

## Capitolo 1

# Basi di dati e sistemi di gestione di basi di dati

Le informazioni sono sempre state una risorsa vitale in qualsiasi organizzazione, indipendentemente dal grado di informatizzazione della stessa. Pensiamo ad esempio alle banche, che fanno della gestione delle informazioni il loro principale obiettivo ed esistono da ben prima dell'avvento dei calcolatori. Oggi, la crescente diffusione dell'informatica ed il fatto che molte organizzazioni offrano principalmente servizi (ad esempio, consulenze di vario genere, servizi di prenotazione o di assistenza remota) piuttosto che produrre beni materiali, hanno reso l'informazione una delle risorse maggiormente strategiche in ogni contesto. La diffusione di Internet e del Web ha ulteriormente accresciuto tale tendenza, in quanto ha reso più agevole lo scambio delle informazioni, abbattendo di fatto le barriere geografiche. In questo contesto, appare pertanto ovvio come un ruolo di primaria importanza sia ricoperto da tutti quegli strumenti in grado di acquisire, elaborare, trasmettere ed archiviare informazioni. Questi strumenti sono solitamente costituiti da risorse umane, da strumenti preposti all'archiviazione delle informazioni e da procedure manuali od automatizzate per il trattamento delle informazioni.

L'importanza della gestione dell'informazione ha motivato notevoli sforzi ed investimenti in ricerca e sviluppo con l'obiettivo di ottenere sistemi di gestione dell'informazione sempre più sofisticati e completi. Questi sforzi hanno portato allo sviluppo di tecniche in grado di minimizzare i tempi ed i costi di archiviazione ed elaborazione delle informazioni assicurando al contempo la correttezza e la qualità delle stesse.

L'obiettivo di questo libro è una trattazione quanto più completa possibile dei moderni sistemi per la gestione dell'informazione, discutendone sia gli aspetti legati all'uso sia gli aspetti architetturali. Questo primo capitolo, oltre a delineare brevemente l'evoluzione storica di tali sistemi, introduce i concetti fondamentali necessari alla trattazione seguente.

### 1.1 Concetti introduttivi

Come abbiamo ricordato nell'introduzione al capitolo, una delle principali esigenze di ogni organizzazione è quella di gestire e rendere disponibili le informazioni. Il sistema preposto a tale compito prende il nome di *sistema informativo*. Più precisamente, un sistema informativo è quella componente di un'organizzazione

che gestisce le informazioni di interesse, che si occupa cioè di produrre, acquisire, elaborare, conservare e distribuire le informazioni. Esso è costituito da strumenti, procedure e strutture, sia automatizzate sia manuali, e la sua definizione è del tutto indipendente dal grado di automazione in essere. Ad esempio, uno scaffale in cui vengono archiviate delle pratiche è una componente del sistema informativo, così come lo è un insieme di file in cui sono archiviate le informazioni anagrafiche dei dipendenti di un'azienda. Per sottolineare il fatto che un sistema informativo non presuppone necessariamente l'ausilio di un supporto informatico, denotiamo con il termine *sistema informatico* la parte del sistema informativo che gestisce l'informazione mediante l'ausilio di strumenti e tecnologie informatiche. Oggi, data la capillare diffusione dell'informatica, i termini sistema informativo e sistema informatico sono spesso utilizzati come sinonimi. Nel prosieguo della trattazione utilizzeremo anche noi tale convenzione a meno che sia necessaria un'esplicita distinzione.

Un sistema informativo deve quindi in primo luogo offrire degli strumenti per la rappresentazione dell'informazione, mediante una qualche codifica. Completiranno poi il sistema informativo una serie di programmi applicativi e di sistemi che, operando su tale rappresentazione, realizzano tutte le funzioni necessarie alla gestione delle informazioni. Vediamo quindi di iniziare a chiarire meglio il concetto di informazione. Tutti noi abbiamo una percezione intuitiva del suo significato, ma se chiediamo a persone diverse di fornire una definizione di informazione è molto facile ottenere risposte differenti, proprio per la natura immateriale di tale concetto. Affidiamoci quindi al dizionario della lingua italiana, dove alla voce informazione troviamo: “*Tutto ciò che produce variazioni nel patrimonio conoscitivo di un soggetto detto percettore dell'informazione*” [BPLS91]. Notiamo come, in questa definizione, sia fondamentale il concetto di *utilità*. Se chiedo l'orario di un treno ad una persona che parla una lingua che non conosco, la risposta non è per me un'informazione in quanto non sono in grado di interpretarne il significato e quindi non produce variazioni nel mio patrimonio conoscitivo. Un sistema informativo, quindi, deve in primo luogo fornire una chiave di lettura mediante cui interpretare l'informazione che gestisce. Nei sistemi informatici, le informazioni sono rappresentate sotto forma di *dati*, dove per dato intendiamo una registrazione della descrizione di una qualsiasi caratteristica del dominio di interesse su un supporto che ne garantisca la conservazione e, mediante un insieme di simboli, ne garantisca la comprensibilità e la reperibilità. Consideriamo ad esempio un dato rappresentato dal numero 4; tale dato non fornisce in effetti alcuna informazione. Viceversa, dire che 4 è il numero di film noleggiati da Anna Rossi nell'ultimo mese fornisce un'informazione. Come appare evidente da questo semplice esempio, i dati hanno bisogno di un *contesto interpretativo* che permetta di estrarre da essi le informazioni di interesse per gli utenti. Uno degli obiettivi fondamentali di un sistema informativo è quindi quello di fornire un contesto interpretativo ai dati, in modo da consentire un accesso efficace alle informazioni da essi rappresentate. Con il termine *base di dati* denotiamo, quindi, una collezione di dati tra loro correlati, utilizzati per rappresentare le informazioni di interesse in un sistema informativo.

Un *sistema di gestione di basi di dati (DBMS – Data Base Management System)*, spesso abbreviato nel seguito in *sistema di gestione dati* è, invece, un sistema software, centralizzato o distribuito, che fornisce gli strumenti necessari a gestire le informazioni. Una base di dati può quindi essere alternativamente definita come una collezione di dati gestita da un DBMS.

Le basi di dati ed i DBMS, a cui è dedicato il presente testo, costituiscono il “cuore” di ogni sistema informativo, in quanto sono gli strumenti mediante cui viene realizzata la gestione delle informazioni. Nei prossimi paragrafi, discuteremo le evoluzioni dei sistemi di gestione dati che hanno portato ai moderni DBMS, chiariremo meglio il ruolo del DBMS all’interno di un sistema informativo ed illustreremo i principali servizi che esso offre.

## 1.2 Dai sistemi operativi ai DBMS

La continua evoluzione dei sistemi di gestione dati, iniziata alla fine degli anni sessanta, ha ormai affermato la tecnologia dei DBMS come una componente essenziale nella realizzazione di qualsiasi sistema informativo. I primi sistemi informativi erano basati sull’uso di archivi separati, gestiti dal sistema operativo. A partire da questa tecnologia si è passati ad un approccio in cui i dati vengono organizzati in un unico insieme logicamente integrato, la base di dati appunto, gestito dal DBMS ed in grado di soddisfare il fabbisogno informativo di tutte le applicazioni.

Per meglio comprendere il notevole salto di qualità apportato dalla tecnologia dei DBMS rispetto ai precedenti approcci basati sul semplice uso del file system, consideriamo un esempio applicativo che ci servirà per discutere le problematiche tipiche di tali approcci.

**Esempio 1.1** Supponiamo che una videoteca voglia mantenere informazioni relative ai propri clienti ed ai noleggi che questi hanno effettuato e supponiamo che le applicazioni usino direttamente i servizi del file system per la memorizzazione e l’accesso a tali dati. In base a tale approccio, i dati relativi ai clienti ed ai noleggi sono mantenuti in record memorizzati in vari file su memoria secondaria. Supponiamo che, in aggiunta ai file, esista un insieme di programmi applicativi tra cui:

- un programma di modifica della residenza di un dato cliente;
- un programma per l’inserimento di un nuovo noleggio;
- un programma per l’inserimento e la cancellazione di un cliente;
- un programma che stampa la lista di tutti i clienti della videoteca in ordine alfabetico. □

L’approccio discusso nell’esempio precedente presenta numerosi svantaggi, i principali dei quali sono discussi nel seguito.

**Ridondanza ed inconsistenza nei dati.** Poiché i file di dati ed i programmi sono creati in tempi diversi da progettisti software e programmatore diversi e non esiste una descrizione ad alto livello e centralizzata dei dati, una stessa informazione può essere replicata in file diversi; determinare se ed in che file una certa informazione è memorizzata è estremamente difficile se non impossibile. Con riferimento all’Esempio 1.1, il nome, il cognome ed il numero di tessera di un cliente possono essere memorizzati sia nel file che contiene le informazioni sui clienti sia nel file che contiene le informazioni sui noleggi. La presenza dello stesso dato in file diversi è detta *ridondanza* e può causare alti costi di memorizzazione ed inconsistenze nei dati. Ad esempio, se il numero di tessera di un cliente viene modificato, a seguito dello smarrimento della vecchia tessera, è necessario riportare tale cambiamento non solo nel file dei clienti ma anche in tutti i record relativi ai noleggi effettuati dal cliente in questione. Nei Capitoli 2, 5, 6 e 10 vedremo quali strumenti gli attuali DBMS forniscono per limitare inconsistenze e ridondanze dei dati, sia a livello di rappresentazione dei dati sia a livello di strumenti di progettazione della base di dati.

**Difficoltà nell’accesso ai dati.** La mancanza di una descrizione ad alto livello e centralizzata dei dati ne rende estremamente difficoltoso l’utilizzo al fine di rispondere a nuove esigenze applicative. Supponiamo, ad esempio, che la videoteca preveda un programma di fidelizzazione che qualifica i clienti come VIP a seguito dell’accumulo di un certo numero di punti. Supponiamo che sia necessario stampare periodicamente la lista dei clienti VIP ordinata in ordine alfabetico. Poiché tale richiesta non era stata prevista al momento della progettazione delle applicazioni illustrate nell’Esempio 1.1, non esiste un programma per la generazione della lista dei clienti VIP, mentre ne esiste uno che stampa la lista di tutti i clienti in ordine alfabetico. Sono quindi possibili due alternative:

1. stampare la lista di tutti i clienti ed estrarre manualmente da tale lista quella relativa ai clienti VIP;
2. richiedere che sia sviluppato un nuovo programma che estragga automaticamente la lista dei clienti VIP in ordine alfabetico.

Entrambe le soluzioni hanno delle evidenti controindicazioni. Ad esempio, supponiamo di optare per la seconda soluzione. Supponiamo che dopo qualche tempo sia necessario stampare la lista di tutti i clienti VIP ordinata per numero di punti accumulati. Ovviamente un programma che generi tale lista non è disponibile e pertanto ritorniamo alla situazione precedente. Un DBMS supera queste limitazioni, mettendo a disposizione dei linguaggi (vedi Capitoli 3 e 10) che facilitano l’accesso ai dati secondo modalità non necessariamente note a priori. Tali linguaggi possono poi essere integrati con generici linguaggi di programmazione per lo sviluppo di applicazioni complesse (vedi Capitolo 4) o con delle funzionalità reattive (vedi Capitolo 11) che consentono di eseguire delle operazioni sulla base di dati in modo automatico, al verificarsi di alcuni eventi. Inoltre, il DBMS mette a disposizione delle strutture ausiliarie di accesso (vedi Capitolo 7) che consentono di rispondere più efficientemente alle richieste di utenti ed applicazioni.

**Problemi nell'accesso concorrente ai dati.** Per migliorare le prestazioni, tutti i sistemi permettono di eseguire accessi concorrenti ai dati. L'interazione di modifiche concorrenti, se non opportunamente controllata, può causare inconsistenze dei dati, portando utenti ed applicazioni ad agire su dati non corretti. I sistemi operativi forniscono meccanismi per garantire la mutua esclusione nella modifica dei dati. Questi però non sono sufficienti in un ambiente di basi di dati dove devono essere messi a disposizione meccanismi più sofisticati per garantire la consistenza dei dati. A titolo di esempio, consideriamo ancora il dominio della videoteca e supponiamo che esista un'applicazione che, ad intervalli regolari, conti il numero di noleggi effettuati da ogni cliente della videoteca e generi un report con queste informazioni. Supponiamo che, mentre l'applicazione è in esecuzione, un cliente per cui il conteggio è già stato effettuato noleggi un video. In questo caso, l'esecuzione concorrente delle due applicazioni genera un conteggio non esatto. D'altro canto, non permettere l'esecuzione concorrente di applicazioni degraderebbe in modo non accettabile le prestazioni del sistema. Per risolvere questi problemi, i DBMS mettono a disposizione il concetto di *transazione*, argomento del Capitolo 8. Informalmente, una transazione è una porzione di programma applicativo a cui il DBMS assicura particolari proprietà durante la sua esecuzione che garantiscono, tra le altre cose, la consistenza dei dati in presenza di transazioni concorrenti, senza che il programmatore debba preoccuparsi della gestione di tali problematiche.

**Problemi di protezione dei dati.** Non tutti gli utenti devono poter accedere a tutti i dati presenti nel sistema. Ad esempio, un commesso della videoteca potrebbe essere autorizzato a conoscere la data in cui un cliente ha effettuato un noleggio ma non, per motivi di privacy, il titolo del film che il cliente ha noleggiato. È quindi necessario disporre di meccanismi che consentano un accesso selettivo ai dati: solo a determinati record (selezionati ad esempio in base al valore di alcuni campi) o solo a determinati campi di ogni record. Poiché i file system tipicamente non hanno meccanismi di controllo dell'accesso adeguati a supportare tali requisiti di protezione, questi controlli devono essere implementati a livello di programmi applicativi. In questo caso, però, è difficile verificare che tali controlli siano effettivamente e correttamente incorporati in tutti i programmi applicativi. Vedremo nel Capitolo 9 quali meccanismi offre un DBMS per gestire efficacemente il controllo dell'accesso senza dover ricorrere allo sviluppo di programmi ad-hoc.

**Problemi di integrità dei dati.** Per riflettere correttamente una certa realtà applicativa, è necessario che i dati rispettino determinate condizioni, note come *vincoli di integrità semantica*. Un esempio di vincolo è che la data di restituzione di un video non sia antecedente alla data di noleggio. Un insieme di dati è semanticamente corretto se verifica tutti i vincoli di integrità semantica ad esso associati. Se il sistema di gestione dati non consente la specifica e la verifica automatica di tali vincoli, è necessario implementarli come codice applicativo. L'aggiunta di nuovi vincoli o la modifica di un qualche vincolo rende però necessarie costose modifiche ai programmi applicativi, rendendo di fatto molto difficile assicurare l'integrità se-

mantica dei dati. Vedremo nel Capitolo 3 come i DBMS offrano adeguati strumenti per la specifica di vincoli e la loro verifica automatica.

### 1.3 Obiettivi e servizi di un DBMS

La tecnologia dei DBMS fornisce una soluzione efficace alle problematiche delineate nel paragrafo precedente. Il meccanismo fondamentale che rende possibile questo è una definizione centralizzata ed ad alto livello dei dati detta *schema* (o *schema logico*) della base di dati, condivisa da utenti ed applicazioni, a cui il DBMS assicura requisiti di affidabilità, efficienza, consistenza e protezione. Uno schema logico descrive il contenuto della base di dati tramite un formalismo ad alto livello che esula dai dettagli dell'effettiva implementazione fisica, detto *modello dei dati*. Questo meccanismo, e la definizione di una serie di opportuni linguaggi, rende i dati agevolmente disponibili ad una vasta gamma di utenti ed ambiti applicativi, anche secondo modalità non anticipate al momento della definizione e realizzazione della base di dati. La condivisione dei dati da parte di utenti ed applicazioni evita il problema della ridondanza ed inconsistenza dei dati. Inoltre, poter disporre di una visione centralizzata ed ad alto livello dei dati consente di introdurre un controllo centralizzato sugli stessi, minimizzando quindi i problemi connessi alla protezione e correttezza semantica dei dati.

Gli obiettivi precedentemente discussi sono resi concreti da un insieme di servizi – i principali dei quali sono elencati nella Tabella 1.1 – che un DBMS tipicamente offre e che facilitano l'utilizzo della base di dati e lo sviluppo di applicazioni che ad essa si interfacciano. Alcuni di questi servizi sono direttamente invocabili dagli utenti e dalle applicazioni; altri sono servizi interni al DBMS che permettono di assicurare efficienza e qualità nella gestione dei dati. La realizzazione dei vari servizi implica lo sviluppo di specifici linguaggi, tecniche, strutture dati ed algoritmi che costituiscono nel loro insieme il DBMS inteso come sistema software. Scopo dei capitoli successivi di questo libro è illustrare in dettaglio tali servizi sia per quanto riguarda il loro uso sia, per alcuni di essi, per quanto riguarda gli aspetti realizzativi.

Da un punto di vista architettonale, gli attuali DBMS adottano un'*architettura client-server*, in cui le funzionalità del sistema sono realizzate da due moduli distinti. Il modulo client, che di solito viene eseguito sull'elaboratore dell'utente finale, gestisce principalmente l'interazione tra l'utente ed il DBMS, mettendo a disposizione opportune interfacce per l'utilizzo dei servizi del DBMS e l'accesso ai dati. Il modulo server, invece, si occupa principalmente della memorizzazione e gestione dei dati e contiene un insieme di sotto-moduli che realizzano i servizi elencati nella Tabella 1.1. Ulteriori dettagli sulla struttura interna di un DBMS sono discussi nel Capitolo 7.

Servizio	Descrizione
Descrizione dei dati	Per specificare i dati da memorizzare nella base di dati
Manipolazione dei dati	Per: - accedere ai dati - inserire nuovi dati - modificare dati esistenti - cancellare dati esistenti
Controllo di integrità	Per evitare di memorizzare dati non corretti
Strutture di memorizzazione	Per rappresentare in memoria secondaria i costrutti del modello dei dati
Ottimizzazione di interrogazioni	Per determinare la strategia più efficiente per accedere ai dati
Protezione dei dati	Per proteggere i dati da accessi non autorizzati
Ripristino della base di dati	Per evitare che errori e malfunzionamenti: - determinino una base di dati inconsistente - provochino perdite di dati
Controllo della concorrenza	Per evitare che accessi concorrenti alla base di dati provochino inconsistenze dei dati

Tabella 1.1: Principali servizi offerti da un DBMS

## 1.4 Modelli dei dati

Abbiamo visto nel paragrafo precedente come una delle caratteristiche principali di un DBMS sia il poter disporre di una rappresentazione logica ed ad alto livello dei dati, espressa tramite un opportuno modello dei dati. È a questa rappresentazione logica che utenti ed applicazioni accedono senza dovere, nella maggior parte dei casi, occuparsi di come le strutture dati messe a disposizione dal modello siano effettivamente implementate sui supporti di memorizzazione di un elaboratore. Un modello dei dati deve quindi fornire dei costrutti per rappresentare le “entità” della realtà di interesse mediante un insieme di “concetti” che il DBMS è in grado di interpretare e gestire. Nei paragrafi successivi preciseremo meglio tali concetti ed introdurremo il modello relazionale, il modello dei dati più utilizzato negli attuali DBMS.

### 1.4.1 Concetti di base

Un modello dei dati è un *formalismo*, che consta di tre componenti fondamentali:

- un insieme di strutture dati;
- un linguaggio per specificare le strutture dati previste dal modello, per aggiornare tali strutture e per specificare vincoli su tali strutture;
- un linguaggio per manipolare i dati.

Iniziamo ad occuparci in questo paragrafo del primo aspetto, rimandando gli aspetti legati ai linguaggi al Paragrafo 1.6. Qualsiasi sia il modello dei dati prescelto, esso deve fornire opportune strutture per rappresentare i seguenti concetti:

- **Entità.** Insieme di “oggetti” della realtà applicativa di interesse, aventi caratteristiche comuni.
- **Istanza di entità.** Singolo oggetto della realtà applicativa di interesse, modellato da una certa entità.
- **Attributo.** Proprietà significativa di un’entità, ai fini della descrizione della realtà applicativa di interesse. Ogni entità è caratterizzata da uno o più attributi. Un attributo di un’entità assume uno o più valori per ciascuna delle istanze dell’entità, detti *valori dell’attributo*, in un insieme di possibili valori, detto *dominio dell’attributo*.
- **Associazione.** Corrispondenza tra un certo numero di entità. Anche le associazioni possono avere degli attributi che corrispondono a proprietà dell’associazione.
- **Istanza di associazione.** Corrispondenza tra le istanze di un certo numero di entità.

**Esempio 1.2** Consideriamo la realtà applicativa relativa alla videoteca. Sono esempi di entità: **Cliente**, **Noleggio**, **Video** e **Film**. Ad esempio, l’entità **Cliente** rappresenta l’insieme dei clienti della videoteca. **anna rossi** e **pulp fiction** sono esempi di istanze delle entità **Cliente** e **Film**, rispettivamente. Il nome di un cliente, la data di restituzione del video noleggiato, il regista di un film sono esempi di proprietà che possono essere modellate come attributi delle entità **Cliente**, **Noleggio** e **Film**, rispettivamente. Il legame tra i clienti ed i film che consigliano è un esempio di associazione, che potrebbe avere un attributo **giudizio**, che modella il giudizio espresso dai clienti della videoteca sui film da loro consigliati. Infine, la relazione che lega l’istanza dell’entità **Film** relativa al film “Pulp fiction” di Quentin Tarantino con l’istanza dell’entità **Cliente** relativa al cliente con numero di tessera 6610 è un esempio d’istanza dell’associazione **Consiglia**. □

Qualsiasi modello dei dati deve quindi rispondere a due domande fondamentali: (*i*) come rappresentare le entità ed i loro attributi; (*ii*) come rappresentare le associazioni ed i loro attributi. I modelli dei dati differiscono principalmente per come rappresentano le associazioni; a seconda dello specifico modello dei dati, la rappresentazione può avvenire tramite strutture, valori o puntatori. Per quanto riguarda invece la rappresentazione delle entità, la maggioranza dei modelli usa strutture concettualmente simili ai record, in cui ogni componente rappresenta un attributo.

#### 1.4.2 Modello relazionale

Un esempio molto semplice di modello dei dati è il *modello relazionale* (discusso in dettaglio nel Capitolo 2), attualmente utilizzato dalla maggioranza dei DBMS in commercio. Il modello relazionale è basato su una singola struttura dati: la

**Film**

<b>titolo</b>	<b>regista</b>	<b>anno</b>	<b>genere</b>	<b>valutaz</b>
underground	emir kusturica	1995	drammatico	3.20
edward mani di forbice	tim burton	1990	fantastico	3.60
nightmare before christmas	tim burton	1993	animazione	4.00
ed wood	tim burton	1994	drammatico	4.00
mars attacks	tim burton	1996	fantascienza	3.00
il mistero di sleepy hollow	tim burton	1999	horror	3.50
big fish	tim burton	2003	fantastico	3.10
la sposa cadavere	tim burton	2005	animazione	3.50
la fabbrica di cioccolato	tim burton	2005	fantastico	4.00
io non ho paura	gabriele salvatores	2003	drammatico	3.50
nirvana	gabriele salvatores	1997	fantascienza	3.00
mediterraneo	gabriele salvatores	1991	commedia	3.80
pulp fiction	quentin tarantino	1994	thriller	3.50
le iene	quentin tarantino	1992	thriller	4.00

**Video**

<b>colloc</b>	<b>titolo</b>	<b>regista</b>	<b>tipo</b>
1111	underground	emir kusturica	v
1112	underground	emir kusturica	d
1113	big fish	tim burton	v
1114	big fish	tim burton	d
1115	edward mani di forbice	tim burton	d
1116	nightmare before christmas	tim burton	v
1117	nightmare before christmas	tim burton	d
1118	ed wood	tim burton	d
1119	mars attacks	tim burton	d
1120	il mistero di sleepy hollow	tim burton	d
1121	la sposa cadavere	tim burton	d
1122	la fabbrica di cioccolato	tim burton	d
1123	la fabbrica di cioccolato	tim burton	d
1124	io non ho paura	gabriele salvatores	d
1125	nirvana	gabriele salvatores	d
1126	mediterraneo	gabriele salvatores	d
1127	pulp fiction	quentin tarantino	v
1128	pulp fiction	quentin tarantino	d
1129	le iene	quentin tarantino	d

Figura 1.1: Base di dati relazionale relativa alla videoteca

*relazione*. Una relazione viene spesso rappresentata come una tabella con righe (dette *tuple*) e colonne contenenti dati di tipo specificato, come ad esempio interi e stringhe. Nel seguito useremo i termini relazione e tabella come sinonimi. Una tupla tipicamente rappresenta un’istanza dell’entità modellata dalla tabella a cui la tupla appartiene; una colonna (o attributo) di una tabella rappresenta, invece, un particolare attributo dell’entità modellata dalla tabella stessa.

**Esempio 1.3** La Figura 1.1 mostra come le informazioni relative ai film e ai video di una videoteca possano essere organizzate in due tabelle, di nome **Film** e **Video**, rispettivamente. Ad esempio, per ciascun film a disposizione nella videoteca memorizziamo il titolo, il regista, l’anno in cui è stato girato, il genere e la valutazione della critica. Tali informazioni sono rappresentate da opportuni attributi della tabella **Film**. Ogni attributo corrisponde ad una colonna della tabella, mentre ogni riga rappresenta uno specifico film offerto dalla videoteca.  $\square$

Per quanto riguarda la rappresentazione delle associazioni, il modello relazionale utilizza una rappresentazione cosiddetta *per valore*, in quanto le associazioni non sono rappresentate esplicitamente (mediante costrutti quali puntatori) ma implicitamente, replicando alcune colonne di una tabella in quella con cui vogliamo stabilire un’associazione, oppure creando una tabella ad-hoc per modellare l’associazione contenente alcune colonne delle tabelle che vogliamo mettere in relazione. Una trattazione completa di tali concetti esula dagli scopi di questo capitolo e sarà oggetto del Capitolo 2. Nel seguito introdurremo comunque un semplice esempio per chiarire meglio i concetti discussi.

**Esempio 1.4** Consideriamo ancora la base di dati della Figura 1.1. L’associazione tra un video ed il film che contiene è ottenuta replicando nella tabella **Video** gli attributi **titolo** e **regista** della tabella **Film**. In questo modo, è possibile sapere, dato un video, il genere del film corrispondente o la sua valutazione. Come sarà più chiaro nel Capitolo 2, questo tipo di modellazione presuppone che non esistano nella tabella **Film** due tuple distinte con gli stessi valori per gli attributi **titolo** e **regista**.  $\square$

Attributi con la proprietà di identificare univocamente le tuple di una relazione vengono chiamati *chiavi*, mentre i corrispondenti attributi nell’altra relazione, inseriti per modellare l’associazione tra le tuple delle due relazioni, prendono il nome di *chiavi esterne* (vedi Capitolo 2).

La corrispondenza tra i concetti illustrati nel Paragrafo 1.4.1 ed i costrutti messi a disposizione dal modello relazionale è riassunta nella Tabella 1.2.

#### 1.4.3 Schemi ed istanze

Indipendentemente dal modello dei dati prescelto, in un DBMS distinguiamo solitamente tra la descrizione dei dati ed il contenuto effettivo della base di dati. La descrizione dei dati memorizzati in una base di dati, specificata tramite il modello

Concetto	Costrutto relazionale
Entità	Relazione
Istanza di entità	Tupla
Attributo	Attributo di una relazione
Associazione	Relazione/attributi di una relazione
Istanza di associazione	Tupla Valori uguali per chiavi e chiavi esterne

Tabella 1.2: Corrispondenza tra concetti generali di un modello dei dati e costrutti del modello relazionale

dei dati, è detta *schema della base di dati*, mentre l'insieme dei dati presenti in un dato momento in una base di dati è detto *istanza della base di dati*; tale insieme cambia molto spesso nel tempo dato che nuovi dati sono immessi di frequente, ed i dati presenti possono essere modificati o cancellati. Le modifiche allo schema sono invece molto meno frequenti anche se possibili. Lo schema fornisce pertanto una descrizione *intensionale* del contenuto della base di dati (in un certo qual senso indica il “tipo” dei dati contenuti nella base di dati, a prescindere dal contenuto vero e proprio). Il primo passo nello sviluppo di una base di dati è pertanto rappresentato dalla definizione dello schema della base di dati; successivamente vengono immessi i dati veri e propri che devono conformarsi alla definizione data dallo schema. Come vedremo nel Paragrafo 1.6, questa distinzione si riflette anche nei linguaggi messi a disposizione da un DBMS.

In una base di dati relazionale, lo schema della base di dati è costituito dall'insieme degli schemi delle relazioni in essa presenti. Lo schema di una relazione, a sua volta, è costituito dal nome della relazione e dal nome e dai domini dei suoi attributi. Con riferimento alla Figura 1.1, lo schema della base di dati è costituito dal nome delle due tabelle e dai nomi e domini dei loro attributi, mentre l'istanza è costituita dall'insieme di tuple contenute nelle due tabelle.

## 1.5 Livelli nella rappresentazione dei dati

Come è già stato evidenziato, uno degli scopi di un DBMS è fornire una rappresentazione ad alto livello di un insieme di dati, nascondendo i dettagli che riguardano la loro effettiva memorizzazione. Inoltre, una base di dati è spesso usata da gruppi di applicazioni e/o utenti per cui non tutti i dati presenti nella base di dati sono di interesse; a tali applicazioni e/o utenti viene spesso messa a disposizione una visione, detta *vista*, di una parte dello schema della base di dati. Una base di dati può pertanto essere osservata a tre diversi livelli di astrazione (vedi Figura 1.2):

- **Livello fisico.** È il livello più basso in cui viene definito lo *schema fisico* della base di dati, precisando *come* i dati sono effettivamente memorizzati tramite strutture di memorizzazione (file, record, ecc.).

