Tabella*
FROM *Utente* {, *Utente* }

dove le autorizzazioni possibili sono le seguenti:

- **SELECT** per consentire l'accesso a tutte le colonne della tabella.
- **INSERT** per consentire l'inserzione di ennuple nella tabella.
- **UPDATE** per consentire la modifica dei campi delle ennuple della tabella. La forma ristretta **UPDATE**(*Attributi*) autorizza la modifica solo dei campi specificati.
- **DELETE** per consentire la cancellazione di ennuple dalla tabella.
- **ALL PRIVILEGES** per consentire tutte le operazioni.

Con il comando **GRANT** si autorizza l'uso di una tabella a tutti (**PUBLIC**) o solo ad alcuni utenti. Se viene specificato **WITH GRANT OPTION** gli utenti hanno a loro volta il diritto di concedere le stesse autorizzazioni ad altri utenti ancora (di nuovo con la clausola **WITH GRANT OPTION**).

Con il comando **REVOKE** si tolgono delle autorizzazioni a certi utenti, oppure, semplicemente, il diritto che hanno a trasferirle ad altri utenti (**REVOKE GRANT OPTION FOR**). La rimozione di un'autorizzazione (o del diritto a trasmetterla) ad un certo utente provoca in maniera automatica la rimozione della stessa autorizzazione a tutti gli altri utenti che l'avevano ricevuta da questo.

Il meccanismo delle autorizzazioni, insieme alle tabelle virtuali, offre una grande flessibilità nel controllo dell'accesso e della modifica dei dati. Ad esempio, per accedere ad una tabella virtuale V che usa nella propria definizione una tabella R è sufficiente avere i diritti relativi a V. In questo modo è possibile fornire ad un utente la possibilità di vedere o modificare solo quella parte di una tabella reale che è riflessa nella tabella virtuale, senza concedergli alcun diritto sulla tabella reale.

7.7 Schemi esterni

Uno schema esterno, secondo la proposta ANSI/X3/SPARC citata nel Capitolo 1, è la definizione della struttura della base di dati per una certa classe di utenti e della modalità d'uso dei dati ad essi consentita.

I DBMS relazionali offrono soluzioni diverse per definire schemi esterni.

1. Una base di dati è descritta da un unico schema S e con il meccanismo delle autorizzazioni si dichiara chi può accedere alle tabelle base o virtuali presenti nello schema e con quali modalità. Esiste, quindi, un comando **CREATE SCHEMA**, ma non un comando **CREATE EXTERNAL SCHEMA**.
2. Si possono definire più schemi S_i che usano tabelle base o virtuali presenti nello schema S che descrive la base di dati. Per ogni schema S_i si autorizzano gli utenti con le possibilità previste nel caso precedente. Con questa soluzione ogni schema S_i ha il ruolo di schema esterno come previsto dalla proposta ANSI/X3/SPARC.

In entrambi i casi, le tabelle virtuali sono il meccanismo fondamentale per modificare la visione della struttura dei dati memorizzati e per garantire l'indipendenza logica delle applicazioni.

7.8 Cataloghi

I sistemi relazionali prevedono che le informazioni relative allo schema (tabelle, viste, vincoli, *trigger*, utenti, autorizzazioni, indici ecc.), i cosiddetti *metadati*, vengano memorizzati in opportune tabelle interrogabili da utente, dette *catalogo del sistema*.

Vediamo alcuni attributi di un campione di possibili tabelle del catalogo per raccogliere informazioni su utenti, basi di dati, tabelle, colonne e indici:

PASSWORD (NomeUtente, ParolaChiave)
SYSDB(NomeBaseDati, Proprietario, Cammino, Commenti)
SYSTABLE (NomeTabella, Proprietario, BaseODerivata, NumeroColonne,
NomeArchivioFisico, Commenti)
SYSCOLS (NomeColonna, Tabella, Numero, Tipo, Lunghezza, Default, Commenti)
SYSINDEX (NomeIndice, Tabella, Proprietario, NumeroColonna, Commenti)

Altre tabelle, una decina, contengono informazioni sui vincoli, le tabelle virtuali, le autorizzazioni ecc.

Altre informazioni raccolte nei cataloghi di sistema riguardano aspetti quantitativi sui dati, le *statistiche*, e sono utilizzate dall'ottimizzatore delle interrogazioni.

Normalmente le tabelle del catalogo sono consultabili dall'utente, ma non modificabili per impedire interferenze con il funzionamento del sistema. Spesso queste tabelle sono delle "viste" di tabelle più complesse o vengono ricopiate dal sistema in strutture dati in memoria temporanea e ottimizzate per la valutazione delle interrogazioni.

Un utilizzo importante dei metadati è la loro consultazione interattiva da parte degli utenti (o dell'amministratore) usando l'SQL. Per ragioni di completezza i metadati saranno autoreferenziati: ad esempio, la tabella SYSTABLE conterrà anche una riga relativa a se stessa. Quindi la struttura dei metadati stessi può essere consultata (naturalmente conoscendo precedentemente un insieme minimo di informazioni essenziali).

7.9 Strumenti per l'amministrazione di basi di dati

Oltre al catalogo dei dati, molti altri strumenti sono offerti dai produttori di DBMS o da ditte indipendenti per assistere l'amministratore di basi di dati nelle numerose attività di sua competenza. Vediamone alcuni per le attività considerate più critiche:

1. progettazione concettuale di basi di dati e generazione dei comandi per la definizione dei relativi schemi relazionali;
2. definizione della memoria da assegnare alle tabelle e agli indici e sua riorganizzazione in caso di eccessiva frammentazione;
3. controllo dell'esecuzione di comandi SQL per migliorare le prestazioni delle applicazioni critiche;
4. pianificazione ed esecuzione delle procedure per la generazione di copie di sicurezza della base di dati;
5. controllo del funzionamento del DBMS e generazione di opportune statistiche su utilizzazione della memoria permanente e del buffer, operazioni di I/O, blocchi dei dati e condizioni di stallo delle transazioni attive ecc.

7.10 Conclusioni

Sono state presentate le caratteristiche dell'SQL per la definizione e amministrazione di basi di dati.

Il linguaggio messo a disposizione a questo scopo dai vari sistemi è in realtà, in genere, molto più complesso del sottoinsieme qui illustrato, ed è ricco di opzioni che differiscono in modo sostanziale da un sistema ad un altro. Questo non dipende tanto dalla povertà dello standard, quanto dal fatto che le attività di cui si occupa l'amministratore della base di dati, ovvero la definizione dello schema fisico, la gestione degli

schemi, degli utenti, delle autorizzazioni e dell'affidabilità, coinvolgono quegli aspetti fisici sui quali i vari sistemi differiscono in modo molto sensibile.

Esercizi

1. Si definisca uno schema relazionale con i comandi **CREATE TABLE** per trattare le informazioni e i vincoli d'integrità sui dipendenti di un'azienda, con attributi CodiceFiscale, Nome, AnnoAssunzione e Stipendio, e sui loro familiari a carico, con attributi Nome, AnnoNascita e RelazioneDiParentela.
2. Si supponga che sia stato definito uno schema relazionale con le istruzioni:

   CREATE TABLE R (K **CHAR**(8) **NOT NULL**, A **CHAR**(8), B **CHAR**(8))
   PRIMARY KEY(K)

   GRANT SELECT ON R **TO** caio

   e siano stati immessi dei dati in R. Usando i comandi:

   DROP TABLE Nome;
   CREATE TABLE Nome (Attributo Tipo, ...) **AS** ExprSQL'
   CREATE VIEW Nome (Attributo, ...) **AS** ExprSQL;
   GRANT SELECT ON Nome **TO** Utente;

   mostrare come si possa modificare lo schema in modo che i dati di R vengano memorizzati esclusivamente nelle tabelle:

   R1(K **CHAR**(8) **NOT NULL**, A **CHAR**(8))
   R2(K **CHAR**(8) **NOT NULL**, B **CHAR**(8))

   e l'utente "caio" possa continuare a lavorare sulla base di dati come se esistesse la tabella:

   R(K **INTEGER**(8) **NOT NULL**, A **INTEGER**(8), B **INTEGER**(8)).

3. Definire lo schema per la base di dati relazionale ottenuta dall'Esercizio 4.3.
4. Si mostri come trattare il vincolo di chiave esterna con i *trigger*.
5. Si ricorda che date due sottoclassi C_1 e C_2 di una classe C, diciamo che:

   a) C_1 e C_2 soddisfano il vincolo di copertura di C se $C_1 \cup C_2 = C$;
   b) C_1 e C_2 soddisfano il vincolo di disgiunzione se non hanno nessun elemento in comune.

   In generale si possono quindi avere quattro tipi di situazioni:

   a) copertura disgiunta o partizione (vincolo di copertura e di disgiunzione);
   b) copertura non disgiunta (solo vincolo di copertura);
   c) sottoinsiemi disgiunti (solo vincolo di disgiunzione);

d) sottoinsiemi non disgiunti (nessun vincolo).

Si supponga di rappresentare C_1, C_2 e C con tre relazioni RC_1, RC_2 ed RC, con RC_1 ed RC_2 che contengono gli attributi propri di C_1 e C_2 e una chiave esterna per RC. Si supponga di poter dichiarare nello schema solo il vincolo di chiave primaria e di chiave esterna. Si mostri se è possibile rappresentare i vincoli dei quattro tipi di sottoclassi con il comando **CREATE TABLE**.

6. Si risolva l'esercizio precedente usando i *trigger*.

Note bibliografiche

Per maggiori dettagli sui comandi SQL per definire e amministrare basi di dati si rimanda ai testi citati nel Capitolo 1, mentre per le estensioni previste dei vari sistemi commerciali è necessario consultare i relativi manuali. Un testo molto utile per la messa a punto delle basi di dati che ogni DBA dovrebbe consultare è [Shasha and Bonnet, 2002].

I *trigger* e le basi di dati attive, con esempi di sistemi e metodologie di progetto, sono discussi in [Widom and Ceri, 1996], [Zaniolo et al., 1997].

Capitolo 8

SQL PER PROGRAMMARE LE APPLICAZIONI

Uno dei principali obiettivi dei proponenti dell'SQL fu che il linguaggio dovesse essere utilizzabile direttamente da utenti occasionali per interrogare basi di dati, senza dover ricorrere a programmi sviluppati da esperti. Per questa ragione fu subito proposta anche una versione grafica del linguaggio per nascondere la sintassi SQL, chiamata *Query By Example* (*QBE*). L'obiettivo però è stato raggiunto solo in parte, perché ancora oggi l'uso interattivo di basi di dati con il linguaggio SQL è limitato a persone con competenze tecniche, nonostante i progressi fatti con le interfacce grafiche di ambienti interattivi tipo Microsoft Access per Windows, versione moderna del *QBE*. Nella maggioranza dei casi occorre invece sviluppare applicazioni interattive in modo che gli utenti del sistema informatico possano usare i servizi offerti senza nessuna competenza specifica. Per questo è necessario disporre di opportuni linguaggi di programmazione che consentano sia l'accesso a basi di dati che la realizzazione delle interfacce grafiche per il dialogo con gli utenti.

Un altro motivo per cui tali linguaggi sono necessari è la necessità di scrivere applicazioni che usano basi di dati per una grande varietà di compiti diversi: applicazioni gestionali, analisi complesse, applicazioni web ecc. In tutti questi casi la tendenza attuale è di usare SQL come componente dedicata all'accesso ai dati all'interno di un linguaggio di programmazione adatto al compito specifico.

La differenza fra le varie soluzioni è data soprattutto dalla modalità di integrazione fra SQL e il linguaggio di programmazione. In generale le soluzioni più comuni possono essere raggruppate nelle seguenti categorie:

1. Linguaggi di programmazione generale, come C, C++, Java, Visual Basic, la cui sintassi viene estesa con costrutti SQL per operare sulla base di dati (si dice anche che *ospitano* l'SQL).
2. Linguaggi tradizionali con i quali l'uso della base di dati avviene attraverso la chiamata di funzioni di una opportuna libreria (*Application Programming Interface, API*).
3. Linguaggi cosiddetti della *quarta generazione* (4^{th} *Generation Languages, 4GL*), come Informix 4GL, Oracle PL/SQL, e Sybase Transact/SQL. Questi sono linguaggi di programmazione di tipo generale costruiti "attorno" ad SQL, nel senso che lo estendono con costrutti di controllo ed altri tipi di dati, ma si basano sul modello dei dati relazionali.

8.1 Linguaggi che ospitano l'SQL

Un approccio classico allo sviluppo di applicazioni per basi di dati relazionali prevede l'immersione dei costrutti di SQL in un linguaggio di programmazione tradizionale (come COBOL, C, Basic, Java, C++) con una sintassi "aggiuntiva", senza alterare nè la sintassi corrente nè il sistema dei tipi.[1] Poiché in questi linguaggi non è previsto il tipo relazione e il tipo ennupla, vengono usati opportuni accorgimenti per scambiare dati fra la "parte" SQL e la parte tradizionale del linguaggio.

I vantaggi di questo approccio sono diversi:

- il costo ridotto di addestramento dei programmatori, che continueranno ad usare un linguaggio già conosciuto per la parte di applicazione che non tratta direttamente la gestione dei dati persistenti;
- la semplicità della sintassi di estensione, che rende i programmi più comprensibili rispetto ad approcci basati su chiamate di funzione;
- la possibilità di usare meccanismi di controllo dei tipi per validare durante la compilazione la correttezza dell'uso dei comandi SQL;
- la possibilità di ottimizzare le interrogazioni durante la compilazione del programma;
- la possibilità di attivare meccanismi di sicurezza basati sull'analisi del codice SQL.

Lo svantaggio principale, invece, è il fenomeno detto di *impedence mismatch*: la differenza fra i tipi di dati del linguaggio e quelli relazionali obbliga a curare la conversione dei dati fra i due diversi modelli. Ad esempio, per trattare il risultato di una interrogazione SQL (una relazione, cioè un insieme) occorre usare costrutti iterativi propri del linguaggio, operando su un solo elemento alla volta.

In questa sezione, si mostra come vengono immersi i comandi SQL (*Embedded SQL*) nel linguaggio Java. Il linguaggio risultante, detto SQLJ, presenta una serie di vantaggi:

- è un linguaggio standard, definito dall'organismo internazionale di standard ISO, quindi i programmi scritti con questo linguaggio sono utilizzabili su DBMS diversi, al contrario di altre soluzioni basate su linguaggio ospite;
- il traduttore dal linguaggio esteso al linguaggio di base, Java, è a sua volta scritto in Java, così come il sistema di supporto per l'esecuzione (*SQLJ runtime*), quindi, sfruttando la portabilità di Java, questa è una soluzione disponibile su moltissimi sistemi operativi;
- vengono usati gli aspetti "object-oriented" di Java per semplificare la comunicazione fra le due componenti del linguaggio.

In ogni caso, qualunque sia il linguaggio ospite che si utilizza, un'approccio al problema dell'uso di basi di dati relazionali all'interno di un linguaggio di programmazione deve risolvere i seguenti aspetti fondamentali:

1. In inglese vengono detti *SQL embedded languages*, cioè linguaggi con SQL immerso.

1. come indicare la connessione alla base di dati e fornire le credenziali di utente;
2. come specificare i comandi SQL, associando ad eventuali loro parametri dei valori calcolati dal programma;
3. come accedere ai risultati di un comando SQL, in particolare ad una relazione restituita da un'interrogazione;
4. come gestire le condizioni anormali di esecuzione di un comando, ed eventualmente come gestire questo aspetto in relazione alle proprietà di atomicità di una transazione.

Questi punti verranno esaminati nel seguito per il linguaggio SQLJ, ma si tenga presente che le soluzioni adottate sono simili a quelle offerte da altri linguaggi.

8.1.1 Connessione alla base di dati

Il concetto di *connessione*, o *contesto di connessione* (*connection context*), indica in generale il contesto di lavoro a cui dei comandi SQL fanno riferimento. Un contesto specifica la base di dati a cui si fa riferimento, con quale nome di utente si accede, e qual'è il tipo dei dati utilizzati. Un contesto è un oggetto di una classe (o tipo oggetto) Java particolare, detta *classe di contesto*, che viene dichiarata e istanziata come nel seguente esempio:[2]

```
Class.forName(DatabaseDriver);
#sql context ClasseContesto;
ClasseContesto contesto = new ClasseContesto(url, utente, password);
```

La prima riga serve per caricare nel sistema il *driver* di accesso alla base di dati, cioè la libreria specifica al DBMS che si vuole utilizzare. Ad esempio, se volessimo accedere ad un sistema Oracle con il driver JDBC della stessa azienda, dovremmo dare come parametro "oracle.jdbc.driver.OracleDriver".

La seconda riga è scritta con la sintassi SQLJ estesa (ogni comando SQLJ deve iniziare con il simbolo #sql), e indica che nel programma viene definita una nuova classe di contesto, con campi e metodi predefiniti, chiamata in questo esempio ClasseContesto.

Infine nell'ultima riga si crea una nuova istanza di questa classe, un oggetto di nome contesto, che verrà usato in tutte le operazioni successive. Il costruttore prende come parametri tre stringhe: la prima è il riferimento alla base di dati, la seconda è il nome dell'utente e la terza è la parola chiave dell'utente specificato.[3]

2. Vi sono in realtà diversi modi per creare una classe di contesto e una sua istanza. Nell'esempio viene mostrato il modo più semplice.
3. Ad esempio il riferimento alla base di dati Oracle di nome acmedb sul server db.acme.com sarà: "jdbc:oracle:thin:@db.acme.com:1521:acmedb" .