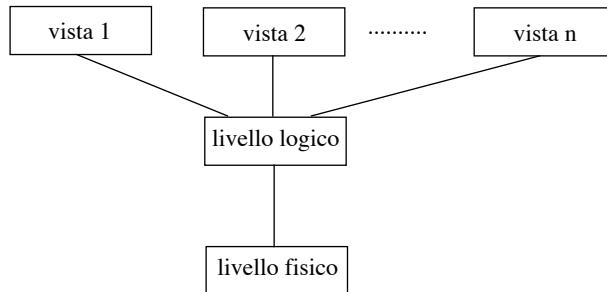


Figura 1.2: Livelli nella rappresentazione dei dati

- **Livello logico.** È il secondo livello di astrazione in cui viene descritto lo *schema logico*, cioè *quali* sono i dati memorizzati nella base di dati, eventuali associazioni tra di essi e vincoli di integrità semantica e di autorizzazione. L'intera base di dati è descritta tramite un numero limitato di strutture dati, relativamente semplici, che costituiscono il modello dei dati (relazioni nel caso del modello relazionale).
- **Livello esterno o livello delle viste.** È il livello di astrazione più alto; descrive una porzione dell'intero schema della base di dati; possono essere definite più viste di una stessa base di dati.

I primi due livelli sono sempre presenti, mentre il terzo è principalmente utilizzato in basi di dati il cui schema è di dimensione medio-grandi.

L'introduzione di questi tre livelli assicura alcune importanti proprietà ai dati, conosciute con il nome di *indipendenza fisica* e *logica*, che facilitano l'accesso ai dati e lo sviluppo di applicazioni. Indipendenza fisica significa che utenti ed applicazioni che accedono alla rappresentazione logica dei dati sono indipendenti da qualsiasi modifica a livello di rappresentazione fisica in quanto modifiche a tale livello non influenzano la rappresentazione dei dati al livello logico. Ad esempio, nel caso del modello relazionale, utenti ed applicazioni agiscono su tabelle. Tradurre ogni operazione effettuata sulle tabelle (ad esempio, la creazione di una tabella) nelle corrispondenti operazioni a livello fisico (ad esempio su file) è compito del DBMS e può avvenire in modo del tutto trasparente rispetto ad utenti ed applicazioni. Analogamente, la presenza delle viste permette di ottenere un certo grado di indipendenza logica, cioè di nascondere (entro certi limiti) modifiche alla rappresentazione dei dati al livello logico alle applicazioni/utenti che accedono alla rappresentazione esterna (tramite vista) dei dati. Ad esempio, nel contesto relazionale la modifica di una tabella non influenza su utenti ed applicazioni che operano su una vista che non contiene tale tabella.

## 1.6 Linguaggi di un DBMS

Oltre a fornire un modello dei dati, un DBMS mette a disposizione un insieme di linguaggi che permettono agli utenti di interagire con il DBMS per descrivere e manipolare i dati di interesse e specificare vincoli. Tra questi linguaggi i più rilevanti sono:

- **Linguaggio di definizione dei dati (DDL – Data Definition Language).** Il DDL consente di specificare ed aggiornare lo schema di una base di dati. In particolare, il DDL concretizza il modello dei dati fornendo la notazione che permette di specificarne le strutture dati. Esso deve supportare la specifica del nome della base di dati, come pure di tutte le unità logiche elementari della base di dati (ad esempio tabelle e colonne nel caso del modello relazionale) e di eventuali vincoli di integrità semantica e di autorizzazione. Deve inoltre prevedere comandi per l'aggiornamento delle strutture dati previste dal modello (ad esempio aggiunta/eliminazione di una nuova colonna in una tabella o modifica del dominio di un attributo nel caso del modello relazionale).
- **Linguaggio di manipolazione dei dati (DML – Data Manipulation Language).** Una base di dati, organizzata logicamente tramite il modello dei dati e definita tramite il DDL, è accessibile agli utenti ed alle applicazioni tramite il DML. Le operazioni fornite da questo linguaggio servono quindi per gestire le istanze della base di dati e sono fondamentalmente quattro:
  - inserimento, per l'immissione di nuovi dati;
  - ricerca, per il ritrovamento dei dati di interesse; un'operazione di ricerca è spesso detta *interrogazione* (o *query*) da cui il nome di *linguaggio di interrogazione* (o *query language*) per quella componente del DML che permette di esprimere le operazioni di ricerca;
  - cancellazione, per l'eliminazione di dati obsoleti;
  - modifica, per variare dati esistenti.
- **Linguaggio di definizione delle strutture di memorizzazione (SDL – Storage Definition Language).** La corrispondenza fra le strutture logiche, specificate nello schema della base di dati, e le strutture di memorizzazione deve essere opportunamente definita. Nella maggior parte dei DBMS attuali la definizione di tale corrispondenza è eseguita automaticamente dal DBMS stesso una volta che lo schema logico è definito. Tuttavia, l'utente esperto può influenzare le scelte operate dal DBMS richiedendo ad esempio l'allocazione di strutture ausiliarie di accesso (vedi Capitolo 7) per velocizzare determinati accessi ai dati. Tali richieste vengono effettuate tramite i comandi del linguaggio di definizione delle strutture di memorizzazione.

Una distinzione fondamentale riguardante i linguaggi di manipolazione dei dati è tra *linguaggi procedurali*, detti anche *operazionali*, e *linguaggi non procedurali*,

detti anche *dichiarativi*. Per grado di proceduralità intendiamo il grado di dettaglio con cui è necessario specificare i vari passi che il DBMS deve eseguire per effettuare le operazioni richieste. Nei linguaggi procedurali è responsabilità del programmatore specificare sia il risultato che vuole ottenere sia come ottenerlo. In sostanza, una volta effettuato l'accesso ad un'istanza di un'entità (ad esempio un record) della base di dati, il programmatore deve specificare a quale altra istanza accedere per effettuare l'operazione richiesta. I DBMS di prima generazione (vedi Paragrafo 1.7), come ad esempio i DBMS basati sul modello Codasyl, erano caratterizzati da linguaggi procedurali. Nei linguaggi non procedurali, invece, dobbiamo solo specificare quali caratteristiche devono avere i dati su cui vogliamo operare, tipicamente indicando condizioni sugli attributi, lasciando al DBMS la responsabilità di decidere come accedere effettivamente a tali dati. Ad esempio, se diciamo che siamo interessati a conoscere il nome del film contenuto nel video con codice di collocazione 1116, stiamo utilizzando un approccio dichiarativo in quanto non esplicitiamo come effettivamente reperire tale informazione nella base di dati. Il linguaggio SQL (vedi Capitolo 3), che è il linguaggio standard per la definizione e manipolazione di dati relazionali, è un esempio di linguaggio dichiarativo ed a questa caratteristica deve molta della sua fortuna. In effetti, l'evoluzione dei linguaggi per la manipolazione dei dati è stata caratterizzata dall'affermarsi di linguaggi dichiarativi. Il vantaggio di tali linguaggi è duplice: da una parte sono di facile utilizzo, anche per utenti poco esperti di informatica, in quanto con tali linguaggi è solo necessario dichiarare le caratteristiche che il risultato deve possedere senza dover specificare anche il procedimento operativo per ottenere tale risultato. D'altro canto, non specificare il modo operativo con cui ottenere il risultato consente al DBMS di applicare tutta una serie di strategie (alcune delle quali trattate nel Capitolo 7) per eseguire in modo ottimizzato l'operazione limitando il numero di accessi a disco necessari.

### 1.7 Evoluzione dei modelli dei dati

L'evoluzione dei DBMS è stata guidata dall'evoluzione dei modelli dei dati. Lo sviluppo di diversi modelli dei dati caratterizza infatti ogni nuova generazione di DBMS. La tendenza nello sviluppo di nuovi modelli dei dati è sempre stata di aumentare il potere espressivo del modello dei dati, rendendolo in grado di rappresentare sempre meglio ed in modo diretto la realtà di interesse alle più svariate applicazioni e, nel contempo, di rendere la rappresentazione dei dati attraverso il modello il più indipendente possibile da aspetti implementativi.

I primi DBMS, sviluppati negli anni '60, erano caratterizzati dal *modello gerarchico*, mentre successivamente sono comparsi DBMS caratterizzati dal *modello reticolare*, standardizzato dalla Conferenza sui Linguaggi per Sistemi di Dati (Codasyl). La generazione successiva, sviluppata durante gli anni '70, viene segnata dall'avvento dei DBMS basati sul modello relazionale, conosciuti come *RDBMS – Relational Database Management System*. La semplicità delle strutture di rappresentazione dati adottate dal modello relazionale ha nettamente differenziato questo

modello rispetto ai precedenti, permettendo lo sviluppo di linguaggi di definizione e manipolazione dei dati semplici da utilizzare. Il modello relazionale costituisce la base su cui è stato sviluppato il linguaggio standard SQL, oggi utilizzato nella maggior parte delle applicazioni di gestione dati. Inoltre, il modello relazionale, a differenza dei precedenti, è caratterizzato da una base matematica che ha dato notevoli impulsi a ricerche, anche teoriche, in ambito basi di dati. Oggi la tecnologia relazionale è alla base dei principali DBMS presenti sul mercato (quali Oracle, leader mondiale del mercato dei DBMS, IBM DB2, Microsoft SQL Server, Informix – recentemente acquisito da IBM, Sybase SQL Server, ed i DBMS open source MySQL e PostgreSQL).

Gli inizi degli anni ‘80 hanno visto inoltre l'affacciarsi di nuove applicazioni, rese possibili dalle innovazioni hardware, quali ad esempio le applicazioni CAD (*Computer-Aided Design*), CAM (*Computer-Aided Manufacturing*), CASE (*Computer-Aided Software Engineering*), GIS (*Geographical Information System*) ed applicazioni multimediali. Tali applicazioni sono caratterizzate dall'esigenza di rappresentare direttamente oggetti applicativi con strutture complesse. Una risposta a tali esigenze è costituita dai *DBMS orientati ad oggetti* (*OODBMS – Object-Oriented Database Management System*) che, come indica il nome, integrano la tecnologia dei DBMS con il paradigma di orientamento ad oggetti sviluppato nell'area dei linguaggi di programmazione e delle metodologie di progettazione del software. Questa linea di tendenza è stata caratterizzata da sviluppi industriali ed applicazioni significative, oltreché da uno sforzo di standardizzazione.

L'intensa attività di ricerca e sviluppo degli anni ‘80 ha evidenziato che, se da un lato determinate funzionalità come ad esempio la possibilità di modellare oggetti complessi e gerarchie di tipi, sono essenziali per un modello dei dati, dall'altro la compatibilità con il modello relazionale ed in particolare l'uso del linguaggio SQL sono requisiti altrettanto cruciali. Gli inizi degli anni ‘90 hanno visto pertanto la confluenza della tecnologia dei DBMS relazionali con la tecnologia dei DBMS orientati ad oggetti, il cui risultato sono i *DBMS relazionali ad oggetti* (*ORDBMS – Object-Relational Database Management System*), argomento del Capitolo 10. Questo tipo di DBMS, pur avendo un modello dei dati estremamente flessibile e ricco (che incorpora di fatto molti dei costrutti del modello ad oggetti), mantiene la stessa filosofia del modello relazionale e del linguaggio SQL con, ovviamente, le opportune estensioni.

Un'altra evoluzione particolarmente rilevante in ambito basi di dati è stata quella di dotare i DBMS della possibilità di definire *trigger*, cioè regole attive che effettuano azioni nel sistema automaticamente al verificarsi di determinati eventi (ad esempio inserimento di dati con particolari valori). I DBMS che forniscono tali funzionalità sono chiamati *DBMS attivi* e verranno trattati nel Capitolo 11.

Tra le varie tendenze evolutive ricordiamo inoltre i *sistemi di gestione dati duttivi* ed i *sistemi di gestione dati “intelligenti”*. I primi integrano la tecnologia delle basi di dati con la programmazione logica. La principale caratteristica di tali sistemi è di fornire meccanismi di inferenza, basati su regole, che permettono di derivare informazioni aggiuntive dai dati memorizzati nella base di dati. Tali siste-

```

<elencoFilm>
  <film>
    <titolo>underground</titolo>
    <regista>emir kusturica</regista>
    <anno>1995</anno>
    <genere>drammatico</genere>
    <valutaz>3.20</valutaz>
  </film>
  <film>
    <titolo>edward mani di forbice</titolo>
    <regista>tim burton</regista>
    <anno>1990</anno>
    <genere>fantastico</genere>
    <valutaz>3.60</valutaz>
  </film>
  <film>
    <titolo>nightmare before christmas</titolo>
    <regista>tim burton</regista>
    <anno>1993</anno>
    <genere>animazione</genere>
    <valutaz>4.00</valutaz>
  </film>
  ...
</elencoFilm>

```

Figura 1.3: Documento XML corrispondente alla tabella Film della Figura 1.1

mi sono caratterizzati da fondamenti teorici abbastanza consolidati. Gli sviluppi industriali e le applicazioni sono stati però molto limitati e ristretti ad applicazioni di tipo scientifico. I secondi, invece, integrano la tecnologia delle basi di dati con vari paradigmi e tecniche sviluppate nell'area dell'intelligenza artificiale. Esempi tipici sono i sistemi basati sui modelli di rappresentazione della conoscenza.

Infine, l'inizio del nuovo millennio ha definitivamente visto l'affermarsi di Internet e del Web come strumenti principali di comunicazione e condivisione dell'informazione. In tale ambito, un ruolo di primaria importanza è giocato dal linguaggio *XML* (*eXtensible Markup Language*), un linguaggio sviluppato dal *World Wide Web Consortium – W3C*, che ormai costituisce uno standard per la rappresentazione e lo scambio di informazioni su Web. XML è un meta-linguaggio per la definizione di linguaggi di markup atti a modellare dati strutturati di differenti domini applicativi. XML fornisce un modo per definire tag e relazioni strutturali tra di essi, senza porre alcuna restrizione sui tag che possono essere definiti. Questa estensibilità consente di personalizzare l'insieme di tag in base al dominio applicativo ed alla semantica dei dati che vogliamo modellare. A titolo di esempio, la Figura 1.3 riporta un documento XML che modella le informazioni contenute nella relazione *Film* della Figura 1.1. Per ragioni di spazio, sono riportate solo le prime tre tuple contenute nella relazione. Come vediamo dalla Figura 1.3, un documento XML ha una strutturazione gerarchica ed i tag sono utilizzati per

rappresentare la semantica dei dati (nel nostro caso entità ed attributi).

XML, come è ovvio che sia, ha notevolmente influenzato gli ultimi sviluppi in ambito DBMS; gli approcci proposti per la gestione di dati XML possono essere classificati in due categorie principali: lo sviluppo di DBMS che hanno XML come modello nativo (*XML-Native DBMS*), come ad esempio il DBMS Tamino; estensioni ai DBMS relazionali o relazionali ad oggetti con funzionalità per la gestione di dati XML (*XML-Enabled DBMS*), quali ad esempio Oracle 10g. In tale contesto, una parte del nuovo standard SQL, SQL/XML, è dedicata alla gestione di dati XML nel contesto di un DBMS relazionale.

## 1.8 Utenti di un DBMS

L'uso di un DBMS nella gestione dati introduce diverse professionalità che, a vario titolo, si occupano di compiti connessi alla gestione della base di dati e del DBMS. Tra le principali professionalità, identifichiamo le seguenti figure:

- **Amministratore della base di dati (DBA – Database Administrator).** L'introduzione di questa tipologia professionale è conseguenza del fatto che oggi i DBMS sono strumenti sempre più complessi ed utilizzati da una vasta categoria di utenti. I principali compiti del DBA sono l'amministrazione ed il controllo della base di dati. In particolare, il DBA deve stabilire, in accordo alle politiche dell'organizzazione, le regole per l'utilizzo dei dati e le procedure che assicurino la protezione (vedi Capitolo 9) e l'integrità dei dati. Deve inoltre occuparsi delle opportune procedure di ripristino (vedi Capitolo 8) della base di dati, migliorare, ove possibile, l'efficienza (vedi Capitolo 7) del sistema e mantenere i contatti con gli utenti per determinare eventuali nuove esigenze.
- **Progettista di basi di dati.** Tale figura professionale deve provvedere, in base ai requisiti dell'organizzazione e delle varie applicazioni, alla progettazione della base di dati (molto spesso questa attività è affidata a consulenti specializzati). In particolare, è compito del progettista definire lo schema logico e fisico della base di dati, ed eventuali viste sugli stessi, in base ad un'attenta analisi dei requisiti informativi del dominio di interesse. Alla progettazione dello schema logico di una base di dati sono dedicati i Capitoli 5 e 6, mentre alcuni dettagli sulla progettazione fisica verranno forniti nel Capitolo 7.
- **Programmatore applicativo.** Il programmatore applicativo è preposto allo sviluppo dei programmi applicativi che operano sulla base di dati ed effettua il mantenimento di tali programmi. È da notare che, a seconda delle dimensioni dei programmi applicativi, di solito sono presenti più programmatore applicativi e che inoltre i programmatore che hanno inizialmente sviluppato i programmi non sono necessariamente gli stessi che eseguono la manutenzione. Allo sviluppo di applicazioni che si interfacciano ad una base di dati ( dette anche applicazioni per basi di dati) è dedicato il Capitolo 4.

- **Progettista, sviluppatore di DBMS.** Un DBMS, essendo un sistema software, necessita di essere progettato, sviluppato e mantenuto così come una qualsiasi applicazione software. Naturalmente, la complessità del sistema necessita di adeguate professionalità per svolgere tale compito.

Per quanto riguarda i semplici utenti possiamo classificarli in due categorie: *utenti parametrici* ed *utenti occasionali*. I primi sono utenti che usano sistematicamente la base di dati, come parte integrante delle loro mansioni; l'uso che tali utenti fanno dei dati può essere anche complesso ma è fondamentalmente noto a priori. È quindi possibile sviluppare appositi programmi applicativi che supportino le funzioni necessarie a tali utenti. I secondi sono utenti le cui richieste non sono predicibili a priori e che tuttavia nella maggior parte dei casi fanno un uso molto semplice della base di dati. Per tali utenti sono spesso predisposte interfacce interattive molto semplici da utilizzare.

### Note bibliografiche

Dato che una buona conoscenza dei DBMS è ormai indispensabile per qualsiasi professionista dell'informatica, esistono numerosi testi che trattano l'argomento. Tra i testi classicamente usati a livello universitario se ne possono citare molti. In particolare, il testo di Silberschatz et al. [SKS01] ed il testo di Elmasri e Navathe [EN03], recentemente tradotto in italiano [EN04, EN05], trattano l'argomento in modo approfondito, dando molto spazio agli aspetti più tecnologici e non eccedendo in quelli teorici. Un altro testo molto esauriente è quello di Ramakrishnan e Gehrke [RG02], di cui è disponibile anche la versione italiana [RG04]. Molto completo, anche relativamente agli aspetti architetturali, è il testo di Garcia-Molina, Ullman e Widom [GMUW02]. Per una trattazione più teorica segnaliamo il testo di Ullman [Ull90], che dà ampio spazio all'uso della logica nelle basi di dati ed alle tecniche di ottimizzazione per linguaggi logici per basi di dati ma nel contempo copre approfonditamente tutti gli argomenti classicamente trattati nei corsi di basi di dati, ed il testo di Abiteboul, Hull e Vianu [AHV95], che presenta una trattazione completa degli aspetti di base, trattando inoltre questioni di espressività e complessità relative ai linguaggi per basi di dati. Sono inoltre da segnalare i testi di Atzeni et al. [ACPT02, ACF<sup>+</sup>03] che trattano in modo esauriente gli argomenti di base, offrendo inoltre una panoramica su argomenti avanzati quali l'accesso a basi di dati da Web e le basi di dati distribuite. Per una presentazione concisa ed essenziale, benché rigorosa, dei concetti fondamentali relativi ai sistemi di gestione dati relazionali suggeriamo invece il testo di Bressan e Catania [BC05]. Infine, per dettagli relativi al linguaggio XML rimandiamo il lettore a [XML].

## Capitolo 7

# Memorizzazione dei dati ed elaborazione delle interrogazioni

Finora abbiamo considerato modelli dei dati ad alto livello, cioè a livello *logico*. Tale livello è quello corretto per gli utenti della base di dati. Tuttavia, un fattore importante nell'accettazione di un DBMS da parte dell'utente è dato dalle sue prestazioni. Le prestazioni del DBMS dipendono dall'efficienza delle strutture dati e dall'efficienza del sistema nell'operare su tali strutture. Nei capitoli precedenti abbiamo introdotto le nozioni di tupla e relazione e i linguaggi per la loro manipolazione. A livello fisico, tali tuple saranno memorizzate in record di file su memoria secondaria ed una manipolazione efficiente verrà garantita dall'uso di opportune tecniche di elaborazione delle interrogazioni. In particolare, per velocizzare la ricerca dei dati vengono in genere utilizzate particolari strutture di accesso, dette *indici*, che consentono di accedere direttamente ai record corrispondenti alle tuple con un certo valore per un attributo, senza scandire l'intero contenuto del file. La scelta delle strutture di memorizzazione e di indicizzazione più efficienti dipende dal tipo di accessi che si eseguono sui dati. Normalmente, ogni DBMS ha le proprie strategie di implementazione di un modello dei dati; tuttavia, l'utente può influenzare le scelte fatte dal sistema. Le scelte dell'utente a questo riguardo costituiscono la *progettazione fisica* della base di dati e si concretizzano mediante l'utilizzo di opportuni comandi forniti dai DBMS. Una volta determinate le strutture di memorizzazione dei dati ed eventuali strutture ausiliarie di accesso, è compito del DBMS determinare, per ogni operazione di manipolazione dei dati, la strategia più efficiente per eseguirla, date le strutture disponibili. Benché tale processo riguardi tutte le operazioni di manipolazione dei dati, gran parte del processo di ottimizzazione è relativo alle operazioni di interrogazione, poiché il costo delle operazioni è dominato dal costo per il ritrovamento delle tuple, che, essendo specificato mediante condizioni dichiarative, offre notevoli margini di ottimizzazione. Per tale motivo ci concentreremo principalmente sull'elaborazione e l'ottimizzazione delle interrogazioni.

In questo capitolo, dopo aver brevemente introdotto l'architettura generale di un sistema di gestione dati, discuteremo le tecniche utilizzate per la memorizzazione fisica e l'indicizzazione dei dati. Illustreremo poi le principali problematiche nell'elaborazione di interrogazioni e descriveremo brevemente l'attività di progettazione fisica.

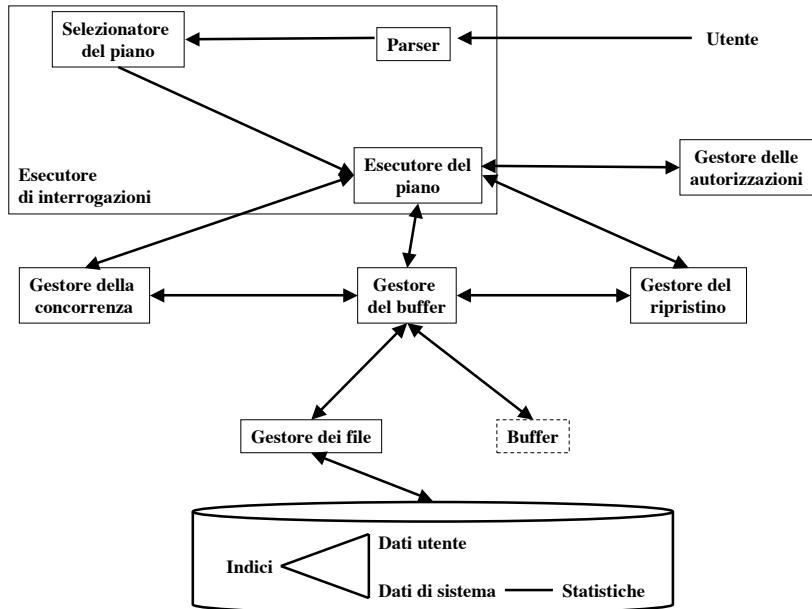


Figura 7.1: Architettura di un DBMS

### 7.1 Architettura di un DBMS

Le basi di dati memorizzano in modo persistente grosse quantità di dati. La maggior parte delle basi di dati sono memorizzate in maniera permanente su supporti di memorizzazione secondaria (cui ci riferiremo anche come memoria di massa), tipicamente su dischi magnetici. I dati in memoria secondaria non possono essere elaborati direttamente dalla CPU, ma devono essere prima copiati in memoria principale, in un'opportuna area di *buffer*.

Un DBMS deve garantire una gestione efficiente, concorrente, affidabile e sicura dei dati, preservandone inoltre l'integrità. Ciascuno degli aspetti precedenti è supportato nel DBMS da specifiche componenti, o sottosistemi, che complessivamente rappresentano l'architettura del sistema, illustrata graficamente nella Figura 7.1. Nella Figura 7.1, le componenti funzionali sono indicate da rettangoli, mentre il disco (memoria secondaria) è indicato con un cilindro e le aree in memoria principale con un rettangolo tratteggiato. Un DBMS è quindi costituito da diverse componenti funzionali che includono:

- **gestore dei file**, che gestisce l'allocazione dello spazio su disco e le strutture dati usate per rappresentare le informazioni memorizzate su disco;

- **gestore del buffer**, responsabile del trasferimento delle informazioni tra disco e memoria principale;
- **esecutore di interrogazioni**, responsabile dell'esecuzione delle richieste utente e costituito da:
  - **parser**, che traduce i comandi del DDL e del DML in un formato interno (*parse tree*);
  - **selezionatore del piano**, che stabilisce il modo più efficiente di processare una richiesta utente;
  - **esecutore del piano**, che processa le richieste utente in accordo al piano di esecuzione selezionato;
- **gestore delle autorizzazioni**, che controlla che gli utenti abbiano gli opportuni diritti di accesso ai dati;
- **gestore del ripristino**, che assicura che la base di dati rimanga in uno stato consistente a fronte di cadute o malfunzionamenti del sistema;
- **gestore della concorrenza**, che assicura che le esecuzioni concorrenti di processi procedano senza conflitti.

Come evidenziato nella Figura 7.1, il DBMS memorizza, oltre ai dati utente, anche dati di sistema (quali informazioni sullo schema della base di dati e sulle autorizzazioni), strutture ausiliarie di accesso a tali dati (indici) e dati statistici (quali il numero di tuple in una relazione), utilizzati dal selezionatore del piano per determinare la migliore strategia di esecuzione. Tutte le diverse componenti accedono a dati utente e/o di sistema nello svolgimento delle loro funzioni (ad esempio, il selezionatore del piano accede ai dati statistici ed il gestore delle autorizzazioni accede alle autorizzazioni), interagendo quindi con il gestore del buffer, anche se, per semplicità, nella Figura 7.1 alcune frecce sono state omesse.

In questo capitolo, introdurremo innanzitutto brevemente la gestione dei dati di sistema. Ci concentreremo poi sulle problematiche inerenti alla memorizzazione dei dati e l'ottimizzazione di interrogazioni, che concorrono alla realizzazione di una gestione efficiente dei dati, introducendo, quindi, il gestore dei file, il gestore del buffer e l'esecutore di interrogazioni. Il gestore delle autorizzazioni, responsabile dell'integrità e della riservatezza dei dati, ha come componente principale un meccanismo di controllo dell'accesso quali quelli che verranno discussi nel Capitolo 9. I gestori di concorrenza e ripristino, garanti, rispettivamente, dell'accesso concorrente ai dati e della loro affidabilità, saranno discussi nel Capitolo 8.

**Cataloghi di sistema.** Un DBMS descrive i dati che gestisce, incluse le informazioni sullo schema della base di dati e gli indici, tramite *meta-dati*. Tali meta-dati sono memorizzati in relazioni speciali, dette *cataloghi* di sistema, e sono utilizzati dalle diverse componenti del DBMS. In tali cataloghi vengono innanzitutto memorizzate informazioni di schema. Ad esempio, per ogni relazione vengono

nomeA	nomeR	pos	tipo
nomeA	Attributi	1	VARCHAR(20)
nomeR	Attributi	2	VARCHAR(20)
pos	Attributi	3	INTEGER
tipo	Attributi	4	VARCHAR(10)
titolo	Film	1	VARCHAR(30)
regista	Film	2	VARCHAR(20)
anno	Film	3	DECIMAL(4)
genere	Film	4	CHAR(15)
valutaz	Film	5	NUMERIC(3,2)
codCli	Cliente	1	DECIMAL(4)
...	...	...	...
dataRest	Noleggio	4	DATE

Figura 7.2: Catalogo relativo allo schema della videoteca

mantenuti il nome della relazione, il nome e il tipo di ciascuno degli attributi, il nome di ciascun indice definito sulla relazione, gli eventuali vincoli definiti sulla relazione. Analogamente, per ogni indice vengono memorizzati il nome, il tipo, gli attributi su cui è definito; per ogni vista vengono memorizzati il nome e la definizione. Nei cataloghi vengono inoltre memorizzate statistiche sulle relazioni e sugli indici (vedi Paragrafo 7.4.2), oltre ad informazioni sugli utenti del DBMS e sui loro diritti di accesso (vedi Capitolo 9).

Un aspetto elegante di un DBMS relazionale è che i cataloghi di sistema sono essi stessi relazioni. Ad esempio, le informazioni sulle relazioni e sulle viste saranno contenute in un catalogo avente come attributi: il nome della relazione, il nome dell'utente creatore, il tipo (relazione/vista) ed il numero di attributi. In un altro catalogo verranno ad esempio memorizzate le informazioni sugli attributi delle relazioni e delle viste. Tale relazione conterrà una tupla per ogni attributo di ogni relazione o vista, con attributi: il nome dell'attributo, il nome della relazione o vista a cui l'attributo appartiene, la posizione ordinale dell'attributo tra gli attributi della stessa relazione o vista, il tipo dell'attributo. Un esempio di tale relazione, che supponiamo essere di schema **Attributi**(*nomeA, nomeR, pos, tipo*), relativamente alla base di dati della videoteca, è presentato nella Figura 7.2. Notiamo come la relazione contenga anche le informazioni relative agli attributi della relazione stessa.

Il fatto che i cataloghi di sistema siano relazioni permette di interrogarli come tutte le altre relazioni e di applicare ad essi tutte le tecniche per l'implementazione e la gestione di relazioni. I DBMS esistenti utilizzano schemi di catalogo differenti, ma le informazioni contenute sono pressapoco le stesse.