8.1.2 Comandi SQL

Una volta aperta una connessione, è possibile usare i comandi SQL premettendovi il simbolo #sql e il nome del contesto all'interno del quale devono essere eseguiti. Ad esempio, se nella base di dati esiste una tabella Province con attributi Nome, Sigla, stringhe di caratteri, e NumeroComuni un numero intero, potremmo creare una nuova Provincia con la riga seguente:

```
#sql [contesto] INSERT INTO Province VALUES ("Milano", "MI", 166 );
```

Normalmente, però, i dati in un comando SQL non sono costanti, ma valori di variabili del programma che vanno usate aggiungendo il prefisso ":". Se volessimo così effettuare un'inserzione usando dati letti da terminale, potremmo scrivere:

```
String provincia, sigla;
int numeroComuni;
... lettura dei valori corretti nelle tre variabili sopra definite ...
#sql [contesto] INSERT INTO Province VALUES (:provincia, :sigla, : numeroComuni);
```

L'uso delle variabili consente il passaggio di dati non solo dal programma alla base di dati, come nell'esempio precedente, ma anche nella direzione opposta, dalla base di dati al programma, usando la versione del comando **SELECT** esteso con la clausola **INTO** per assegnare a delle variabili il valore degli attributi dell'*unica* ennupla del risultato di una **SELECT**, come nel seguente esempio:

```
int numeroProvince;
#sql [contesto] SELECT COUNT(*) INTO :numeroProvince FROM Province
System.out.println("In Italia ci sono" + numeroProvince + " province.");
```

Questa forma del comando **SELECT** non si può usare se il risultato è un insieme di ennuple, ma per accedere ad esse una per volta si utilizza il meccanismo dei *cursori* (chiamati in SQLJ, *iterator*, o *result set iterators*).

8.1.3 I cursori

Un cursore è un oggetto che rappresenta un insieme di ennuple, con metodi che permettono di estrarre dall'insieme un elemento alla volta. Ad esempio, se vogliamo stampare tutte le province con più di 60 comuni, potremmo scrivere:

```
#sql iterator IteratoreProvince (String nome, int comuni);
IteratoreProvince province;
#sql [contesto] province = {SELECT    Nome, NumeroComuni
                            FROM      Province
                            WHERE     NumeroComuni > 60};
while (province.next()) {
    System.out.println(province.nome() + " " + province.comuni());
}
province.close();
```

La prima riga, analogamente alla dichiarazione di una classe di contesto, è una dichiarazione di una classe di iteratore di nome IteratoreProvince. Gli oggetti istanza di questa classe verranno usati per scorrere insiemi di ennuple con due campi, una stringa e un intero.

La seconda riga dichiara una variabile di tipo iteratore, che viene inizializzata nella terza riga al risultato di un comando **SELECT**. Il risultato del comando è quindi un oggetto iteratore che ha il tipo compatibile con IteratoreProvince: infatti la **SELECT** effettua la proiezione su due attributi, il primo stringa e il secondo intero.

Un oggetto iteratore ha associato in ogni istante un'ennupla del risultato, che chiameremo *riga corrente*, oppure un valore nullo (all'inizio e alla fine dopo la scansione di tutte le ennuple). Il metodo booleano **NEXT** ha un duplice scopo: ritorna vero o falso per indicare se ci sono ancora ennuple da scorrere, e fa diventare corrente l'ennupla successiva (o la prima, quando viene chiamato la prima volta). Quindi è sufficiente usarlo all'interno della condizione del comando **WHILE** per scorrere in sequenza tutte le ennuple del risultato e terminare il ciclo quando non ce ne sono più. Il corpo del ciclo, invece, mostra come viene usata l'ennupla corrente: attraverso dei metodi, con i nomi uguali a quelli dati nella dichiarazione della classe di iteratore, si selezionano i campi associati. Così il metodo nome restituisce il valore dell'attributo Nome dell'ennupla corrente, mentre il metodo comuni restituisce il valore dell'attributo NumeroComuni. Alla fine della scansione l'iteratore viene chiuso, rilasciando così le risorse impegnate per la gestione del risultato.

8.1.4 Transazioni

Come abbiamo già visto nel Capitolo 1, una transazione è un programma che non può avere un effetto parziale: la transazione termina correttamente, oppure, se si verifica un errore, ne vengono disfatti tutti gli effetti ed è come se la transazione non fosse mai iniziata. Come si concretizza questo concetto all'interno di un programma per basi di dati?

I vari linguaggi rispondono in maniera diversa a questa domanda. SQLJ, in particolare, stabilisce la seguente regola iniziale: *ogni singolo comando SQL è considerato una transazione a sè stante* (regola di *autocommit on*). Dato che non sempre questa regola è soddisfacente, ad esempio perchè vogliamo fare una serie di letture e scritture che potrebbero essere completamente annullate in seguito al verificarsi di qualche evento particolare, è possibile imporre la regola di *autocommit off*: una transazione inizia con il primo comando SQL, prosegue con i comandi successivi, e termina solo quando si dà un comando esplicito di terminazione (con successo **COMMIT** o con fallimento **ROLLBACK**).

Se non viene specificato altrimenti, viene seguita la regola *autocommit on*; si può specificare quale delle due regole seguire con un quarto parametro al costruttore di contesto: se questo è true si vuole il comportamento standard, se invece è false si vuole disabilitare l'*autocommit*. In questo secondo caso, una transazione inizia con il primo comando SQL e viene terminata dando il comando SQLJ:

```
#sql [contesto] COMMIT;
```

se vogliamo far terminare la transazione con successo e quindi salvare le modifiche, oppure

#sql [contesto] **ROLLBACK**;

se vogliamo far fallire la transazione annullando tutte le modifiche effettuate dall'inizio.

Si noti che la gestione del fallimento delle transazioni è indipendente dal verificarsi di errori SQL all'interno del programma. Infatti, quando un comando SQL provoca un errore, questo viene riflesso nel programma Java sotto forma di una eccezione di classe SQLException. Sta al programmatore gestire l'eccezione e decidere come comportarsi rispetto alla transazione corrente (in caso di *autocommit off*, dato che nell'altro caso la transazione consiste solo dell'operazione SQL che ha provocato errore e che quindi non viene eseguita).

Spetta quindi al programmatore la decisione della regola da seguire: nei casi più semplici è sufficiente lasciare quella default (*autocommit on*), mentre nei casi più complessi sarà necessario impostare *autocommit off* e gestire esplicitamente l'inizio e la fine delle transazioni. Alla fine di questo capitolo, nella Sezione 8.4, questo argomento verrà discusso in dettaglio e verranno mostrati degli esempi.

Esempio 8.1
Usando la base di dati del capitolo precedente, si mostra un programma per (a) la stampa dell'ammontare di un certo ordine, (b) l'aggiornamento della zona di un agente e (c) la stampa delle coppie (clienti, ammontare degli ordini) ordinate in modo decrescente secondo l'ammontare.

```
import java.sql.*;
import sqlj.runtime.*;

public class Esempio {
    public static void main(String [] args) {
        try {
            // Connessione alla base di dati
            Class.forName("oracle.jdbc.driver.OracleDriver");
            #sql context ClasseContesto;
            ClasseContesto contesto =
                new ClasseContesto("jdbc:oracle:thin:@localhost:1521:mydb",
                        "utente1","password1");

            // Ricerca dell'ammontare di un certo ordine
            // Lettura dei parametri
            String numeroAgente = leggi("Numero agente: ");
            String numeroCliente = leggi("Numero cliente: ");
            String numeroOrdine = leggi("Numero ordine: ");
            int ammontare;
```

```
        #sql [contesto] SELECT   Ammontare INTO  :ammontare
                        FROM     Ordini
                        WHERE    CodiceAgente = :numeroAgente
                        AND      CodiceCliente = :numeroCliente
                        AND      NumOrdine = :numeroOrdine;

        // Stampa risultato
        System.out.println("L'ammontare e': " + ammontare);

        // Aggiornamento della zona di un agente
        numeroAgente = leggi("Numero agente: ");
        String zona = leggi("Zona: ");

        #sql [contesto] UPDATE   Agenti SET Zona = :zona
                        WHERE    CodiceAgente = :numeroAgente;

        // Stampa le coppie (numero cliente, ammontare ordini)
        // ordinate in modo decrescente secondo l'ammontare
        #sql iterator IteratoriClientiOrdini(String codCliente, int ammontare);
        IteratoriClientiOrdini risultato;
        #sql [contesto] risultato = SELECT      CodiceCliente, SUM(Ammontare)
                                    FROM        Ordini
                                    GROUP BY    CodiceCliente
                                    ORDER BY    SUM(Ammontare) DESC;

        //Stampa intestazione tabella
        System.out.println("Cliente Uscite");
        while (risultato.next()) {
            System.out.println(risultato.codCliente() + " " + risultato.ammontare());
        }
        risultato.close();

        } catch (SQLException e) {
                System.err.println("SQLException " + e);
                } finally {
                        try {
                            contesto.close(); // disconnessione dal DBMS
                          } catch (SQLException e) {
                                System.err.println("SQLException " + e);
                                }
                        }
}
```

```
    static BufferedReader reader =
        new BufferedReader(new InputStreamReader(System.in));

    static String leggi(String messaggio) throws IOException {
        System.out.println(messaggio);
        return reader.readLine();
    }
}
```

8.2 Linguaggi con interfaccia API

L'uso della base di dati con i comandi SQL immersi nel linguaggio ospite richiede la presenza di un compilatore apposito, o, come avviene più frequentemente, di un precompilatore che trasforma il programma nel linguaggio esteso in un programma nel linguaggio base con opportune chiamate al sistema a run-time del DBMS.

Ci sono tuttavia casi in cui non è disponibile il precompilatore, oppure, per motivi di efficienza e di flessibilità, ci si vuole interfacciare direttamente dal programma con il sistema di gestione di basi di dati.

In tali casi viene fornita una libreria di funzioni *API* da richiamare nel linguaggio tradizionale utilizzato passando opportuni parametri, che possono essere, ad esempio, comandi SQL sotto forma di stringhe di caratteri, oppure informazioni di controllo o per lo scambio dei dati.

La possibilità di passare al DBMS i comandi SQL sotto forma di stringhe, se da un lato comporta lo svantaggio che eventuali errori nei comandi possono essere individuati solo durante l'esecuzione del programma, e non durante la sua compilazione, dall'altro permette l'utilizzo del cosiddetto *Dynamic SQL*. Con questo termine si intendono programmi in cui i comandi SQL non sono noti durante la scrittura del programma, ma vengono calcolati durante la sua esecuzione, ad esempio leggendoli come stringhe da terminale, o generandoli con informazioni ricavate dall'accesso al catalogo della base di dati.

In Figura 8.1 viene mostrata la differenza fra i due approcci: quello del linguaggio ospite (linee continue) e delle chiamate dirette delle librerie messe a disposizione dal DBMS, nel caso del linguaggio Java (linee tratteggiate).

Le principali soluzioni adottate, per mantenere la portabilità del codice, prevedono l'utilizzo di librerie che, sebbene siano diverse per i vari DBMS e ambienti di esecuzione, hanno un'interfaccia standard, che viene usata all'interno dei programmi applicativi sempre nello stesso modo. Il programma, una volta compilato rispetto all'interfaccia, viene quindi eseguito insieme alla libreria opportuna per il suo ambiente di esecuzione e per il DBMS a cui deve accedere. Un vantaggio ulteriore di questo approccio è che queste librerie prevedono in generale che il DBMS risieda su un elaboratore diverso da quello in cui si esegue il programma, e si occupano di gesti-

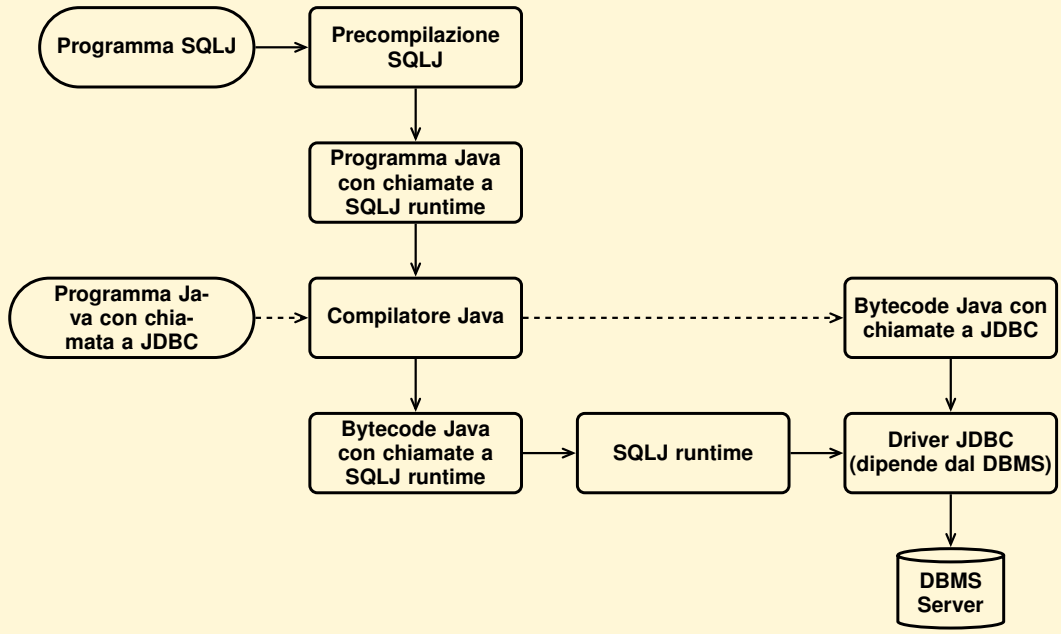


Figura 8.1: Sviluppo di applicazioni con linguaggio ospite e con chiamate ad API del DBMS

re in maniera trasparente tutte le problematiche di comunicazione con un approccio *client-server*.

8.2.1 L'API ODBC

L'API ODBC (*Open Data Base Connectivity*) è, attualmente, uno standard molto diffuso per l'utilizzo di DBMS relazionali. ODBC è stato definito dalla Microsoft, e specifica un DDL ed un DML relazionali basati sullo standard SQL CLI (*Call Level Interface*) proposto dal comitato X/Open.

Uno strumento che implementi l'API ODBC è composto principalmente da un insieme di *driver* o librerie ODBC. Un *driver* ODBC per il DBMS X è una libreria che traduce le chiamate generiche ODBC in chiamate allo specifico sistema X e gliele invia, appoggiandosi ad un sistema di comunicazione (Figura 8.2).

ODBC permette di accedere ad un sottoinsieme limitato di SQL, quello implementato da tutti i sistemi relazionali, ma permette anche di usare estensioni non standard presenti su di uno specifico sistema, anche se questo riduce la portabilità dell'applicazione. Sono disponibili anche *driver* ODBC per sistemi di archiviazione; questi *driver* interpretano in proprio le istruzioni SQL.

Mentre ODBC è un'API utilizzabile per lo sviluppo di applicazioni in C o in altri linguaggi di programmazione, ne esistono anche delle versioni integrate in ambienti

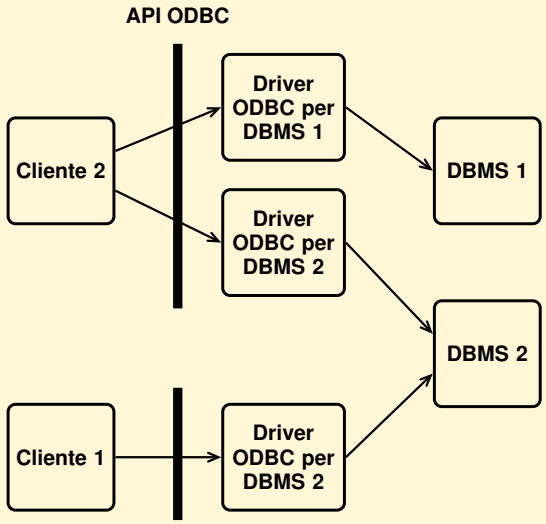


Figura 8.2: Uso dei *driver* ODBC

di sviluppo di vario tipo, come Visual Basic di Microsoft, che ne permettono anche l'uso all'interno di strumenti di produttività individuale. In particolare, tutti gli strumenti più diffusi per l'implementazione di interfacce hanno la capacità di sfruttare *driver* ODBC.

Il seguente esempio mostra un semplice programma Visual Basic che può essere richiamato all'interno di un foglio elettronico Excel, per inserire i risultati di una interrogazione SQL in una colonna del foglio di calcolo con nome Risultato:

```
Procedure TrovaNomiStudenti()

' Si apre la connessione alla base di dati
connessione = SQLOpen("DSN=MioDatabase;UID=ut1;PWD=pw1")

' Si esegue l'interrogazione (senza recuperare i dati)
SQLExecQuery connessione; "SELECT Nome FROM Studenti"

' Si dichiara che i risultati saranno restituiti a partire
' dalla prima cella del foglio di calcolo
SQLBind connessione; 1
FogliDiLavoro("Risultati").Celle(1; 1)

' Si recuperano effettivamente i dati
SQLRetrieve connessione
```

```
' Si chiude la connessione
SQLClose connessione
End Procedure
```

8.2.2 L'API JDBC

L'API JDBC (*Java Data Base Connectivity*) può essere definita come la versione Java di ODBC, e ha implementazioni analoghe a quelle di ODBC. L'interesse per questo tipo di API è duplice:

- grazie alla portabilità dei programmi Java, con JDBC si ha il vantaggio di poter sviluppare applicazioni indipendenti non solo dal tipo di sistema che gestisce la base di dati, ma anche dal tipo di piattaforma su cui l'applicazione viene eseguita;
- JDBC è un'interfaccia per un linguaggio ad oggetti, Java, progettato per evitare quelle caratteristiche del C e C++ che rendono la programmazione rischiosa.

JDBC è lo strumento utilizzato per la realizzazione del linguaggio SQLJ: tutti i comandi SQL di tale linguaggio vengono trasformati dal precompilatore in chiamate alle primitive JDBC. In effetti, dato che i compilatori SQLJ sono ancora scarsamente diffusi, la programmazione di basi dati relazionali dal linguaggio Java è fatta molto spesso con JDBC.

La logica con cui dal programma si utilizza il DBMS è molto simile a quella di SQLJ. Le differenze principali sono nel fatto che non è possibile nessun controllo dei tipi, mentre altre differenze dipendono dalle classi diverse che sono usate per accedere ai dati:

- **Connection** per stabilire il collegamento con una base di dati: una connessione è l'equivalente di un contesto in SQLJ;
- **Statement** per costruire il comando SQL da inviare al sistema tramite una *connection* , e
- **ResultSet** per ricevere il risultato (equivalente al *result set iterator* di SQLJ).

Vediamo un semplice esempio d'uso di JDBC.

```
import java.sql.*;
class StampaNomiStudenti {
    public static void main(String argv[]) {
        try {

            // URL della base di dati da usare
            String url = "jdbc:odbc:MioDatabase";

            // si apre la connessione con la base di dati
            Connection con = DriverManager.getConnection(url, "utente1", "password1");
```

```
        // si esegue un comando SELECT
        Statement stmt = con.createStatement();
        ResultSet risultato = stmt.executeQuery("SELECT Nome FROM Studenti");

        // scansione del risultato usando il metodo next
        System.out.println("Nomi trovati:");
        while (risultato.next()) {
            String nome = risultato.getString("Nome");

            // Stampa del nome trovato
            System.out.println(" Nome = " + nome);
        }

        risultato.close();
        stmt.close();
        con.close();
    }
    ...
  }
}
```

Si noti la differenza dell'uso del ResultSet rispetto a SQLJ: dato che a tempo di compilazione non si conosce la struttura della base di dati, non si possono usare metodi specifici per accedere ai campi dell'ennupla corrente. Si usano invece metodi generici (getString, getInteger ecc.) che prendono come parametro o il nome dell'attributo, oppure il suo numero d'ordine.

La gestione delle transazioni è analoga a quella di SQLJ, attraverso il meccanismo di *autocommit* (che può essere esplicitamente manipolato con il metodo setAutocommit(Boolean) di una connessione). Si può anche richiedere l'esecuzione della transazione con un certo livello di isolamento con un opportuno valore del parametro del metodo setTransactionIsolation.

Per passare valori dal programma ai comandi SQL, dato che sono solamente stringhe, si può procedere in due modi diversi:

- usando la concatenazione di stringhe, come nell'esempio seguente, in cui viene usato il valore della variabile matricola per indicare la matricola dello studente da ricercare:

  ```
  stmt.executeQuery("SELECT Nome FROM Studenti WHERE Matricola = " + matricola);
  ```

- usando la classe PreparedStatement, che le cui istanze corrispondono a comandi SQL all'interno dei quali vi sono dei parametri indicati da "?" che si possono sostituire con valori qualunque, in maniera ordinata, come nel seguente frammento di programma:

```
PreparedStatement pstmt =
        con.prepareStatement("SELECT Nome FROM Studenti WHERE Matricola = ?");
pstmt.setInt(1, matricola);
risultato = pstmt.executeQuery();
```

Si noti che il secondo metodo è preferibile per evitare che caratteri particolari all'interno dei parametri, come le virgolette, interferiscano con le operazioni di concatenamento di stringhe o con la sintassi del comando SQL.

Infine, è importante sottolineare come l'interfaccia JDBC permetta anche lo sviluppo di applicazioni con SQL dinamico (*Dynamic SQL*), cioè in cui i comandi SQL che devono essere eseguiti sono calcolati a tempo di esecuzione.

Esempio 8.2
Si consideri ad esempio un programma che, consultando il catalogo, propone all'utente la lista delle tabelle della base di dati chiedendogli quale di queste vuole interrogare. Una volta che l'utente ha fatto la sua scelta, il programma potrebbe proporre l'elenco degli attributi, e permettere di all'utente di dare i valori di alcuni attributi per restringere la ricerca, e di indicare gli attributi di cui vuole la visualizzazione. In base a queste informazioni il programma potrebbe sintetizzare la query sotto forma di stringa, con la sicurezza che sia comunque corretta perché generata utilizzando le informazioni del catalogo dei dati.

Un programma come quello descritto nell'esempio precedente può essere facilmente scritto con JDBC utilizzandone le funzionalità che permettono di interrogare, in maniera indipendente dal particolare DBMS usato, il catalogo dei dati (i metadati). Esiste infatti il metodo getMetaData di un oggetto Connection che restituisce un oggetto istanza della classe predefinita DataBaseMetaData, con moltissimi metodi per conoscere tutti gli elementi del catalogo dei dati (tabelle, attributi, chiavi, chiavi esterne, procedure memorizzate ecc.). Inoltre, per visualizzare il risultato di una interrogazione espressa con una stringa calcolata a tempo di esecuzione, si può usare il metodo getMetaData dell'istanza di Result Set restituita. Questo metodo produce un oggetto di tipo ResultSetMetaData, che descrive completamente il risultato dell'interrogazione (numero di attributi, loro tipo, lunghezza ecc.).

8.3 Linguaggi integrati

Per ovviare al problema dell'*impedence mismatch* presente nell'uso di SQL con un linguaggio ospite, si integrano i costrutti di un linguaggio relazionale e di un linguaggio di programmazione in un unico linguaggio. Il primo esempio di linguaggio progettato con questo obiettivo è stato il Pascal/R, in cui il sistema dei tipi del Pascal è esteso con un nuovo tipo di dato, la relazione, e di costrutti per agire su relazioni [Schmidt, 1977].

Una direzione completamente diversa è stata invece presa da molti costruttori di sistemi relazionali commerciali, con i cosiddetti linguaggi della quarta generazione (4GL): l'idea è quella di costruire un linguaggio di programmazione estendendo l'SQL con costrutti di controllo di tipo generale, spesso derivati da linguaggi tipo BASIC.

L'esempio che presenteremo in questa sezione è il PL/SQL, linguaggio del sistema ORACLE.

Le caratteristiche principali del PL/SQL sono le seguenti:

- Si possono definire variabili o costanti dei tipi dei domini relazionali, sia esplicitamente (come in **DECLARE** x **CHAR**(2); che dichiara x di tipo stringa di due caratteri), che uguali al tipo di una colonna (come in **DECLARE** x Clienti.CognomeENome%**TYPE**; che dichiara x dello stesso tipo della colonna CognomeENome della tabella Clienti). Si possono, inoltre, dichiarare variabili di un tipo ennupla, come in **DECLARE** Cliente Clienti%**ROWTYPE**.
- Si possono definire cursori, equivalenti agli iteratori SQLJ, per operare in maniera iterativa su un insieme di ennuple restituite da una **SELECT**. Un cursore è associato ad una espressione SQL, come ad esempio in **DECLARE CURSOR** c1 **IS SELECT** Zona **FROM** Agenti, e quindi può essere usato in due modi: all'interno di un costrutto **FOR**, per scandire tutte le ennuple del risultato dell'espressione associata (come in **FOR** z **IN** c1 **LOOP** ... **END**), oppure con dei comandi per eseguire il ciclo in maniera esplicita (**OPEN**, **FETCH**, **CLOSE**).
- I comandi possibili sono l'assegnamento, tutti comandi SQL (per cui il linguaggio è in effetti un soprainsieme dell'SQL), e i tradizionali costrutti di controllo (**IF**, **WHILE**, **LOOP** ecc.).
- Esiste un meccanismo di gestione delle eccezioni, sia generate da operazioni illegali, che esplicitamente dall'utente. Il verificarsi di un'eccezione provoca l'annullamento delle modifiche alla base di dati.
- Si possono definire funzioni e procedure con parametri (**PROCEDURE**) in maniera usuale.
- Si possono definire moduli (**PACKAGE**) che racchiudono procedure, funzioni, cursori e variabili collegate. Un modulo ha una parte pubblica, che elenca le entità esportabili dal modulo, ed una privata, che contiene la realizzazione delle procedure pubbliche ed eventuali variabili e procedure locali al modulo.
- Sia le singole procedure e funzioni che i moduli possono essere *memorizzati* nella base di dati, assegnando loro un nome (**CREATE PROCEDURE**, **CREATE PACKAGE**). Il sistema memorizza procedure e moduli in forma compilata, ma mantiene anche il codice sorgente e tiene traccia delle dipendenze in esso presenti, per ricompilarlo nel caso di modifiche dei dati usati (ad esempio, se viene modificata la definizione di una tabella).
- Si possono definire dei *trigger* (**CREATE TRIGGER**), come discusso nel capitolo precedente.

Un programma in PL/SQL viene scritto sotto forma di blocco anonimo (anonymous block), e può essere presentato, interattivamente oppure in un archivio di testo, ad uno dei vari tool di ORACLE per l'esecuzione. Usando i comandi **CREATE PROCEDURE** o

CREATE PACKAGE si può memorizzare una procedura o un modulo nella base di dati. Procedure e componenti di moduli memorizzati possono quindi essere richiamate ed usate da altri programmi, anche remoti.

Esempio 8.3
Riferendoci allo schema già presentato dei clienti, agenti e venditori, si vuole: (*a*) stampare l'ammontare di un certo ordine; (*b*) aggiornare la zona di un agente e (*c*) stampare le coppie (clienti, ammontare degli ordini) ordinate in modo decrescente secondo l'ammontare.

```
DECLARE
    NumCliente Clienti.CodiceCliente%TYPE;
    NumOrdine Ordini.NumFattura%TYPE;
    NumAgente Agenti.CodiceAgente%TYPE;
    NomeZona Agenti.Zona%TYPE;
    VAmmontare Ordini.Ammontare%TYPE;
    TotaleAmmontare INTEGER;
BEGIN
/* Ricerca dell'ammontare di un certo ordine */
PRINT "Scrivi CodiceAgente, CodiceCliente, NumOrdine";
READ NumAgente, NumCliente, NumOrdine;

SELECT Ammontare INTO VAmmontare
FROM   Ordini
WHERE  CodiceAgente=NumAgente AND CodiceCliente=NumCliente
AND    NumOrdine=NumOrdine;

PRINT VAmmontare;
/* Aggiornamento della zona di un agente */
PRINT "Scrivi CodiceAgente, Zona";
READ NumAgente, NomeZona;

UPDATE Agenti
SET    Zona = NomeZona
WHERE  CodiceAgente = NumAgente;

/* Creazione di un cursore per i clienti e l'ammontare */
DECLARE
   c CURSOR IS
      SELECT    CodiceCliente, SUM(Ammontare) AS Totale
      FROM      Ordini
      GROUP BY  CodiceCliente
      ORDER BY  SUM(Ammontare) DESC;
```

```
/* Uso del cursore; */
/* coppia e' implicitamente di tipo c%ROWTYPE */
FOR coppia IN c LOOP
    PRINT coppia.CodiceCliente, coppia.Totale
END LOOP; END;
```

8.4 La programmazione di transazioni

Come specificato nel Capitolo 1, una transazione è un programma che il DBMS esegue garantendone atomicità e serializzabilità.

L'atomicità viene garantita (concettualmente) facendo sì che, quando una transazione fallisce, tutti i suoi effetti sulla base di dati siano disfatti.

La serializzabilità è garantita in genere con il meccanismo del bloccaggio dei dati (*record and table locking*). Prima di leggere, o modificare, un dato una transazione lo deve bloccare in lettura o, rispettivamente, in scrittura. Quando una transazione T1 cerca di ottenere un blocco in scrittura su di un dato già bloccato da T2, T1 viene messa in attesa, finché T2 non termina e quindi rilascia il blocco. Con questa tecnica si garantisce la serializzabilità delle transazioni ed il loro isolamento, ovvero il fatto che una transazione non veda mai le modifiche fatte da un'altra transazione non ancora terminata. Le richieste di blocco sono fatte automaticamente dal sistema senza intervento del programmatore.

Se si adotta l'approccio di considerare ogni comando SQL come una transazione (*autocommit on*), possiamo non essere in grado di disfare un'operazione che abbiamo già fatta e che ha portato la base di dati in uno stato non consistente. È quindi necessario, a volte, gestire esplicitamente le transazioni (*autocommit off*).

La prima possibilità da considerare in questi casi, è quella di far sì che l'intero programma diventi una transazione. Questo approccio, però, funziona bene in casi semplici, ma in generale è necessario poter prevedere altri comportamenti:

- Quando il programma scopre una condizione anomala, che impedisce il completamento di un suo compito, è spesso opportuno avere la possibilità di disfare solo una parte delle operazioni fatte, cercando di aggirare l'anomalia usando del codice alternativo.
- Quando un programma può richiedere un tempo lungo per terminare le proprie operazioni, ad esempio perché interagisce con l'utente, può essere opportuno spezzare il programma in più transazioni, in modo da poter rilasciare i blocchi, per permettere l'esecuzione di altre transazioni che necessitano degli stessi dati.

I comandi **COMMIT** e **ROLLBACK**. permettono di ottenere questi comportamenti, spezzando un programma in più transazioni. Vediamo come questo può essere ottenuto nel caso di SQLJ, assumendo di operare quindi attraverso un contesto con *autocommit off*.

Una transazione viene considerata iniziata dal sistema quando un programma esegue un'operazione su una tabella (ad esempio, **SELECT**, **UPDATE**, **INSERT**, **DELETE**).[4]

La transazione quindi prosegue finché:

1. viene eseguito il comando #sql [contesto] **COMMIT**; che comporta la terminazione normale della transazione e quindi il rilascio dei blocchi sui dati usati, che diventano così disponibili ad altre transazioni;
2. viene eseguito il comando #sql [contesto] **ROLLBACK**; (*abort transaction*) che comporta la terminazione prematura della transazione, e quindi (*a*) il disfacimento di tutte le modifiche fatte dalla transazione (per garantire la proprietà dell'atomicità) e (*b*) il rilascio dei blocchi sui dati usati;
3. il programma termina senza errori e quindi la transazione termina normalmente;
4. il programma termina con fallimento e provoca la terminazione prematura della transazione.

Nell'esempio precedente, quindi, il programma andrebbe riscritto per portare le parti di codice che interagiscono con l'utente al di fuori della transazione.

Per ottenere questo risultato è sufficiente organizzare il programma come due transazioni: la prima che comincia dopo la prima interazione con l'utente, consistente di un unico **SELECT**, e la seconda, che comprende il comando **UPDATE**, e la dichiarazione e l'uso del cursore. La modifica da fare al programma è quindi l'inserzione del comando:

#sql [contesto] **COMMIT**;

dopo il primo **SELECT**, per forzare la terminazione della prima transazione, dato che la seconda termina con la fine del programma.

In realtà, quando un programma esegue transazioni occorre cercare di renderle meno "estese" possibili, in modo da permettere il più alto grado di concorrenza possibile fra programmi diversi. Così, nell'esempio precedente, è preferibile suddividere ancora la seconda transazione in due, dato che si eseguono due operazioni non correlate (l'aggiornamento della zona di un agente e la stampa delle coppie cliente, ammontare ordini).

La struttura finale del programma diventa quindi:

Class Esempio;
Dichiarazioni e Inizializzazioni

{ Lettura dal terminale dei dati di un ordine }

{ Prima transazione: ricerca ordine }
#sql [contesto] **COMMIT**;

4. Di solito i comandi che modificano lo schema (CREATE, DROP e ALTER) sono eseguiti in modo atomico e i loro effetti non possono essere annullati.

{ Stampa risultato prima transazione }
{ Lettura dal terminale dei dati per la seconda transazione }

{ Seconda transazione: aggiornamento zona di un agente }
#sql [contesto] **COMMIT**;

{ Terza transazione: recupero e stampa dei dati sui clienti e totali ordini }
#sql [contesto] **COMMIT**;
END programma.