## 7.2 Memorizzazione dei dati e gestione del buffer

In questo paragrafo introdurremo le nozioni alla base della memorizzazione fisica dei dati su disco e discuteremo due aspetti cruciali nel minimizzare il costo di

esecuzione dei comandi di manipolazione dei dati: l'organizzazione in cluster e la gestione del buffer.

### 7.2.1 File, record e blocchi

I dati sono trasferiti tra il disco e la memoria principale in unità chiamate *blocchi* (*o pagine*); un blocco è una sequenza di byte contigui su disco. La dimensione del blocco dipende dal sistema operativo e varia tipicamente tra 4 e 16 KB. Poiché il costo di I/O di un blocco domina il costo delle operazioni tipiche delle basi di dati, le strategie di memorizzazione dei dati e di ottimizzazione delle interrogazioni hanno come scopo principale la minimizzazione dei blocchi trasferiti. Evidenziamo inoltre che trasferire blocchi contigui ha un costo decisamente inferiore che trasferire gli stessi blocchi in ordine casuale.

I dati sono generalmente memorizzati in forma di *record*. Ogni record è costituito da un insieme di valori collegati, dove ogni valore è formato da uno o più byte e corrisponde ad un particolare *campo* del record. I record corrispondono in generale a tuple delle relazioni ed i campi del record ai loro attributi. Una collezione di nomi di campi a cui sono associati i tipi corrispondenti costituisce un *tipo di record*. Il tipo associato ad un campo specifica quali valori il campo può assumere; il tipo di un campo è generalmente uno dei tipi predefiniti. Il numero di byte necessari per la memorizzazione di un valore di un certo tipo è fissato per ogni sistema.

**Esempio 7.1** Consideriamo ad esempio un sistema in cui un numero intero può essere memorizzato in 4 byte, un numero reale in 4, una data in 4, una stringa di lunghezza  $k$  in  $k$ . Una tupla della relazione *Film* della Figura 2.1 potrebbe corrispondere ad un record di tipo `struct Film {char titolo[30]; char regista[20]; int anno; char genere[15]; real valutaz}` per la cui memorizzazione sarebbero quindi necessari 73 byte.  $\square$

Ciascun record di un file ha un identificatore unico chiamato *record id* o *RID*, tramite cui è possibile identificare l'indirizzo su disco del blocco che contiene il record. Generalmente il RID è costituito da un identificatore di blocco associato all'identificatore del record all'interno del blocco.

Un *file* è una sequenza di record. In molti casi, tutti i record memorizzati in un file sono dello stesso tipo. Se tutti i record memorizzati in un file hanno la stessa dimensione (in byte), parliamo di *file con record a lunghezza fissa*. Se, al contrario, nel file sono memorizzati record di dimensioni diverse, abbiamo un *file con record a lunghezza variabile*. Un file può contenere record a lunghezza variabile per varie ragioni. Il file può innanzitutto contenere record di tipi differenti e quindi di dimensioni differenti. Questo può succedere se record contenenti informazioni collegate ma di tipi differenti sono memorizzati in posizione contigua sullo stesso blocco di disco. Ad esempio, ciò accade se vogliamo memorizzare le informazioni relative ai noleggi effettuati da un cliente vicino a quelle del cliente (vedi Paragrafo 7.2.2). Il file può inoltre contenere record tutti dello stesso tipo, ma uno o più campi

di tali record possono avere dimensione variabile, essere opzionali o, nel caso di modelli dei dati quale quello relazionale ad oggetti (vedi Capitolo 10), possono assumere più di un valore.

In un file con record a lunghezza fissa, ogni record ha gli stessi campi, e la lunghezza dei campi è fissa, quindi si può identificare la posizione di partenza di ogni campo rispetto alla posizione di partenza del record. Questo facilita l'accesso ai campi del record. Viceversa, nel caso di file con record a lunghezza variabile, sono possibili diverse rappresentazioni: una prima possibilità è l'aggiunta di un simbolo speciale *end-of-record* che indica la fine di ogni record. Alternativamente, si può rappresentare un file con record a lunghezza variabile mediante file con record a lunghezza fissa. La scelta del tipo di rappresentazione dipende dalle caratteristiche dei record contenuti nel file.

A seconda dell'organizzazione primaria dei dati scelta (vedi Paragrafo 7.3), il file che contiene i record dei dati può essere:

- un *file heap* (o file seriale), in cui i record dei dati vengono memorizzati uno dopo l'altro in ordine di inserimento;
- un *file ordinato* (o file sequenziale), in cui i record dei dati sono memorizzati mantenendo l'ordinamento su uno o più campi, in questo caso al file è associato un indice ad albero (vedi Paragrafo 7.3.2) su tali campi;
- un *file hash*, in cui i record dei dati sono memorizzati in una posizione nel file che dipende dal valore ottenuto dall'applicazione di una funzione hash ad uno o più campi del record (vedi Paragrafo 7.3.3);
- un file indice ad albero *integrato*, in cui i record dati sono memorizzati all'interno delle entrate di un indice ad albero (vedi Paragrafo 7.3.2) costruito su uno o più campi di tali record.

Un file può essere visto come una collezione di record. Tuttavia, poiché i dati sono trasferiti in blocchi da memoria secondaria a memoria principale, è importante assegnare i record ai blocchi in modo tale che uno stesso blocco contenga record tra loro correlati. Memorizzando sullo stesso blocco record che sono spesso richiesti insieme si risparmiano infatti accessi a disco. Questo motiva la definizione delle tecniche di organizzazione in cluster discusse nel paragrafo seguente.

### 7.2.2 Organizzazione in cluster

Una strategia di memorizzazione efficiente per alcune interrogazioni è basata sull'organizzazione in *cluster* (*clustering*, o *raggruppamento*) delle tuple che hanno lo stesso valore di uno o più attributi, che prendono il nome di *chiave*<sup>1</sup> del cluster. Clusterizzare una relazione sul valore di uno o più attributi significa organizzare la memorizzazione fisica delle tuple di tale relazione in base al valore di tali attributi,

---

<sup>1</sup>Il termine chiave utilizzato in questo capitolo è diverso da quello introdotto nel Capitolo 2, cui faremo riferimento in questo capitolo come *chiave primaria*.

6610	anna	rossi	01055664433	05-Ott-1979	via scribanti 16	16131 genova
1126	15-Mar-2006	6610	16-Mar-2006			
1112	16-Mar-2006	6610	18-Mar-2006			
1114	16-Mar-2006	6610	17-Mar-2006			
1124	20-Mar-2006	6610	21-Mar-2006			
1115	20-Mar-2006	6610	21-Mar-2006			
1116	21-Mar-2006	6610	?			
1117	21-Mar-2006	6610	?			
6635	paola	bianchi	0104647992	12-Apr-1976	via dodecaneso 35	16146 genova
1111	01-Mar-2006	6635	02-Mar-2006			
1115	01-Mar-2006	6635	02-Mar-2006			
1117	02-Mar-2006	6635	06-Mar-2006			
1118	02-Mar-2006	6635	06-Mar-2006			
1119	08-Mar-2006	6635	10-Mar-2006			
1120	08-Mar-2006	6635	10-Mar-2006			
1121	15-Mar-2006	6635	18-Mar-2006			
1122	15-Mar-2006	6635	18-Mar-2006			
1113	15-Mar-2006	6635	18-Mar-2006			
1129	15-Mar-2006	6635	20-Mar-2006			
1127	22-Mar-2006	6635	?			
1125	22-Mar-2006	6635	?			
6642	marco	verdi	3336745383	16-Ott-1972	via lagustena 35	16131 genova
1111	04-Mar-2006	6642	05-Mar-2006			
1116	08-Mar-2006	6642	09-Mar-2006			
1118	10-Mar-2006	6642	11-Mar-2006			
1119	15-Mar-2006	6642	16-Mar-2006			
1128	18-Mar-2006	6642	20-Mar-2006			
1124	21-Mar-2006	6642	22-Mar-2006			
1122	22-Mar-2006	6642	?			
1113	22-Mar-2006	6642	?			

Figura 7.3: Co-clustering delle relazioni Noleggio e Cliente sull'attributo codCli

quindi le tuple della stessa relazione con lo stesso valore per tali attributi saranno memorizzate fisicamente contigue, nello stesso blocco o su blocchi adiacenti. Tale organizzazione in cluster è associata all'uso di tecniche di indice (ad albero o hash), come organizzazione primaria dei dati, come discusso nel Paragrafo 7.3. Ad esempio, se la memorizzazione fisica dei record corrispondenti alle tuple della relazione Noleggio è quella della Figura 2.2, la relazione è clusterizzata sull'attributo dataNol. Con tale strategia di memorizzazione dei dati possiamo ritrovare in maniera efficiente tutti i noleggi effettuati in un certo giorno, o in un certo periodo, perché i record corrispondenti sono memorizzati nello stesso blocco o in blocchi adiacenti. Viceversa, un clustering su codCli sarebbe più efficiente per ritrovare tutti i noleggi effettuati da un certo cliente e un clustering su colloc per ritrovare tutti i noleggi di un certo video.

È anche possibile organizzare in cluster due o più relazioni, parleremo in questo caso di *co-clustering*. Nel co-clustering vengono memorizzate fisicamente contigue le tuple delle due relazioni che hanno lo stesso valore per gli attributi associati alla chiave del cluster, tipicamente gli attributi che sono chiave esterna di una relazione sull'altra, che vengono utilizzati per effettuarne il join.

**Esempio 7.2** Consideriamo le relazioni Noleggio e Cliente, contenenti le tuple della Figura 2.2, e l'interrogazione SQL che ritrova nome, cognome e data di

nascita dei clienti che hanno effettuato dei noleggi, oltre alla collocazione del video noleggiato ed alla data di noleggio:

```
SELECT nome, cognome, dataN, colloc, dataNol
FROM Noleggio NATURAL JOIN Cliente;
```

Una strategia di memorizzazione efficiente per questo tipo di interrogazione è basata sul co-clustering delle relazioni sull'attributo di join (`codCli`), come illustrato nella Figura 7.3. Il co-clustering può rendere tuttavia inefficiente l'esecuzione di altre interrogazioni. Ad esempio, l'esecuzione dell'interrogazione:

```
SELECT * FROM Cliente;
```

richiede l'accesso ad un numero di blocchi maggiore rispetto ad una strategia di memorizzazione in cui si usa un file separato per ogni relazione. Inoltre, per poter ritrovare le tuple della relazione `Cliente` può essere necessario collegarle con dei puntatori.  $\square$

Un cluster è quindi una struttura di memorizzazione che contiene dati di una o più relazioni. Ogni cluster ha una chiave, costituita da uno o più attributi. Nell'inserire una relazione in un cluster alcuni attributi della relazione vengono associati alla chiave del cluster. Le tuple delle relazioni inserite nel cluster che hanno lo stesso valore per gli attributi associati alla chiave del cluster vengono memorizzate fisicamente contigue. La scelta degli attributi su cui clusterizzare una relazione, così come la scelta di effettuare il co-clustering di relazioni, dipende dalle operazioni da eseguire su di esse e viene effettuata durante la fase di progettazione fisica della base di dati (vedi Paragrafo 7.5). Nel Paragrafo 7.3.4 presenteremo i comandi SQL per organizzare in cluster una o più relazioni.

### 7.2.3 Gestione del buffer

L'obiettivo principale delle strategie di memorizzazione è minimizzare gli accessi a disco. Un altro modo, oltre a quello discusso nel paragrafo precedente, per ottenere lo stesso risultato è mantenere più blocchi possibile in memoria principale. A questo scopo viene utilizzato un *buffer* che permette di tenere in memoria principale copia di alcuni blocchi di disco. Il gestore del buffer di un DBMS usa politiche di gestione che sono più sofisticate delle politiche usualmente utilizzate dai sistemi operativi. In particolare, per motivi legati alla gestione del ripristino (vedi Capitolo 8), in alcuni casi un blocco non può essere trasferito su disco, mentre in altri casi è necessario forzare la copiatura di un blocco su disco, anche se il suo spazio non è stato reclamato. Inoltre, un sistema operativo usa tipicamente politiche di tipo *least recently used* (LRU) per gestire il buffer. In accordo a tali politiche, il blocco contenente dati a cui si è acceduto meno recentemente viene copiato su disco ed eliminato dal buffer. Una politica di questo tipo è adeguata quando non si sa predire il pattern degli accessi. Un DBMS è invece spesso in grado di predire meglio il tipo dei riferimenti futuri, come discusso dall'esempio seguente.

**Esempio 7.3** Consideriamo l'operazione di join:

**Noleggio  $\bowtie$  Cliente**

e supponiamo che le relazioni **Noleggio** e **Cliente** siano memorizzate su file diversi. Per eseguire tale join, una possibilità è considerare ogni tupla di **Noleggio** e confrontarla con ogni tupla di **Cliente** (vedi Paragrafo 7.4.3.4, iterazione orientata ai blocchi). In tal caso, una volta che una tupla della relazione **Noleggio** è stata usata, non è più necessaria. Quindi, non appena le tuple di un blocco sono state esaminate, il blocco non serve più; il gestore del buffer deve pertanto liberare tale blocco (strategia *toss immediate*). Per quanto riguarda invece i blocchi della relazione **Cliente**, il blocco a cui si è acceduto più recentemente sarà utilizzato di nuovo solo dopo che tutti gli altri blocchi saranno stati esaminati. Pertanto, la strategia migliore per i blocchi del file **Cliente** è quella di rimuovere l'ultimo blocco esaminato (*most recently used* - MRU). Affinché tale strategia funzioni correttamente, però, è necessario tenere in memoria il blocco correntemente esaminato fino a che non ne sono state considerate tutte le tuple.  $\square$

### 7.3 Strutture ausiliarie di accesso

Spesso le interrogazioni accedono solo ad un piccolo sottoinsieme dei dati. Per risolvere efficientemente le interrogazioni può quindi essere utile utilizzare delle strutture ausiliarie che permettano di determinare direttamente i record che verificano una data condizione, senza dover accedere a tutti i dati. In tale contesto, il termine *chiave di ricerca* indica un attributo, od un insieme di attributi, usati per la ricerca.<sup>2</sup> Per migliorare l'accesso ai dati vengono utilizzati in genere due tipi di organizzazioni:

- **Organizzazioni primarie.** Tali organizzazioni impongono un criterio di memorizzazione dei dati, possono essere organizzazioni ad albero o basate su tecniche hash.
- **Organizzazioni secondarie.** Tali organizzazioni fanno ricorso ad indici (separati dal file dei dati) che sono normalmente organizzati ad albero (ma sono anche possibili indici hash).

Poiché un'organizzazione primaria impone un criterio di allocazione dei dati, mentre un'organizzazione secondaria no, è possibile avere per gli stessi dati un'organizzazione primaria ed una o più organizzazioni secondarie. In generale, esistono quindi più modalità (cammini) di accesso ai dati. La Figura 7.4 illustra una possibile struttura di memorizzazione per la relazione **Film**. Tale struttura utilizza un'organizzazione primaria, basata su struttura hash sull'attributo **regista**, e due organizzazioni secondarie, una ad albero sull'attributo **anno** e l'altra hash

---

<sup>2</sup>Ricordiamo nuovamente che questo concetto è diverso dal concetto di *chiave primaria* introdotto relativamente all'identificazione delle tuple nel Capitolo 2.

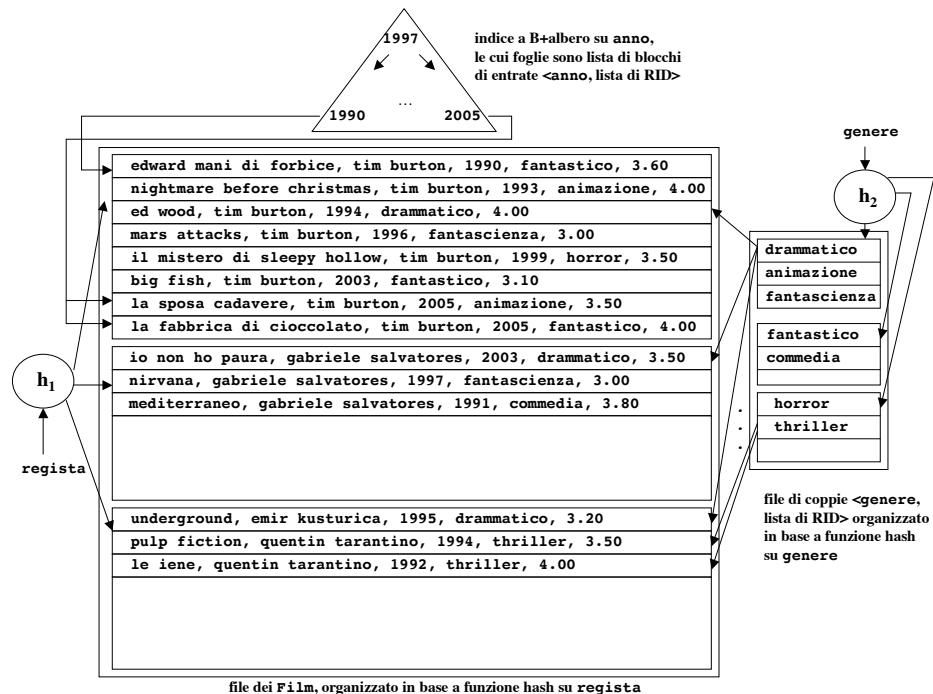


Figura 7.4: Organizzazioni primarie e secondarie

sull'attributo **genere**. L'uso di più indici rende l'esecuzione delle interrogazioni più efficiente, ma rende in generale più costosi gli aggiornamenti. Quando si esegue l'inserimento o la cancellazione di un record è necessario infatti modificare tutti gli indici allocati sul file.

Un indice è costituito da un insieme di *entrate*, ciascuna corrispondente ad un valore  $k_i$  della *chiave* dell'indice, ovvero dell'attributo (degli attributi) su cui è definito l'indice. Le diverse tecniche di indicizzazione (alberi o hash) differiscono essenzialmente nel modo in cui organizzano l'insieme di entrate. Un'entrata dell'indice contiene abbastanza informazioni per localizzare uno o più record di dati che hanno valore  $k_i$  della chiave. Ci sono tre diverse alternative rispetto a che cosa memorizzare nell'entrata dell'indice relativa ad un valore di chiave  $k_i$ :

1. l'entrata contiene un record dati con valore di chiave  $k_i$ , parliamo in questo caso di *indice integrato*;
2. l'entrata contiene una coppia  $(k_i, r_i)$  dove  $r_i$  è il RID del record (eventualmente il solo) con valore di chiave  $k_i$ ;
3. l'entrata contiene una coppia  $(k_i, l_i)$  dove  $l_i$  è una lista di RID di record con valore di chiave  $k_i$ .

L'alternativa (1) corrisponde ad un'organizzazione primaria dei dati, mentre le alternative (2) e (3) corrispondono ad organizzazioni secondarie. Gli indici

<b>Contenuto delle entrate dell'indice</b>	<i>Indice integrato</i> indice le cui entrate contengono i record dei dati (alternativa (1))	<i>Indice separato</i> indice le cui entrate contengono riferimenti ai record dei dati (alternative (2) e (3))
<b>Unicità dei valori di chiave</b>	<i>Indice su chiave primaria</i> indice la cui chiave è chiave primaria per la relazione	<i>Indice su chiave secondaria</i> indice la cui chiave non è chiave primaria per la relazione
<b>Corrispondenza tra posizione entrate dell'indice e record dei dati</b>	<i>Indice clusterizzato</i> indice in cui vi è corrispondenza tra le posizioni delle entrate dell'indice e quelle dei record dati	<i>Indice non clusterizzato</i> indice in cui non vi è corrispondenza tra le posizioni delle entrate dell'indice e quelle dei record dati
<b>Numero di entrate nell'indice</b>	<i>Indice denso</i> indice il cui numero di entrate è pari al numero di valori di $k_i$	<i>Indice sparso</i> indice il cui numero di entrate è minore del numero di valori di $k_i$
<b>Numero di livelli</b>	<i>Indice a singolo livello</i> indice organizzato su un singolo livello	<i>Indice multi-livello</i> indice organizzato su più livelli
<b>Numero di attributi nella chiave</b>	<i>Indice su singolo attributo</i> indice la cui chiave è un singolo attributo	<i>Indice multi-attributo</i> indice la cui chiave è costituita da due o più attributi

Tabella 7.1: Classificazione dei vari tipi di indice

che utilizzano le alternative (2) e (3) vengono a volte anche indicati come *indici separati*. In riferimento alla Figura 7.4, entrambe le organizzazioni secondarie (su **anno** e **genere**) utilizzano l'alternativa (3). Se vogliamo costruire più di un indice su una collezione di record, come discusso precedentemente, uno solo tra gli indici userà l'alternativa (1), per evitare di memorizzare i dati più di una volta. Il vantaggio nell'uso di un indice di tipo (2) o (3) nasce dal fatto che la chiave è solo parte dell'informazione contenuta in un record. Pertanto, l'indice occupa meno spazio del file dei dati. Nel seguito di questo paragrafo introdurremo innanzitutto una classificazione delle varie tipologie di indici per poi discutere le due tecniche di indice più comunemente utilizzate: gli indici ad albero e gli indici hash.

### 7.3.1 Tipologie di indici

La Tabella 7.1 riassume la classificazione degli indici in base alle loro caratteristiche principali, che verrano discusse in quanto segue.

**Indici su chiave primaria e su chiave secondaria.** Un indice *su chiave primaria* ha come chiave un insieme di attributi che include la chiave primaria della relazione su cui è definito. Esisterà quindi al più un record con tale valore per ogni possibile valore di chiave. Un indice è invece *su chiave secondaria* se la chiave dell'indice non include la chiave primaria della relazione su cui è definito l'indice

e possono quindi in generale esistere più record con lo stesso valore di chiave. Ad esempio, in riferimento alla relazione `Noleggio`, l'indice su `(colloc,dataNol)` è un indice su chiave primaria, mentre l'indice su `codCli` è un indice su chiave secondaria. Nel caso di indice su chiave primaria possono essere utilizzate le alternative (1) o (2) discusse nel paragrafo precedente, mentre non ha senso utilizzare l'alternativa (3), poiché esiste un unico record per ogni valore di chiave. Per gli indici su chiave secondaria, al contrario, può essere usata l'alternativa (3). Possono però essere anche usate le alternative (1) o (2), nelle seguenti modalità: (i) con entrate *duplicate*, cioè con lo stesso valore di chiave, che quindi compare in più entrate dell'indice; (ii) inserendo un ulteriore livello di indirezione: l'entrata per la chiave  $k_i$  dell'indice contiene il riferimento ad un blocco in cui sono memorizzati i record dei dati oppure i RID dei record dei dati (a seconda che si tratti di alternativa (1) o (2)) aventi valore di chiave  $k_i$ . Il vantaggio rispetto all'alternativa (3) è che le entrate dell'indice hanno tutte la stessa dimensione. Il vantaggio di (ii) rispetto ad (i) è invece che non devono essere modificati gli algoritmi di ricerca ed inserimento nell'indice. Anche se la soluzione (ii) ha lo svantaggio evidente del livello di indirezione aggiuntivo introdotto, risulta comunque la più utilizzata.

**Indici clusterizzati e non clusterizzati.** Quando un file è organizzato in modo tale che vi è una corrispondenza tra la posizione delle entrate dell'indice e quelle dei record dati corrispondenti, l'indice è detto *clusterizzato* (o *primario*,<sup>3</sup> od *organizzato in cluster*); altrimenti è un indice *non clusterizzato* (o *secondario*, o *non organizzato in cluster*). Un indice che usa l'alternativa (1), cioè un indice integrato, è organizzato in cluster per definizione. Un indice che usa l'alternativa (2) o (3) è clusterizzato se è un indice ad albero ed i record dei dati sono ordinati sui campi chiave dell'indice. Poiché le entrate di un indice ad albero sono ordinate in base al valore della chiave, se i record dei dati sono ordinati sui campi corrispondenti si ha anche in questo caso una corrispondenza tra la posizione delle entrate dell'indice e quelle dei record dei dati corrispondenti.

La Figura 7.5 illustra un esempio di indice non clusterizzato. In particolare, illustra un indice sull'attributo `colloc` della relazione `Noleggio`, non clusterizzato, denso, su chiave secondaria che usa l'alternativa (3). La Figura 7.6 illustra invece un esempio di indice clusterizzato. In particolare, illustra un indice sull'attributo `dataNol` della relazione `Noleggio`, clusterizzato, sparso, su chiave secondaria che usa l'alternativa (3).

**Indici densi e sparsi.** Relativamente al numero di entrate dell'indice possiamo distinguere tra indici densi ed indici sparsi. Un *indice denso* contiene un'entrata per ogni valore della chiave di ricerca nel file; al contrario, in un *indice sparso* le entrate dell'indice sono create solo per alcuni valori della chiave. La Figura 7.6 illustra un indice sparso su `dataNol`. Per localizzare un record, utilizzando

---

<sup>3</sup>Alcuni testi usano il termine indice primario per indicare indici su chiave primaria. Per evitare ambiguità, eviteremo il termine indice primario ed useremo invece i termini indice su chiave primaria ed indice clusterizzato.

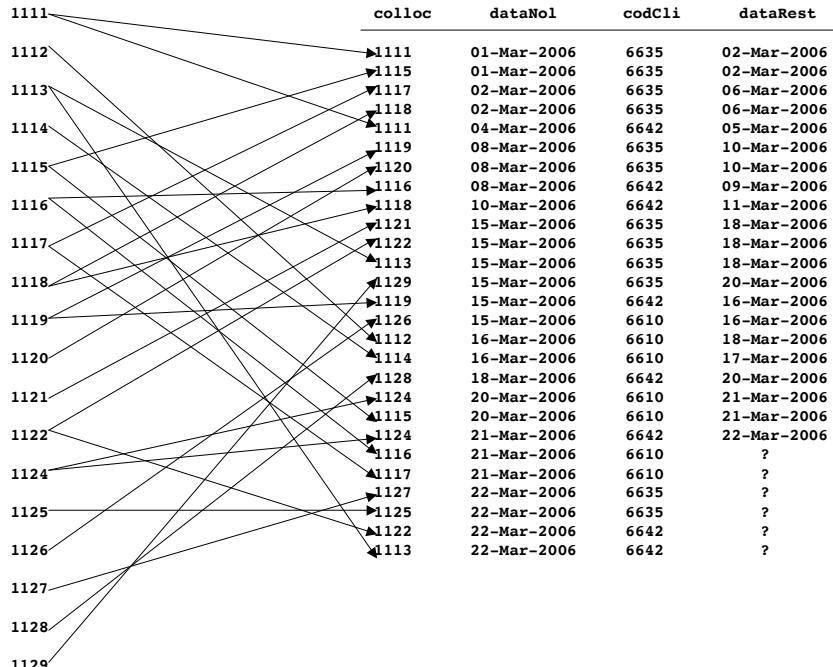


Figura 7.5: Indice non clusterizzato

un indice sparso, viene eseguita una scansione fino a trovare il record con il più alto valore della chiave che sia minore o uguale al valore cercato. Viene quindi effettuata una ricerca sequenziale nel file dei dati, a partire da tale record, fino a trovare il record cercato. Gli indici non clusterizzati sono indici densi anziché sparsi; il file dei dati, infatti, non è ordinato in base alla chiave dell'indice. In riferimento all'indice su `colloc` illustrato nella Figura 7.5, infatti, se l'entrata per 1126 non fosse presente, l'indice non sarebbe di alcuna utilità per ritrovare i noleggi di tale video. Un indice denso consente una ricerca più veloce, ma impone maggiori costi di aggiornamento. Un indice sparso è viceversa meno efficiente, ma impone minori costi di aggiornamento. Poiché la strategia è di minimizzare il numero di blocchi trasferiti, un compromesso spesso adottato consiste nell'avere un'entrata nell'indice per ogni blocco.

**Indici multi-livello.** Un indice a singolo livello è un indice le cui entrate sono costruite sui record dei dati. Un indice multi-livello, viceversa, è un indice le cui entrate sono costruite su un altro indice. Molto spesso, infatti, un indice, anche se sparso, può essere di dimensioni notevoli. Ad esempio, un file di 100'000 record, con 10 record per blocco, richiede un indice con 10'000 entrate; assumendo che un blocco contenga 100 entrate dell'indice, sono necessari 100 blocchi. Se

	<b>colloc</b>	<b>dataNol</b>	<b>codCli</b>	<b>dataRest</b>
01-Mar-2006	1111	01-Mar-2006	6635	02-Mar-2006
	1115	01-Mar-2006	6635	02-Mar-2006
	1117	02-Mar-2006	6635	06-Mar-2006
	1118	02-Mar-2006	6635	06-Mar-2006
	1111	04-Mar-2006	6642	05-Mar-2006
	1119	08-Mar-2006	6635	10-Mar-2006
	1120	08-Mar-2006	6635	10-Mar-2006
	1116	08-Mar-2006	6642	09-Mar-2006
10-Mar-2006	1118	10-Mar-2006	6642	11-Mar-2006
	1121	15-Mar-2006	6635	18-Mar-2006
	1122	15-Mar-2006	6635	18-Mar-2006
	1113	15-Mar-2006	6635	18-Mar-2006
	1129	15-Mar-2006	6635	20-Mar-2006
	1119	15-Mar-2006	6642	16-Mar-2006
	1126	15-Mar-2006	6610	16-Mar-2006
	1112	16-Mar-2006	6610	18-Mar-2006
	1114	16-Mar-2006	6610	17-Mar-2006
	1128	18-Mar-2006	6642	20-Mar-2006
20-Mar-2006	1124	20-Mar-2006	6610	21-Mar-2006
	1115	20-Mar-2006	6610	21-Mar-2006
	1124	21-Mar-2006	6642	22-Mar-2006
	1116	21-Mar-2006	6610	?
	1117	21-Mar-2006	6610	?
	1127	22-Mar-2006	6635	?
	1125	22-Mar-2006	6635	?
	1122	22-Mar-2006	6642	?
	1113	22-Mar-2006	6642	?

Figura 7.6: Indice sparso (clusterizzato)

l'indice è piccolo, può essere tenuto in memoria principale. Molto spesso, però, è necessario tenerlo su disco e quindi la scansione dell'indice può richiedere parecchi trasferimenti di blocchi (se l'indice occupa un numero  $b$  di blocchi e si usa una ricerca binaria, è necessario accedere ad un numero di blocchi pari a  $1 + \log_2 b$ , circa 7 nel nostro esempio). È quindi necessario trattare l'indice come un file ed allocare un indice sparso sull'indice stesso. Si parla in questo caso di *indice sparso a due livelli*. Se l'indice esterno è mantenuto in memoria principale, nell'esempio precedente è necessario accedere ad un solo blocco dell'indice.

**Indici multi-attributo.** Un indice multi-attributo è un indice la cui chiave è costituita da più di un attributo. Viceversa, un indice la cui chiave è costituita da un solo attributo è detto indice su singolo attributo. Ad esempio, l'indice su  $(\text{colloc}, \text{dataNol})$  per la relazione *Noleggio* è un indice multi-attributo, mentre l'indice su  $\text{codCli}$  è un indice su singolo attributo. L'ordinamento tra i valori di chiave è dato dall'ordinamento lessicografico. L'indice multi-attributo è definito su una *lista* di attributi, cioè diversi ordinamenti degli attributi chiave danno luogo ad indici differenti. Un indice su  $(\text{colloc}, \text{dataNol})$  è cioè differente, ad esempio, da un indice su  $(\text{dataNol}, \text{colloc})$ . Un indice multi-attributo permette di determinare efficientemente le tuple che soddisfano condizioni di uguaglianza o di intervallo (nel caso di indici ad albero) su tutti gli attributi nella chiave o su un *prefisso* della lista di attributi. L'indice su  $(\text{dataNol}, \text{colloc})$ , ad esempio, è utile per condizioni quali  $\text{dataNol} = \text{DATE } '15-\text{Mar-2006}' \text{ AND colloc} = 1111$ ,  $\text{dataNol}$

= DATE '15-Mar-2006' oppure `dataNol BETWEEN DATE '15-Mar-2006' AND DATE '18-Mar-2006'`. Viceversa, l'indice non aiuta, ad esempio, a determinare tutti i noleggi per il video 1120.

Un indice multi-attributo può supportare una più vasta gamma di interrogazioni rispetto ad un indice a singolo attributo. Inoltre, poiché le entrate dell'indice contengono più informazioni sul record dei dati, gli indici multi-attributo offrono maggiori opportunità di poter valutare interrogazioni accedendo solo all'indice e non al file dei dati (come vedremo nel Paragrafo 7.4, una valutazione basata soltanto sull'indice non necessita di accedere ai dati, ma trova tutti i valori degli attributi richiesti nelle entrate dell'indice). D'altra parte, un indice multi-attributo deve essere aggiornato più frequentemente ed è di dimensione maggiore rispetto ad un indice su singolo attributo.

### 7.3.2 Indici ad albero

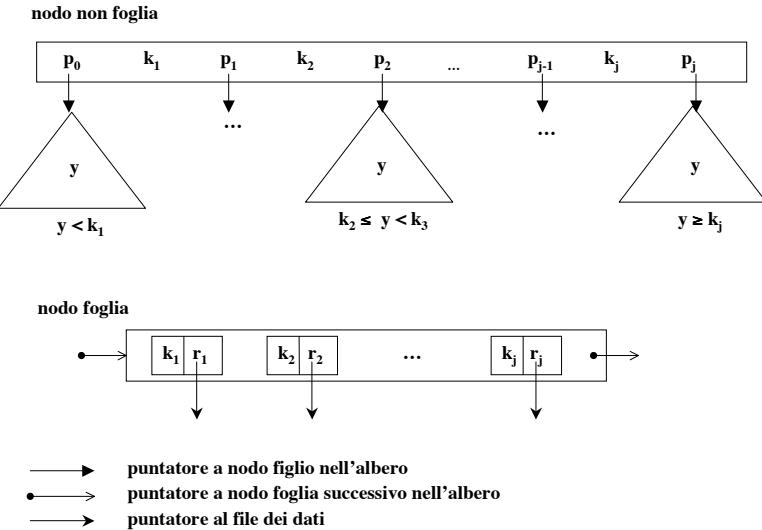
Gli indici ad albero sono alberi binari di ricerca bilanciati per memoria secondaria che rispondono ai seguenti requisiti fondamentali:

- **bilanciamento**, l'indice è bilanciato rispetto ai blocchi e non ai singoli nodi, in quanto è il numero di blocchi acceduti a determinare il costo I/O di una ricerca;
- **occupazione minima**, per evitare un sotto-utilizzo della memoria viene stabilito un limite inferiore all'utilizzazione dei blocchi;
- **efficienza di aggiornamento**, il costo delle operazioni di aggiornamento è comunque limitato.

In un indice ad albero, quindi, ogni nodo corrisponde ad un blocco, memorizza molti valori di chiave e tipicamente ha centinaia di figli. Il costo delle operazioni (ricerca, inserimento e cancellazione) in tali strutture è lineare nell'altezza dell'albero<sup>4</sup> e logaritmico nel numero di elementi memorizzati nell'indice. La struttura impone un limite minimo (oltre all'ovvio limite massimo corrispondente alla dimensione del nodo) al numero di elementi contenuti in un nodo. Il numero di figli di un nodo è dato dal numero di elementi contenuti nel nodo più uno. Se ogni nodo non foglia avesse  $n$  figli, un albero di altezza  $h$  avrebbe  $n^h$  nodi foglia. Nella pratica, i nodi non hanno lo stesso numero di figli, ma usando il valore medio  $m$ ,  $m^h$  è una buona approssimazione del numero dei nodi foglia. Sempre nella pratica,  $m$  è almeno 100, quindi un albero di altezza quattro contiene 100 milioni di nodi foglia.

Gli indici ad albero più utilizzati sono i B<sup>+</sup>-alberi, il formato dei cui nodi è illustrato nella Figura 7.7. Nella figura, entrambi i tipi di nodi, sia quelli interni sia i nodi foglia, hanno  $j$  elementi,  $k_1, \dots, k_j$  sono i valori di chiave,  $p_0, \dots, p_j$  sono i

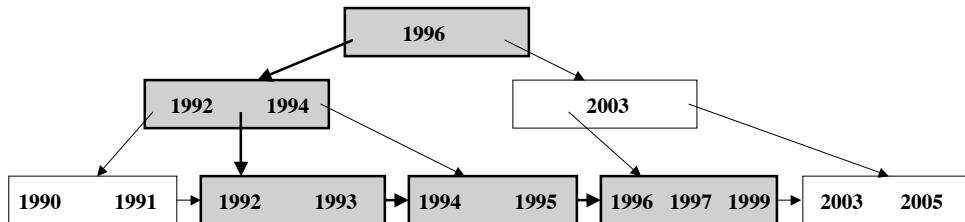
<sup>4</sup>Con altezza dell'albero indichiamo la lunghezza di un cammino dalla radice ad una foglia. La proprietà di bilanciamento dell'albero assicura che tutti i cammini radice-foglia abbiano la stessa lunghezza.

Figura 7.7: Formato di un nodo di un B<sup>+</sup>-albero

puntatori ai nodi figli e  $r_1, \dots, r_j$  sono RID (facciamo riferimento ad un albero che utilizza l'alternativa (2) discussa in precedenza). Come evidenziato dalla figura, in un B<sup>+</sup>-albero le entrate dell'indice sono contenute nelle foglie, mentre i nodi interni costituiscono una “mappa” che consente una rapida localizzazione delle chiavi e memorizzano dei separatori. La funzione dei separatori è determinare il giusto cammino nella ricerca di una chiave. Il sotto-albero sinistro di un separatore contiene valori di chiave minori del separatore. Il sotto-albero destro contiene valori di chiave maggiori od uguali al separatore.<sup>5</sup> Nel caso di chiavi alfanumeriche è possibile utilizzare separatori di lunghezza ridotta risparmiando spazio e, eventualmente, riducendo l'altezza dell'albero. I nodi foglia sono inoltre collegati a lista.

**Esempio 7.4** Consideriamo l'indice a B<sup>+</sup>-albero sull'attributo **anno** della relazione **Film**, introdotto nella Figura 7.4. Un B<sup>+</sup>-albero contenente i valori di tale attributo è mostrato nella Figura 7.8. In realtà, per illustrare le caratteristiche dell'albero, abbiamo utilizzato nodi che contengono al massimo 3 elementi, quindi con capacità decisamente inferiore a quella dei nodi utilizzati in pratica. Per semplicità, inoltre, non abbiamo evidenziato nella figura i riferimenti al file dei dati. L'albero nella figura contiene 11 elementi (cioè 11 entrate dell'indice), ha 5 foglie ed altezza 3. Ricordiamo che, rispetto alla classificazione nella Tabella 7.1, tale indice è un indice su chiave secondaria, denso, su singolo attributo (non

<sup>5</sup>È possibile anche adottare la convenzione che i valori uguali siano contenuti nel sotto-albero sinistro.

Figura 7.8: B<sup>+</sup>-albero sull'attributo anno della relazione Film

specifichiamo se clusterizzato o meno perché non sappiamo come è organizzato il file dei dati). □

La ricerca di una chiave avviene nel B<sup>+</sup>-albero fino ad individuare una foglia, nella quale si trovano le entrate dell'indice, cioè le chiavi ed i corrispondenti riferimenti ai dati. Una volta trasferita la radice in memoria, si esegue la ricerca tra le chiavi contenute nel nodo in esame fino a determinare la presenza o l'assenza nel nodo della chiave cercata. Se la chiave non viene trovata, si continua la ricerca nell'unico sotto-albero del nodo corrente che può contenere l'elemento. Se invece il nodo è foglia, significa che la chiave non è presente nell'albero. Il collegamento a lista dei nodi foglia consente di risolvere in modo efficiente anche ricerche su intervalli: viene ricercato nell'albero l'estremo sinistro dell'intervalle arrivando ad una foglia, la lista dei nodi foglia viene quindi attraversata fino a raggiungere l'estremo destro dell'intervalle.

**Esempio 7.5** In riferimento al B<sup>+</sup>-albero nella Figura 7.8, se vogliamo ritrovare tutti i film usciti tra il 1992 e il 1998, si raggiungerà la foglia dell'indice corrispondente al valore di chiave 1992 e ci si sposterà poi nelle foglie dell'indice fino a trovare il valore di chiave 1999. Il cammino seguito nell'albero per effettuare la ricerca è evidenziato nella figura marcando gli archi attraversati in grassetto ed ombreggiando i nodi visitati. □

Le operazioni di inserimento e cancellazione richiedono innanzitutto una ricerca dell'elemento, ma possono richiedere ulteriori aggiustamenti dell'albero per mantenere la proprietà di bilanciamento ed i vincoli sul numero minimo e massimo di elementi contenuti in un nodo. L'idea chiave su cui si basano gli algoritmi per l'inserimento e la cancellazione è che le modifiche partono sempre dalle foglie e l'albero cresce o si accorcia verso l'alto. Ad esempio, nel caso dell'inserimento, non si creano nuovi figli dalle foglie, ma, se necessario, si crea una nuova foglia allo stesso livello delle altre e si propaga un valore di chiave (separatore) verso l'alto. I nodi ai livelli superiori non sono necessariamente pieni e quindi possono "assorbire" le informazioni che si propagano dalle foglie. La propagazione degli

effetti sino alla radice può provocare l'aumento dell'altezza dell'albero. Il costo delle operazioni è comunque lineare nell'altezza dell'albero.

### 7.3.3 Indici hash

L'uso di indici ad albero ha lo svantaggio di richiedere la scansione di una struttura dati per localizzare i dati; questo perché nelle tecniche di tipo "tabellare" l'associazione (chiave, indirizzo) è mantenuta in forma esplicita. Un'organizzazione hash, invece, utilizza una *funzione hash*  $h$  che trasforma ogni valore di chiave in un indirizzo. In queste organizzazioni, i record di un file sono raggruppati in  $M$  *bucket*, dove un bucket consiste in un blocco primario ed eventuali blocchi addizionali organizzati in una lista. La funzione hash, data una chiave, restituisce l'indirizzo (blocco o bucket di blocchi) da cui partire per cercare i record con quel valore di chiave (in relazione al fatto che vengano utilizzate le alternative (1), (2) o (3) discusse nel Paragrafo 7.3). Ad esempio, nella Figura 7.4, la funzione hash  $h_1$ , dato un **regista**, restituisce l'indirizzo del bucket contenente i record dati dei **Film** di tale regista (organizzazione primaria), viceversa, la funzione hash  $h_2$ , dato un **genere**, restituisce l'indirizzo del bucket contenente la lista dei RID dei **Film** di quel genere (organizzazione secondaria). Le organizzazioni hash sono usate principalmente per l'organizzazione primaria dei dati.

Una funzione hash  $h$  è una funzione dall'insieme dei possibili valori per la chiave all'insieme  $0, \dots, M - 1$  dei possibili indirizzi che deve verificare le seguenti proprietà:

- **distribuzione uniforme delle chiavi nello spazio degli indirizzi:** ogni indirizzo deve essere generato con la stessa probabilità;
- **distribuzione casuale delle chiavi:** eventuali correlazioni tra i valori delle chiavi non devono tradursi in correlazioni tra gli indirizzi generati.

Poiché tali proprietà dipendono dall'insieme delle chiavi su cui si opera non esiste una funzione universale ottima. Una delle funzioni hash che si comporta meglio, e quindi tra le più utilizzate in pratica, è quella basata sul metodo della divisione: la chiave numerica viene divisa per  $M$  e l'indirizzo è ottenuto considerando il resto. La funzione  $h$  è quindi definita come  $h(k) = k \bmod M$  dove, come usuale,  $\bmod$  indica il resto della divisione intera. Affinché  $h$  distribuisca bene è però necessario che  $M$  sia un numero primo oppure un numero non primo con nessun fattore primo minore di 20.

Per effettuare la ricerca, dato un valore di chiave  $k$ , calcoliamo semplicemente  $h(k)$ . Ogni indirizzo generato dalla funzione hash individua una pagina logica o bucket. Il costo delle operazioni è costante, se la struttura è ben progettata, ed ogni indirizzo corrisponde ad un singolo blocco. Il numero di elementi che possono essere allocati nello stesso bucket determina la *capacità*  $c$  dei bucket. Se una chiave viene assegnata ad un bucket che già contiene  $c$  chiavi si verifica un *trabocco* (*overflow*). La presenza di overflow può richiedere, dipendentemente dalla specifica organizzazione, l'uso di un'area di memoria separata, detta *area*

*di overflow.* L'area di memoria costituita dai bucket indirizzabili dalla funzione hash è detta *area primaria*. Alle pagine dell'area primaria si accede direttamente, mentre alle pagine dei trabocchi si accede mediante riferimenti memorizzati nelle pagine dell'area primaria.

Una funzione hash genera  $M$  indirizzi, tanti quanti sono i bucket dell'area primaria. Se il valore di  $M$  per una data organizzazione è costante, l'organizzazione è detta *statica*. In questo caso il dimensionamento dell'area primaria è parte integrante del progetto dell'organizzazione. Se l'area primaria può espandersi e contrarsi, per meglio adattarsi al volume effettivo dei dati da gestire, allora l'organizzazione è detta *dinamica*. In questo caso è necessario l'utilizzo di più funzioni hash.

**Confronto tra indici ad albero ed hash.** L'uso di un indice ad albero piuttosto che hash dipende dal tipo di interrogazioni. Se ad esempio la maggior parte delle interrogazioni che coinvolgono la chiave  $A_i$  ha la forma:

```
SELECT A1,A2,...,An FROM R WHERE Ai=C;
```

la tecnica hash è preferibile. La scansione di un indice ad albero ha infatti un costo proporzionale al logaritmo del numero di valori in  $R$  per  $A_i$  mentre in una struttura hash il tempo di ricerca è indipendente dalla dimensione della relazione.

Le strutture ad albero sono invece preferibili se le interrogazioni che coinvolgono la chiave  $A_i$  usano condizioni di intervallo come:

```
SELECT A1,A2,...,An FROM R WHERE Ai BETWEEN C1 AND C2;
```

perché è difficile determinare funzioni hash che mantengano l'ordine. In generale, è difficile prevedere a priori il tipo di interrogazioni per cui in pratica sono maggiormente utilizzate strutture di indici ad albero.

#### 7.3.4 Definizione di cluster ed indici in SQL

La maggior parte dei DBMS relazionali fornisce varie primitive che permettono al progettista della base di dati di definire la configurazione fisica dei dati. Queste primitive sono rese disponibili all'utente come comandi del linguaggio SQL. Lo standard SQL:2003, tuttavia, non include alcun comando per la definizione di strutture di memorizzazione o di indici. In realtà, lo standard non richiede neppure che le implementazioni di SQL supportino gli indici. Ovviamente, nella pratica, ogni DBMS relazionale in commercio supporta uno o più tipi di indice ed, in genere, alloca automaticamente indici, ad esempio, sugli attributi chiave primaria di relazione. Poiché esistono però numerose differenze tra i vari DBMS per quanto riguarda gli aspetti di organizzazione fisica, i comandi per la configurazione fisica dei dati differiscono notevolmente tra i vari sistemi. I comandi più importanti sono il comando per la creazione di cluster ed il comando per la creazione di indici, oltre

all'inserimento di una relazione in un cluster. Nel seguito faremo riferimento ad una versione semplificata della sintassi Oracle.

#### 7.3.4.1 Definizione di cluster

Il comando per la creazione di un cluster ha il seguente formato:

```
CREATE CLUSTER <nome cluster>
(<nome colonna> <dominio> [,<nome colonna> <dominio>]*)
[{:INDEX | HASHKEYS <intero> [HASH IS <espressione>]}];
```

dove:

- <nome cluster> è il nome del cluster che viene definito.
- La lista di coppie <nome colonna> <dominio> è la specifica delle colonne del cluster, cioè della chiave del cluster.

Come possiamo inoltre notare dal formato del comando, ogni cluster ha sempre associata una struttura di accesso ausiliaria. Le scelte possibili sono:

- INDEX. Viene allocato un indice di tipo B<sup>+</sup>-albero. Tale scelta, che è quella di default, conviene se si hanno frequenti interrogazioni di tipo intervallo sulla chiave del cluster o se le relazioni possono aumentare di dimensione in modo impredicibile. Un cluster su cui è allocato un indice è detto *cluster di tipo index*.
- HASH. Viene usata una struttura di accesso di tipo hash. Un cluster su cui viene utilizzata un'organizzazione hash è detto *cluster di tipo hash*.

Se il cluster è di tipo index, prima di poter manipolare le relazioni nel cluster è necessario creare un indice sul cluster tramite il comando **CREATE INDEX**, discusso nel Paragrafo 7.3.4.2.

La clausola **HASHKEYS** richiede l'uso dell'hash come struttura di accesso e specifica il numero di valori della funzione hash. Questo valore se non è un numero primo viene arrotondato dal sistema al primo numero primo maggiore. Tale intero viene usato come argomento dell'operazione usata dal sistema per generare i valori della funzione hash; sia  $M$  l'intero dato (od il suo arrotondamento), i valori generati sono compresi tra 0 ed  $M - 1$ . Nei cluster di tipo hash, la clausola **HASH IS** permette la specifica della funzione hash da utilizzare. Il DBMS fornisce sempre una funzione hash interna che viene usata come default. È tuttavia possibile non usare questo default e specificare, tramite l'opzione **HASH IS**, come valori della funzione hash i valori dell'espressione indicata in questa opzione. L'espressione deve però assumere come valori solo interi positivi e deve fare riferimento ad una o più tra le colonne del cluster.

**Esempio 7.6** I seguenti comandi definiscono, rispettivamente, un cluster NolCli di tipo index ed un cluster NolCliH di tipo hash. Entrambi i cluster hanno come chiave un'unica colonna cod, di tipo DECIMAL(4).

```
CREATE CLUSTER NolCli (cod DECIMAL(4));

CREATE CLUSTER NolCliH (cod DECIMAL(4)) HASHKEYS 10;
```

Dato che l'opzione HASHKEYS ha valore 10, il numero di valori generati dalla funzione hash è 11. Come funzione hash viene utilizzata quella di default fornita dal sistema.  $\square$

#### 7.3.4.2 Definizione di indici

Il comando per la creazione di un indice ha il seguente formato:

```
CREATE INDEX <nome indice>
ON {<nome relazione>(<lista nomi colonne>) |
    CLUSTER <nome cluster>}
[ {ASC | DESC}];
```

dove:

- <nome indice> è il nome dell'indice che viene creato.
- La clausola ON specifica l'oggetto su cui è allocato l'indice. Tale oggetto può essere: una relazione, ed in tal caso devono essere specificati i nomi delle colonne su cui l'indice è allocato, oppure un cluster, ed in tal caso non è necessario specificare alcuna colonna in quanto l'indice viene allocato automaticamente sulle colonne chiave del cluster. L'indice può essere allocato su più colonne; i valori della chiave su cui l'indice è costruito sono ottenuti come concatenazione di tutti i valori delle colonne su cui l'indice è allocato. Normalmente esiste un limite al numero di colonne su cui un singolo indice può essere allocato (in Oracle ad esempio tale limite è 16).
- Le opzioni (mutuamente esclusive) ASC e DESC specificano rispettivamente se i valori della chiave dell'indice devono essere ordinati in modo crescente o decrescente. ASC è l'opzione di default.

**Esempio 7.7** I seguenti comandi definiscono un indice idxCliente sull'attributo codCli della relazione Noleggio ed un indice idxNol sul cluster NolCli definito nell'Esempio 7.6.

```
CREATE INDEX idxCliente ON Noleggio (codCli);

CREATE INDEX idxNol ON CLUSTER NolCli;
```

Supponiamo che la relazione `Noleggio` sia inserita nel cluster `NolCli`, come verrà mostrato nell'Esempio 7.8. Rispetto alla classificazione introdotta nella Tabella 7.1, per la relazione `Noleggio` entrambi gli indici sono esempi di indici su singolo attributo e su chiave secondaria. L'indice `idxCliente` sarà clusterizzato (perché definito sul cluster `NolCli`), mentre l'indice `idxCliente no` (o non necessariamente). La scelta se mantenere tali indici come densi o come sparsi viene presa dal sistema, anche se, come discusso, gli indici non clusterizzati sono sicuramente densi e quelli clusterizzati generalmente sparsi.  $\square$

#### 7.3.4.3 Inserimento di relazioni in un cluster

Un cluster può includere una o più relazioni. Nel caso di una singola relazione, il cluster è usato per raggruppare le tuple della relazione aventi lo stesso valore per le colonne chiave del cluster. Nel caso di più relazioni, il cluster viene usato per effettuarne il co-clustering, cioè raggruppare le tuple di tutte le relazioni aventi lo stesso valore per la chiave del cluster, permettendo pertanto esecuzioni efficienti per operazioni di join sulle colonne che sono parte della chiave del cluster (vedi Paragrafo 7.2.2). Una relazione deve essere inserita nel cluster al momento della creazione; pertanto il comando `CREATE TABLE` include un'ulteriore clausola `CLUSTER` che permette di specificare il cluster in cui inserire la relazione.

**Esempio 7.8** Supponiamo di voler inserire nel cluster `NolCli` definito nell'Esempio 7.6 le relazioni `Noleggio` e `Cliente`. Alla chiave `cod` del cluster vogliamo far corrispondere, in entrambe le relazioni, la colonna `codCli`, il cui dominio, `DECIMAL(4)`, coincide con il dominio della chiave del cluster. I comandi per la definizione delle due relazioni sono i seguenti (per brevità, nelle definizioni delle relazioni abbiamo omesso gran parte degli attributi):

```
CREATE TABLE Noleggio (colloc DECIMAL(4) NOT NULL,
                      codCli DECIMAL(4),
                      ...)
CLUSTER NolCli (codCli);

CREATE TABLE Cliente (codCli DECIMAL(4) PRIMARY KEY,
                     ...)
CLUSTER NolCli (codCli);  $\square$ 
```

Come possiamo notare dall'esempio precedente, i nomi delle colonne delle relazioni su cui si esegue il clustering non devono necessariamente avere lo stesso nome della colonna del cluster. Alla colonna del cluster, che ha nome `cod`, vengono fatte corrispondere le colonne di nome `codCli` nelle relazioni `Noleggio` e `Cliente`. Le colonne del cluster e quelle delle relazioni devono essere però lo stesso numero ed i domini corrispondenti devono essere compatibili.

## Capitolo 2

# Modello relazionale

Il modello relazionale, sebbene non sia stato il modello dei dati usato nei primi DBMS, ha rivoluzionato, sin dalla sua definizione nel 1970 ad opera di Codd, il mondo delle basi di dati ed è rapidamente divenuto il modello dei dati più diffuso, oggi comunemente adottato dalla larga maggioranza dei DBMS disponibili a livello commerciale. Il modello relazionale è basato su una semplice struttura dati – la *relazione* – ed è caratterizzato da precise basi matematiche, avendo come fondamento teorico la teoria degli insiemi e la logica dei predicati del primo ordine. I principali vantaggi del modello relazionale rispetto ai modelli dei dati di precedente definizione possono essere individuati nella semplice rappresentazione dei dati e nella facilità con cui possono essere espresse interrogazioni anche complesse. La semplicità del modello relazionale, infatti, ha reso possibile lo sviluppo di linguaggi *dichiarativi* e semplici da usare per specificare ricerche sui dati, che consentono l'accesso ai dati da parte di utenti ed applicazioni anche in base a modalità non previste a priori. Le interrogazioni sulle relazioni possono essere espresse in due formalismi di base:

- **algebra relazionale**, in cui le interrogazioni sono espresse applicando operatori specializzati alle relazioni;
- **calcolo relazionale**, in cui le interrogazioni sono espresse per mezzo di formule logiche che devono essere verificate dalle tuple ottenute come risposta all'interrogazione.

Un risultato teorico stabilisce che, sotto determinate assunzioni, i due formalismi hanno lo stesso potere espressivo: ognuno di essi può esprimere qualsiasi interrogazione che l'altro formalismo può esprimere, ma non di più. Questi formalismi, ovviamente, non costituiscono veri linguaggi per basi di dati, in quanto mancano delle operazioni di modifica alle relazioni e di numerose funzionalità utili nell'uso pratico. Le operazioni di modifica e varie funzionalità addizionali saranno illustrate nella trattazione del linguaggio SQL (vedi Capitolo 3), linguaggio standard per basi di dati basate sul modello relazionale, sviluppato sulla base di algebra e calcolo relazionale.

Questo capitolo, oltre ad introdurre il modello dei dati relazionale, illustra gli aspetti fondamentali relativi ai linguaggi di interrogazione per basi di dati relazionali, presentando le caratteristiche principali di algebra e calcolo relazionale.

## 2.1 Modello dei dati

In questo paragrafo introdurremo le principali nozioni alla base del modello dei dati relazionale.

### 2.1.1 Relazioni

Il concetto alla base del modello relazionale è la *relazione*. Per definire la nozione di relazione è necessario partire dalla nozione di *dominio*. Un dominio è un insieme (anche infinito) di valori.

**Esempio 2.1** L'insieme dei numeri interi è un dominio. L'insieme delle stringhe di caratteri è un dominio. L'insieme  $\{0,1\}$  è un dominio.  $\square$

Nel seguito indicheremo con  $\mathcal{D}$  l'insieme di tutti i domini, che assumeremo contenere i numeri interi (**int**), i numeri reali (**real**), le stringhe (**string**) e le date (**date**). I valori dei domini costituiscono i valori atomici che popoleranno la base di dati, a partire dai quali vengono costruite le *tuple* delle relazioni, effettuando un prodotto cartesiano di tali valori.

**Definizione 2.1 (Prodotto cartesiano)** Siano  $D_1, D_2, \dots, D_k \in \mathcal{D}$   $k$  domini. Il prodotto cartesiano di tali domini, indicato con  $D_1 \times D_2 \times \dots \times D_k$ , è definito come l'insieme  $\{(v_1, v_2, \dots, v_k) \mid v_1 \in D_1, \dots, v_k \in D_k\}$ .  $\diamond$

Gli elementi appartenenti al prodotto cartesiano sono detti *tuple*. Il prodotto cartesiano di  $k$  domini ha *grado*  $k$ .

**Definizione 2.2 (Relazione)** Siano  $D_1, D_2, \dots, D_k \in \mathcal{D}$  domini. Una relazione su  $D_1, D_2, \dots, D_k$  è un sottoinsieme finito del prodotto cartesiano  $D_1 \times D_2 \times \dots \times D_k$ .  $\diamond$

Una relazione, sottoinsieme del prodotto cartesiano di  $k$  domini, ha grado  $k$ . Ogni tupla di una relazione di grado  $k$  ha pertanto  $k$  componenti, una per ogni dominio su cui è definita la relazione cui la tupla appartiene. Dati una relazione  $R$  di grado  $k$ , una tupla  $t \in R$  ed un intero  $i \in \{1, \dots, k\}$ , la notazione  $t[i]$  denota la  $i$ -esima componente di  $t$ . La *cardinalità* di una relazione è il numero di tuple appartenenti alla relazione. È importante notare che mentre i domini che partecipano ad un prodotto cartesiano possono essere insiemi infiniti (questa è anzi la situazione più comune), una relazione è sempre un insieme finito.

In base alle definizioni, una relazione è un insieme di tuple, in quanto tali ordinate al loro interno: l' $i$ -esimo valore di ciascuna tupla proviene dall' $i$ -esimo dominio su cui è definito il prodotto cartesiano; è cioè definito un ordinamento fra i domini su cui è definita la relazione. Viceversa, essendo una relazione un insieme, non è definito alcun ordinamento fra le tuple di una relazione. Inoltre, sempre poiché una relazione è un insieme, le tuple di una relazione sono distinte l'una dall'altra, cioè non vi sono tuple duplicate.

**Esempio 2.2** Dati  $D_1 = \{0, 1, 2\}$  e  $D_2 = \{d, v\}$ , il prodotto cartesiano  $D_1 \times D_2$  ha grado 2 e contiene le seguenti tuple:  $\{(0, d), (0, v), (1, d), (1, v), (2, d), (2, v)\}$ .