Esempio 8.4
Supponiamo che nella base di dati esista anche la tabella:

Magazzino(Prodotto, Quantità, Prezzo)

Si consideri il seguente frammento di programma, che potrebbe servire ad un venditore al momento della richiesta di un ordine da parte di un cliente. Nel programma si legge la quantità di prodotto disponibile in magazzino e il prezzo corrente. Quindi si legge la quantità ordinata, e si crea l'ordine.

```
import java.sql.*;
import sqlj.runtime.*;

public class Esempio2 {

    public static void main(String [] args) {

        try {
            // Connessione alla base di dati
            Class.forName("oracle.jdbc.driver.OracleDriver");
            #sql context ClasseContesto;
            ClasseContesto contesto =
                    new ClasseContesto("jdbc:oracle:thin:@localhost:1521:mydb",
                            "utente1","password1");

            // Ricerca della quantita' di un certo prodotto
            // Lettura dei parametri
            String numProdotto = leggi("Codice prodotto: ");

            // inizio della prima transazione
            int quantita, prezzo;
            #sql [contesto] SELECT   Quantita, Prezzo INTO  :quantita, :prezzo
                            FROM     Magazzino
                            WHERE    Prodotto = :numProdotto;
```

```
//si termina la transazione prima di interagire con l'utente
#sql [contesto] COMMIT;

// Stampa risultato
System.out.println("La quantita' e': " + quantita);
System.out.println("Il prezzo e': " + prezzo);

int quantitaRichiesta = new Integer(leggi("Quantita' ordinata: ")).intValue();
int prezzoProposto = prezzo;

// inizio della seconda transazione
#sql [contesto] SELECT   Quantita, Prezzo INTO  :quantita, :prezzo
                FROM     Magazzino
                WHERE    Prodotto = :numProdotto;

if (quantita >= quantitaRichiesta && prezzo == prezzoProposto) {
    #sql [contesto] UPDATE   Magazzino
                    SET      Quantita = :quantitaRichiesta
                    WHERE    Prodotto = :numProdotto;
    ... soddisfacimento dell'ordine ...
} else { // se la quantita' e' diventata insufficiente o il prezzo e' cambiato
            #sql [contesto] ROLLBACK;
            System.out.println("Richiesta non evasa per cambiamento" +
                " quantita' oppure prezzo");
...
```

Questo esempio mostra come la necessità di dividere un programma in più transazioni per aumentare il grado di concorrenza comporti alcune complicazioni.

Infatti la seconda transazione, quella che effettua l'ordine vero e proprio, invece di aggiornare direttamente il valore della quantità del prodotto, richiede una *nuova* lettura per controllare l'effettiva disponibilità della merce e il suo prezzo. Il controllo è dovuto al fatto che fra la chiusura della prima transazione e l'inizio della seconda, può essere stata eseguita un'altra transazione che ha modificato la quantità (ad esempio per fare un altro ordine), o il prezzo (ad esempio per aggiornare i prezzi dei prodotti). Il controllo quindi è indispensabile per evitare di vendere la stessa merce due volte o ad un prezzo diverso da quello stabilito. A seconda dell'esito del controllo, può essere necessario provocare il fallimento della transazione.

8.4.1 Ripetizione esplicita delle transazioni

Un'altra situazione da prevedere nella programmazione di transazioni è la ripetizione della transazione quando questa viene interrotta dal sistema per sbloccare una condi-

zione di *stallo* (*deadlock*), che si presenta quando due o più transazioni sono bloccate in attesa l'una dei dati dell'altra, e viceversa.

In questo caso, il DBMS scopre lo stallo e interrompe, secondo una propria strategia, una delle transazioni in modo che le altre possano proseguire.

Il fatto che una transazione venga interrotta per sbloccare uno stallo viene segnalato al programma con (**SQLCODE** = **DEADABORT**), e quindi si può decidere di far ripartire la transazione.

Esempio 8.5
Supponiamo di dover fare un aggiornamento che coinvolge molte ennuple, come cambiare il supervisore di tutti gli agenti di una certa zona. Si può usare il seguente frammento di codice, che prova ripetutamente, fino ad un massimo di quattro volte, l'operazione in caso di interruzione della transazione per uno stallo.

```
int tentativi = 0;
boolean riuscito = false;

while ((tentativi < 4) && ! riuscito) {
    try {
        #sql [contesto] UPDATE   Agenti
                        SET      Supervisore = :nuovoSupervisore
                        WHERE    Zona = :nomeZona;
        riuscito = true;
    } catch (SQLException e) {
        #sql [contesto] ROLLBACK;
        tentativi = tentativi + 1;
    }
}
if (! riuscito) {
    System.out.println("Aggiornamento non eseguito per troppi stalli!");
    ...
}
```

In generale, se si decide di far ripartire una transazione interrotta dal sistema, bisogna ricordarsi che mentre i suoi effetti sulla base di dati vengono disfatti automaticamente, i valori di variabili del programma, eventualmente significativi per la corretta esecuzione della transazione, rimangono inalterati perché non sono sotto il controllo del sistema e quindi vanno inizializzati dal programma prima di far ripartire la transazione.

8.4.2 Transazioni con livelli diversi di isolamento

Con l'aumentare del numero di transazioni eseguite concorrentemente in modo serializzabile si può ridurre l'effettivo grado di concorrenza del sistema a causa del fatto

che aumenta la probabilità di avere transazioni in attesa di dati bloccati da altre o interrotte per il verificarsi di situazioni di stallo. Per questa ragione i sistemi commerciali prevedono la possibilità di programmare transazioni rinunciando alla proprietà della serializzabilità e quindi di isolamento delle transazioni.

Nella proposta dell'SQL-92, con il comando **SET TRANSACTION** si può scegliere uno dei seguenti livelli di isolamento per consentire gradi di concorrenza decrescenti:

```
SET TRANSACTION ISOLATION LEVEL [ READ UNCOMMITTED |
                                  READ COMMITTED   |
                                  REPEATABLE READ  |
                                  SERIALIZABLE     ]
```

Il primo livello di isolamento, *read uncommitted*, detto anche *dirty read* o *degree of isolation 0*, consente transazioni che fanno solo operazioni di lettura (quelle di modifica sono proibite) che vengono eseguite dal sistema senza bloccare in lettura i dati. La conseguenza di questo modo di operare è che una transazione può leggere dati modificati da un'altra non ancora terminata, che vengono detti *sporchi* perché potrebbero non essere più nella base di dati se la transazione che li ha cambiati venisse abortita.

Il secondo livello di isolamento, *read committed*, detto anche *cursor stability* o *degree of isolation 1*, prevede che i blocchi in lettura vengano rilasciati subito, mentre quelli in scrittura vengono rilasciati alla terminazione della transazione. In questo modo, quando una transazione T modifica un dato, quel dato non può essere letto da altri fino a che T non abbia effettuato un commit o un rollback. La conseguenza di questo modo di operare è che una transazione può fare *letture non ripetibili*, ovvero letture successive degli stessi dati possono dare risultati diversi perché i dati sono stati modificati da altre transazioni terminate nell'intervallo tra la prima e la seconda lettura.

Il terzo livello di isolamento, *repeatable read*, detto anche *degree of isolation 2*, prevede che i blocchi in lettura e scrittura siano assegnati solo su ennuple di tabelle e vengano rilasciati alla terminazione della transazione. Questa soluzione evita il problema delle letture non ripetibili, ma non è ancora il tipo di isolamento che occorre per avere transazioni serializzabili in quanto consente ad altre transazioni di fare inserzioni di ennuple nella stessa tabella, creando il fenomeno cosiddetto dei *fantasmi* (*phantoms*). Ad esempio, supponiamo che esista una tabella di riepilogo sulle vendite dei prodotti:

```
Vendite(Prodotto,TotaleAmmontare)
```

che venga mantenuta aggiornata dalla transazione che inserisce un nuovo ordine per un prodotto.

Supponiamo che vengano eseguite due transazioni T_1 e T_2 con il livello di isolamento *repeatable read*: T_1 inserisce un ordine per il prodotto 200 e quindi aggiorna la riga corrispondente della tabella Vendite; T_2 legge gli ordini del prodotto 200, calcola il totale e controlla che sia quello della tabella Vendite.

Allora può capitare che le due transazioni vengano eseguite come segue:

T_1	T_2
	Legge gli ordini di un certo prodotto e calcola il totale
Inserisce un nuovo ordine per lo stesso prodotto e aggiorna la tabella Vendite	
	Controlla la somma

in cui T_2 segnala che il valore della tabella Vendite è scorretto, anche se non è vero (dato che è stato inserito un nuovo ordine e aggiornata la somma). Questo è possibile proprio perché T_2 blocca solo le righe degli ordini che legge, e non tutta la tabella, pertanto T_1 può effettuare l'inserzione e la modifica di Vendite.

Per evitare il problema occorre invece che l'insieme delle righe di Ordini lette da T_2 non venga modificato da T_1, ad esempio bloccando tutta la tabella, come fanno quei sistemi che garantiscono il quarto livello di isolamento, *serializable*, detto anche *degree of isolation 3*.

In alcuni sistemi il blocco della tabella può essere anche richiesto esplicitamente dalla transazione con il comando:

LOCK TABLE *Tabella* **IN** [**SHARE** | **EXCLUSIVE**] **MODE**

per semplificare la gestione dei blocchi da parte del sistema o per operare correttamente anche con un livello di isolamento diverso da quello *serializable*. Il comando **LOCK** non è previsto da SQL-92.

La programmazione di transazioni con i primi tre livelli di isolamento richiede in generale un maggior impegno per garantire la corretta evoluzione della base di dati, perché non è possibile astrarre dal fatto che la transazione è eseguita insieme ad altre. In tabella sono riassunti i fenomeni che si possono presentare operando con i diversi livelli di isolamento.

Livello	**Letture sporche**	**Letture non ripetibili**	**Dati fantasmi**
READ UNCOMMITTED	Possibile	Possibile	Possibile
READ COMMITTED	Non possibile	Possibile	Possibile
REPEATABLE READ	Non possibile	Non possibile	Possibile
SERIALIZABLE	Non possibile	Non possibile	Non possibile

8.5 Conclusioni

Sono state presentate le caratteristiche di tre tipi di strumenti per lo sviluppo di applicazioni che usano basi di dati: linguaggi che ospitano SQL, linguaggi che usano interfacce API e linguaggi integrati della quarta generazione.

L'attenzione è stata posta su due aspetti principali: come scambiare informazioni con una base di dati da un programma e come programmare le transazioni. C'è un altro aspetto importante che non è stato discusso per ragioni di spazio: come programmare la parte dell'applicazione dedicata alle interazioni con gli utenti con opportune interfacce grafiche. Ogni sistema offre linguaggi dotati dei meccanismi per farlo ed esistono anche strumenti di ditte indipendenti finalizzati a questo scopo.

Esercizi

1. Si consideri il seguente frammento di codice SQLJ:

```
#sql [contesto]
        SELECT   Ammontare INTO :ammontare
        FROM     Ordini
        WHERE    CodiceAgente = :numAgente
```

   L'elaborazione del codice procede in tre fasi: (a) precompilazione dei frammenti SQL in Java, (b) compilazione del programma Java risultante, (c) esecuzione. Durante l'esecuzione, il controllo si alterna tra macchina astratta Java e il DBMS. Esemplifichiamo ora alcuni motivi per cui tale codice potrebbe essere scorretto. Immaginando che ogni errore sia scoperto prima possibile, specificare chi dei quattro attori (precompilatore, compilatore, macchina astratta Java e DBMS) segnala ciascun errore.

   a) Sintassi SQL: il programmatore potrebbe scrivere WEHRE anziché **WHERE**.
   b) Nomi: il programmatore potrebbe avere sbagliato a scrivere il nome della relazione, oppure quello dell'attributo Ammontare, oppure quello della variabile :ammontare.
   c) Tipi: il tipo di Ammontare e quello di :ammontare potrebbero essere incompatibili.
   d) Variazione dello schema: quando il programma viene eseguito la relazione Ordini potrebbe essere stata cancellata, oppure il tipo dell'attributo Ammontare potrebbe essere cambiato.
   e) Univocità: il valore di :NumAgente potrebbe non essere associato ad alcun agente, oppure essere associato a più agenti.

2. Si consideri la versione API del frammento di codice dell'esempio precedente.

```
PreparedStatement pstmt =
    con.prepareStatement(
        "SELECT Ammontare
          FROM Ordini WHERE CodiceAgente = ?");
pstmt.setString(1, codAgente);
risultato = pstmt.executeQuery();
int ammontare = risultato.next().getInt(0);
```

In questo caso l'elaborazione di tale frammento procede come segue: compilazione del programma Java, esecuzione da parte della macchina astratta Java che interagisce con il DBMS. Anche in questo caso si cerchi di individuare in quale momento verrebbe scoperto ciascuno degli errori sopra elencati.

3. Specificare vantaggi e svantaggi della programmazione di applicazioni usando un linguaggio integrato anziché un'API.
4. Si considerino le seguenti applicazioni in ambito bancario. Indicare il livello di isolamento più opportuno per ciascuna di esse, spiegando la risposta.
   a) Per ogni cliente della banca, contare le operazioni effettuate negli ultimi 500 giorni, ed aggiungere il nome del cliente ad un elenco se le operazioni sono più di trecento. L'elenco servirà a scopi di marketing.
   b) Effettuare un trasferimento fondi, sottraendo un ammontare da un conto per aggiungerlo ad un altro.
   c) Gestire un prelievo allo sportello come segue: il cassiere legge sul terminale il saldo corrente del cliente; se questo supera la cifra richiesta dal cliente, il cassiere comunica al sistema la cifra ed effettua il pagamento (specificare quali di queste operazioni sarebbero racchiuse nella transazione).

Note bibliografiche

Per approfondire il problema della programmazione delle applicazioni in SQL si veda [van der Lans, 2001] e [Kifer et al., 2005]. Molte informazioni su JDBC e SQLJ sono disponibili sulla rete, in particolare ai siti http://java.sun.com/products/jdbc/ e http://www.sqlj.org.

Capitolo 9

REALIZZAZIONE DEI DBMS

Si presentano l'architettura dei DBMS relazionali centralizzati e alcune delle tecniche utilizzate per realizzarne le funzionalità essenziali: la gestione dei dati, delle interrogazioni, della concorrenza e dell'affidabilità. Una conoscenza, sia pure elementare, di tali tecniche è indispensabile per utilizzare questi sistemi in maniera efficace.

9.1 Architettura dei sistemi relazionali

In Figura 9.1 sono mostrati i moduli principali di una possibile architettura di un sistema relazionale centralizzato.

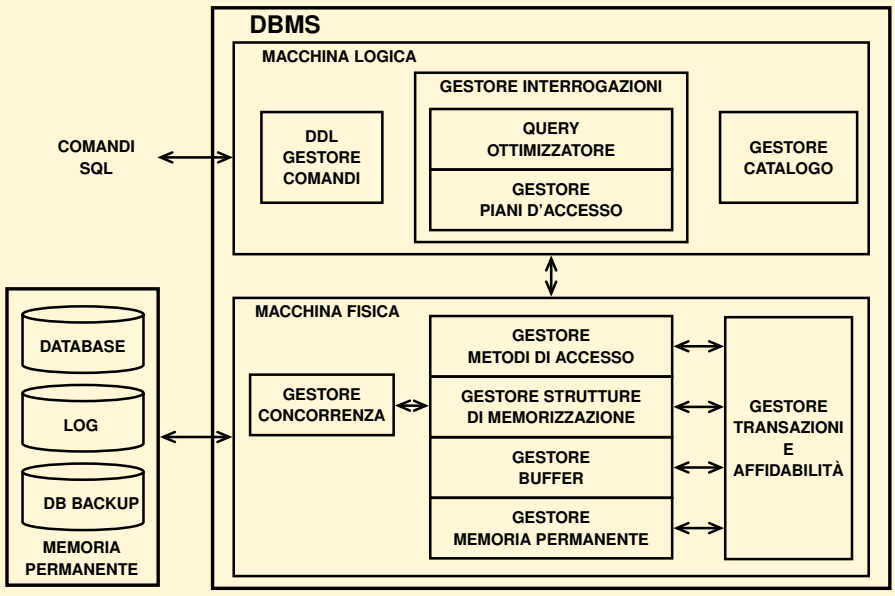


Figura 9.1: Architettura di un DBMS relazionale

Un DBMS è organizzato su due livelli, che chiameremo la *macchina logica* e la *macchina fisica*. La macchina logica comprende i moduli che trattano i comandi del linguaggio

SQL e stabiliscono come eseguirli usando gli operatori forniti dalla macchina fisica, che gestisce la memoria permanente e le strutture per la memorizzazione e recupero dei dati.

Nei sistemi commerciali le funzionalità di questi moduli non sono nettamente separate come la figura potrebbe far pensare, ma questa schematizzazione consente di comprendere meglio gli scopi di ognuno.

Nelle prossime sezioni si esaminano brevemente i moduli dedicati alla gestione dei dati, delle interrogazioni, della concorrenza e dell'affidabilità. Di ogni modulo viene descritto il livello di astrazione fornito e le funzionalità che rendono disponibili agli altri moduli. Per approfondire gli argomenti si veda [Albano, 2001].

9.2 Gestore della memoria permanente

Il *gestore della memoria permanente* offre una visione di una basi di dati come un insieme di file di blocchi di caratteri (*pagine fisiche*) di grandezza prefissata, compresa generalmente fra 1 e 8 Kbyte, che sono l'unità minima di trasferimento fra la memoria permanente e quella temporanea. Esso consente agli altri livelli di usare la memoria permanente astraendo dalle diverse modalità di gestione dei file dei sistemi operativi.

I dati memorizzati nella memoria permanente sono quelli descritti nello schema logico, le strutture ausiliarie per agevolare gli accessi alla base di dati (indici), e i dati di servizio necessari per il funzionamento del sistema.

9.3 Gestore del buffer

Il *gestore del buffer* gestisce uno spazio della memoria temporanea destinato a contenere un insieme di pagine fisiche trasferite dalla memoria permanente. Una pagina fisica è rappresentato in memoria temporanea come una struttura logica detta *pagina* e gli altri livelli del sistema hanno una visione della memoria permanente come un insieme di pagine utilizzabili in memoria temporanea astraendo da quando esse vengano trasferite fra i due tipi di memoria.