Esempi di relazioni, sottoinsiemi del prodotto cartesiano  $D_1 \times D_2$ , sono:  $\{(0, d), (0, v), (1, d)\}$  e  $\{(1, d), (2, v)\}$ . Le due relazioni hanno, rispettivamente, cardinalità 3 e 2. Data la tupla  $t = (0, d)$  appartenente alla prima delle due relazioni precedenti,  $t[1] = 0$  e  $t[2] = d$ .  $\square$

Una formulazione più conveniente del modello relazionale associa un nome, detto *nome di attributo*, ad ogni componente delle tuple in una relazione. La coppia (nome di attributo, dominio) è detta *attributo*. L'uso degli attributi permette di denotare le componenti di ogni tupla *per nome* piuttosto che *per posizione*, come è invece necessario nella formulazione precedente della nozione di relazione. Un altro importante vantaggio nell'uso degli attributi è di fornire maggiori informazioni semantiche sulle proprietà che ogni componente delle tuple in una relazione modella. In nome di una relazione e l'insieme dei suoi attributi ne costituiscono lo *schema*, come specificato dalla seguente definizione.

**Definizione 2.3 (Schema di relazione)** Siano  $R$  un nome di relazione,  $\{A_1, A_2, \dots, A_n\}$  un insieme di nomi di attributi,  $\text{dom} : \{A_1, A_2, \dots, A_n\} \rightarrow \mathcal{D}$  una funzione totale che associa ad ogni nome di attributo in  $\{A_1, A_2, \dots, A_n\}$  il corrispondente dominio. La coppia  $(R(A_1, A_2, \dots, A_n), \text{dom})$  è uno schema di relazione.  $U_R$  denota l'insieme dei nomi di attributi di  $R$ , cioè  $\{A_1, A_2, \dots, A_n\}$ .  $\diamond$

Nel seguito, quando saranno chiare dal nome dell'attributo o non saranno rilevanti per la trattazione, ometteremo le informazioni relative al dominio degli attributi ed indicheremo lo schema di una relazione  $R$  semplicemente con  $R(A_1, A_2, \dots, A_n)$ . Un insieme di schemi di relazione costituisce uno *schema di base di dati*, come specificato dalla definizione seguente.

**Definizione 2.4 (Schema di base di dati)** Siano  $S_1, S_2, \dots, S_n$  schemi di relazioni, con nomi di relazione diversi,  $\mathcal{S} = \{S_1, S_2, \dots, S_n\}$  è detto schema di base di dati.  $\diamond$

Le definizioni seguenti introducono le nozioni di tupla e di relazione sulla base della precedente definizione di schema di relazione.

**Definizione 2.5 (Tupla e relazione)** Sia  $(R(A_1, A_2, \dots, A_n), \text{dom})$  uno schema di relazione. Una tupla  $t$  definita su  $R$  è un insieme di funzioni totali  $f_1, f_2, \dots, f_n$ , dove  $f_i : A_i \rightarrow \text{dom}(A_i)$ ,  $i = 1, \dots, n$ , associa all'attributo di nome  $A_i$  un valore del dominio di tale attributo. Una relazione definita su uno schema di relazione è un insieme finito di tuple definite su tale schema; tale relazione è anche detta istanza dello schema.  $\diamond$

Dato uno schema di relazione  $(R(A_1, A_2, \dots, A_n), \text{dom})$ , una tupla  $t$  su tale schema può essere rappresentata tramite la notazione  $[A_1 : v_1, A_2 : v_2, \dots, A_n : v_n]$  dove  $v_i$ ,  $i = 1, \dots, n$ , è un valore appartenente a  $\text{dom}(A_i)$ . La notazione  $t[A_i]$ ,

$i = 1, \dots, n$ , denota il valore dell'attributo  $A_i$  della tupla  $t$  (cioè  $v_i$ ). Analogamente, dato un insieme  $A \subseteq U_R$  di attributi,  $t[A]$  denota la tupla costituita dalle componenti di  $t$  i cui attributi appartengono all'insieme  $A$ .

Dalle definizioni precedenti possiamo notare che, dato uno schema di relazione, è in realtà possibile definire più relazioni che siano istanze di tale schema; pertanto può essere utile distinguere tra il nome di relazione usato in uno schema di relazione ed il nome (i nomi) della relazione (relazioni) istanza (istanze) di tale schema. Nell'uso comune, tuttavia, e nel resto della trattazione, il nome di relazione utilizzato nello schema è usato per denotare anche la relazione istanza dello schema. Pertanto ogni schema di relazione ha, ad ogni dato istante, una sola relazione istanza. Ovviamente tale istanza cambia il suo *stato* nel tempo a seguito di modifiche sui dati.

**Esempio 2.3** Consideriamo la relazione `Film` della base di dati della videoteca illustrata nella Figura 1.1 (qui riprodotta come Figura 2.1). In tale figura, ed in tutto il testo, le relazioni sono rappresentate come tabelle in cui ad ogni colonna è associato un nome di attributo ed ogni riga corrisponde ad una tupla. Lo schema della relazione `Film` è:

```

Film(titolo,regista,anno,genere,valutaz)
dom(titolo) = dom(regista) = dom(genere) = string
dom(anno) = int, dom(valutaz) = real.

```

La relazione è un sottoinsieme di:  $\text{string} \times \text{string} \times \text{int} \times \text{string} \times \text{real}$ . Una tupla appartenente a tale relazione è: `[titolo:'ed wood',regista:'tim burton',anno:1994,genere:'drammatico',valutaz:4.00]`.

Consideriamo ora le relazioni di Figura 2.2, sempre relative alla base di dati della videoteca. Il nome della prima relazione, che modella informazioni sui clienti della videoteca, è `Cliente` e lo schema è:

```

Cliente(codCli, nome, cognome, telefono, dataN, residenza)
dom(nome) = dom(cognome) = dom(telefono) = dom(residenza) = string
dom(codCli) = int, dom(dataN) = date.

```

Tali attributi rappresentano, rispettivamente, il codice identificativo, il nome, il cognome, il numero di telefono, la data di nascita e l'indirizzo di residenza del cliente.  $\square$

### 2.1.2 Valori nulli

Un aspetto importante nella modellazione dei dati riguarda il fatto che non sempre sono disponibili tutte le informazioni sulle entità del dominio applicativo che vengono rappresentate nella base di dati. In termini del modello relazionale questo vuol dire che alcune tuple possono non avere un valore per un qualche attributo.

Non permettere l'inserzione nella base di dati di queste tuple sarebbe troppo rigido rispetto alle più comuni esigenze applicative. L'approccio adottato è quello di introdurre un valore speciale, detto *valore nullo*, il quale denota la mancanza di un valore. Usare ad esempio 0 per indicare un valore nullo non è una soluzione appropriata in quanto 0 è un valore legale per vari domini (ad esempio quelli numerici) e quindi il suo uso non permetterebbe di distinguere il caso in cui 0 sia effettivamente il valore dell'attributo dal caso in cui 0 indichi il valore nullo. Nella trattazione assumiamo di denotare il valore nullo con il simbolo ‘?’ (sono tuttavia possibili altre convenzioni). Il valore nullo è un valore ammissibile per ogni dominio. I linguaggi come SQL permettono comunque di specificare nella definizione di una relazione quali attributi non possono mai assumere valore nullo (vedi Capitolo 3). La presenza di valori nulli introduce interessanti questioni riguardanti l'esecuzione di interrogazioni sui dati. Tali questioni saranno trattate approfonditamente nel Capitolo 3 nel contesto del linguaggio SQL. Nell'esempio seguente, ed in tutto il testo, useremo la convenzione di evidenziare negli schemi con un circoletto gli attributi che possono assumere valori nulli.

**Esempio 2.4** Consideriamo la relazione `Noleggio` illustrata nella Figura 2.2, che modella i noleggi effettuati dai clienti della videoteca. Lo schema della relazione `Noleggio` è:

```
Noleggio(colloc,dataNol,codCli,dataRest.)
dom(codCli) = dom(colloc) = int, dom(dataNol)= dom(dataRest)= date.
```

Tali attributi rappresentano, rispettivamente, la collocazione del video noleggiato, la data di inizio noleggio, il codice identificativo del cliente che effettua il noleggio e la data di restituzione del video. Le tuple corrispondenti ai noleggi in corso hanno valore nullo per l'attributo `dataRest`.  $\square$

### 2.1.3 Chiavi

Una *chiave* di una relazione è un insieme di attributi che distingue fra loro le tuple della relazione, come formalizzato dalla seguente definizione.

**Definizione 2.6 (Chiave e super-chiave)** Sia  $R(A_1, \dots, A_n)$  uno schema di relazione. Un insieme  $X \subseteq U_R$  di attributi di  $R$  è chiave di  $R$  se verifica entrambe le seguenti proprietà:

1. qualsiasi sia lo stato di  $R$ , non esistono due tuple distinte di  $R$  che abbiano lo stesso valore per tutti gli attributi in  $X$ ;
2. nessun sottoinsieme proprio<sup>1</sup> di  $X$  verifica la proprietà (1).

---

<sup>1</sup>Dati due insiemi  $S$  ed  $S'$ ,  $S'$  è sottoinsieme proprio di  $S$  (indicato come  $S' \subset S$ ) se tutti gli elementi di  $S'$  appartengono ad  $S$  ed esiste almeno un elemento di  $S$  che non appartiene ad  $S'$ .

## Film

titolo	regista	anno	genere	valutaz
underground	emir kusturica	1995	drammatico	3.20
edward mani di forbice	tim burton	1990	fantastico	3.60
nightmare before christmas	tim burton	1993	animazione	4.00
ed wood	tim burton	1994	drammatico	4.00
mars attacks	tim burton	1996	fantascienza	3.00
il mistero di sleepy hollow	tim burton	1999	horror	3.50
big fish	tim burton	2003	fantastico	3.10
la sposa cadavere	tim burton	2005	animazione	3.50
la fabbrica di cioccolato	tim burton	2005	fantastico	4.00
io non ho paura	gabriele salvatores	2003	drammatico	3.50
nirvana	gabriele salvatores	1997	fantascienza	3.00
mediterraneo	gabriele salvatores	1991	commedia	3.80
pulp fiction	quentin tarantino	1994	thriller	3.50
le iene	quentin tarantino	1992	thriller	4.00

## Video

colloc	titolo	regista	tipo
1111	underground	emir kusturica	v
1112	underground	emir kusturica	d
1113	big fish	tim burton	v
1114	big fish	tim burton	d
1115	edward mani di forbice	tim burton	d
1116	nightmare before christmas	tim burton	v
1117	nightmare before christmas	tim burton	d
1118	ed wood	tim burton	d
1119	mars attacks	tim burton	d
1120	il mistero di sleepy hollow	tim burton	d
1121	la sposa cadavere	tim burton	d
1122	la fabbrica di cioccolato	tim burton	d
1123	la fabbrica di cioccolato	tim burton	d
1124	io non ho paura	gabriele salvatores	d
1125	nirvana	gabriele salvatores	d
1126	mediterraneo	gabriele salvatores	d
1127	pulp fiction	quentin tarantino	v
1128	pulp fiction	quentin tarantino	d
1129	le iene	quentin tarantino	d

Figura 2.1: Base di dati relativa alla videoteca (1)

**Cliente**

codCli	nome	cognome	telefono	dataN	residenza
6610	anna	rossi	01055664433	05-Ott-1979	via scribanti 16 16131 genova
6635	paola	bianchi	0104647992	12-Apr-1976	via dodecaneso 35 16146 genova
6642	marco	verdi	3336745383	16-Ott-1972	via lagustena 35 16131 genova

**Noleggio**

colloc	dataNol	codCli	dataRest
1111	01-Mar-2006	6635	02-Mar-2006
1115	01-Mar-2006	6635	02-Mar-2006
1117	02-Mar-2006	6635	06-Mar-2006
1118	02-Mar-2006	6635	06-Mar-2006
1111	04-Mar-2006	6642	05-Mar-2006
1119	08-Mar-2006	6635	10-Mar-2006
1120	08-Mar-2006	6635	10-Mar-2006
1116	08-Mar-2006	6642	09-Mar-2006
1118	10-Mar-2006	6642	11-Mar-2006
1121	15-Mar-2006	6635	18-Mar-2006
1122	15-Mar-2006	6635	18-Mar-2006
1113	15-Mar-2006	6635	18-Mar-2006
1129	15-Mar-2006	6635	20-Mar-2006
1119	15-Mar-2006	6642	16-Mar-2006
1126	15-Mar-2006	6610	16-Mar-2006
1112	16-Mar-2006	6610	18-Mar-2006
1114	16-Mar-2006	6610	17-Mar-2006
1128	18-Mar-2006	6642	20-Mar-2006
1124	20-Mar-2006	6610	21-Mar-2006
1115	20-Mar-2006	6610	21-Mar-2006
1124	21-Mar-2006	6642	22-Mar-2006
1116	21-Mar-2006	6610	?
1117	21-Mar-2006	6610	?
1127	22-Mar-2006	6635	?
1125	22-Mar-2006	6635	?
1122	22-Mar-2006	6642	?
1113	22-Mar-2006	6642	?

Figura 2.2: Base di dati relativa alla videoteca (2)

*Un insieme di attributi che verifica la proprietà (1), ma non la proprietà (2), è detto super-chiave di R.*  $\diamond$

Una relazione può avere più di un insieme  $S$  di attributi che verificano le proprietà (1) e (2) della precedente definizione. In tal caso viene utilizzato a volte il termine *chiavi candidate* per indicare tutte le chiavi. Notiamo inoltre che una relazione ha sicuramente almeno una chiave. Infatti, poiché una relazione, essendo un insieme di tuple, non contiene tuple duplicate, è sicuramente possibile distinguere una tupla da ogni altra tupla della relazione considerando *tutti* i suoi attributi. L'insieme di attributi in  $U_R$ , infatti, soddisfa sempre la proprietà (1) precedente.

Nel caso in cui una relazione abbia più chiavi candidate, è possibile selezionare tra queste una *chiave primaria*. Le altre chiavi vengono a volte indicate come *chiavi alternative*. Intuitivamente, come discuteremo nel Paragrafo 2.1.4, possiamo fare riferimento ad una tupla mediante i valori dei campi di una sua chiave. In linea di principio, possiamo usare qualunque chiave, non solo la primaria, per riferirci ad una tupla. Usare la chiave primaria è però preferibile perché è ciò per cui il DBMS ottimizza le operazioni (vedi Capitolo 7). Come discuteremo nel Capitolo 6, è quindi importante selezionare oculatamente la chiave primaria. Un criterio nella scelta della chiave primaria è scegliere tra le chiavi candidate quella che contiene il minor numero di attributi o quella più frequentemente usata nelle interrogazioni. Un insieme di attributi selezionato come chiave primaria deve inoltre verificare la proprietà che nessuno degli attributi possa assumere valore nullo. I linguaggi come SQL permettono comunque di specificare nella definizione di una relazione quali attributi costituiscono la chiave primaria e quali eventuali chiavi alternative (vedi Capitolo 3).

Nell'esempio seguente ed in tutto il testo utilizzeremo la convenzione di sottolineare gli attributi chiave primaria di una relazione. Eventuali altre chiavi candidate verranno indicate in italico, quando la notazione non crea ambiguità, o elencate a parte, altrimenti. Come evidenziato dall'esempio seguente, le chiavi delle relazioni vengono individuate mediante esame del dominio applicativo e dei relativi vincoli.

**Esempio 2.5** Consideriamo la base di dati della videoteca costituita dalle relazioni nelle Figure 2.1 e 2.2. Un cliente viene identificato attraverso il codice, quindi una chiave della relazione **Cliente** è l'attributo **codCli**. Se assumiamo che non possano esistere clienti con lo stesso nome e cognome, nati lo stesso giorno, una chiave alternativa per **Cliente** è costituita dai tre attributi **nome**, **cognome** e **dataN**. L'attributo **codCli** è preferibile, perché la chiave risulta costituita da un solo attributo. Assumiamo invece che possano esistere clienti che condividono lo stesso numero di telefono, quindi l'attributo **telefono** non è chiave per **Cliente**.

Una chiave della relazione **Film** è data dagli attributi (**titolo**, **regista**). Notiamo, infatti, che possono esistere film con lo stesso titolo, quindi il titolo da solo non permette di identificare un film. Al contrario, assumiamo che uno stesso regista non possa realizzare due film con lo stesso titolo. Poiché assumiamo che

potrebbero in generale essere realizzati due film con lo stesso titolo nello stesso anno,  $(\text{titolo}, \text{anno})$  non viene individuata come chiave alternativa. Notiamo che, ad esempio,  $(\text{titolo}, \text{regista}, \text{anno})$  è una super-chiave per la relazione `Film`.

Per quanto riguarda la relazione `Video`, la collocazione permette di identificare univocamente un video, quindi la chiave è costituita dal solo attributo `colloc`.

Consideriamo infine la relazione `Noleggio`. Se assumiamo che ogni video venga noleggiato (e quindi restituito) al più una volta ogni giorno,  $(\text{colloc}, \text{dataNol})$  e  $(\text{colloc}, \text{dataRest})$  sono chiavi candidate per `Noleggio`. Al contrario, siccome uno stesso cliente può di solito noleggiare (e/o restituire) più video nello stesso giorno e può noleggiare più volte lo stesso video, né  $(\text{codCli}, \text{dataNol})$ , né  $(\text{codCli}, \text{dataRest})$ , né  $(\text{codCli}, \text{colloc})$  sono chiavi per la relazione `Noleggio`. Poiché `dataRest` può assumere valore nullo per i noleggi in corso e le chiavi primarie non possono contenere valori nulli, viene selezionata  $(\text{colloc}, \text{dataNol})$  come chiave primaria per `Noleggio`.

Possiamo quindi riassumere lo schema della base di dati come segue:

```
Cliente(codCli, nome, cognome, telefono, dataN, residenza)
Film(titolo, regista, anno, genere, valutaz.)
Video(colloc, titolo, regista, tipo)
Noleggio(colloc, dataNol, codCli, dataRest.).
```

Lo schema evidenzia anche che per la valutazione di un film sono ammessi valori nulli, in caso tale valutazione non sia nota.  $\square$

#### 2.1.4 Chiavi esterne

Un aspetto importante nella modellazione di una qualsiasi realtà applicativa riguarda la rappresentazione dei collegamenti tra informazioni memorizzate in relazioni diverse, cioè le associazioni tra entità introdotte nel Capitolo 1. Nel modello relazionale, tali collegamenti sono rappresentati tramite l'uso di *chiavi esterne*.

**Definizione 2.7 (Chiave esterna)** Siano  $R$  ed  $R'$  due relazioni, sia  $Y \subseteq U_{R'}$  una chiave per  $R'$  e sia  $X \subseteq U_R$  un insieme di attributi di  $R$  tale che  $Y$  e  $X$  contengano lo stesso numero di attributi e di dominio compatibile.<sup>2</sup>  $X$  è una chiave esterna di  $R$  su  $R'$  se, qualsiasi siano gli stati di  $R$  ed  $R'$ , per ogni tupla  $t$  di  $R$  esiste una tupla  $t'$  di  $R'$  tale che  $t[X] = t'[Y]$ .  $R$  viene detta relazione referente e  $R'$  viene detta relazione riferita.  $\diamond$

Il vincolo d'integrità semantica che assicura che, se una tupla  $t$  di  $R$  fa riferimento, tramite i valori di una chiave esterna, ai valori della chiave di una tupla  $t'$  di  $R'$ ,  $t'$  sia effettivamente presente in  $R'$ , viene indicato con il termine *vincolo di integrità referenziale*.

---

<sup>2</sup>Due domini sono compatibili se sono uguali o se i valori di uno possono essere comunque accettati come valori dell'altro. Ad esempio, interi e reali sono compatibili.

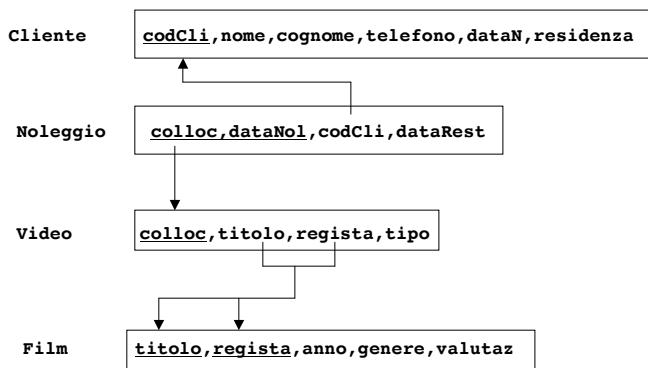


Figura 2.3: Rappresentazione grafica dello schema della base di dati della videoteca

Le chiavi esterne permettono di collegare tra loro tuple di relazioni diverse e costituiscono, pertanto, un meccanismo per realizzare le associazioni. Abbiamo visto nel Capitolo 1 come l'approccio alla modellazione delle associazioni basato su chiavi esterne sia detto *per valore*; infatti una tupla che debba riferire un'altra tupla include tra i suoi attributi degli attributi il cui *valore* è il valore della chiave della seconda tupla.

Nell'esempio seguente ed in tutto il testo useremo la convenzione di indicare, quando questo non crea ambiguità, le chiavi esterne utilizzando il nome della relazione riferita come apice del nome dell'attributo che appartiene alla chiave esterna. Altrimenti, le chiavi esterne e le corrispondenti relazioni riferite verranno elencate a parte. Per evidenziare le chiavi esterne e le corrispondenti relazioni riferite, gli schemi relazionali vengono a volte rappresentati mediante una notazione grafica quale quella presentata nella Figura 2.3 relativamente alla base di dati della videoteca.

**Esempio 2.6** Consideriamo nuovamente la base di dati della videoteca. Ogni video contiene un determinato film. L'informazione del film contenuto nel video è rappresentata dagli attributi **titolo** e **regista** in **Video**, che sono una chiave esterna sulla relazione **Film**. La relazione **Noleggio**, a sua volta, contiene due chiavi esterne: **codCli** su **Cliente**, rappresentante il cliente che ha effettuato il noleggio, e **colloc** su **Video**, rappresentante il video noleggiato. Lo schema con l'indicazione delle chiavi esterne diventa quindi:

```

Cliente(codCli, nome, cognome, telefono, dataN, residenza)
Film(titolo, regista, anno, genere, valutaz.)
Video(colloc, titoloFilm, registaFilm, tipo)
Noleggio(collocVideo, dataNol, codCliCliente, dataRest).

```

Le relazioni nelle Figure 2.1 e 2.2 verificano il vincolo di integrità referenziale. Infatti ogni tupla della relazione **Video** ha come valore degli attributi **titolo** e **regista** valori che sono anche valori della chiave di una qualche tupla della relazione **Film**. Analogamente, ogni tupla della relazione **Noleggio** ha come valore dell'attributo **colloc** un valore che è valore della chiave di una tupla della relazione **Video** e come valore dell'attributo **codCli** un valore che è valore della chiave di una tupla della relazione **Cliente**. Supponiamo, invece, che la relazione **Noleggio** contenga, in aggiunta alla tuple presenti nella Figura 2.2, la tupla:

[**colloc**:1137, **dataNol**:22-Mar-2006, **codCli**:6635, **dataRest**:?].

Tale tupla viola il vincolo d'integrità referenziale in quanto non esiste alcun video con numero di collocazione 1137; pertanto tale tupla riferisce un video non esistente.  $\square$

L'integrità referenziale può essere violata da inserimenti e modifiche (del valore della chiave esterna) nella relazione referente e da cancellazioni e modifiche (del valore della chiave) nella relazione riferita. Data la rilevanza di tale vincolo, i linguaggi per basi di dati quali SQL permettono all'utente, nella definizione di chiavi esterne, di specificare quali azioni eseguire nel caso in cui operazioni di modifica violino tale vincolo (vedi Capitolo 3).

**Esempio 2.7** Consideriamo lo schema dell'Esempio 2.6 e la chiave esterna **codCli** da **Noleggio** su **Cliente**. Possiamo avere violazioni dell'integrità referenziale:

- se inseriamo un nuovo noleggio: l'inserimento in **Noleggio** della tupla [**colloc**:1126, **dataNol**:22-Mar-2006, **codCli**:6638, **dataRest**:?] causa una violazione del vincolo, perché non è presente in **Cliente** una tupla con valore di chiave 6638;
- se modifichiamo il codice del cliente che ha effettuato il noleggio: la modifica del codice di cliente di una tupla della relazione in 6638 causa la violazione del vincolo;
- se cancelliamo un cliente: la cancellazione del cliente con codice 6635 renderebbe invalide 12 tuple della relazione **Noleggio**;
- se modifichiamo il codice di un cliente: la modifica del codice del cliente Paola Bianchi da 6635 a 6638 renderebbe invalide 12 tuple della relazione **Noleggio**.  $\square$

Notiamo che nello schema dell'Esempio 2.6 abbiamo scelto di utilizzare lo stesso nome per gli attributi (chiave e chiave esterna) nelle due relazioni. Tale scelta è comoda perché permette l'utilizzo dell'operazione di join naturale, che vedremo nel Paragrafo 2.2, ma non è strettamente necessaria. Ad esempio, l'attributo che modella il cliente che effettua il noleggio potrebbe chiamarsi **cliente** in **Noleggio** invece di **codCli**. In particolare, gli attributi avranno sicuramente nomi diversi tutte le volte che la relazione referente e la relazione riferita coincidono, cioè la

chiave esterna contiene un riferimento alla relazione stessa. Ad esempio, la relazione `Film` potrebbe contenere come chiave esterna su `Film` stessa una coppia di attributi (`titoloPre`, `registaPre`), contenente titolo e regista del film di cui il film è eventualmente il seguito. Come evidenziato dall'Esempio 2.6, inoltre, una relazione può contenere più chiavi esterne, eventualmente anche sulla stessa relazione. Rimarchiamo inoltre che le chiavi esterne, come del resto le chiavi, devono essere esplicitamente specificate in uno schema di relazione. Il fatto di avere attributi con lo stesso nome e domini compatibili in relazioni diverse non offre di per sé alcuna garanzia relativamente al mantenimento dell'integrità referenziale. Notiamo infine che, se non esplicitamente impedito mediante la specifica di un apposito vincolo, le chiavi esterne possono assumere valore nullo.

Per concludere la trattazione del modello dei dati relazionale, sottolineiamo che i vincoli di integrità che il modello permette di rappresentare direttamente, e che abbiamo discusso in questo paragrafo, non sono sufficienti a garantire che il contenuto della base di dati rispecchi in modo fedele le informazioni del dominio applicativo da rappresentare. Discuteremo più in dettaglio nel Capitolo 3 (vedi Paragrafo 3.4) altre categorie di vincoli, oltre ai vincoli di dominio, di obbligatorietà, di chiave e di chiave esterna qui discussi, cui il modello relazionale non offre diretto supporto.

## 2.2 Algebra relazionale

L'algebra relazionale è costituita da varie operazioni per la manipolazione delle relazioni. Più precisamente, l'algebra è composta da cinque operazioni di base: *proiezione*, *selezione*, *prodotto cartesiano*, *unione* e *differenza*. Queste operazioni definiscono completamente l'algebra relazionale. Ogni operazione ha come argomento una o due relazioni (a seconda dell'operazione) e restituisce come risultato una relazione; è pertanto possibile applicare un'operazione al risultato di un'altra operazione (proprietà detta di *chiusura*). Esistono operazioni addizionali, che possono essere espresse in termini delle cinque operazioni di base. Tali operazioni non estendono il potere espressivo dato dalle cinque operazioni di base, ma sono utili come abbreviazione. Tra le operazioni addizionali, l'operazione detta di *join* è la più rilevante. È da notare che la definizione delle varie operazioni è leggermente diversa a seconda che si consideri la definizione del modello relazionale per posizione o la definizione per nome (vedi Paragrafo 2.1); la differenza principale consiste nel modo in cui vengono denotate le componenti delle tuple in alcune delle operazioni che hanno una o più di tali componenti come ulteriore argomento. Nella trattazione seguente, considereremo la definizione delle operazioni in base alla notazione per nome. In riferimento a tale notazione, viene introdotta un'ulteriore operazione, di *ridenominazione*, che permette di modificare i nomi degli attributi.

## Capitolo 5

# Progettazione concettuale

Progettare una base di dati significa definire in modo preciso il suo contenuto e come questo debba essere rappresentato tramite il modello dei dati scelto. L'obiettivo è quello di arrivare ad uno schema che modelli in modo completo e corretto la realtà d'interesse e risponda nel modo più efficiente possibile alle esigenze di utenti ed applicazioni. La progettazione di una base di dati si inserisce nel quadro più ampio della progettazione di un sistema informativo che prevede, oltre al progetto della base di dati, quello delle applicazioni che di esso fanno parte. In questo capitolo, ci concentreremo sulla progettazione della base di dati, rimandando il lettore a testi di ingegneria del software per gli aspetti connessi alla progettazione delle applicazioni.

La progettazione di una base di dati è uno dei compiti maggiormente strategici e delicati dell'intero progetto di un sistema informativo, poichè da essa dipende il buon funzionamento dell'intero sistema. Tale attività ha subito un'evoluzione nel corso del tempo, man mano che i sistemi di gestione dati diventavano sempre più utilizzati e complessi ed aumentava la quantità e la tipologia delle informazioni da essi gestite. Oggi, la progettazione di basi di dati avviene secondo un approccio sistematico, così come accade da tempo per la progettazione del software, il cui elemento centrale è una metodologia di progettazione che consente di dominare l'inerente complessità di tale compito ed ottenere un risultato in grado di soddisfare i requisiti specificati sia in termini di funzionalità sia di qualità.