Poiché il buffer può contenere molte pagine, quando si opera su dati usati di recente esiste una certa probabilità che tali dati siano ancora disponibili nel buffer, evitandone così la rilettura dal disco. Similmente, gli aggiornamenti ai dati vengono in realtà effettuati all'interno del buffer, e i dati sono riportati sul disco solo quando è necessario liberare il buffer o quando il protocollo per la gestione dell'affidabilità lo richiede (si veda la Sezione 9.8).

I record all'interno delle pagine sono memorizzati come stringhe di caratteri con un prefisso contenente informazioni di servizio seguito dai valori dei campi. Il prefisso può contenere, ad esempio, una marca per la cancellazione logica, la lunghezza del record, il numero degli attributi, l'identificatore interno del record, unico all'interno della base di dati e assegnato automaticamente dal sistema ad ogni nuovo record. Un record con una dimensione inferiore a quella di una pagina si memorizza tutto in una pagina.

Quando un record viene inserito in una relazione, il sistema gli assegna un identificatore, chiamato TID (*Tuple Identifier*), o RID (*Row Identifier*), che diventa il riferimento da usare nelle strutture dati. L'esatta natura del TID può variare da un sistema ad un altro, ma l'obiettivo comune a tutti è di garantire che un TID sia un'informazione che non cambia fintantoché il record esista nella base di dati. La soluzione più comune è la seguente: un TID è una coppia (P, j), dove P è il numero di pagina e j è la posizione relativa in un vettore memorizzato nella pagina, contenente il riferimento all'inizio del record (*slot array*) (Figura 9.2). Se il record si sposta nella pagina, ad esempio in seguito a modifiche che ne cambiano la lunghezza, basta aggiornare il contenuto del vettore, senza cambiare il TID. Nel caso in cui la modifica del record ne comporti uno spostamento in un'altra pagina, il record viene sostituito con la coppia (P', j'), un altro TID, dove P' è l'indirizzo della nuova pagina e j' è la posizione relativa in P'. Anche in questo caso, il TID originariamente assegnato al record non cambia. Se successivamente, nell'accedere al record si scopre che nella pagina P esiste sufficiente spazio libero per contenerlo, allora questo può essere riportato nella pagina originaria.

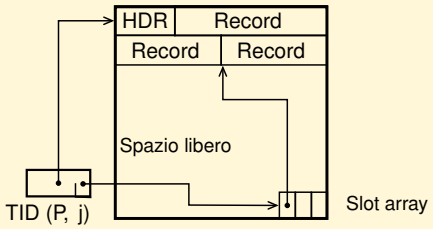


Figura 9.2: Riferimenti ai record

9.4 Gestore delle strutture di memorizzazione

Il *gestore delle strutture di memorizzazione* offre agli altri livelli del sistema una visione dei dati permanenti organizzati in collezioni di record e indici astraendo dalle strutture fisiche usate per la loro memorizzazione in memoria permanente.

Avendo stabilito come si può memorizzare in modo persistente una collezione di record dotati di un TID univoco, resta ancora da stabilire:

- come individuare la pagina in cui inserire un nuovo record quando questo viene aggiunto alla collezione;
- come gestire le situazioni in cui una sequenza di cancellazioni o di inserimenti rendono una pagina troppo vuota o troppo piena;
- quali strutture ausiliarie prevedere per facilitare l'esecuzione delle ricerche.

Un'*organizzazione dei dati* è un insieme di algoritmi per gestire le operazioni su una collezione di record che risponde a queste tre domande. In questa sezione presentiamo brevemente le organizzazioni più importanti.

Organizzazione seriale e sequenziale

L'organizzazione *seriale* (*heap*) è il modo più semplice di organizzare i record di una collezione perché essi si memorizzano consecutivamente nell'ordine in cui vengono inseriti. Le pagine possono essere contigue nella memoria permanente oppure no, e in quest'ultimo caso sono collegate a lista. Questa soluzione, per la sua semplicità, è usata da tutti i DBMS quando non ne vengono richieste altre. Le operazioni di inserzioni di nuovi record sono veloci, mentre le ricerche di un record per valore di una chiave o di piccoli sottoinsiemi di record sono lente se i dati sono molti. Per questa ragione si usa per piccole collezioni o quando interessa operare principalmente su grandi sottoinsiemi di record.

Quando i record di una collezione sono memorizzati consecutivamente nell'ordine dei valori di un insieme di attributi $A_1, \ldots, A_n$, si parla di organizzazione *sequenziale* su $A_1, \ldots, A_n$. Questa organizzazione permette di trovare velocemente i record che hanno un valore specificato degli attributi di ordinamento, ma è più complessa di quella seriale se deve garantire l'ordinamento dei dati in presenza di molte inserzioni di nuovi record. Per questa ragione, i DBMS che la prevedono richiedono di usarla per collezioni statiche e dopo aver caricato tutti i dati. Se poi vengono fatte inserzioni si perde l'ordinamento dei dati. Per avere invece i dati ordinati di collezioni dinamiche, i DBMS prevedono un'altra soluzione con i dati memorizzati con una struttura ad albero, descritta più avanti.

Per rendere efficienti le operazioni di ricerca di piccoli sottoinsiemi di collezioni dinamiche di record, in particolare di un record noto il valore di una chiave, si usa una delle seguenti organizzazioni: *procedurale*, *ad albero* o *con indice*.

Organizzazione procedurale

L'organizzazione *procedurale*, o *hash*, prevede l'esistenza di un opportuno algoritmo (*trasformazione della chiave*) che, applicato al valore della chiave, restituisce l'indirizzo della pagina in cui memorizzare, e successivamente cercare, il record; in caso di insuccesso, esiste un criterio per proseguire la ricerca in modo da completare l'operazione con pochi accessi supplementari. Se ben configurata, questa organizzazione permette in genere di ritrovare un record, a partire dal valore della chiave primaria, con un solo accesso alla memoria permanente.

Organizzazione ad albero

L'organizzazione ad *albero* di una collezione C sulla chiave A prevede l'utilizzo di una struttura dati in memoria permanente detta B^+*-albero* che generalizza l'albero di ricerca bilanciato, con le seguenti caratteristiche:

1. Permette di trovare il record con un valore della chiave A, se esiste, con pochi accessi alla memoria permanente.

2. Mantiene la collezione C ordinata sulla chiave A.

Questa organizzazione è la più complessa tra quelle illustrate ed è lievemente meno efficiente della precedente quando si deve rispondere ad un'interrogazione con condizione $A = k$. Tuttavia, la seconda caratteristica sopra elencata la rende molto adatta a rispondere ad interrogazioni con condizioni tipo $A \leqslant k$, $A \geqslant k$, o $k_1 \leqslant A \leqslant k_2$.

Indici

Un indice su un attributo A di una relazione R è una struttura dati con un ruolo analogo all'indice analitico di un libro, costituito da un insieme di pagine che contengono ciascuna un insieme di righe di testo. Un indice analitico è un insieme ordinato di termini con associato il numero della pagina dove vengono usati, di solito dove il termine viene introdotto per la prima volta. Per trovare la pagina dove appare un termine, invece di sfogliarle tutte una dopo l'altra, si consulta l'indice analitico e si passa alla pagina segnalata.

L'organizzazione dei dati con indice è simile all'organizzazione di un libro: i record nei DBMS sono memorizzati in pagine di un file, come le righe di un libro in pagine di testo, ma l'indice nei DBMS consente di trovare rapidamente i record di una pagina in base al valore di un attributo, mentre l'indice analitico di un libro non fornisce l'informazione sulla riga del testo che contiene il termine, ma solo l'informazione sulla pagina dove occorre. Più precisamente valgono le seguenti considerazioni.

Definizione 9.1

Un indice I su un attributo A di una relazione R, dal punto di vista logico, è una relazione ordinata con due attributi $I(A, \text{TID})$, con gli elementi ordinati su A e valori $(A := a_i, \text{TID} := r_j)$, dove a_i è un valore di A presente in un record di R, ed r_j è un riferimento (TID) al record di R in cui A vale a_i. Se A non è una chiave, nell'indice sono presenti tanti record con lo stesso valore a_i di A quanti sono i record di R in cui A vale a_i.

Di solito un indice è organizzato a B^+- albero per permettere di trovare con pochi accessi, a partire da un valore v, i record di R in cui il valore di A è in una relazione specificata con v.

In Figura 9.3 sono mostrati due esempi di indici su due attributi della relazione R, la chiave K e A, supponendo per semplicità che (a) i record di R siano di lunghezza uguale e memorizzati con l'organizzazione seriale e (b) i TID siano interi che rappresentano la posizione dei record nella relazione, traducibili in modo ovvio nella coppia (numero della pagina, posizione nella pagina), nota la capacità della pagina.
Un indice può anche essere definito su di un insieme $A_1, \ldots, A_n$ di attributi. In questo caso, l'indice contiene un record per ogni combinazione di valori assunti dagli attributi $A_1, \ldots, A_n$ nella relazione, e può essere utilizzato per rispondere in modo efficiente ad interrogazioni che specifichino un valore per ciascuno di questi attributi.

Organizzazioni statiche e dinamiche

In tutti i metodi descritti, non abbiamo specificato come si gestiscono le situazioni in

R

TID	K	A	...
1	k5	d	...
2	k3	b	...
3	k7	a	...
4	k6	c	...
5	k2	b	...
6	k4	g	...
7	k1	c	...
...	...	...	...

IdxK

K	TID
k1	7
k2	5
k3	2
k4	6
k5	1
k6	4
k7	7
...	...

IdxA

A	TID
a	3
b	2
b	5
c	4
c	7
d	1
g	6
...	...

Figura 9.3: Esempio di relazione con due tipi di indici

cui una pagina diventa troppo vuota o troppo piena. A seconda di come si affronta questo problema, l'organizzazione dei dati che ne scaturisce può essere *statica* o *dinamica*. Un'organizzazione è detta *statica* se, una volta dimensionata per una certa quantità di dati, non si riconfigura automaticamente in seguito ad un aumento dei dati memorizzati, il quale comporta, quindi, un degrado delle prestazioni, che si elimina con una riorganizzazione. Un'organizzazione è detta invece *dinamica* se è in grado di adeguarsi alla quantità di dati memorizzati, preservando le prestazioni senza bisogno di riorganizzazioni. Tutte le organizzazioni sopra descritte ammettono una versione statica ed una dinamica.

Scelta dell'organizzazione

La scelta dell'organizzazione più opportuna per ogni relazione costituisce il nocciolo della progettazione fisica, ed è un compito arduo che necessita di esperienza e di strumenti di supporto. La scelta dell'organizzazione non è mai definitiva, ma varia durante l'uso del sistema, e in particolare quando si osservano delle prestazioni poco soddisfacenti. La scelta dell'organizzazione per una relazione è in genere guidata dall'applicazione che la usa ed è la più importante da eseguire in modo efficiente. In estrema sintesi, se tale applicazione utilizza un'alta percentuale dei dati si sceglie un'organizzazione seriale, se l'applicazione seleziona un piccolo insieme di record in base al valore di un attributo A si sceglie un'organizzazione ad albero, mentre se seleziona un unico record sulla base del valore di una chiave si sceglie un'organizzazione procedurale. Per facilitare l'esecuzione di ogni altra applicazione è possibile aggiungere indici sugli attributi coinvolti, tenendo conto delle indicazioni date nel Capitolo 7.

Esempio 9.1
Il linguaggio per la definizione dello schema della base di dati prevede comandi per scegliere le strutture per memorizzare i dati. Vediamo le soluzioni adottate da alcuni sistemi commerciali.

Tutti i DBMS memorizzano una relazione $R(A_1 : T_1, ..., A_n : T_n)$ con un'organizzazione seriale, in assenza di altre specifiche. Per agevolare le operazioni, si

utilizzano indici su alcuni attributi o su combinazioni di attributi. Gli indici sono memorizzati ad albero dinamico e si definiscono con il comando:

CREATE [**UNIQUE**] **INDEX** Nome **ON** R(A_i)

L'opzione **UNIQUE** si usa per indici su attributi chiave.

Nel sistema INGRES, una volta caricati i dati, è possibile trasformare l'organizzazione di una relazione in sequenziale (**HEAPSORT**), *hash* statica, oppure ad albero statico (*ISAM*), con uno dei seguenti comandi:

MODIFY R **TO HEAPSORT ON** A_i **ASC**
MODIFY R **TO HASH ON** A_i
MODIFY R **TO ISAM ON** A_i

Le dichiarazioni delle organizzazioni prevedono inoltre la possibilità di imporre che i valori della chiave siano unici (ad esempio, **HASH UNIQUE ON** A_i) oppure che i valori siano memorizzati compressi, eliminando i caratteri bianchi (ad esempio, **CHEAPSORT**).È possibile poi aggiungere indici con il comando

CREATE INDEX Nome **ON** R(A_i)

che vengono trattati dal sistema come relazioni binarie con attributi (A_i, TID), dove i valori di TID sono gli identificatori interni delle ennuple. Un attributo può essere sostituito da una combinazione di più attributi, fino ad un massimo di sei. Questi indici possono a loro volta essere organizzati in modo sequenziale, *hash* o ISAM, come ogni altra relazione.

In Oracle l'organizzazione dinamica ad albero di una relazione, detta IOT (*index organized table*), si dichiara al momento della creazione di una tabella, con la chiave primaria e la specifica **ORGANIZED INDEX**

CREATE TABLE R(Pk Tipo **PRIMARY KEY**, ...) **ORGANIZED INDEX**;

In SQL Server l'organizzazione dinamica ad albero di una relazione si ottiene definendo sulla chiave primaria un **CLUSTERED INDEX**.

CREATE TABLE R(Pk Tipo **PRIMARY KEY**, ...)
CREATE CLUSTERED INDEX RAlbero **ON** R(Pk)

9.5 Gestore dei metodi di accesso

Il *gestore dei metodi di accesso* offre operatori per costruire, o eliminare, collezioni di record o indici e per recuperare i record uno dopo l'altro nell'ordine in cui sono memorizzati, oppure attraverso indici, astraendo dalla loro organizzazione fisica.

Gli accessi alle relazioni e agli indici avvengono con modalità simili a quelle previste per i cursori descritte nel Capitolo 8, quali apertura relazione, avanzamento del cursore, posizionamento del cursore sulla base del valore di un attributo, verifica di fine relazione, chiusura.

9.6 Gestore delle interrogazioni

La macchina logica offre una visione dei dati permanenti come un insieme di tabelle relazionali sulle quali si opera con gli operatori dell'SQL. Essa prevede i seguenti moduli:

- Il *gestore delle autorizzazioni* per controllare che solo gli utenti autorizzati facciano uso dei dati con le modalità consentite.
- Il *gestore del catalogo* per trattare i metadati, ovvero le informazioni sulle caratteristiche logiche e fisiche dei dati presenti.
- Il *gestore delle interrogazioni* per controllare la correttezza delle interrogazioni e stabilire la strategia migliore per eseguirle con un opportuno algoritmo detto *piano di accesso*.

Nel seguito ci soffermiamo sulla gestione delle interrogazioni, compito fra i più importanti di un DBMS svolto con le seguenti fasi:

1. Controllo lessicale, sintattico e semantico dell'interrogazione e sua rappresentazione in forma di albero logico.
2. Riscrittura algebrica dell'albero logico.
3. Ottimizzazione fisica e generazione del piano di accesso.
4. Esecuzione del piano di accesso.

Una volta controllata la correttezza dell'interrogazione, essa viene rappresentata con un albero logico, usando i seguenti operatori dell'algebra relazionale estesa per operare su multinsiemi, come accade in SQL.

- Proiezione senza l'eliminazione dei duplicati: $\pi^{\mathbf{b}}_X(O)$, con X alcuni attributi di O.
- Eliminazione dei duplicati da un multinsieme: $\boldsymbol{\delta}(O)$.
- Ordinamento del risultato: $\boldsymbol{\tau}_X(O)$, con X alcuni attributi di O.
- Operatori insiemistici senza l'eliminazione dei duplicati: $O_1 \cup^{\mathbf{b}} O_2$, $O_1 \cap^{\mathbf{b}} O_2$, $O_1 -^{\mathbf{b}} O_2$.

Gli operatori dell'algebra relazionale su insiemi per la restrizione, prodotto, giunzione e raggruppamento con aggregazioni non cambiano per applicarli a multinsiemi.

9.6.1 Riscrittura algebrica

Durante la riscrittura algebrica si applicano tecniche di trasformazione dell'interrogazione, basate sulle proprietà algebriche degli operatori relazionali, per trovarne una

forma equivalente eseguibile con costi inferiori. Un possibile algoritmo di riscrittura algebrica è riportato nel Capitolo 4.

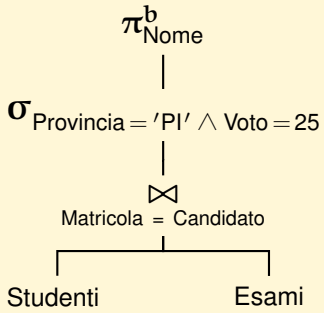
Vediamo alcuni esempi di riscritture algebriche, con riferimento alle seguenti relazioni, con Candidato chiave esterna di Esami per Studenti:

Studenti(Matricola, Nome, Provincia, AnnoNascita)
Esami(Materia, Candidato, Voto, Lode)

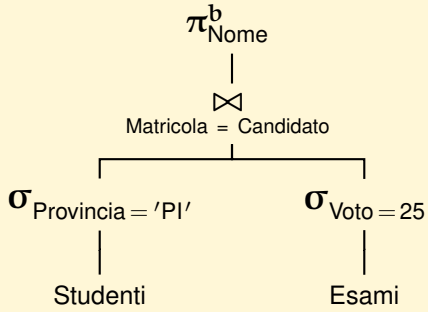
Esempio 9.2
Si consideri la seguente interrogazione:

```
SELECT   Nome
FROM     Studenti, Esami
WHERE    Matricola = Candidato AND Provincia = 'PI' AND Voto > 25;
```

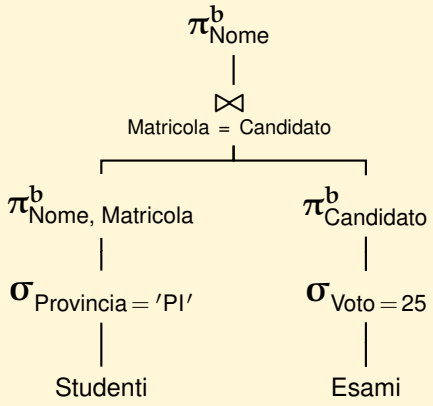
rappresentata con l'albero logico iniziale:



Un esempio tipico di riscrittura algebrica è l'anticipazione delle restrizioni e delle proiezioni rispetto alle giunzioni, allo scopo di ridurre la dimensione degli argomenti di questi ultimi. Con l'anticipazione delle restrizioni l'albero logico iniziale diventa:



Applicando anche l'anticipazione della proiezione, il precedente albero logico diventa:



Un'altra trasformazione, che di solito si prende in considerazione, è la normalizzazione delle condizioni nella forma normale prodotti di somme e i **NOT** di predicati vengono eliminati negando l'operatore di confronto (per es. **NOT** A>10 diventa A $\leqslant$ 10).

Le trasformazioni precedenti sono relativamente semplici, mentre quelle più complesse si hanno quando nell'interrogazione sono presenti *sottoselect* o viste, che in questa fase si tenta di eliminare, anche se non è sempre possibile farlo. In generale l'eliminazione delle *sottoselect* o delle viste aumenta poi le possibilità di generare piani di accesso migliori. Per esempio:

```
SELECT   Matricola, Nome
FROM     Studenti
WHERE    Matricola IN (SELECT   Candidato
                       FROM     Esami
                       WHERE    Materia = 'BD');
```

viene trasformata nell'interrogazione equivalente:

```
SELECT   DISTINCT Matricola, Nome
FROM     Studenti, Esami
WHERE    Matricola = Candidato AND Materia = 'BD';
```

Supponendo che esistano le viste:

```
CREATE   VIEW       VistaStudentiPisani
     AS  SELECT     *
         FROM       Studenti
         GROUP BY   Provincia ='PI';
```

```
CREATE   VIEW       VistaEsamiBD
      AS SELECT     *
         FROM       Esami
         GROUP BY   Materia = 'BD';
```

l'interrogazione:

```
SELECT   Matricola, Nome
FROM     VistaStudentiPisani, VistaEsamiBD
WHERE    Matricola = Candidato;
```

viene trasformata in:

```
SELECT   Matricola, Nome
FROM     Studenti, Esami
WHERE    Matricola = Candidato AND Materia = 'BD' AND Provincia = 'PI';
```

9.6.2 Ottimizzazione fisica

Durante l'ottimizzazione fisica, si stabilisce come eseguire nel modo "migliore" l'albero logico di un'interrogazione considerando i parametri fisici in gioco, quali la dimensione delle relazioni, l'organizzazione dei dati e la presenza di indici. Il problema è particolarmente difficile perché, come si vedrà più avanti, ogni operazione dell'algebra relazionale può essere realizzata in modi diversi ed occorre utilizzare opportune strategie per stimare i costi delle possibili alternative e scegliere quella con costo inferiore.

L'ottimizzazione fisica delle interrogazioni non verrà considerata nel seguito perché esula dai fini di questo testo e per approfondimenti si rinvia ai testi citati nelle note bibliografiche. L'attenzione sarà invece posta sul formalismo usato nei DBMS per mostrare il risultato dell'ottimizzazione fisica con una rappresentazione ad albero dell'interrogazione detta *albero fisico* o *piano di accesso*, in cui i nodi rappresentano degli *operatori fisici* che realizzano un algoritmo per eseguire un'operazione dell'algebra relazionale, o una sua parte.

La conoscenza del formalismo dei piani di accesso è utile perché i DBMS, di solito, su richiesta di chi formula un'interrogazione, mostrano il piano di accesso per eseguirla in modo che si possa (a) capire eventualmente come mai il sistema impieghi più tempo del previsto a produrre la risposta e (b) decidere poi opportune azioni correttive riguardanti l'organizzazione fisica dei dati per consentire al sistema di generare piani di accesso migliori.

Ogni DBMS ha i propri operatori fisici e per comodità si useranno quelli del sistema **JRS** sia perché sono più semplici di quelli dei sistemi commerciali sia perché si possono facilmente fare delle prove per prendere familiarità con l'ottimizzazione fisica delle interrogazioni.

Descriviamo ora gli operatori fisici del sistema **JRS** che prenderemo in considerazione per realizzare le seguenti operazioni su relazioni della base di dati o risultato di un'altra operazione:

- Scansione di tabelle memorizzate con l'organizzazione seriale.
- Proiezione dei record di un multinsieme senza l'eliminazione di duplicati.
- Eliminazione dei duplicati da un multinsieme.
- Restrizione dei record di un multinsieme a quelli che soddisfano una condizione.
- Ordinamento dei record di un multinsieme.
- Giunzione dei record di due multinsiemi.
- Raggruppamento dei record di un multinsieme.

Gli operatori fisici, come quelli dell'algebra relazionale, ritornano collezioni di record con una struttura che dipende dal tipo di operatore.

Operatori per la scansione di relazioni (R)

- **TableScan**(R): ritorna la collezione dei record di R.
- **SortScan**(R, $\{A_i\}$): ritorna la collezione dei record di R ordinati sui valori degli $\{A_i\}$ in modo crescente (in realtà si può ordinare anche in modo decrescente, ma per semplicità si omette questa possibilità).

Si noti che l'argomento di questi operatori è R, il nome di una relazione, quindi devono necessariamente stare su una foglia di un albero fisico.

Operatore per la proiezione con duplicati (π^b)

L'operatore fisico ha come argomento la collezione dei record O restituiti da un altro operatore fisico:

- **Project**(O, $\{A_i\}$): ritorna la proiezione dei record di O sugli attributi $\{A_i\}$, senza l'eliminazione dei duplicati.

Operatore per l'eliminazione di duplicati (δ)

Per eliminare i duplicati, un modo per procedere è di ordinare prima la collezione dei record. Altri modi di procedere sono possibili per eseguire questa operazione che non richiedono l'ordinamento dei dati, ma per semplicità non si prendono in considerazione.

- **Distinct**(O): ritorna la collezione dei record *diversi* di O, che devono essere *ordinati*.

Operatore per l'ordinamento (τ)

Per ordinare i dati ritornati da un operatore fisico O si usa il seguente operatore fisico:

- **Sort**(O, $\{A_i\}$): ritorna la collezione dei record di O ordinati sugli $\{A_i\}$.

A differenza del **SortScan**, che prima ordina i record di una relazione in memoria permanente e poi li ritorna, il **Sort** prima memorizza i record di O in una relazione temporanea, poi li ordina e infine li ritorna. Se non c'è spazio disponibile nel buffer, la relazione temporanea viene memorizzata in memoria permanente.

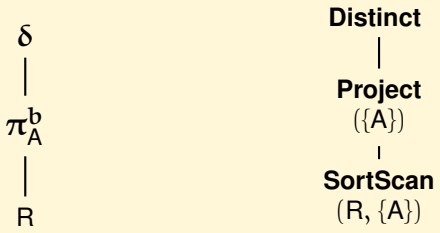
Per ordinare una relazione in memoria permanente si usa l'algoritmo di *ordinamento per fusione* (*merge sort*).

Esempio 9.3
In questi esempi, e in quelli che vedremo più avanti, l'obiettivo è solo di mostrare l'uso del formalismo dei piani di accesso, senza pretendere che il piano sia quello migliore per eseguire un'interrogazione, avendo evitato, per ragioni di spazio, di approfondire il problema dell'ottimizzazione fisica basata sui costi di esecuzione delle operazioni dell'algebra relazionale. Di ogni interrogazione SQL si presenta prima l'albero logico, con l'eventuale anticipazione delle restrizioni, e poi un possibile albero fisico, con riferimento alle seguenti relazioni:

R(A, B)
S(C, D)

Si consideri l'interrogazione:

SELECT **DISTINCT** A
FROM R;



Si lascia al lettore, come esercizio, definire un altro possibile albero fisico equivalente al precedente sostituendo il **SortScan** con un **TableScan**.

Operatori per la restrizione (σ)

L'operazione $\sigma_{\psi}(R)$ può essere realizzata in modi diversi. In assenza di indici si esaminano tutti i record di R e si controlla quali di essi soddisfano la condizione ψ. I casi più interessanti si presentano quando la condizione riguarda attributi sui quali sono definiti degli indici.

Se la condizione è del tipo $A\,\theta\,c$ (*condizione semplice*), con c una costante e θ un predicato di confronto ($=, <, \leqslant, >, \geqslant$), i record che soddisfano la condizione si trovano usando l'indice su A (si veda la Sezione 9.2).

Se la condizione ψ è il prodotto logico di condizioni semplici $\psi = \psi_1 \wedge \psi_2$ su attributi con indice, si può procedere in due modi. Il primo prevede l'uso di un solo indice e si basa sul fatto che $\sigma_\psi(R) = \sigma_{\psi_1}(\sigma_{\psi_2}(R))$, pertanto si usa l'indice per risolvere la restrizione $\sigma_{\psi_2}(R)$ e si controlla sul risultato la condizione ψ_1. Il secondo metodo usa entrambi gli indici disponibili calcolando gli insiemi dei riferimenti S_1 e S_2 ai record che soddisfano le condizioni ψ_1 e ψ_2 per recuperare poi i record con identificatori in $S_1 \cap S_2$.

Analogamente, se la condizione ψ è la somma logica di condizioni semplici $\psi = \psi_1 \vee \psi_2$ su attributi con indice, si possono calcolare gli insiemi dei riferimenti S_1 e S_2 dei record che soddisfano le condizioni ψ_1 e ψ_2 per recuperare poi i record con identificatori in $S_1 \cup S_2$.

Vediamo gli operatori fisici che realizzano questi algoritmi, supponendo di usare al più un solo indice:

- **Filter**(O, ψ): ritorna la collezione dei record di O che soddisfano la condizione ψ.
- **IndexFilter**(R, Idx, ψ): ritorna la collezione dei record di R che soddisfano la condizione ψ, con l'uso dell'indice Idx definito su attributi di R. La condizione ψ è un prodotto logico di predicati che interessano solo i valori degli attributi sui quali è definito l'indice. Il risultato è ordinato sugli attributi di R sui quali è definito l'indice.

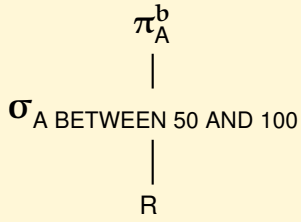
  L'operatore esegue due operazioni: utilizza l'indice per trovare rapidamente i riferimenti ai record che soddisfano la condizione e poi recupera i record di R. Queste due operazioni potrebbero essere realizzate con due operatori fisici che di solito nei DBMS commerciali sostituiscono l'operatore **IndexFilter**(R, Idx, ψ), ma in questo testo non si considerano per semplificare i piani di accesso delle interrogazioni: **TIDIndexFilter**(Idx, ψ), che ritorna un insieme di TID, e **TableAccess**(O, R), che ritorna la collezione dei record di R con i TID in O.

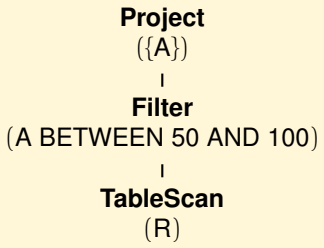
L'**IndexFilter** ha come argomento la relazione R sulla quale è definito l'indice (e non un altro operatore), pertanto in un piano di accesso l'**IndexFilter** può essere solo una foglia e non un nodo interno dell'albero.

Esempio 9.4
Si mostrano alcuni esempi d'uso degli operatori fisici visti in precedenza per definire possibili piani di accesso.

1. Interrogazione **SELECT-FROM-WHERE** (**SFW**)

```
SELECT  A
FROM    R
WHERE   A BETWEEN 50 AND 100;
```

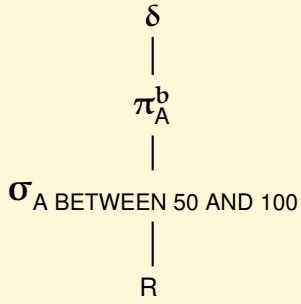


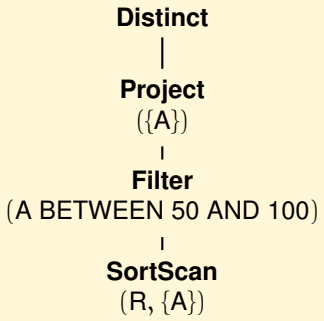


Quando l'interrogazione è semplice c'è una corrispondenza uno a uno tra gli operatori logici e fisici dei due alberi.
Un altro possibile albero fisico equivalente al precedente si ottiene scambiando il **Project** con il **Filter**.

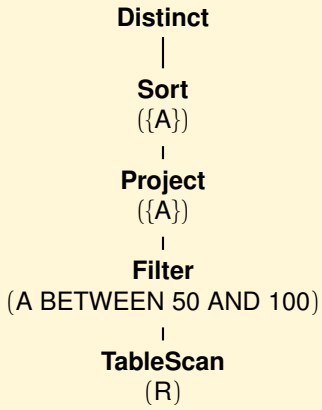
2. Interrogazione **SFW** con **DISTINCT**

```
SELECT   DISTINCT A
FROM     R
WHERE    A BETWEEN 50 AND 100;
```





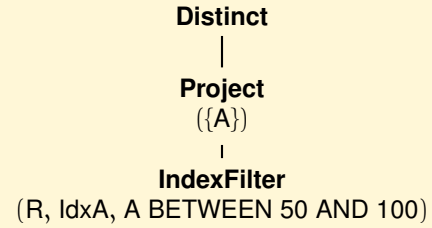
In questo esempio, con **SortScan** si ottiene la collezione ordinata dei record di R, che vengono filtrati dal **Filter**, proiettati dal **Project** e si eliminano i duplicati con **Distinct**.
Se A fosse una chiave, il **DISTINCT** e l'ordinamento di R sarebbero inutili.
Un altro possibile albero fisico equivalente al precedente si ottiene ordinando il risultato del **Project** e non R:



3. Interrogazione **SFW**, con indice

```
SELECT   DISTINCT A
FROM     R
WHERE    A BETWEEN 50 AND 100;
```

con l'ipotesi che esista un indice su A.
L'albero logico è identico al caso precedente.

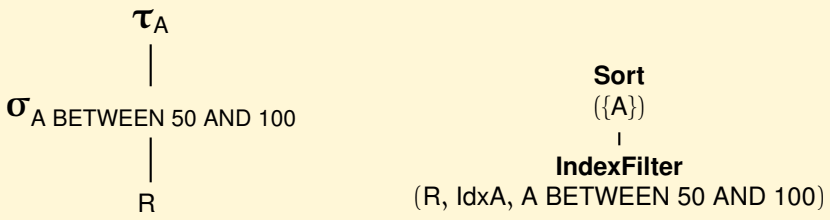


L'indice su A consente di usare l'operatore **IndexFilter** che, restituendo la collezione dei record di R che soddisfano la condizione, ordinati su A, rende inutile il **Sort**. Dopo aver generato un possibile piano di accesso, è bene controllare se alcuni operatori possono essere eliminati.

4. Interrogazione **SFW** con **ORDER BY** e indice

```
SELECT     *
FROM       R
WHERE      A BETWEEN 50 AND 100
ORDER BY   A;
```

con l'ipotesi che esista un indice su A.



Il **Sort** è usato per trattare l'**ORDER BY**, ma in questo caso è inutile perché l'**IndexFilter** ritorna la collezione dei record di R, che soddisfano la condizione, ordinati su A. L'albero fisico finale diventa

Operatori per la giunzione (⋈)

La giunzione è l'operazione più complessa dell'algebra relazionale e può essere eseguita in più modi. Vediamo i metodi principali supponendo di dover eseguire la giunzione $R \underset{B=D}{\bowtie} S$ delle relazioni $R(A, \underline{B})$ e $S(\underline{C}, D)$.