Un primo passo in questa direzione consiste nel suddividere il processo di progettazione in una serie di sotto-fasi, ognuna con delle finalità ben precise, meno complesse del compito originario. La buona riuscita delle varie fasi della progettazione è strettamente connessa, come vedremo nei paragrafi successivi, alla possibilità di disporre di adeguati strumenti di supporto. Ogni fase deve inoltre prevedere delle verifiche di qualità, secondo una serie di criteri e con un insieme di strumenti che dipendono dalla specifica fase. Più precisamente, la progettazione di una base di dati è organizzata come un processo incrementale costituito da tre passi principali a cui è aggiunta una fase iniziale di *raccolta ed analisi dei requisiti*. Il primo passo è la *progettazione concettuale*, il cui compito è generare una prima rappresentazione ad alto livello della base di dati che stiamo sviluppando, del tutto indipendente dalla sua implementazione. La progettazione concettuale è seguita dalla *progettazione logica* in cui viene generata una rappresentazione della

base di dati direttamente implementabile in uno specifico DBMS. Infine, nella fase di *progettazione fisica* vengono definite e preciseate le strutture di memorizzazione fisica dei dati e le eventuali strutture ausiliarie di accesso.

In questo capitolo, dopo aver illustrato con maggiore dettaglio le fasi della progettazione, ci concentreremo sulla progettazione concettuale. In particolare, descriveremo il modello Entità-Relazione, che è oggi il modello maggiormente utilizzato per la fase di progettazione concettuale. La progettazione logica sarà invece oggetto del capitolo successivo, mentre alcuni aspetti legati alla progettazione fisica saranno illustrati nel Capitolo 7.

### 5.1 Fasi della progettazione

Vediamo ora più in dettaglio le fasi di cui è composta la progettazione di una base di dati. Tutte e tre le fasi di progettazione vera e propria (cioè progettazione concettuale, logica e fisica) sono basate sull'utilizzo di un opportuno *modello* attraverso cui viene generata una rappresentazione della base di dati ad un determinato livello di astrazione. A questa fase si aggiunge una fase preliminare di raccolta ed analisi dei requisiti, il cui significato è di seguito precisato:

- **Raccolta ed analisi dei requisiti.** In questa fase vengono definite informalmente le caratteristiche che la base di dati dovrà possedere. Le specifiche generate in questa fase riguardano: *(i)* i requisiti informativi, ovvero le caratteristiche e la tipologia dei dati che vogliamo rappresentare nella base di dati; *(ii)* i requisiti rispetto alla tipologia di operazioni che dovranno essere svolte sui dati e la loro frequenza, ovvero il *carico di lavoro*, illustrato più in dettaglio nel Capitolo 6; *(iii)* i requisiti sui vincoli d'integrità e di autorizzazione, ovvero le proprietà che ai dati devono essere assicurate, in termini di correttezza e protezione; *(iv)* informazioni circa la popolosità della base di dati, ovvero il *volume dei dati*. Le informazioni sul volume dei dati saranno utilizzate in fase di progettazione logica (vedi Capitolo 6) e fisica (vedi Capitolo 7) per ottimizzare gli schemi generati. Sebbene nel corso degli anni siano state proposte alcune metodologie a supporto di questa fase, essa è principalmente condotta in modo informale, tramite interviste con gli utenti ed analisi delle applicazioni e delle basi di dati esistenti, della normativa inerente il dominio d'interesse e dell'ambiente operativo. Il risultato è un documento in linguaggio naturale di specifica dei requisiti.
- **Progettazione concettuale.** Lo scopo di questa fase è generare una prima rappresentazione formale e ad alto livello del contenuto informativo della base di dati, partendo dal documento di specifica dei requisiti prodotto nella fase precedente. Tale formalizzazione è basata sull'utilizzo di un opportuno *modello concettuale*, di un modello cioè che consente di descrivere ad alto livello le informazioni che dovranno essere memorizzate nella base di dati e le loro associazioni, trascurando gli aspetti implementativi. Tale rappresentazione, ottenuta solitamente tramite linguaggi grafici, è del tutto indipen-

dente dal modello logico che sarà utilizzato per la realizzazione della base di dati e costituisce una prima rappresentazione non ambigua del contenuto della base di dati. Il risultato di questa fase è lo *schema concettuale* della base di dati, sviluppato utilizzando il modello concettuale prescelto, più una documentazione a corredo di tale schema, contenente ulteriori informazioni, quali vincoli d'integrità non direttamente rappresentabili tramite lo schema concettuale (vedi paragrafi successivi per ulteriori dettagli). Durante questa fase viene inoltre condotta una verifica di qualità sugli schemi generati, che può portare ad un parziale ridisegno degli stessi.

- **Progettazione logica.** In questa fase viene prodotto, a partire dall'output della fase precedente, lo schema logico della base di dati. In fase di progettazione logica, è quindi necessario conoscere il modello dei dati del DBMS che verrà utilizzato per implementare la base di dati (ad esempio, relazionale, orientato ad oggetti, ecc.). In questa fase vengono inoltre specificati, mediante il DDL del DBMS target, eventuali vincoli d'integrità, identificati in sede di progettazione concettuale in base alle esigenze evidenziate dalla fase di raccolta ed analisi dei requisiti. Vengono inoltre definite opportune autorizzazioni se il dominio applicativo richiede un accesso differenziato ai dati. Il risultato è lo schema logico della base di dati nel DBMS target.
- **Progettazione fisica.** In questa fase vengono definiti e precisati alcuni dettagli circa la memorizzazione fisica dei dati. Ad esempio, in base alle informazioni sul carico di lavoro contenute nel documento di specifica dei requisiti, possiamo decidere di definire delle strutture ausiliarie di accesso per velocizzare alcune interrogazioni (vedi Capitolo 7 per ulteriori dettagli). Il risultato di questa fase è lo schema fisico della base di dati.

A valle della fase di progettazione logica può essere condotta, tramite opportuni strumenti formali, un'analisi di qualità dello schema logico ottenuto, che può portare ad una parziale ristrutturazione dello schema. Tale attività, nel caso del modello relazionale, prende il nome di *normalizzazione* (vedi Capitolo 6).

La Figura 5.1 riassume le varie fasi della progettazione ed i loro output per un DBMS relazionale, dove il tratteggio indica una fase opzionale.

È bene notare che nella pratica le fasi della progettazione non seguono un processo strettamente sequenziale, in quanto il risultato di una fase può richiedere una parziale riesecuzione delle fasi precedenti. Inoltre, l'output di ogni fase viene di solito generato mediante un processo iterativo che raffina lo schema ottenuto (fisico, logico o concettuale) fino ad arrivare allo schema finale. Infine, sottolineiamo come l'utilizzo di un modello appropriato per le diverse fasi della progettazione è fondamentale per il buon esito dell'attività di progettazione. Ad esempio, nel caso della progettazione concettuale oggetto di questo capitolo, poter disporre di un modello concettuale consente di verificare in modo non ambiguo le caratteristiche della base di dati prima della sua effettiva implementazione, diminuendo quindi di molto i costi necessari per correggere errori di modellazione nello schema

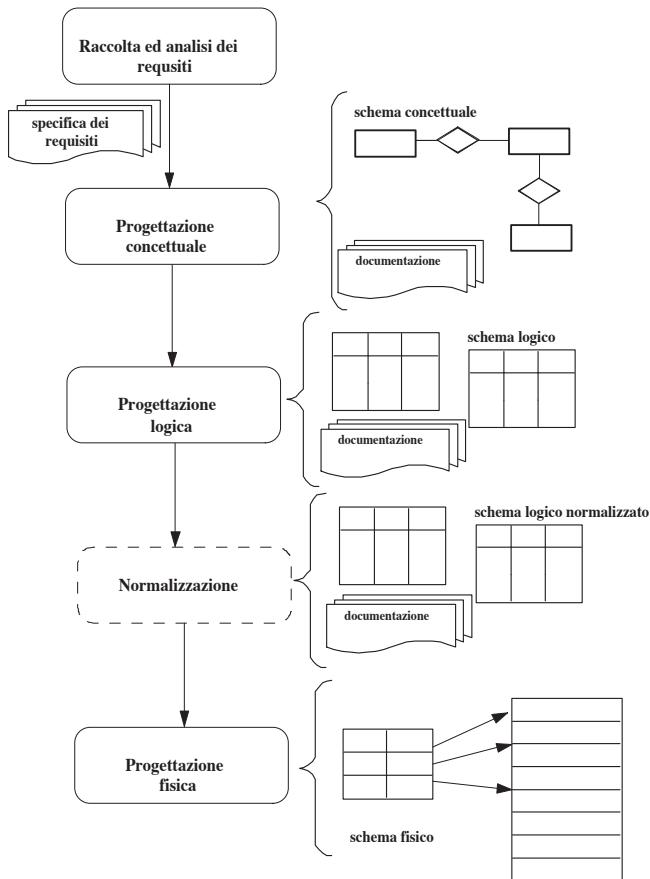


Figura 5.1: Fasi della progettazione di una base di dati

finale. Inoltre, la rappresentazione formale generata tramite il modello costituisce uno strumento attraverso cui comunicare sia con i futuri utenti della base di dati sia con gli addetti alle fasi successive e rappresenta anche una buona fonte di documentazione per eventuali modifiche o ristrutturazioni della base di dati.

## 5.2 Modello Entità-Relazione

Il *modello Entità-Relazione* (nel seguito denotato con *modello ER*), introdotto nel 1976 da Peter Chen, è attualmente uno dei modelli più utilizzati nell'ambito della progettazione concettuale di basi di dati, grazie alla sua semplicità e ricchezza semantica. Il modello possiede una tipologia di rappresentazione dei dati basata su un utilizzo congiunto di metodi grafici e linguistici. Nel seguito, indicheremo



Figura 5.2: Entità ed associazioni

con il termine *diagramma ER* (o *schema ER*) la rappresentazione grafica di uno schema concettuale ER.

Nella formulazione originaria, il modello ER includeva soltanto i concetti di *entità*, *relazione* ed *attributo*. Successivamente, il modello è stato esteso con costrutti quali gli attributi composti e le gerarchie di generalizzazione. Nei paragrafi successivi illustreremo in dettaglio i costrutti di base del modello ER e le estensioni sopra citate.

### 5.2.1 Costrutti di base

Gli elementi costitutivi del modello ER sono: entità, relazioni ed attributi.

Un'*entità* rappresenta una collezione di oggetti della realtà che vogliamo modellare (quali ad esempio clienti, film e video) che possiedono caratteristiche comuni. Gli oggetti appartenenti ad una certa entità rappresentano le *istanze* dell'entità. Graficamente, un'entità viene rappresentata tramite un rettangolo, contenente il suo nome, come illustrato nella Figura 5.2.

Una *relazione* (o *associazione*) rappresenta un legame logico tra più entità. Nel seguito, utilizzeremo il termine associazione per denotare una relazione, per evitare ambiguità con il concetto di relazione del modello relazionale. Le istanze di un'associazione denotano combinazioni di istanze delle entità che prendono parte all'associazione. Graficamente un'associazione viene rappresentata tramite un rombo, contenente il suo nome, connesso alle entità che partecipano all'associazione, come illustrato nella Figura 5.2.

**Esempio 5.1** L'associazione della Figura 5.2 modella il legame che intercorre tra i clienti di una videoteca ed i film che consigliano. Ogni istanza dell'associazione **Consiglia** è una coppia  $(c, f)$ , dove  $c$  è un'istanza dell'entità **Cliente** ed  $f$  è un'istanza dell'entità **Film**. La coppia  $(c, f)$  indica che il cliente rappresentato dall'istanza  $c$  ha consigliato il film rappresentato dall'istanza  $f$ .  $\square$

Formalmente, le istanze di un'associazione sono un sottoinsieme del prodotto cartesiano degli insiemi di istanze di ogni entità che partecipa all'associazione. Notiamo come questo implichì che nell'insieme di istanze di un'associazione non possano esistere duplicati. Vedremo, nei paragrafi successivi, come ciò abbia delle conseguenze rilevanti sulla scelta dei costrutti più idonei per modellare la realtà applicativa d'interesse.

Le associazioni sono classificate in base al loro *grado*, cioè al numero di entità che mettono in relazione, in: *associazioni unarie*, cioè associazioni che mettono in relazione istanze della stessa entità, *associazioni binarie*, cioè associazioni

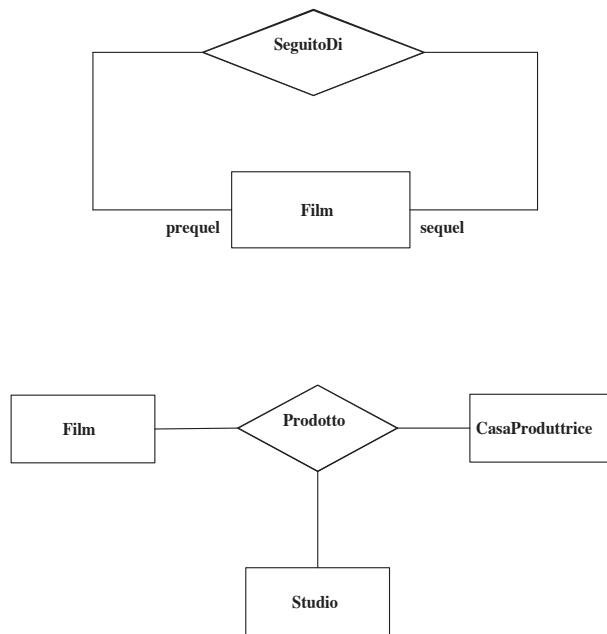


Figura 5.3: Associazioni unarie e ternarie

che mettono in relazione istanze di due diverse entità ed *associazioni n-arie*, cioè associazioni che mettono in relazione istanze di più di due entità.

La Figura 5.3 presenta alcuni esempi di associazioni unarie e ternarie, mentre l'associazione della Figura 5.2 è un esempio di associazione binaria.

**Esempio 5.2** In riferimento alla Figura 5.3, le istanze dell'associazione **SeguitoDi** sono coppie  $(f_1, f_2)$ , dove  $f_1$  ed  $f_2$  sono istanze dell'entità **Film**, che rappresentano, rispettivamente, un film ed il suo seguito. Ad esempio, la coppia **(L'era glaciale, L'era glaciale 2 - il disgelo)** rappresenta il fatto che il film “L'era glaciale 2 - il disgelo” è il seguito del film “L'era glaciale”. Le istanze dell'associazione **Prodotto** sono invece triple  $(f, p, s)$  che modellano il fatto che un certo film  $f$  è prodotto dalla casa di produzione  $p$  nello studio  $s$ .  $\square$

Per specificare la funzione che un'entità esercita nell'ambito di un'associazione possiamo far uso del concetto di *ruolo*. La definizione del ruolo è necessaria nelle associazioni, quali quelle unarie, in cui la stessa entità può partecipare più volte all'associazione, ma con ruoli diversi. Ad esempio, nella Figura 5.3 sono stati specificati i ruoli che le istanze dell'entità **Film** giocano nell'associazione **SeguitoDi**.

Un *attributo* rappresenta una proprietà posseduta da un'entità od associazione. Gli attributi sono inseriti in un diagramma ER accanto al costrutto a cui sono relativi e denotati tramite un pallino collegato al relativo costrutto.

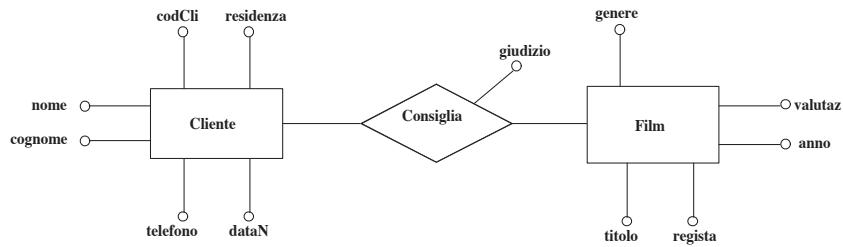


Figura 5.4: Attributi di entità ed associazioni

**Esempio 5.3** La Figura 5.4 mostra il diagramma della Figura 5.2 completato con i relativi attributi. Notiamo che il giudizio che un cliente esprime su di un film è modellato mediante un attributo dell'associazione **Consiglia**, in quanto tale informazione non è specifica né di un certo cliente né di un certo film ma piuttosto dell'associazione tra un cliente ed i film che esso consiglia. □

Un attributo è *mono-valore* se può assumere al più un valore, è *multi-valore* altrimenti.

**Esempio 5.4** Con riferimento alla Figura 5.4, l'attributo dataN dell'entità **Cliente** è un esempio di attributo mono-valore, in quanto un cliente non può avere più di una data di nascita, mentre telefono è un esempio di attributo che potrebbe essere multi-valore in quanto un cliente potrebbe avere più recapiti telefonici. □

Un attributo può a sua volta essere formato da un certo numero di componenti (sotto-attributi). Attributi di questo tipo vengono chiamati *attributi composti*. Ad esempio, l'attributo **residenza** nel diagramma ER della Figura 5.4 potrebbe essere considerato come composto dagli attributi **città**, **via**, **no** e **cap**. Graficamente gli attributi composti sono rappresentati mediante un ovale collegato sia all'entità od associazione a cui si riferiscono, sia ai sotto-attributi. La Figura 5.5 mostra la rappresentazione grafica dell'attributo composto **residenza**.

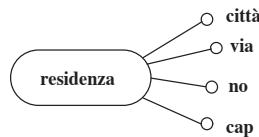


Figura 5.5: Attributo composto

Ad ogni attributo è associato un dominio che denota l'insieme dei valori che l'attributo può assumere. Esempi di possibili domini sono i domini standard, quali interi, reali, booleani, date, caratteri, stringhe di caratteri; gli intervalli di valori, ad esempio di interi o caratteri; gli insiemi di valori, o i domini definiti per

enumerazione dall'utente. Gli intervalli di valori vengono generalmente rappresentati tramite la notazione  $[v_1, v_2]$ , dove  $v_1$  e  $v_2$  denotano, rispettivamente, il limite inferiore e superiore dell'intervallo; gli insiemi di valori, che servono per modellare i domini degli attributi multi-valore, vengono denotati tramite la notazione  $\text{set\_of}(d)$ , dove  $d$  è un dominio, mentre i domini definiti per enumerazione dall'utente vengono rappresentati tramite la notazione  $\{v_i, \dots, v_j\}$ , dove  $v_i, \dots, v_j$  indicano i valori che compongono il dominio.

Ulteriori domini possono inoltre essere definiti come combinazione dei domini visti finora. Tali domini vengono chiamati *domini composti* per distinguerli dai *domini semplici*, cioè dai domini finora presi in esame. I domini composti servono per assegnare un dominio agli attributi composti. L'insieme dei valori associati ad un dominio composto è dato dal prodotto cartesiano degli insiemi di valori associati ai domini componenti. Un dominio composto  $D$  è rappresentato tramite la notazione  $D = D_1 \times D_2 \times \dots \times D_n$ , dove  $D_1, \dots, D_n$  sono i domini componenti del dominio  $D$ . Ogni valore di  $D$  è una tupla  $(d_1, \dots, d_n)$ , tale che  $d_1 \in D_1, \dots, d_n \in D_n$ .

**Esempio 5.5** Consideriamo il diagramma ER della Figura 5.4 e supponiamo che l'attributo **residenza** abbia la struttura illustrata nella Figura 5.5. Una possibile dichiarazione per gli attributi dell'entità **Cliente** è la seguente:

```
codCli: int
nome: string
cognome: string
dataN: date
telefono: set_of(string)
residenza: string × string × string × integer.
```

Una possibile dichiarazione per l'attributo **giudizio** dell'associazione **Consiglia** è la seguente:

```
giudizio: dom_giudizio
```

dove il dominio **dom\_giudizio** è definito come segue:

```
dom_giudizio: [0,5].
```

□

Le informazioni sui domini degli attributi non sono direttamente rappresentabili in un diagramma ER. Queste informazioni, fondamentali per una corretta progettazione logica, devono essere inserite nella documentazione a corredo dei diagrammi ER, che viene elaborata durante la fase di progettazione concettuale (vedi Paragrafo 5.3 per ulteriori dettagli sull'argomento).

Oltre ai costrutti di base illustrati fino ad ora, il modello ER consente la specifica di ulteriori costrutti, che ne aumentano il potere espressivo, quali *vincoli di integrità* e *gerarchie di generalizzazione*, che verranno illustrati nei paragrafi successivi.

### 5.2.2 *Vincoli di integrità*

Il modello ER fornisce costrutti per definire due diverse classi di vincoli di integrità: *vincoli di cardinalità*, sia per associazioni sia per attributi, e *vincoli di identificazione*, per entità.

#### 5.2.2.1 *Vincoli di cardinalità per associazioni*

I vincoli di cardinalità per associazioni stabiliscono il numero minimo e massimo di istanze di un'associazione a cui le istanze delle entità coinvolte nella associazione possono partecipare. Ad esempio, se consideriamo l'associazione **Consiglia** della Figura 5.2 e specifichiamo una cardinalità massima pari a cinque per l'entità **Cliente** rispetto a tale associazione, imponiamo il vincolo che ogni cliente non possa consigliare più di cinque film, mentre una cardinalità minima pari ad uno indica che ogni cliente della videoteca deve consigliare almeno un film. Graficamente, i vincoli di cardinalità vengono rappresentati tramite una coppia di valori interi ( $c\_min, c\_max$ ), collocata vicino alla linea che connette l'associazione con ciascuna entità che mette in relazione. I valori  $c\_min$  e  $c\_max$  indicano, rispettivamente, il numero minimo e massimo di istanze dell'associazione a cui un'istanza dell'entità può partecipare. In generale, il valore assegnato a  $c\_min$  e  $c\_max$  può essere un qualunque intero non negativo, con l'ovvio vincolo che il valore di  $c\_min$  sia minore o uguale a quello di  $c\_max$ . Tuttavia, i valori più comunemente utilizzati sono zero ed uno per la cardinalità minima, uno ed  $n$  per la cardinalità massima, dove ' $n$ ' è un simbolo che indica un qualsiasi numero intero maggiore di uno. Il significato di questi valori è il seguente:

- Se la cardinalità minima di un'entità rispetto ad un'associazione è zero, allora la partecipazione dell'entità all'associazione è *opzionale*, possono cioè esistere delle istanze dell'entità che non partecipano ad alcuna istanza dell'associazione.
- Se la cardinalità minima di un'entità rispetto ad un'associazione è uno, allora la partecipazione dell'entità all'associazione è *obbligatoria*. Questo implica che non può esistere un'istanza dell'entità che non sia coinvolta in almeno un'istanza dell'associazione.
- Se la cardinalità massima di un'entità rispetto ad un'associazione è uno, allora ogni istanza dell'entità può partecipare a non più di un'istanza dell'associazione. Se, in aggiunta, la cardinalità minima è uno allora ogni istanza dell'entità partecipa ad una ed una sola istanza dell'associazione.
- Se la cardinalità massima di un'entità rispetto ad un'associazione è  $n$ , allora non esiste limite al numero massimo di istanze dell'associazione a cui ogni istanza dell'entità può partecipare. In questo caso, se la cardinalità minima dell'entità rispetto all'associazione è zero, ogni istanza dell'entità può partecipare ad un numero qualsiasi di istanze dell'associazione, anche nessuna.

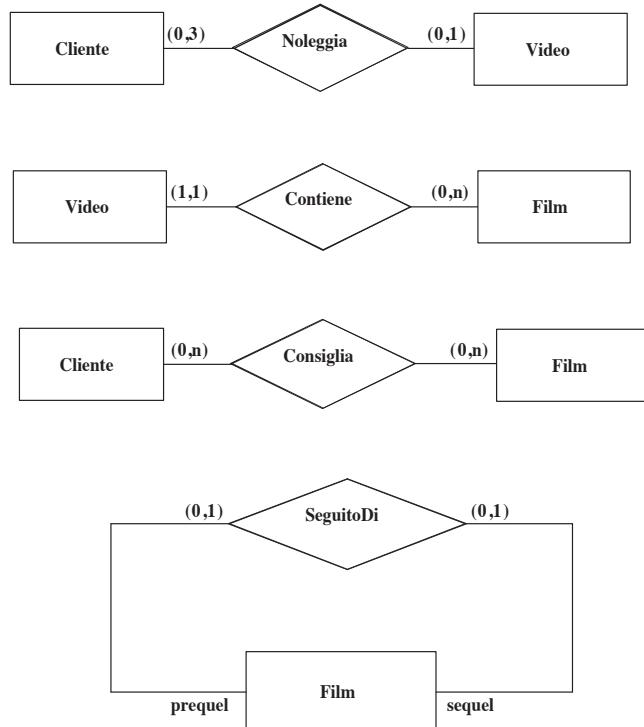


Figura 5.6: Vincoli di cardinalità per associazioni

**Esempio 5.6** La Figura 5.6 mostra possibili vincoli di cardinalità per alcune associazioni relative al dominio della videoteca. Le associazioni **Noleggia** e **Contiene** modellano, rispettivamente, l'associazione che lega un cliente ai video correntemente in noleggio e quella che lega un video al film che contiene. In base ai vincoli specificati, un cliente può avere contemporaneamente in noleggio non più di tre video, mentre un video può essere noleggiato da non più di un cliente contemporaneamente. I vincoli di cardinalità per l'associazione **Contiene** stabiliscono invece che la videoteca può possedere un numero arbitrario di video relativi ad un certo film (anche nessuno), mentre un video contiene uno ed un solo film. I vincoli di cardinalità relativi all'associazione **Consiglia** stabiliscono che ogni cliente può consigliare un numero arbitrario di film (anche nessuno) ed un film può essere consigliato da più clienti. Infine, l'ultima associazione mostra come, nel caso di associazioni unarie, i vincoli di cardinalità siano espressi rispetto ai ruoli che l'entità ricopre nell'associazione. In questo caso, ogni film ha al più un seguito, mentre un film può essere il seguito di non più di un film.  $\square$

Se in un diagramma ER vengono omessi dei vincoli di cardinalità per alcune associazioni, assumiamo **(0..n)** come valore di default.

Infine, i vincoli di cardinalità massima di un'associazione binaria o unaria inducono una classificazione delle associazioni che riveste particolare importanza durante la fase di progettazione logica (vedi Capitolo 6). Consideriamo innanzitutto il caso di un'associazione binaria  $A$  tra le entità  $E_1$  ed  $E_2$ . Possono essere distinti i seguenti casi:

- Se la cardinalità massima sia di  $E_1$  sia di  $E_2$  rispetto ad  $A$  è uno, allora  $A$  è un'associazione *uno a uno*, in quanto ogni istanza delle due entità è in corrispondenza con al più un'istanza dell'altra.
- Se la cardinalità massima rispetto ad  $A$  di una tra le entità  $E_1$  ed  $E_2$  è  $n$ , mentre quella dell'altra entità è uno, allora  $A$  è un'associazione *uno a molti*.
- Infine, se la cardinalità massima sia di  $E_1$  sia di  $E_2$  rispetto ad  $A$  è  $n$ , allora  $A$  è un'associazione *molti a molti*.

Un'analoga classificazione vale per le associazioni unarie considerando la cardinalità massima dei ruoli giocati dall'entità nell'associazione considerata.

**Esempio 5.7** L'associazione **Consiglia** della Figura 5.6 è un'associazione molti a molti, mentre l'associazione **SeguitoDi** è un esempio di associazione uno a uno. L'associazione **Contiene** è un esempio di associazione uno a molti.  $\square$

La definizione dei vincoli di cardinalità deve essere attentamente valutata in fase di progettazione, in base al dominio applicativo in considerazione. Come vedremo nel Capitolo 6, i vincoli di cardinalità per le associazioni sono fondamentali per una corretta traduzione dello schema concettuale nello schema logico.

#### 5.2.2.2 *Vincoli di cardinalità per attributi*

Oltre che per le associazioni, è possibile specificare vincoli di cardinalità anche per attributi. Abbiamo visto in precedenza come il modello ER consenta la specifica sia di attributi mono sia multi-valore. I vincoli di cardinalità per attributi, imponendo delle condizioni sul numero minimo e massimo di valori che un attributo può assumere, permettono di specificare se l'attributo è mono o multi-valore (cardinalità massima) e se può assumere o meno un valore nullo (cardinalità minima). In particolare, la cardinalità minima di un attributo rappresenta il numero minimo di valori dell'attributo che possono essere associati ad un'istanza dell'associazione o dell'entità a cui l'attributo si riferisce. Un attributo con cardinalità minima zero è un attributo che può assumere un valore nullo, mentre se la cardinalità minima è uno allora l'attributo deve necessariamente assumere almeno un valore. Rispetto ai vincoli di cardinalità minima, un attributo viene detto *opzionale* se ha cardinalità minima pari a zero, *obbligatorio* in caso contrario. La cardinalità massima di un attributo rappresenta, invece, il numero massimo di valori dell'attributo che possono essere associati ad un'istanza della corrispondente associazione od entità. Un attributo con cardinalità massima pari a uno, è un attributo che può assumere

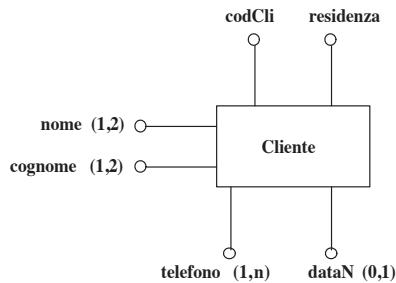


Figura 5.7: Vincoli di cardinalità per attributi

un solo valore, mentre un attributo con cardinalità massima maggiore di uno può assumere un insieme di valori. Quindi, un attributo è multi-valore se ha cardinalità massima maggiore di uno, mono-valore in caso contrario.

Graficamente i vincoli di cardinalità per gli attributi vengono rappresentati tramite la coppia ( $c\_min, c\_max$ ) associata alla linea che connette l'attributo all'entità od associazione a cui si riferisce, dove  $c\_min$  e  $c\_max$  indicano, rispettivamente, la cardinalità minima e massima dell'attributo. Per attributi per cui non è specificato un vincolo di cardinalità assumiamo, come valore di default, il valore (1,1), che può essere omesso dalla rappresentazione grafica. Il valore 'n' per la cardinalità massima indica che l'attributo può assumere un numero di valori arbitrario.

**Esempio 5.8** La Figura 5.7 illustra alcuni vincoli di cardinalità per gli attributi dell'entità **Cliente**. Il vincolo di cardinalità (1,2) associato agli attributi **nome** e **cognome** indica che un cliente può avere un doppio nome e cognome, ma ne deve essere specificato almeno uno. Anche **telefono** è un esempio di attributo multi-valore ma non è specificato alcun limite superiore al numero di recapiti telefonici che possono essere associati ad un cliente. L'attributo **dataN** è invece un esempio di attributo mono-valore: ad ogni cliente può essere associata al più una data di nascita ma possono esistere clienti per cui tale informazione non è registrata. Infine, gli attributi **codCli** e **residenza** non hanno alcun vincolo di cardinalità associato; per essi vale quindi il valore di default (1,1).  $\square$

### 5.2.2.3 Vincoli di identificazione

Un'ulteriore classe di vincoli specificabili nel modello ER è la classe dei *vincoli di identificazione* per entità. Definire un vincolo d'identificazione per un'entità  $E$  significa specificare un insieme di attributi e/o di entità che possiedono la proprietà di identificare univocamente le istanze di  $E$ . Tali insiemi di entità e/o di attributi vengono solitamente chiamati *identificatori* o *chiavi*. Gli identificatori possono essere suddivisi in tre classi a seconda della loro natura. La forma più semplice d'identificatore, chiamato *identificatore interno*, è costituita da uno o più attributi

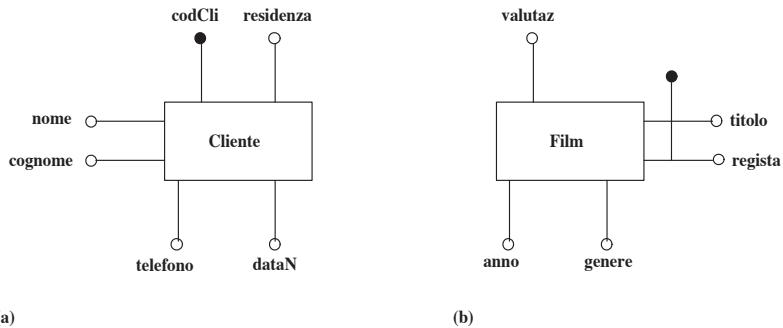


Figura 5.8: Identificatore: (a) interno semplice; (b) interno composto

dell'entità a cui si riferisce. Possono però esistere delle situazioni in cui nessun insieme di attributi di un'entità ha la proprietà d'identificare univocamente le sue istanze. Ad esempio, se consideriamo il dominio della videoteca e supponiamo che lo schema ER descriva video appartenenti a diverse videoteche (ad esempio tutte le videoteche appartenenti ad una certa catena di distribuzione), allora il codice di collocazione di un video non lo identifica univocamente, in quanto video appartenenti a negozi diversi potrebbero avere lo stesso codice di collocazione. Per identificare univocamente un video serve, in aggiunta al suo codice di collocazione, anche la videoteca a cui appartiene. In questo caso l'identificatore è *misto*, cioè costituito sia da attributi dell'entità sia da entità ad essa collegate. Infine, un'entità ha un *identificatore esterno* quando è identificata da una o più entità ad essa collegate.

Gli identificatori possono essere ulteriormente classificati in *identificatori semplici*, formati cioè da un solo elemento, ed *identificatori composti*, cioè formati da più di un elemento. Graficamente un identificatore è rappresentato tramite un circolino nero, come mostrato nelle Figure 5.8 e 5.9. L'esempio seguente chiarisce meglio i concetti appena introdotti.

**Esempio 5.9** La Figura 5.8 mostra alcuni esempi di identificatori interni. L'attributo *codCli* dell'entità *Cliente* è un esempio di identificatore interno semplice; definire tale attributo come identificatore significa imporre che non possano esistere due istanze dell'entità *Cliente* con lo stesso valore per l'attributo *codCli*. Sempre in riferimento alla Figura 5.8, la coppia di attributi *titolo* e *regista* è un esempio di identificatore interno composto per l'entità *Film*. Possono quindi esistere film con lo stesso titolo ma non diretti dallo stesso regista. Come abbiamo accennato in precedenza, l'entità *Video* della Figura 5.9(a) potrebbe essere identificata dall'attributo *colloc*, se lo schema descrivesse video appartenenti ad una sola videoteca. In caso contrario, un identificatore corretto per l'entità *Video* è costituito dall'attributo *colloc* e dall'entità *Videoteca*, in quanto nello stessa videoteca non possono esistere due video con lo stesso codice di collocazione.

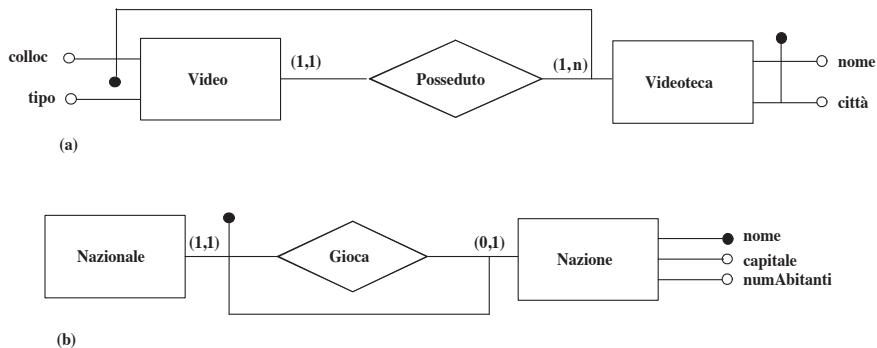


Figura 5.9: Identificatore: (a) misto; (b) esterno

L'identificatore della Figura 5.9(a) è quindi un esempio d'identificatore misto. Infine, l'entità **Nazione** della Figura 5.9(b) è un esempio di identificatore esterno per l'entità **Nazionale**, in quanto ogni nazionale può essere identificata univocamente dalla nazione che rappresenta.  $\square$

Dagli esempi della Figura 5.9 appare evidente come l'identificazione di un'entità *E* tramite un'altra sia possibile solo se *E* partecipa con cardinalità (1,1) all'associazione che lega le due entità. Un'entità le cui istanze vengono identificate mediante l'associazione con altre entità viene chiamata *entità debole*.

**Esempio 5.10** Le entità **Video** e **Nazionale** della Figura 5.9 sono esempi di entità deboli. L'identificazione delle istanze di **Video** tramite l'entità **Videoteca** è possibile solo perché un video appartiene ad una ed una sola videoteca. Un discorso analogo vale per l'entità **Nazionale**.  $\square$

È importante notare che un'entità può possedere più identificatori. Ad esempio, in riferimento al diagramma della Figura 5.8(a), un identificatore alternativo per l'entità **Cliente** potrebbe essere costituito dagli attributi **nome**, **cognome** e **dataN**. Un identificatore è *minimale* se non è sottoinsieme proprio di un altro identificatore. Durante la fase di progettazione concettuale è bene evidenziare per ogni entità tutti i possibili identificatori minimali; la scelta di come utilizzarli nel corrispondente schema logico verrà effettuata durante la fase di progettazione logica (vedi Capitolo 6 per ulteriori dettagli).

### 5.2.3 Gerarchie di generalizzazione

Nel modello ER è possibile organizzare le entità in *gerarchie di generalizzazione*. Questo consente la definizione di un insieme di entità (dette *entità figlie*) come specializzazione di un'altra entità (detta *entità padre*), rappresentante le proprietà in comune a tutte le entità figlie. L'associazione che viene instaurata tra un'entità

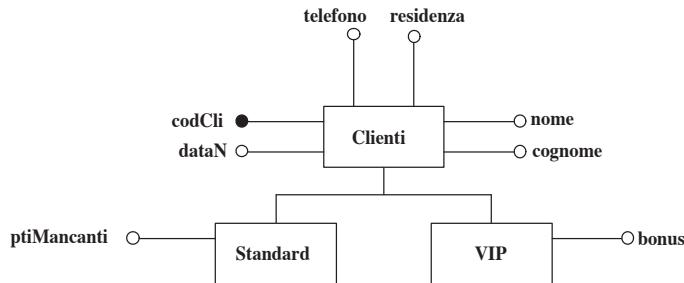


Figura 5.10: Gerarchia di generalizzazione

e la sua generalizzazione è un'associazione di contenimento. Più precisamente, un'entità  $E$  è una generalizzazione delle entità  $E_1, \dots, E_n$  (ed  $E_1, \dots, E_n$  sono specializzazioni di  $E$ ) se ogni istanza delle entità  $E_1, \dots, E_n$  è anche istanza di  $E$ . Tutte le proprietà dell'entità padre (attributi, identificatori ed associazioni) sono anche proprietà delle entità figlie. Le gerarchie di generalizzazione vengono graficamente rappresentate tramite la notazione illustrata nella Figura 5.10.

**Esempio 5.11** Supponiamo che la nostra videoteca preveda di erogare un bonus (da spendere per noleggi di video nella videoteca stessa) per i clienti, denominati VIP, che hanno effettuato un certo numero di noleggi nell'anno corrente. Il programma di fidelizzazione consente di accumulare punti ad ogni noleggio. Quando il numero di punti supera una certa soglia, il cliente viene qualificato come VIP e gli viene erogato un bonus di un certo ammontare. Per i clienti che non hanno ancora ricevuto il bonus è importante conoscere il numero di punti che li separa dall'acquisizione del bonus stesso. Questi requisiti possono essere modellati tramite la gerarchia di generalizzazione illustrata nella Figura 5.10. Tutte le istanze delle entità Standard e VIP sono anche istanze dell'entità padre Cliente, in quanto rappresentano specifiche tipologie di clienti. Notiamo inoltre che nella rappresentazione grafica vengono associate alle entità figlie solo le proprietà che esse hanno in più rispetto alle proprietà possedute dall'entità padre. Gli attributi nome, cognome, codCli, dataN, telefono e residenza, definiti per l'entità Cliente, sono anche attributi delle entità Standard e VIP, in base alla gerarchia di generalizzazione che lega le tre entità. Notiamo inoltre come l'associazione Consiglia della Figura 5.2 sia a sua volta ereditata dalle entità Standard e VIP.  $\square$

A seconda dei legami che intercorrono tra le istanze dell'entità padre e le istanze delle entità figlie, una generalizzazione delle entità  $E_1, \dots, E_n$  nell'entità  $E$  può essere una:

- **Generalizzazione totale.** La generalizzazione è totale se ogni istanza dell'entità padre  $E$  è istanza di almeno una delle entità figlie  $E_1, \dots, E_n$ .

- **Generalizzazione parziale.** La generalizzazione è parziale se esiste almeno un'istanza dell'entità padre  $E$  che non è istanza di alcuna delle entità figlie  $E_1, \dots, E_n$ .

**Esempio 5.12** La generalizzazione della Figura 5.10 è un esempio di generalizzazione totale se assumiamo che le uniche tipologie di clienti della videoteca siano clienti VIP e standard. Viceversa, se la videoteca gestisce anche clienti occasionali, clienti cioè che effettuano un noleggio senza registrarsi al programma di fidelizzazione, allora la generalizzazione diventa un esempio di generalizzazione parziale.  $\square$

Inoltre, le generalizzazioni possono essere ulteriormente classificate in:

- **Generalizzazioni esclusive.** Una generalizzazione è esclusiva se ogni istanza dell'entità padre  $E$  è istanza di non più di una delle entità figlie  $E_1, \dots, E_n$ .
- **Generalizzazioni condivise.** Una generalizzazione è condivisa se esiste almeno un'istanza dell'entità padre  $E$  che è istanza di più di una delle entità figlie  $E_1, \dots, E_n$ .

**Esempio 5.13** La generalizzazione della Figura 5.10 è un esempio di generalizzazione esclusiva in quanto un cliente non può essere contemporaneamente VIP e standard. Viceversa, la generalizzazione che lega l'entità **Film** con le entità figlie **FilmAnimazione** e **FilmEssay** è un esempio di generalizzazione condivisa in quanto ci possono essere film d'animazione che sono anche film d'essay.  $\square$

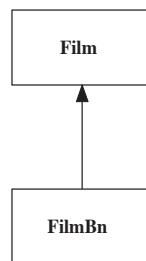


Figura 5.11: Associazione di sottoinsieme

Le due classificazioni precedentemente esposte sono ortogonali. Questo implica che le generalizzazioni possono essere di quattro tipi diversi: *totali esclusive*, *totali condivise*, *parziali esclusive* e *parziali condivise*. Le informazioni sulle tipologie di generalizzazioni presenti nello schema concettuale, non direttamente rappresentabili nel diagramma ER, andranno inserite nella documentazione a corredo dello schema stesso (vedi Paragrafo 5.4 per un esempio di tale documentazione).

Un caso particolare di generalizzazione è rappresentato dall'associazione di *sottoinsieme* che viene instaurata quando un'entità ha una sola entità figlia. Definire un'entità  $E_1$  come sottoinsieme di un'entità  $E_2$  significa specificare che ogni istanza dell'entità  $E_2$  è anche istanza dell'entità  $E_1$ . Per l'associazione di sottoinsieme vale naturalmente il principio di ereditarietà illustrato per le generalizzazioni. L'associazione di sottoinsieme è una generalizzazione parziale ed esclusiva. La rappresentazione grafica dell'associazione di sottoinsieme è illustrata nella Figura 5.11, dove l'entità **FilmBn**, che rappresenta l'insieme dei Film in bianco e nero offerti dalla videoteca, è definita come sottoinsieme dell'entità **Film**. Ciò implica che le istanze di **FilmBn** sono anche istanze dell'entità **Film**.

I principali simboli grafici messi a disposizione dal modello ER sono riassunti nella Tabella 5.1.

Costrutto	Simbolo
Entità	
Associazione	
Attributo	
Attributo composto	
Gerarchia di generalizzazione	
Relazione di sottoinsieme	
Identificatore	
Vincolo di cardinalità	

Tabella 5.1: Principali costrutti del modello ER e relativi simboli grafici

### 5.3 Metodologie di progettazione

Nel paragrafo precedente abbiamo introdotto il modello ER, cioè lo strumento tramite cui è possibile definire lo schema concettuale di una base di dati. Questo naturalmente è solo una delle componenti per arrivare, dal documento di specifica dei requisiti, alla realizzazione di uno schema concettuale che modelli nel modo migliore tale specifica. Nel seguito illustreremo alcune linee guida per condurre le

varie fasi che portano alla generazione di uno schema concettuale. Nel Paragrafo 5.4 illustreremo invece un esempio di progettazione concettuale in base alle linee guida discusse in questo paragrafo.

### ***5.3.1 Raccolta ed analisi dei requisiti***

Va innanzitutto osservato che la fase di raccolta ed analisi dei requisiti (vedi Paragrafo 5.1) è per sua stessa natura un'attività difficilmente standardizzabile che di solito viene condotta da personale esperto, in stretta collaborazione con coloro che dovranno utilizzare la base di dati. Prima di procedere con la progettazione concettuale è però utile effettuare alcune operazioni sul documento di specifica che rendano più agevoli le fasi successive. Innanzitutto, è bene leggere attentamente il documento di specifica e cercare di evidenziare tutte le possibili ambiguità che il documento contiene, dovute sia ad imprecisioni sia all'intrinseca ambiguità del linguaggio naturale. Ad esempio, se il documento di specifica stabilisce che la base di dati dovrà contenere informazioni anagrafiche sui clienti della videoteca, è opportuno precisare quali siano esattamente queste informazioni (nome, cognome, codice fiscale, altro), prima di procedere con la progettazione concettuale. Questa attività sul documento di specifica deve essere condotta in stretta relazione con i committenti del progetto che potranno fornire tutti i chiarimenti del caso. Un'ulteriore operazione che è opportuno effettuare sul documento di specifica dei requisiti è l'unificazione dei sinonimi. Spesso, utilizziamo termini diversi per denotare lo stesso concetto, questi però andranno rappresentati in modo univoco all'interno della base di dati che stiamo progettando. Ad esempio, nel documento di specifica della base di dati della videoteca potrebbero comparire i termini cassetta e videocassetta per denotare lo stesso concetto. Occorre quindi scegliere tra i vari sinonimi il termine che meglio denota il concetto che vogliamo esprimere e sostituire con questo tutte le occorrenze degli altri. È inoltre opportuno separare, nel documento di specifica, le frasi che riguardano i dati (che saranno quelle principalmente usate in fase di progettazione) da quelle che sono riferite ad operazioni. Tali operazioni preliminari sul documento di specifica sono fondamentali per il buon progetto della base di dati. Infatti, rilevare e correggere incompletezze ed ambiguità già nel documento di specifica consente di effettuare le successive fasi su dati corretti, con un conseguente vantaggio in termini di impiego di risorse e qualità del progetto. Tanto più gli errori vengono rilevati nelle fasi alte della progettazione, tanto meno costerà la loro correzione.

Il passo successivo è quello di generare uno schema ER consistente con le specifiche. Tale attività, soprattutto per basi di dati di medie e grosse dimensioni, è un'attività incrementale che genera lo schema finale attraverso una serie di raffinamenti successivi e la produzione di un certo numero di schemi intermedi. Nel seguito approfondiremo l'argomento.

### 5.3.2 Progettazione concettuale

Anche nel caso della progettazione concettuale non esiste una metodologia standard per condurla e molto dipende dalla capacità e dall'esperienza del progettista. Possiamo però fornire alcune linee guida che facilitano la generazione di un buon schema concettuale.

#### 5.3.2.1 Scelta dei costrutti

Il primo passo nella generazione di uno schema ER è decidere, tramite un'attenta analisi delle specifiche, quali costrutti del modello ER (entità, attributi, associazioni, ecc.) utilizzare per modellare i vari concetti presenti nel documento di specifica. A tale scopo, possono essere utili le seguenti indicazioni:

- Se un concetto presente nel documento di specifica descrive un insieme omogeneo di oggetti rilevanti per il dominio d'interesse e caratterizzati da un insieme di proprietà comuni, allora il costrutto più idoneo per la sua rappresentazione nello schema ER è quello di entità.

**Esempio 5.14** Facendo riferimento all'esempio della videoteca, è opportuno rappresentare il concetto di cliente tramite un'entità, in quanto il cliente denota una classe rilevante di oggetti per il dominio d'interesse (tutti i clienti della videoteca) ed inoltre di tali clienti vogliamo memorizzare una serie di proprietà comuni (nome, cognome, ecc.). □

Generalmente, sono buoni candidati a essere modellati come entità i nomi che ricorrono frequentemente nel documento di specifica.

- Se evidenziamo nel documento di specifica concetti che sono casi particolari di altri, allora è bene modellarli mediante una gerarchia di generalizzazione o di sottoinsieme, che metta in associazione i vari concetti.

**Esempio 5.15** Se il documento di specifica identifica due tipologie di clienti: VIP e standard, allora nel corrispondente diagramma ER potrebbe essere instaurata una gerarchia di generalizzazione tra le entità **Cliente**, **VIP** e **Standard** (vedi Figura 5.10). □

La modellazione tramite gerarchia di generalizzazione è opportuna quando l'insieme di proprietà e/o di associazioni che vogliamo modellare per le entità figlie differisce da quelle dell'entità padre, come nel caso delle entità della Figura 5.10. Altrimenti, è anche possibile una modellazione alternativa, che consiste nell'inserire nell'entità che modella il concetto più generale un attributo che identifica i vari tipi di istanze che possono appartenere all'entità.

**Esempio 5.16** Supponiamo che la videoteca consenta il noleggio sia di videocassette sia di dvd, ma che non vogliamo memorizzare informazioni aggiuntive su questi due tipi di supporti (ad eccezione del fatto che entrambi sono disponibili per il noleggio). È possibile, in analogia all’Esempio 5.15, instaurare una gerarchia di generalizzazione avente **Video** come entità padre e **DVD** e **VHS** come figli. Alternativamente, potremmo inserire un attributo **tipo** nell’entità **Video** il cui valore specifica se il video è una videocassetta o un dvd.  $\square$

- Se un concetto rappresenta una proprietà elementare, senza ulteriori proprietà associate, allora il costrutto più idoneo per la sua rappresentazione è quello di attributo. Ad esempio, nel caso della videoteca il concetto di nome può essere opportunamente rappresentato tramite un attributo, in quanto descrive una proprietà elementare dell’entità **Cliente**. Leggendo il documento di specifica dobbiamo poi determinare se gli attributi sono semplici o composti e definire gli opportuni vincoli di cardinalità.

Possono esistere però casi in cui non è facile scegliere se modellare un concetto come attributo o come entità, o casi in cui entrambe le modellazioni possono essere appropriate. Questo accade soprattutto quando un concetto candidato ad essere modellato come attributo è formato da un insieme di componenti, come illustrato nel seguente esempio.

**Esempio 5.17** Supponiamo che siamo interessati a memorizzare la residenza dei clienti della videoteca, intesa come via, numero civico, città e cap. Una possibilità è rappresentare tale concetto come un attributo composto, con sotto-attributi per ognuna delle sue componenti (vedi Figura 5.5). Questa modellazione è appropriata quando non esistono molti clienti che condividono la stessa residenza. Viceversa, se questo accade di frequente, possiamo creare un’entità **Residenza**, collegata tramite un’associazione all’entità **Cliente**, ed avente come attributi **via**, **no**, **città** e **cap**.  $\square$

- Se nel documento di specifica è presente un concetto che denota un legame logico tra concetti che sono stati modellati come entità, allora tale concetto va modellato come associazione nello schema ER corrispondente.

**Esempio 5.18** Con riferimento all’esempio della videoteca, il concetto di video consigliato da un certo cliente può essere modellato tramite un’associazione tra **Film** e **Cliente** (vedi Figura 5.2).  $\square$

Generalmente le associazioni possono essere rilevate nel documento di specifica identificando verbi che mettono in associazione concetti che sono stati modellati tramite entità. Il grado dell’associazione dipende dal numero di concetti che mette in relazione. Un’associazione può anche avere delle proprietà associate, quali ad esempio il giudizio associato al video consigliato

da un cliente, che possono essere modellate come attributi. Come per gli attributi, non è sempre facile stabilire se un concetto sia meglio modellato da un'entità oppure da un'associazione, come dimostra il seguente esempio.

**Esempio 5.19** Consideriamo il concetto di noleggio di un video da parte di un cliente. Una modellazione intuitiva è tramite un'associazione **Noleggia** che lega le entità **Cliente** a **Video**. Questa rappresentazione però non permette di modellare il fatto che un cliente abbia noleggiato più volte lo stesso video, in quanto non possono esserci due istanze identiche della stessa associazione. Potrebbe quindi essere appropriata nel caso in cui volessimo memorizzare solo le informazioni relative ai noleggi in corso (associando per esempio un attributo **dataNol** all'associazione **Noleggia**). Nel caso in cui, invece, volessimo mantenere anche le informazioni riguardanti noleggi conclusi, il noleggio dovrebbe essere modellato tramite un'entità **Noleggio** collegata da un'associazione uno a molti all'entità **Cliente**, con attributi **dataNol** e **dataRest**, quest'ultimo opzionale per modellare anche i noleggi in corso (vedi Figura 5.15).  $\square$

Infine, è di fondamentale importanza stabilire i corretti vincoli di cardinalità per le associazioni identificate ed i vincoli di identificazione per le entità. Anche queste informazioni vanno dedotte dal documento di specifica o richieste ai committenti della base di dati, ove non specificate.

#### 5.3.2.2 Generazione di diagrammi ER

Come accennato in precedenza, il diagramma ER finale viene di solito generato mediante un processo iterativo, che genera lo schema finale mediante raffinamento e/o integrazione di schemi intermedi. Tale processo può essere condotto con le usuali strategie *top-down* e *bottom-up* utilizzate anche per la progettazione del software, o con una strategia mista che combina opportunamente le strategie top-down e bottom-up (vedi il paragrafo successivo per un esempio di tale modo di procedere). Nella strategia top-down, si parte dal documento di specifica e si genera un primo schema che descrive il dominio con pochi costrutti ed un notevole livello di astrazione (ad esempio questo primo schema può contenere solo le entità ed associazioni maggiormente rilevanti e nessun attributo). Ad ogni passo della strategia, lo schema ottenuto al passo precedente viene precisato aggiungendo sempre maggiori dettagli fino a rappresentare tutte le informazioni richieste dalle specifiche. Naturalmente, i diagrammi ottenuti ai vari passi dovranno essere tra loro coerenti, nel senso che lo schema ottenuto ad un certo passo sarà un raffinamento dello schema ottenuto al passo precedente, e non un diagramma che modella le informazioni di interesse in modo completamente diverso. Ad esempio, se ad un certo passo abbiamo ottenuto un diagramma contenente una certa entità *E*, nel passo successivo possiamo aggiungere attributi ad *E*, oppure instaurare una gerarchia di generalizzazione avente *E* come padre. Nella strategia bottom-up,

invece, si procede in modo diametralmente opposto. Le specifiche iniziali vengono suddivise in sotto-specifiche. Il processo viene iterato fino a quando ogni sotto-specifica descrive una componente elementare della realtà d'interesse. Ad esempio, se applichiamo la strategia bottom-up alla videoteca, potremmo partizionare le specifiche in tre gruppi riguardanti clienti, video e film rispettivamente. Per ognuna delle sotto-specifiche ottenute generiamo un diagramma ER, tali diagrammi vengono poi integrati per ottenere lo schema finale. Il passo d'integrazione, in analogia ai passi di raffinamento della strategia top-down, dovrà avvenire in base ad alcune regole che assicurano la correttezza dello schema integrato. Ad esempio, dati due diagrammi ER  $ER_1$  ed  $ER_2$  è possibile integrarli definendo un'associazione che mette in relazione entità di  $ER_1$  con entità di  $ER_2$ , oppure instaurando una gerarchia di generalizzazione che coinvolge entità dei due schemi.

#### 5.3.2.3 Documentazione di diagrammi ER

Lo schema ER finale deve poi essere corredata da una documentazione in linguaggio naturale, che serve per precisare tutto ciò che non è esprimibile usando i costrutti del modello ER ma che è utile in fase di progettazione logica. La documentazione a corredo di un diagramma ER deve innanzitutto contenere le informazioni sui domini dei vari attributi presenti nello schema; deve inoltre contenere tutti quei vincoli imposti dal dominio e che non sono rappresentabili come vincoli di cardinalità ed identificazione del modello ER. Un esempio di tali vincoli per il nostro dominio di riferimento è il fatto che un cliente non possa noleggiare più di tre video contemporaneamente. Tali vincoli dovranno essere riportati nella documentazione a corredo dello schema ER e utilizzati nella fase di progettazione logica (vedi Capitolo 6) per decidere come implementarli (ad esempio, nel caso del modello relazionale, mediante la specifica di vincoli CHECK, asserzioni o l'implementazione di procedure ad-hoc). La documentazione deve contenere inoltre i vincoli di autorizzazione, se necessari (vedi Capitolo 9). Un esempio di tali vincoli è il fatto che clienti minorenni non possano accedere al noleggio di determinate categorie di film. Infine, la documentazione deve contenere informazioni sulle principali scelte progettuali effettuate per ottenere lo schema ER finale, soprattutto ove siano possibili più alternative. Un esempio di documentazione di schemi ER è fornita nel Paragrafo 5.4.

#### 5.3.2.4 Verifiche di qualità

Indipendentemente dalla strategia adottata, è opportuno effettuare frequenti verifiche di correttezza e completezza durante la generazione degli schemi intermedi, interagendo con le persone che hanno commissionato il progetto della base di dati. Una volta ottenuto il diagramma ER finale, è poi opportuno effettuare su di esso alcune verifiche finali di qualità. In primo luogo, bisogna verificare la correttezza, sia dal punto di vista sintattico sia semantico. Correttezza sintattica significa verificare che i costrutti del modello ER siano stati utilizzati in accordo con quanto previsto dal modello (ad esempio un'associazione può collegare entità

*Vogliamo realizzare una base di dati per una videoteca. La videoteca consente il noleggio di circa 1'000 film. Per ogni film, vogliamo memorizzare il titolo, il regista, l'anno di produzione, il genere e la valutazione della critica, se presente. Ogni film è disponibile per il noleggio in un certo numero di video. Ogni videocassetta o dvd disponibile nella videoteca (circa 3'000) è identificato da un codice di collocazione e dal tipo di supporto (videocassetta o dvd). La base di dati dovrà inoltre memorizzare informazioni sui clienti della videoteca (circa 2'000) e sui video che hanno noleggiato. Il numero di noleggi giornalieri alla videoteca è circa 200. Per ogni utente della videoteca vogliamo mantenere il suo nome, cognome, data di nascita, residenza e telefono. Ogni cliente è identificato da un codice che corrisponde al numero della tessera rilasciatagli per usufruire dei servizi della videoteca. Ogni cliente può avere contemporaneamente in noleggio un certo numero di video (non più di tre). Per ogni noleggio, vogliamo memorizzare la data in cui il noleggio è stato effettuato e, per i noleggi conclusi, la data di restituzione. Ogni cliente può inoltre consigliare dei film ad altri clienti, esprimendo per essi un giudizio. La videoteca prevede un programma di fidelizzazione dei clienti. Ogni noleggio consente l'accumulo di un certo numero di punti. Quando i punti accumulati superano una certa soglia, i clienti vengono qualificati come clienti VIP e hanno diritto ad un bonus. La base di dati dovrà memorizzare, per ogni cliente VIP, il valore corrente del suo bonus. Per i clienti standard, ovvero quelli che non hanno ancora accumulato punti sufficienti per accedere alla categoria VIP, vogliamo memorizzare il numero di punti mancanti ad accedere a tale categoria.*

Figura 5.12: Requisiti per la base di dati della videoteca

ma non attributi, la cardinalità massima di un attributo non può essere inferiore a quella minima e così via). Deve inoltre essere effettuata una verifica di correttezza semantica, ovvero dobbiamo verificare che i costrutti scelti per modellare i concetti presenti nel documento di specifica siano quelli più appropriati. A tal fine, è bene valutare le varie opzioni, se praticabili, per modellare un concetto ed analizzare pro e contro per il dominio considerato. Un'altra verifica che è opportuno effettuare è quella di completezza; dovremo cioè verificare, rileggendo ancora una volta il documento di specifica, che tutto ciò che richiede di memorizzare nella base di dati sia stato effettivamente rappresentato nello schema ER. Ulteriori verifiche di qualità riguardano la presenza di dati ridondanti nello schema ER e saranno trattate nel Capitolo 6.