L'algoritmo più ovvio per valutare la giunzione è il *nested loop* che ha la seguente struttura:

Algoritmo 9.1
Nested Loop

for each $r \in R$ **do**
  for each $s \in S$ **do**
    if $r[B] = s[D]$ **then** aggiungi $< r, s >$ al risultato;

Se esiste un indice sull'attributo di giunzione D della relazione S (detta *relazione interna*), un algoritmo più efficiente è l'*index nested loop* che ha la seguente struttura:

Algoritmo 9.2
Index Nested Loop

for each $r \in R$ **do** {
  usa l'indice su D per trovare tutti i record $s \in S$ *tali che* $s[D] = r[B]$;
  aggiungi $< r, s >$ al risultato };

Quando la condizione di giunzione è fra la chiave primaria di R e la chiave esterna di S, e le due relazioni sono ordinate sugli attributi di giunzione, un altro algoritmo interessante è il *merge join* che ha la seguente struttura:

Algoritmo 9.3
Merge Join

```
r = primo record di R; // r = null se R è vuota
s = primo record di S; // s = null se S è vuota
// sia succ(w) il record successivo a w
// o il valore null se w è l'ultimo;
while not r == null and not s == null do
        if r[B] = s[D]
        then{
             while not s == null and r[B] = s[D] do
                    {aggiungi < r, s > al risultato;
                     s = succ(s) };
             r = succ(r) }
        else if r[B] < s[D]
             then r = succ(r)
             else s = succ(s);
```

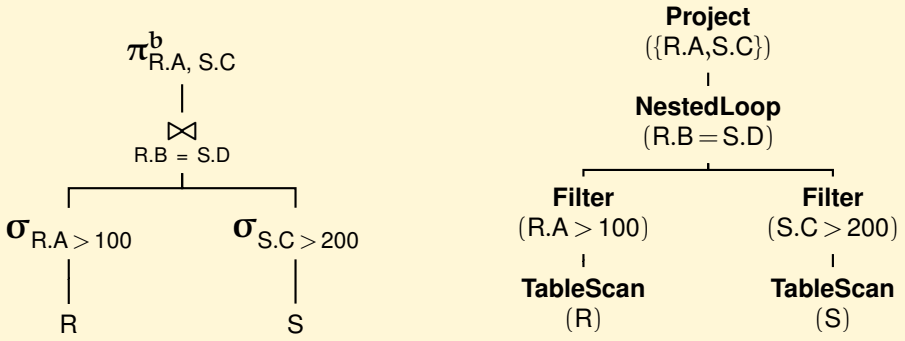
Vediamo gli operatori fisici che realizzano questi algoritmi:

- **NestedLoop**(O_E, O_I, ψ_J): ritorna la giunzione dei record di O_E e O_I che soddisfano la condizione di giunzione ψ_J.
- **IndexNestedLoop**(O_E, O_I, ψ_J): ritorna la giunzione dei record di O_E e O_I che soddisfano la condizione di giunzione ψ_J, supponendo che esista un indice Idx sugli attributi di giunzione della relazione interna.
  Supponiamo per semplicità che la condizione di giunzione sia $O_E.A_i = O_I.A_j$ e che Idx sia un indice su $S.A_j$. L'operatore interno O_I può essere:
  - un **IndexFilter**(S, Idx, ψ_J): per ogni record r di O_E si usa l'indice Idx su $S.A_j$ per trovare i record s di O_I che soddisfano la condizione di giunzione $S.A_j = \underline{O_E.A_i}$, con $\underline{O_E.A_i} = r.A_i$;
  - un **Filter**(O, ψ') con O un **IndexFilter**(S, Idx, ψ_J): per ogni record r di O_E si usa l'indice Idx su $S.A_j$ per trovare i record s di O_I che soddisfano la condizione di giunzione, come nel caso precedente, ma di questi si ritornano solo quelli che soddisfano anche la condizione ψ' del filtro.
- **MergeJoin**(O_E, O_I, ψ_J): ritorna la giunzione dei record di O_E e O_I che soddisfano la condizione di giunzione ψ_J, supponendo che (a) i record di O_E e O_I siano ordinati sui relativi attributi di giunzione e (b) la condizione di giunzione sia fra una chiave di O_E e una chiave esterna di O_I.

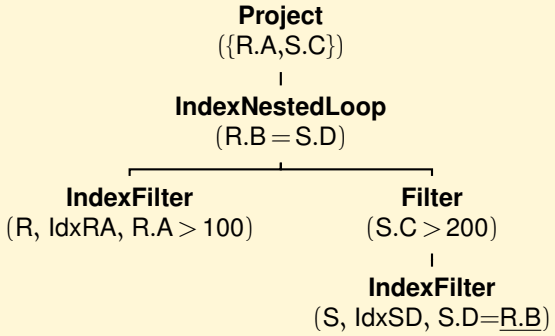
Tutti gli operatori fisici che realizzano la giunzione ritornano collezioni di record che sono la concatenazione di un record r di O_E e di un record s di O_I, ovvero hanno sia gli attributi di r che quelli di s, e preservano l'ordine dei record di O_E.

Esempio 9.5
Si consideri l'interrogazione

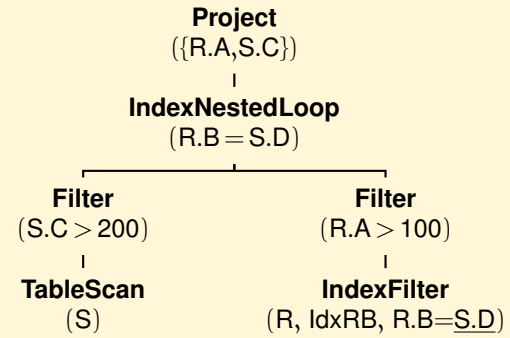
SELECT R.A, S.C
FROM R, S
WHERE R.B = S.D **AND** R.A $>$ 100 **AND** S.C $>$ 200;



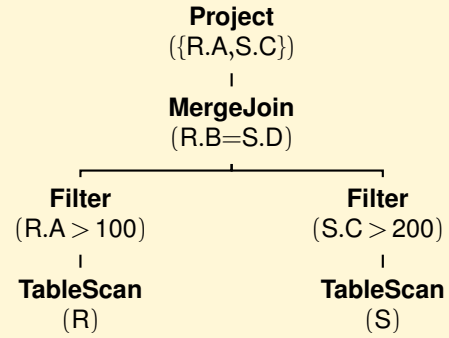
Supponiamo di avere due indici su R.A e su S.D e di usare l'**IndexNestedLoop** per la giunzione; l'albero fisico diventa:



Se ci fosse anche un indice su R.B, si potrebbe usare l'**IndexNestedLoop** con R come relazione interna, ottenendo l'albero fisico:



Infine, se le relazioni R e S fossero ordinate sugli sugli attributi di giunzione R.B e S.D, si potrebbe usare il **MergeJoin** con R come relazione esterna, ottenendo l'albero fisico:



Operatore per il raggruppamento (γ)

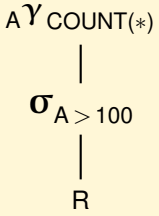
– **GroupBy**(O, $\{A_i\}$, $\{f_i\}$): ritorna una collezione di record, uno per ogni gruppo di record di O, con attributi quelli di raggruppamento $\{A_i\}$ e le funzioni di aggregazione $\{f_i\}$, supponendo che i record di O siano ordinati sugli attributi di raggruppamento. Il risultato è ordinato sugli attributi di raggruppamento.[1]

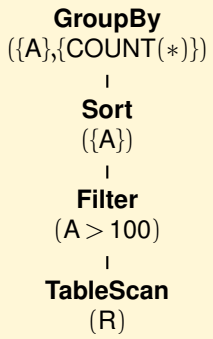
Esempio 9.6
Vediamo possibili piani di alcune interrogazioni con raggruppamento:

1. Interrogazione **SFW** con **GROUP BY**

1. Nei sistemi commerciali è previsto anche un altro operatore **HashGroupBy** che raggruppa i record di O non ordinati usando una tecnica *hash*, ma per semplicità non si prende in considerazione.

SELECT A, **COUNT**(∗)
FROM R
WHERE A > 100
GROUP BY A;

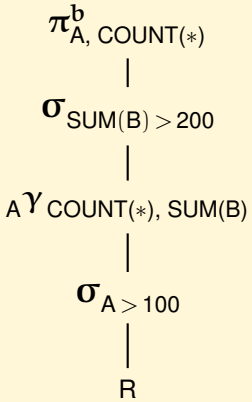


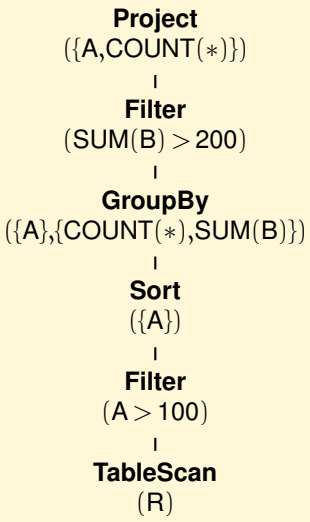


Per usare il **GroupBy** occorre ordinare i dati sui quali opera. Il **Project** è necessario solo se gli attributi della **SELECT** sono un sottoinsieme di quelli dei record prodotti dal **GroupBy**.

2. Interrogazione **SFW** con **GROUP BY** e **HAVING**

SELECT A, **COUNT**(∗)
FROM R
WHERE A > 100
GROUP BY A
HAVING **SUM**(B) > 200;





Si noti che (a) le funzioni di aggregazione di γ e **GroupBy** sono quelle diverse della **SELECT** e dell'**HAVING**, (b) la condizione dell'**HAVING** diventa un **Filter** sul **GroupBy** e (c) per produrre il risultato occorre un **Project**.

Unione, differenza e intersezione

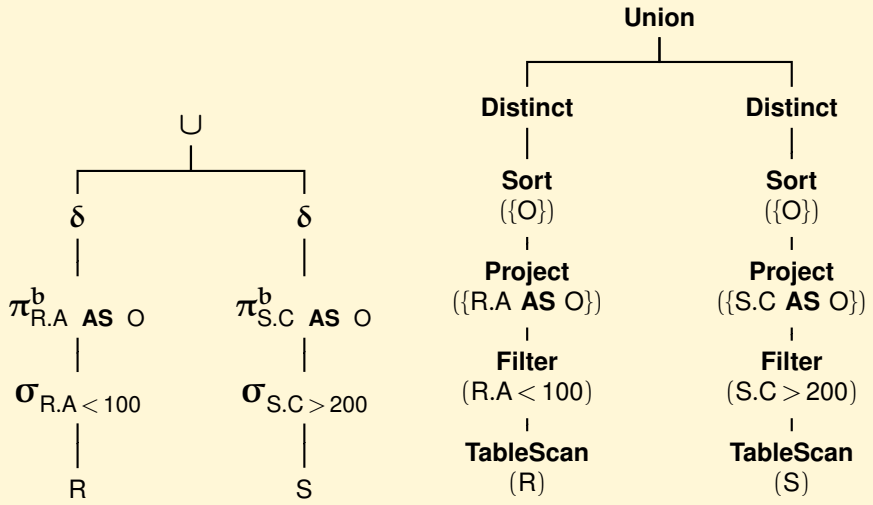
L'operazione di unione è semplice da realizzare se sono ammessi record duplicati nel risultato (operatore **UNION ALL** in SQL). Se i duplicati vanno eliminati (operatore insiemistico **UNION** in SQL), l'operazione è ancora semplice da realizzare nell'ipotesi che i record delle due relazioni siano ordinati e privi di duplicati. In modo analogo si procede per la differenza e l'intersezione insiemistica di due relazioni. Altri modi di procedere sono possibili per eseguire queste operazioni che non richiedono l'ordinamento dei dati e l'assenza di duplicati, ma per semplicità non si prendono in considerazione.

Vediamo gli operatori fisici che realizzano questi algoritmi:

- **Union**(O_E, O_I), **Except** (O_E, O_I), **Intersect** (O_E, O_I): ritornano la collezione dei record delle operazioni insiemistiche, supponendo che i record degli operandi siano dello stesso tipo, ordinati e privi di duplicati.
- **UnionAll** (O_E, O_I): ritorna l'unione dei record degli operandi senza l'eliminazione dei duplicati.

Esempio 9.7
Si consideri l'interrogazione:

```
SELECT  R.A AS O
FROM    R
WHERE   R.A < 100
    UNION
SELECT  S.C AS O
FROM    S
WHERE   S.C > 200;
```



9.6.3 Esecuzione di un piano di accesso

Ogni operatore fisico di un piano di accesso è un *iteratore*, cioè è realizzato con un oggetto con quattro metodi:

- **open**(): inizializza lo stato dell'operatore ed esegue il metodo **open** degli operatori fisici dei suoi argomenti.
- **next**(): ritorna il record successivo del risultato interagendo con gli operatori fisici dei suoi argomenti.
- **isDone**(): ritorna *true* se non ci sono altri record da ritornare, *false* altrimenti.
- **close**(): termina le operazioni dell'operatore fisico e dei suoi argomenti.

Supponendo che AlberoFisico sia il nome di un piano di accesso, il risultato dell'interrogazione si ottiene eseguendo il seguente programma:

```
AlberoFisico.open();                  //inizia le operazioni
while !AlberoFisico.isDone()          //finché ci sono elementi
    print(AlberoFisico.next());       //stampa prossimo record
AlberoFisico.close();                 //termina le operazioni
```

Quando si esegue il metodo **open** della radice dell'albero, esso inizializza lo stato dell'operatore ed esegue l'**open** del figlio e così via fino ad arrivare alle foglie. Terminata la fase di inizializzazione, inizia la fase di generazione del risultato.

Quando si esegue il metodo **next** della radice dell'albero, esso esegue il **next** del figlio per ottenere il prossimo record del risultato, il figlio fa la stessa operazione sul figlio e così via fino ad arrivare alla foglia: i record ritornati dalla foglia faranno la

strada inversa. Ogni operatore quindi è pronto a restituire un record alla volta, risultato dell'elaborazione del record ottenuto dal figlio (o dai figli, nel caso di operatori binari).

Il Sort è l'unico operatore che si discosta da questo schema: il metodo che compie gran parte del lavoro è **open**, che richiede tutti i record al nodo figlio, li memorizza ordinati in una relazione temporanea per poi restituirli uno alla volta su richiesta del nodo padre.

9.7 Gestore della concorrenza

La tecnica utilizzata più comunemente per realizzare il controllo della concorrenza nei sistemi centralizzati è la tecnica del *blocco a due fasi* (*Two Phase Lock, 2PL*) .

Utilizzando questa tecnica, si associa ad ogni dato usato da un'operazione un blocco (*lock*) in lettura o in scrittura. Ogni transazione, prima di eseguire un'azione su di un dato, richiede il blocco corrispondente di lettura o scrittura. Due transazioni non possono avere blocchi *incompatibili* sullo stesso dato, ovvero blocchi con almeno uno dei due di scrittura. Pertanto in ogni momento, per ogni dato possono essere stati assegnati o più blocchi in lettura o, alternativamente, un solo blocco in scrittura. La transazione che richiede un blocco incompatibile su un dato viene messa in attesa, e risvegliata solo quando il dato diventa disponibile. Per garantire la serializzabilità e l'isolamento, ogni transazione viene di solito eseguita con la tecnica del blocco a due fasi stretto, che prevede che i blocchi di una transazione vengano rilasciati tutti assieme, solo dopo che la transazione sia terminata.

La tecnica del blocco prevede che una transazione venga posta in attesa quando richiede un blocco che non può ottenere. Può accadere che questa attesa diventi infinita: supponiamo che T_1 ottenga un blocco in scrittura su X_1 e T_2 ottenga un blocco in scrittura su X_2, e che successivamente T_1 richieda anch'essa un blocco in scrittura su X_2: in questo caso T_1 viene messa in attesa del fatto che T_2 rilasci il proprio blocco. Se a questo punto T_2 richiede un blocco su X_1, anche T_2 va in attesa del fatto che T_1 rilasci il proprio blocco, e si crea una situazione di attesa "circolare", ovvero di *stallo* (*deadlock*).

I metodi comunemente usati per sbloccare una situazione di stallo sono i seguenti.

- Rilevazione tramite grafo delle attese: si costruisce un grafo avente come nodi le transazioni, aggiungendo un arco da T_1 a T_2 ogni volta che T_1 va in attesa del rilascio di un blocco da parte di T_2. Ogni volta che si crea un ciclo nel grafo, una delle transazioni coinvolta viene fatta abortire (tipicamente la più giovane, quella che ha meno risorse o quella il cui aborto ha il minor costo).
- Rilevazione per *time-out*: ogni volta che un'attesa si prolunga oltre un certo limite, la transazione in attesa viene abortita, presupponendo l'esistenza di uno stallo.

Il blocco a due fasi garantisce la serializzabilità, ma limita la concorrenza possibile tra diverse transazioni. Ad esempio, un'applicazione lunga che legge una grande quantità di dati potrebbe non riuscire mai ad acquisire tutti i blocchi necessari, oppure, quando li avesse acquisiti tutti, potrebbe impedire ad ogni altra transazione che vo-

glia effettuare modifiche di partire. Per questo motivo, molti sistemi permettono al programmatore di limitare la quantità di blocchi richiesti dalle applicazioni, anche se questo comporta la perdita della serializzabilità (si veda il Capitolo 8).

9.8 Gestore dell'affidabilità

Compito del gestore dell'affidabilità è di eseguire le operazioni delle transazioni e la loro terminazione garantendo che la base di dati contenga solo gli effetti delle transazioni terminate normalmente e sia protetta da fallimenti di transazione, di sistema e disastri.

Le operazioni delle transazioni possono essere eseguite con algoritmi diversi. Supponiamo che si adotti l'algoritmo *disfare-rifare*, come accade nei sistemi DB2 e Oracle:

- Una modifica di un dato può essere riportata sulla base di dati prima che la transazione termini. Nel caso di fallimento di transazione o di sistema occorre annullare le modifiche fatte dalla transazione sulla base di dati (disfare).
- Una transazione T è considerata terminata normalmente, e viene scritto nel *giornale* (descritto più avanti) il record (T, *commit*), senza che le sue modifiche vengano preventivamente riportate nella base di dati; questo compito viene svolto dal gestore del buffer quando lo ritiene opportuno. Nel caso di fallimento di sistema occorre rifare le modifiche fatte dalle transazioni terminate normalmente perché non si è certi che i loro effetti siano stati riportati sulla base di dati.

La struttura dati che viene utilizzata in maniera cruciale tanto per disfare che per rifare gli effetti delle transazioni è il *giornale delle modifiche* (*log*). Questo è un archivio gestito in maniera sicura (cioè mantenuto in due copie su dispositivi con fallimento indipendente) che contiene, per ogni operazione effettuata da una transazione sui dati, le seguenti informazioni:

- L'identificatore della transazione che ha effettuato l'operazione.
- L'operazione eseguita (inserzione, aggiornamento, cancellazione, inizio transazione, *commit*, *abort*).
- L'identificatore del record modificato.
- Il vecchio ed il nuovo valore del record.

Poiché un malfunzionamento può anche intercorrere tra il momento in cui un'operazione viene eseguita ed il momento in cui tale operazione viene registrata nel giornale, è essenziale registrare tutte le operazioni nel giornale prima che esse vengano eseguite. Più precisamente, è necessario seguire le due regole seguenti:

1. *Regola per disfare* (*Write ahead log*): prima di eseguire una modifica, occorre salvare il vecchio valore nel giornale, in modo che non vada perduto in caso di malfunzionamento.
2. *Regola per rifare* (*Commit Rule*): prima di considerare terminata una transazione, occorre salvare nel giornale i nuovi valori dei dati modificati, in modo da poter rieseguire la transazione in caso di fallimento di sistema o di disastro.

Avendo a disposizione il giornale, ed una vecchia copia della base di dati, la procedura di ripristino in caso di malfunzionamento è la seguente:

- In caso di fallimento di transazione, si disfano gli effetti di tutte le operazioni della transazione, utilizzando le informazioni sul giornale, ed infine si memorizza una marca di *abort* sul giornale stesso.
- In caso di fallimento di sistema, prima di rendere operativa la base di dati, si esegue il comando *restart* che scandisce il giornale per disfare gli effetti delle transazioni attive al momento del fallimento e per rifare gli effetti delle transazioni terminate normalmente. Per limitare poi la porzione di giornale da scandire, i DBMS effettuano ad intervalli brevi e regolari un'operazione di allineamento (*checkpoint*) che consiste nel riportare in memoria permanente tutti gli aggiornamenti effettuati nei buffer, e nel memorizzare poi sul giornale una marca di *checkpoint*. In questo modo, in caso di fallimento di sistema, la procedura di ripristino può partire dallo stato del giornale e della base di dati in linea, avendo la certezza che non c'è bisogno di rifare nessuna delle operazioni eseguite prima del *checkpoint*. Si osservi tuttavia che è necessario disfare le operazioni eseguite prima e dopo il *checkpoint* da transazioni non terminate normalmente.
- In caso di disastro, si porta in linea la vecchia copia stabile dello stato della base di dati e si riapplicano ad essa tutte le operazioni registrate sul giornale da parte di transazioni che abbiano effettuato il *commit*. Se la vecchia copia era stata presa in un momento di attività del sistema, è anche necessario disfare gli effetti di tutte le transazioni che erano in corso al momento della copia e che non avevano ancora effettuato il *commit* al momento del malfunzionamento.

Si osservi che la regola per disfare garantisce che il vecchio valore di un record modificato non vada mai perduto, ma implica che, in seguito ad un malfunzionamento avvenuto poco dopo la scrittura del record nel giornale, non sia possibile sapere se l'operazione fosse stata realmente effettuata sui dati. Si osservi inoltre che un malfunzionamento può avere luogo anche durante il ripristino. Per ambedue questi motivi, è importante che le operazioni di "disfacimento" e "rifacimento" che si effettuano durante il ripristino siano effettuate in maniera "idempotente", ovvero in modo da ottenere l'effetto voluto sia che si stia disfacendo un'operazione realmente effettuata, sia che si stia disfacendo un'operazione già disfatta.

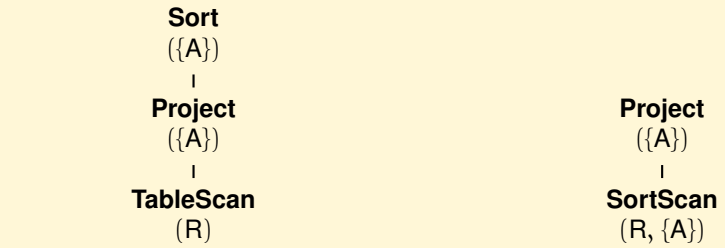
9.9 Conclusioni

Sono state presentate le principali tecniche utilizzate per realizzare le funzionalità fondamentali di un DBMS centralizzato: la gestione dei dati, delle interrogazioni, della concorrenza e dell'affidabilità. Per mostrare come venga eseguita un'interrogazione SQL è stato introdotto il formalismo dei piani di accesso, una versione semplificata di quello usato dai sistemi commerciali per visualizzare il risultato dell'ottimizzazione fisica di un'interrogazione. Questa informazione è molto utile all'amministratore della base di dati per la messa a punto delle strutture di memorizzazione dei dati al fine di un'esecuzione efficiente delle interrogazioni.

Per fare pratica con i piani di accesso si può il sistema **JRS**, il cui ottimizzatore produce sia piani logici che di accesso delle interrogazioni con gli operatori descritti in questo capitolo, e prevede due interfacce grafiche per definire ed eseguire alberi logici o fisici. È interessante confrontare il piano logico o di accesso immaginato per un'interrogazione con quello prodotto da un sistema con un buon ottimizzatore.

Esercizi

1. Dire quali delle seguenti affermazioni è vera o falsa e giustificate la risposta:
   a) I seguenti piani di accesso per l'interrogazione
   SELECT A **FROM** R **ORDER BY** A
   non sono equivalenti:



   b) Operatore logico e operatore fisico sono sinonimi.
   c) Ad ogni interrogazione SQL corrispondono più alberi logici.
   d) Ad ogni albero logico corrispondono più alberi fisici.
   e) La gestione della concorrenza con il blocco dei dati non crea condizioni di stallo.
   f) Con il metodo disfare–rifare, nessun dato modificato da una T può essere riportato nella BD prima che il corrispondente record del giornale sia scritto nella memoria permanente.
   g) Con il metodo rifare, tutte le modifiche di una T devono essere riportate nella BD prima che il record di commit sia scritto nel giornale.
2. Si considerino due relazioni unarie $R(A)$ e $S(B)$ con i seguenti record:
   $R = \{(A := 7), (A := 2), (A := 8), (A := 3), (A := 1), (A := 3), (A := 6)\}$
   $S = \{(B := 4), (B := 2), (B := 1), (B := 3), (B := 2), (B := 7), (B := 3)\}$

   Mostrare (a) il contenuto di un indice sull'attributo A di R e (b) il risultato prodotto dalla giunzione $R \underset{A=B}{\bowtie} S$ usando il NestedLoop.
3. Si consideri la relazione R(A, B, C), con chiave primaria A, e l'interrogazione:

```
SELECT    A, COUNT(*)
FROM      R
```

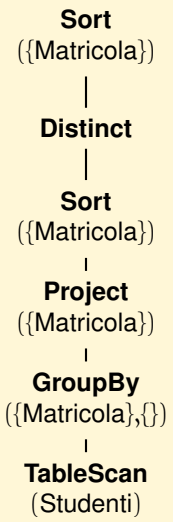
WHERE A $>$ 100
GROUP BY A;

Dare l'albero logico iniziale dell'interrogazione e un possibile piano di accesso. Come cambia la soluzione se nella **SELECT** ci fosse **DISTINCT** A, **COUNT**($*$)?

4. Si consideri la relazione Studenti(Matricola, Nome, AnnoNascita), ordinata sulla chiave primaria Matricola, e l'interrogazione:

SELECT **DISTINCT** Matricola, **COUNT**($*$)
FROM Studenti
WHERE AnnoNascita = 1974
GROUP BY Matricola
ORDER BY Matricola;

Dare l'albero logico iniziale dell'interrogazione e si dica se il seguente piano d'accesso produce il risultato cercato. Se non va bene, lo si modifichi in tre modi:
a) aggiungendo prima solo le parti mancanti (operatori e parametri),
b) semplificando poi il piano eliminando operatori inutili e
c) modificando infine il piano supponendo che esita un indice su AnnoNascita:



5. Si consideri il seguente schema relazionale:

Aule(CodiceA, Edificio, Capienza)
Lezioni(CodiceA, CodiceC, Ora, GiornoSett, Semestre)
Corsi(CodiceC, NomeC, Docente)

e l'interrogazione:

```
SELECT      A.Edificio, COUNT(*)
FROM        Aule A, Lezioni L
WHERE       A.CodiceA = L.CodiceA AND Semestre = 1
GROUP BY    A.Edificio
HAVING      COUNT(*) > 2;
```

Si dia l'albero logico iniziale dell'interrogazione e un piano di accesso che utilizzi indici:
a) nel caso di giunzione con l'operatore NestedLoop e
b) nel caso di giunzione con l'operatore IndexNestedLoop.

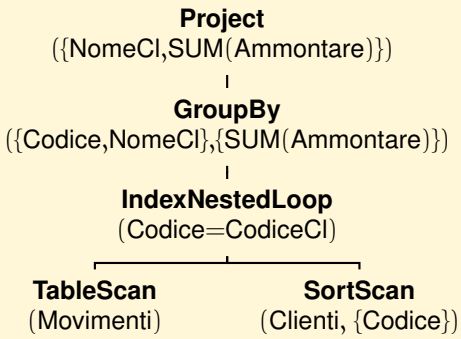
6. Si consideri la base di dati:

Clienti(Codice, NomeCl, AnnoNascita), con chiave primaria Codice
Movimenti(CodiceCl, Ammontare, Tipo), con chiave esterna CodiceCl

e l'interrogazione:

```
SELECT      NomeCl, SUM(Ammontare)
FROM        Clienti, Movimenti
WHERE       Codice = CodiceCl AND AnnoNascita = 1974
GROUP BY    Codice, NomeCl
HAVING      COUNT(*) > 5 ;
```

Dare l'albero logico iniziale dell'interrogazione e si dica (a) se il seguente piano d'accesso è corretto, (b) se produce il risultato cercato. Se non va bene, lo si modifichi aggiungendo le parti mancanti (operatori e parametri).



Note bibliografiche

Per approfondire gli argomenti di questo capitolo si rinvia al libro [Albano, 2001], dedicato alle strutture e algoritmi per la realizzazione di DBMS. Il sistema JRS con la relativa documentazione è disponibile sul sito Web del libro alla pagina http://fondamentidibasididati.it/.

BIBLIOGRAFIA

Abiteboul, S. and Bidoit, N. (1984). An algebra for non normalized relations. In *Proceedings of the ACM SIGACT-SIGMOD Symposium on Principles of Database Systems (PODS)*. 137

Abiteboul, S., Hull, R., and Vianu, V. (1995). *Database Foundations*. Addison-Wesley, Reading, Massachusetts. 27, 139, 182

Albano, A. (2001). *Costruire sistemi per basi di dati*. Addison-Wesley, Milano. 232, 264, 292

Albano, A., Antognoni, G., and Ghelli, G. (2000). View operations on objects with roles for a statically typed database language. *IEEE Transactions on Knowledge and Data Engineering*, 12(4):548–567. 40, 48, 50

Albano, A., Cardelli, L., and Orsini, R. (1985). Galileo: A strongly typed, interactive conceptual language. *ACM Transactions on Database Systems*, 10(2):230–260. Also in S.B. Zdonik and D. Maier, editors, *Readings in Object-Oriented Database Systems*, Morgan Kaufmann Publishers, Inc., San Mateo, California, 1990. 40, 48, 50

Armstrong, W. (1974). Dependency structures of database relationships. In *Proceedings of the IFIP Congress*, pages 580–583. 147

Atzeni, P. and Antonellis, V. D. (1993). *Relational Database Theory*. Morgan Kaufmann Publishers, San Mateo, California. 139, 161, 182

Atzeni, P., Ceri, S., Paraboschi, S., and Torlone, R. (2002). *Basi di dati. Modelli e linguaggi di interrogazione*. McGraw-Hill, Milano. 27, 102

Batini, C., Ceri, S., and Navathe, S. (1992). *Conceptual Database Design. An Entity-Relationship Approach*. The Benjamin/Cummings Publishing Company, Inc., Redwood City, California. 74, 102

Batini, C., Lenzerini, M., and Navathe, S. (1987). A comparative analysis of methodologies for database schema integration. *ACM Computing Surveys*, 18(4):2–18. 91

Beeri, C. and Bernstein, P. (1979). Computational problems related to the design of normal form relational schemas. *ACM Transactions on Database Systems*, 4(1):30–59. 151

Bernstein, P. (1976). Synthesizing third normal form relations from functional

dependencies. *ACM Transactions on Database Systems*, 1(4):277–298. 173, 174

Ceri, S., editor (1983). *Methodology and Tools for Database Design*. North-Holland, Amsterdam. 102

Ceri, S., Gottlob, G., and Tanca, L. (1990). *Logic Programming and Data Bases*. Springer-Verlag, Berlin. 139

Codd, E. (1970). A relational model for large shared data banks. *Communications of the ACM*, 13(6):377–387. 139, 145

Connolly, T. and Begg, C. (2000). *Database Solutions. A step-by-step guide to building databases*. Addison-Wesley, Reading, Massachusetts. 102

Diederich, J. and Milton, J. (1988). New methods and fast algorithms for database normalization. *ACM Transactions on Database Systems*, 13(3):339–365. 157

Elmasri, R. and Navathe, S. (2001). *Sistemi di basi di dati. Fondamenti. Prima edizione italiana*. Addison-Wesley, Milano. 27

Fraternali, P. and Tanca, L. (1995). A structured approach for the definition of the semantics of active databases. *ACM Transactions on Database Systems*, 20(4):414–471. 228

Harel, D. (1987). Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8:231–274. 75

Kifer, M., Bernstein, A., and Lewis, P. M. (2005). *Database Systems*. Addison-Wesley, Reading, Massachusetts, second edition. 27, 213, 262

Lucchesi, C. and Osborn, S. (1978). Candidate keys for relations. *Journal of Computer and System Sciences*, 17(2):270–280. 152

Maciaszek, L. A. (2002). *Sviluppo di sistemi informativi con UML*. Addison-Wesley, Milano. 102

Maier, D. (1983). *The Theory of Relational Databases*. Computer Science Press, Rockville, Maryland. 139, 156, 157, 182

Mannila, H. and Räihä, K. (1992). *The Design of Relational Databases*. Addison-Wesley, Reading, Massachusetts. 182

Marco, T. D. (1979). *Structured Analysis and System Specification*. Prentice Hall, Inc., Englewood Cliffs, New Jersey. 74

Ramakrishnan, R. and Gehrke, J. (2003). *Sistemi di basi di dati*. McGraw-Hill, Milano. 27

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W. (1991). *Object-Oriented Modeling and Design*. Prentice Hall International, Inc., London. 74, 75

Schmidt, J. (1977). Some high level language constructs for data of type relation. *ACM Transactions on Database Systems*, 2(3):247–261. 251

Shasha, D. and Bonnet, P. (2002). *Database Tuning: principles, experiments, and troubleshooting techniques*. Morgan Kaufmann Publishers, San Mateo, California. 238

Silberschatz, H. F., Korth, H. F., and Sudarshan, S. (2002). *Database System Concepts*. McGraw-Hill, New York, 4th edition. 27, 139

Teorey, T. (1999). *Database Modeling and Design. The E-R Approach*. Morgan Kaufmann Publishers, San Mateo, California, third edition. 102

Tsou, D. and Fischer, P. (1982). Decomposition of a relation scheme into Boyce-Codd

Normal Form. *ACM SIGACT News*, 14(3):23–29. 169

Ullman, J. (1983). *Principles of Database Systems*. Computer Science Press, Rockville, Maryland, second edition. 163

Ullman, J. D. and Widom, J. (2001). *A First Course in Database System*. Prentice Hall, Inc., Englewood Cliffs, New Jersey, second edition. 27, 139, 156

van der Lans, R. (2001). *Introduzione a SQL. Seconda edizione italiana*. Addison-Wesley, Milano. 213, 262

Widom, J. and Ceri, S., editors (1996). *Active Database Systems: Trigger and Rules for Advanced Database Programming*. Morgan Kaufmann Publishers, San Mateo, California. 238

Yourdon, E. (1989). *Modern Structured Analysis*. Yourdon Press, Englewood Cliffs, New Jersey. 74

Zaniolo, C., Ceri, S., Faloutsos, C., Snodgrass, R., Subrahmanian, V., and Zicari, R., editors (1997). *Introduction to Advanced Database Systems*. Morgan Kaufmann Publishers, San Mateo, California. 238

INDICE ANALITICO