#### 5.4 Un esempio di progettazione concettuale

In questo paragrafo, mostreremo un esempio di progettazione concettuale per il dominio della videoteca utilizzato come esempio illustrativo in tutto il testo, effettuata in accordo alle metodologie discusse nel paragrafo precedente. A tal fine, supponiamo che i proprietari della videoteca abbiano espresso i requisiti illustrati nel documento di specifica della Figura 5.12. Nel documento di specifica sono anche contenute informazioni sul volume dei dati che saranno utilizzate nelle suc-

Vogliamo realizzare una base di dati per una videoteca. La videoteca consente il noleggio di circa 1 000 film. Per ogni film, vogliamo memorizzare il titolo, il **nome e cognome del regista**, l'anno di produzione, il genere e la valutazione della critica (espressa in una scala di valori decimali da 0 a 5), se presente. Ogni film è disponibile per il noleggio in un certo numero di video (dove video sono sia videocassette che dvd). Ogni video disponibile nella videoteca (circa 3 000) è identificato da un codice di collocazione e dal tipo di supporto (videocassetta o dvd). La base di dati dovrà inoltre memorizzare informazioni sui clienti della videoteca (circa 2 000) e sui video che hanno correntemente in noleggio ed hanno noleggiato in passato. Il numero di noleggi giornalieri alla videoteca è circa 200. Per ogni cliente della videoteca vogliamo mantenere il suo nome, cognome, data di nascita, residenza (intesa come città, via, numero civico e cap) e un insieme di recapiti telefonici. Ogni cliente è identificato da un codice che corrisponde al numero della tessera rilasciatagli per usufruire dei servizi della videoteca. Ogni cliente può avere contemporaneamente in noleggio un certo numero di video (non più di tre). Per ogni noleggio vogliamo memorizzare il giorno, mese ed anno in cui il noleggio è stato effettuato e, per i noleggi conclusi, il giorno, mese ed anno della restituzione. Ogni cliente può inoltre consigliare dei film ad altri clienti, esprimendo per essi un giudizio in una scala di valori interi da 0 a 5. La videoteca prevede un programma di fidelizzazione dei clienti. Ogni noleggio consente l'accumulo di un certo numero di punti. Quando i punti accumulati superano una certa soglia, i clienti vengono qualificati come clienti VIP e hanno diritto ad un bonus di un certo importo espresso in euro. La base di dati dovrà memorizzare, per ogni cliente VIP, il valore corrente del suo bonus. Per i clienti standard, ovvero quelli che non hanno ancora accumulato punti sufficienti per accedere alla categoria VIP, vogliamo memorizzare il numero di punti mancanti ad accedere a tale categoria.

Figura 5.13: Ristrutturazione delle specifiche

cessive fase di progettazione (vedi Capitoli 6 e 7). Le informazioni sul carico di lavoro saranno invece descritte nel Capitolo 6.

#### 5.4.1 Analisi e ristrutturazione delle specifiche

Dall'analisi della Figura 5.12 è facile notare come le specifiche contengano un certo numero di ambiguità. Una prima operazione da compiere è quindi quella di analizzare in dettaglio le specifiche, evidenziando ogni possibile ambiguità o incompletezza e cercare di chiarirla o di completare le specifiche mediante interazioni con chi ha commissionato il progetto. Un'ulteriore operazione che è opportuno effettuare in questa fase è evidenziare tutti i sinonimi presenti nella specifica, scegliere tra questi quello che giudichiamo più significativo ad esprimere il concetto in questione, e sostituire tutte le occorrenze degli altri con il termine scelto. Questa operazione rende più agevole la successiva fase di progettazione concettuale in cui ogni concetto deve essere rappresentato univocamente all'interno del diagramma ER. Esaminiamo quindi le specifiche della Figura 5.12 e cerchiamo di applicare

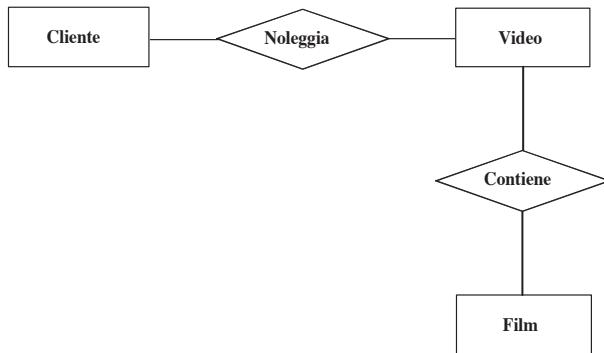


Figura 5.14: Diagramma ER iniziale per la base di dati della videoteca

quanto detto finora. Analizzando con attenzione le specifiche, notiamo subito alcune fonti di ambiguità. Ad esempio, per ogni film è richiesta la memorizzazione delle informazioni sul regista, bisogna però chiarire se siamo interessati al nome e cognome del regista, oppure solo al cognome, o ad altro. Inoltre, per quanto riguarda il giudizio espresso dai clienti e dalla critica è opportuno conoscere se tale giudizio sia un valore numerico, oppure un giudizio qualitativo (ad esempio, buono, mediocre, ottimo, ecc.). Passando alle informazioni sui clienti, occorre precisare cosa intendiamo per residenza (città, via, numero civico, ecc.). Occorre inoltre precisare se un cliente può avere più recapiti telefonici associati (ad esempio un numero di telefono fisso e quello del cellulare). È inoltre opportuno precisare come rappresentare il bonus (ad esempio in termini di punti accumulati oppure mediante un importo in euro). Infine, per quanto riguarda la data di noleggio e quella di restituzione, bisogna chiarire a quale granularità temporale siamo interessati (basta il giorno, oppure anche l'ora, i minuti, ecc.). Dall'esame delle specifiche della Figura 5.12 notiamo inoltre l'utilizzo di alcuni sinonimi (cliente ed utente, video per videocassetta/dvd). Al termine di questa analisi, le specifiche vanno riscritte unificando i sinonimi e chiarendo ambiguità ed imprecisioni. Il risultato, per il nostro esempio, è mostrato nella Figura 5.13, dove le modifiche sono evidenziate in neretto.

#### 5.4.2 Generazione dello schema ER

Dopo queste operazioni preliminari effettuate sul documento di specifica, siamo pronti per iniziare la progettazione concettuale della base di dati. Una delle metodologie maggiormente utilizzate nella pratica per generare il diagramma ER combina le metodologie top-down e bottom-up descritte nel Paragrafo 5.3.2 e consiste nel generare un primo diagramma ER contenente solo le entità e le associazioni maggiormente rilevanti (tralasciando anche gli attributi in questa prima fase). Dall'attenta analisi delle specifiche e considerando le linee guida discusse nel Paragrafo 5.3 le entità principali del dominio applicativo sono: **Cliente**, **Video** e **Film**.

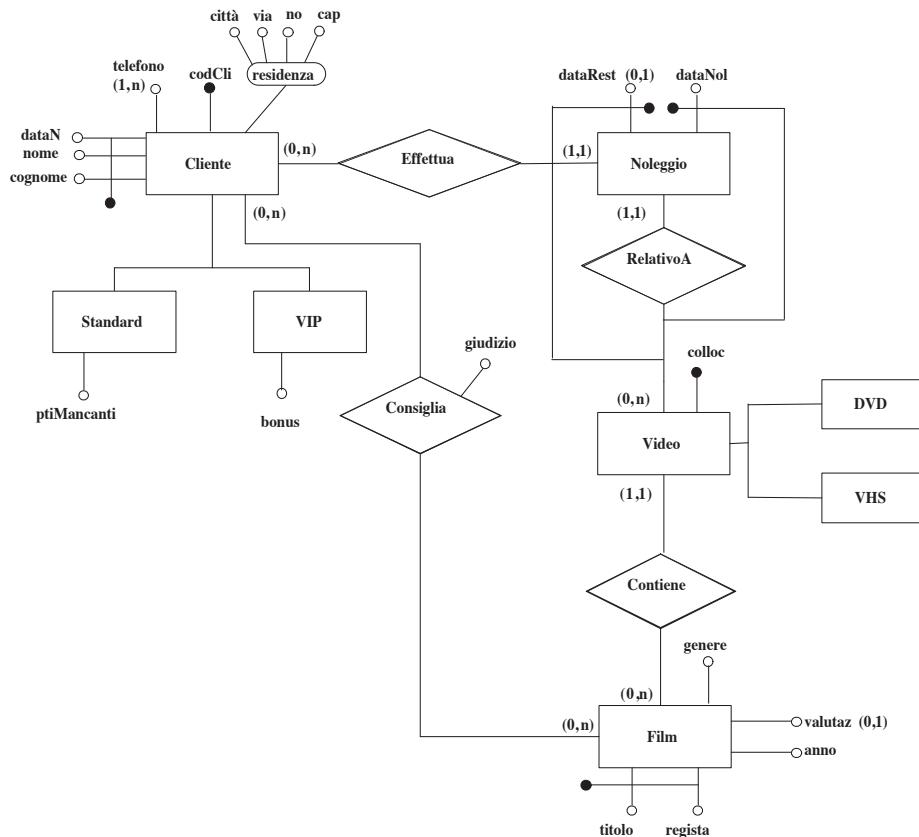


Figura 5.15: Diagramma ER finale per la base di dati della videoteca

Una prima bozza di schema ER potrebbe quindi essere quella rappresentata nella Figura 5.14. A questo punto, lo schema della Figura 5.14 può essere suddiviso in sotto-schemi (ad esempio uno per ogni entità ed associazione), che potranno essere sviluppati autonomamente sulla base delle specifiche ed integrati alla fine tenendo come traccia lo schema preliminare, ottenendo il diagramma della Figura 5.15. A titolo d'esempio, se consideriamo il sotto-schema relativo all'entità `Cliente`, leggendo le specifiche appare evidente come sia necessario mantenere informazioni su due diverse tipologie di clienti: standard e VIP. Può quindi essere instaurata una generalizzazione per modellare tali requisiti. Inoltre, un'attenta analisi delle specifiche evidenzia come non sia opportuno modellare il noleggio di un certo video da parte di un cliente tramite un'associazione, in quanto siamo interessati a memorizzare non solo i noleggi correnti ma anche quelli già effettuati. L'associazione `Noleggia` dello schema preliminare della Figura 5.14 è quindi sostituita da un'entità `Noleggio` in associazione con le entità `Cliente` e `Video`.

Vincoli d'integrità
<b>V<sub>1</sub>:</b> Un cliente non può noleggiare più di tre video contemporaneamente
<b>V<sub>2</sub>:</b> Un video non può essere noleggiato prima dell'uscita del film a cui è relativo
<b>V<sub>3</sub>:</b> La data di noleggio di un video non può essere successiva alla data di restituzione
<b>V<sub>4</sub>:</b> Uno stesso video non può essere noleggiato da due o più clienti diversi contemporaneamente

Tabella 5.2: Vincoli d'integrità per l'esempio della videoteca

È poi necessario definire gli attributi per ogni entità ed associazione e decidere se esistono proprietà che è opportuno modellare tramite attributi composti. Nello schema della Figura 5.15 è presente un attributo composto **residenza** che modella le varie componenti dell'indirizzo di residenza di un cliente. Infine, è necessario specificare gli identificatori ed i vincoli di cardinalità sia per attributi sia per associazioni. Notiamo dalla Figura 5.15 come tutte le entità presenti nello schema sono identificate da identificatori interni, ad eccezione dell'entità **Noleggio** che è caratterizzata da due identificatori misti, costituiti dall'entità **Video** e dagli attributi **dataRest** e **dataNol**, rispettivamente. Questa scelta implica che uno stesso video non possa essere noleggiato e restituito più volte nello stesso giorno.<sup>1</sup> Infine, per quanto riguarda i vincoli di cardinalità per attributi, i soli attributi che hanno una cardinalità diversa da quella di default sono **dataRest** e **valutaz** che possono assumere valore nullo, e **telefono** che è un attributo multi-valore in quanto un cliente può aver associati più numeri di telefono. Rispetto ai vincoli di cardinalità per associazioni, lo schema contiene tre associazioni uno a molti: **Effettua**, **RelativoA** e **Contiene**, ed una associazione molti a molti: **Consiglia**. La partecipazione all'associazione **Effettua** è obbligatoria per **Noleggio**, mentre è opzionale per **Cliente** (un cliente può non aver ancora noleggiato alcun video). Analogamente, l'associazione **RelativoA** è obbligatoria per **Noleggio** ed opzionale per **Video**. L'associazione **Contiene** è opzionale per **Film** ed obbligatoria per **Video**, mentre l'associazione **Consiglia** è opzionale per entrambe le entità partecipanti.

#### 5.4.3 Documentazione dello schema ER

Al diagramma della Figura 5.15 va aggiunta una documentazione in linguaggio naturale che deve contenere, oltre ad eventuali note sulle scelte di progetto, anche tutte quelle informazioni che non sono direttamente rappresentabili nello schema ER ma che sono necessarie in fase di progettazione logica. Più precisamente, la documentazione deve contenere:

<sup>1</sup>Questo vincolo, non presente nelle specifiche iniziali, può essere stabilito tramite colloqui successivi con i commitenti del progetto.

Entità padre	Entità figlie	Tipologia
Cliente	Standard, VIP	Totale/esclusiva
Video	DVD, VHS	Totale/esclusiva

Tabella 5.3: Informazioni sulle gerarchie di generalizzazione per l'esempio della videoteca

- I domini di tutti gli attributi presenti nello schema (vedi Esempio 5.5 per come queste informazioni possono essere rappresentate).
- Un elenco, in linguaggio naturale, di tutti i vincoli d'integrità e di autorizzazione non direttamente rappresentabili nel diagramma ER. Esempi di vincoli da inserire nella documentazione per il dominio della videoteca sono riportati nella Tabella 5.2 dove, per ragioni di semplicità, sono stati omessi i vincoli di autorizzazione. I vincoli possono essere desunti direttamente dal documento di specifica (vincolo  $V_1$  della Tabella 5.2), oppure essere vincoli di correttezza implicitamente desumibili dalle specifiche o rilevati tramite interazioni con i committenti del progetto (vincoli  $V_2$ ,  $V_3$  e  $V_4$  della Tabella 5.2). Anche quest'ultima tipologia di vincoli è di notevole importanza in quanto permette di effettuare dei controlli di correttezza sui dati inseriti nella base di dati.
- La tipologia di gerarchie di generalizzazione presenti nello schema. La Tabella 5.3 riassume tali informazioni per l'esempio della videoteca.

Può inoltre essere opportuno, soprattutto per schemi di dimensioni elevate, inserire nella documentazione a corredo dello schema ER, anche delle tabelle riasuntive delle entità ed associazioni presenti nello schema. Tali tabelle, chiamate anche *dizionari*, danno una visione di sintesi delle entità ed associazioni presenti nello schema. In particolare, il dizionario delle entità contiene, per ogni entità presente nello schema ER, il nome, una breve descrizione, i nomi degli attributi e gli identificatori. Il dizionario delle associazioni contiene, invece, per ogni associazione presente nel diagramma ER, il suo nome, una breve descrizione, il nome degli eventuali attributi ed il nome delle entità che mette in associazione. I dizionari delle entità e delle associazioni per l'esempio della videoteca sono riportati, rispettivamente, nelle Tabelle 5.4 e 5.5.

### Note conclusive

La progettazione è una fase di fondamentale importanza nello sviluppo di una base di dati. In questo capitolo, abbiamo analizzato le varie fasi della progettazione, concentrando sulla progettazione concettuale ed illustrando il principale modello e le principali metodologie a supporto di questa fase. La fase di progettazione logica è trattata nel Capitolo 6, dove illustreremo anche la teoria della normalizzazione

Nome	Descrizione	Attributi	Identifieri
Cliente	Utenti che usufruiscono dei servizi della videoteca	codCli, dataN, nome, cognome, telefono, residenza	codCli, {nome,cognome,dataN}
Standard	Cliente comune della videoteca	ptiMancanti	gli stessi di Cliente
VIP	Cliente VIP della videoteca	bonus	gli stessi di Cliente
Video	Video offerti dalla videoteca	colloc	colloc
Noleggio	Noleggi correnti e conclusi effettuati nella videoteca	dataNol, dataRest	{Video,dataNol} {Video,dataRest}
Film	Film offerti dalla videoteca	titolo, regista valutaz, anno, genere	{titolo,regista}
DVD	DVD offerti dalla videoteca		gli stessi di Video
VHS	Videocassette offerte dalla videoteca		gli stessi di Video

Tabella 5.4: Dizionario delle entità per l'esempio della videoteca

Nome	Descrizione	Attributi	Entità collegate
Effettuata	Noleggi effettuati		Cliente, Noleggio
RelativoA	Video noleggiati		Noleggio, Video
Contiene	Film noleggiati		Video, Film
Consiglia	Film consigliati dai clienti della videoteca	giudizio	Cliente, Film

Tabella 5.5: Dizionario delle associazioni per l'esempio della videoteca

come strumento per la verifica di qualità degli schemi logici ottenuti, mentre alcuni aspetti legati alla progettazione fisica saranno trattati nel Capitolo 7.

A conclusione del capitolo è opportuno sottolineare che, mentre per le basi di dati relazionali le metodologie di progettazione concettuale hanno raggiunto un buon livello di maturità, testimoniato anche dalla disponibilità di numerosi strumenti CASE a supporto dell'attività di progettazione, non esiste ancora un ampio consenso circa lo sviluppo di modelli e metodi per la progettazione concettuale di basi di dati di nuova generazione, quali le basi di dati relazionali ad oggetti, trattate nel Capitolo 10. Per tali basi di dati è possibile utilizzare ancora il modello ER. Il fatto però che il modello relazionale ad oggetti consenta la specifica di metodi associati ai tipi di dato definiti dall'utente, pone la necessità di utilizzare adeguati strumenti concettuali per la progettazione di tali aspetti, che non sono direttamente rappresentabili nel modello ER. Un'alternativa è rappresentata dall'utilizzo di UML (Unified Modeling Language) [Fow03], un modello concettuale

basato sul paradigma orientato ad oggetti, che può essere utilizzato per la fase di progettazione concettuale di una base di dati, indipendentemente dal modello logico che verrà adottato in fase di progettazione logica.

Per ragioni di spazio, alcuni aspetti relativi alla progettazione concettuale non sono stati trattati nel presente capitolo. Ne è un esempio il problema dell'integrazione di schemi concettuali, di particolare rilevanza quando viene utilizzata una metodologia bottom-up per la generazione dello schema concettuale finale, oppure quando devono essere integrati in un unico schema più schemi concettuali (ad esempio in seguito ad una ristrutturazione aziendale o quando devono essere integrate basi di dati di settori diversi della stessa azienda). Per tali aspetti, così come per una trattazione più approfondita delle metodologie di progettazione, rimandiamo il lettore ai testi consigliati nelle note bibliografiche.

### Note bibliografiche

Alla progettazione concettuale di basi di dati è dedicato il libro di Batini, Ceri e Navathe [BCN92]. Tale libro, oltre ad illustrare in dettaglio il modello ER, offre una panoramica dei principali strumenti per la progettazione assistita da calcolatore, che non sono stati trattati in questo capitolo. Il più recente testo di Harrington [Har02] presenta numerosi casi di studio di progettazione di basi di dati. Alla progettazione concettuale ed al modello ER è inoltre dedicato il libro in italiano di Batini et al. [BPLS91]. Il riferimento per il modello Entità-Relazione è [Che76]. Infine, per le metodologie di sviluppo software rimandiamo il lettore a [GFM<sup>+</sup>91].

### Esercizi

**5.1** Vogliamo realizzare una base di dati che mantenga informazioni sulle componenti di arredamento, da utilizzare per effettuare proposte di arredo ai clienti di uno studio di architettura. Ogni componente di arredo (ad esempio tavolo, sedie, ecc.) è caratterizzata da un codice identificativo, un nome, una breve descrizione, dalla data in cui è iniziata la sua produzione e dalle sue dimensioni. Per ogni componente, vogliamo mantenere informazioni sui colori in cui è disponibile e sui materiali (ad esempio legno, laminato plastico, ecc.). Ogni componente può essere disponibile in più colori ed in più materiali. Per ogni componente, vogliamo inoltre mantenere informazioni sull'azienda produttrice. In particolare, per ogni azienda, vogliamo mantenere il nome, l'indirizzo, il numero di telefono e di fax, ed il nome e cognome della persona di riferimento. Ogni componente può essere prodotta da più aziende, mentre la stessa azienda può produrre più componenti utilizzate dallo studio di architettura. Nella base di dati, vogliamo inoltre mantenere informazioni sui progetti che lo studio realizza. Per ogni progetto, vogliamo memorizzare le informazioni sul cliente per cui il progetto è stato sviluppato (nome, cognome, codice fiscale ed indirizzo), sulla data in cui il progetto è stato realizzato e sull'architetto che lo ha realizzato (nome, cognome e recapito). Per semplicità assumiamo che ogni progetto sia realizzato da un solo architetto, mentre lo stesso cliente può avere più progetti ad esso associati. Ogni progetto è composto del progetto di una serie di ambienti. Per ogni ambiente vogliamo mantenere il nome (cucina, soggiorno, ecc.), le dimensioni (lunghezza, larghezza e altezza) ed il colore del pavimento e delle pareti. Vogliamo inoltre mantenere informazioni sulle componenti di arredo che il progetto prevede di collocare in ogni ambiente, con l'indicazione del colore, del materiale scelto e della ditta fornitrice.

- a. Ristrutturare le specifiche secondo la metodologia illustrata nel Paragrafo 5.4.

- b. Definire uno schema ER per lo scenario sopra descritto, evidenziando identificatori e vincoli di cardinalità per attributi ed associazioni.
- c. Motivare le scelte progettuali effettuate, ove più alternative siano possibili.
- d. Produrre la documentazione a supporto dello schema ER generato, indicando in tale documentazione i vincoli del dominio non esprimibili nel modello ER ed i domini degli attributi.

**5.2** Vogliamo realizzare una base di dati a supporto del processo di accettazione e pubblicazione dei lavori scientifici su una rivista. Tale processo può essere schematizzato come segue. Gli autori di articoli scientifici che intendono proporre dei lavori ne devono inviare una copia in formato elettronico alla rivista scelta, accompagnata da una lettera che indichi il settore in cui è inquadrata la ricerca presentata nell'articolo e l'istituto, azienda o università di provenienza degli autori. Quando riceve l'articolo, l'incaricato della rivista assegna innanzitutto un codice unico all'articolo. Inoltre, rintraccia, nell'archivio ESPERTI, cinque nomi di persone esperte nel settore segnalato congiuntamente all'invio del lavoro. In particolare, tali persone non devono appartenere alle stesse organizzazioni a cui appartengono gli autori. Se più di cinque esperti sono disponibili, vengono selezionati gli esperti che hanno correntemente meno lavori da seguire. Quindi, l'incaricato contatta ciascuno di questi, per vedere se è disponibile ad effettuare la recensione del lavoro entro tre mesi. Se riceve qualche risposta negativa, l'incaricato cerca nell'archivio ESPERTI altri nomi, e continua a richiedere fino a quando ha collezionato una rosa di cinque nomi di esperti disponibili ad effettuare la recensione. Quindi, invia una copia del lavoro a ciascuna delle persone identificate e ne tiene una per sé, inserendola in un archivio. Se l'incaricato non riceve risposta entro tre mesi dall'invio dell'articolo, invia un sollecito a ciascuno dei ritardatari. Quando ha collezionato le cinque recensioni, egli decide, in base al giudizio degli esperti, quale dei seguenti messaggi inviare agli autori:

- l'articolo è stato accettato per la pubblicazione senza modifiche;
  - l'articolo è stato accettato ma occorre modificarlo. In tal caso, l'incaricato acclude al messaggio una spiegazione dettagliata delle modifiche necessarie ed attende di ricevere la nuova versione dell'articolo;
  - l'articolo non è stato accettato, ma viene suggerito all'autore di rispedirlo alla rivista dopo averlo modificato. Anche in questo caso, al messaggio viene acclusa una spiegazione delle modifiche, e l'incaricato attende di ricevere la nuova versione dell'articolo;
  - l'articolo viene rifiutato.
- a. Definire lo schema ER di una base di dati ARTICOLI che memorizzi tutte le informazioni riguardanti:
    - (1) gli articoli, i relativi autori, e lo stato corrente di ogni articolo;
    - (2) gli esperti;
    - (3) gli assegnamenti dei lavori agli esperti.
  - b. Produrre la documentazione a supporto dello schema ER generato, indicando in tale documentazione i vincoli del dominio non esprimibili nel modello ER ed i domini degli attributi.

**5.3** Definire uno schema ER relativo ad una base di dati per gestire la programmazione cinematografica giornaliera in un certo insieme di comuni italiani. Per ogni comune, vogliamo memorizzare il nome e la regione di appartenenza. Comuni con lo stesso nome possono appartenere solo a regioni diverse. In ogni comune sono presenti dei cinema. Per ogni cinema, vogliamo memorizzare il nome, l'indirizzo, il numero di telefono ed il numero di sale disponibili. Cinema con lo stesso nome possono essere presenti solo in comuni diversi. Ogni cinema ha un certo numero di sale di proiezione (almeno una). Per ogni sala, vogliamo mantenere informazioni sul nome della sala (sale con lo stesso nome possono appartenere solo a cinema diversi) ed il

numero di posti disponibili. In ogni sala vengono proiettati dei film ad orari stabiliti. Per ogni film, vogliamo memorizzare il titolo, il nome ed il cognome del regista e la durata. Ogni film viene proiettato in una o più sale. In una stessa sala possono essere proiettati anche film diversi, in diversi orari. Per ogni proiezione vogliamo memorizzare l'orario d'inizio e la durata prevista per la proiezione.

**5.4** Vogliamo realizzare una base di dati relativa ad una rete bancaria informatizzata. La rete bancaria è costituita da un consorzio di banche che condividono un insieme di sportelli automatizzati (sportelli Bancomat). Ogni banca è individuata da un codice, ha un proprio nome, un insieme di filiali, ed un insieme di impiegati. Di ogni impiegato vogliamo conoscere il nome, l'indirizzo dell'abitazione e la data d'assunzione. Nel corso della vita lavorativa, un impiegato può prestare servizio in diverse filiali della stessa banca, a condizione di una permanenza minima di un mese in una filiale. Ogni filiale è caratterizzata da un codice, ha un indirizzo, dei propri conti, delle postazioni di cassa e sportelli automatizzati e rilascia delle carte Bancomat. Ogni conto è relativo ad uno o più clienti ed ha una data di apertura, un'eventuale data di estinzione e (se ancora aperto) un ammontare totale. Ogni cliente ha un nome, un indirizzo, ed è identificato dal codice fiscale. Un cliente può possedere una o più carte Bancomat, ognuna delle quali associate ad uno dei suoi conti. Un cliente della banca può inoltre essere un impiegato della stessa, in tal caso non ha alcun costo di gestione per le operazioni eseguite presso le filiali della banca stessa, o presso gli sportelli automatizzati del consorzio a cui la banca appartiene. Ogni cassa o sportello automatizzato ha una certa quantità di denaro disponibile, che viene aggiornata in caso di depositi/prelievi. Ovviamente non è possibile effettuare prelievi superiori al denaro disponibile. Sia presso le casse sia presso gli sportelli automatizzati è possibile eseguire operazioni bancarie, le quali avvengono in una certa data ed ora e sono relative ad un certo conto. Attraverso uno sportello automatizzato è possibile effettuare operazioni di deposito, saldo, lista movimenti e prelievo. Alle operazioni di prelievo ogni banca associa due diversi costi: uno per le operazioni di prelievo eseguite presso stazioni automatizzate di banche appartenenti allo stesso consorzio ed un altro per operazioni eseguite presso stazioni automatizzate di altre banche. Una carta Bancomat ha una password, una data iniziale, una data di scadenza oltre la quale non è più valida, un limite di spesa complessivo giornaliero e mensile ed un limite di spesa per singolo prelievo. Questi limiti, insieme alla durata della carta Bancomat, sono uguali per tutte le carte rilasciate dalla stessa banca.

- a. Ristrutturare le specifiche secondo la metodologia illustrata nel Paragrafo 5.4.
- b. Definire lo schema ER di una base di dati per il dominio sopra descritto.
- c. Produrre la documentazione a supporto dello schema ER generato, indicando in tale documentazione i vincoli del dominio non esprimibili nel modello ER ed i domini degli attributi.

## Capitolo 6

# Progettazione logica e verifica della qualità

Come abbiamo visto nel Capitolo 5, la progettazione concettuale produce uno schema concettuale che rappresenta ad alto livello il contenuto di una base di dati, in modo del tutto indipendente dal particolare sistema di gestione dati che utilizzeremo per la sua implementazione. Ne consegue che, scelto un DBMS, per sviluppare la base di dati è necessario tradurre lo schema concettuale in uno schema logico per il sistema di gestione dati prescelto. Questa fase è chiamata *progettazione logica*. Il risultato della fase di progettazione logica è uno schema logico direttamente implementabile in uno specifico DBMS. Mentre la progettazione concettuale ha come obiettivo primario una rappresentazione formale e non ambigua dei dati di interesse, la progettazione logica rappresenta il punto di partenza per la realizzazione della base di dati e delle relative applicazioni; deve, quindi, tenere conto anche di aspetti prestazionali, non considerati a livello concettuale.

Poiché le attività di progettazione possono essere soggette ad errori ed incompletezze, al termine della fase di progettazione logica è opportuno effettuare un'analisi della qualità dello schema logico ottenuto e, se necessario, ristrutturare parzialmente lo schema. A questo scopo, nell'ambito del modello relazionale, è stata sviluppata la *teoria della normalizzazione*, finalizzata a fornire indicazioni per progettare schemi relazionali di qualità, cioè schemi relazionali che non siano affetti da *anomalie*. In particolare, questa teoria può essere utilizzata per “migliorare” gli schemi logici, generando nuovi schemi equivalenti a quelli di partenza ma non affetti da anomalie. Sebbene gli sviluppi relativi alla teoria della normalizzazione siano stati fortemente teorici, come testimoniato dai numerosissimi lavori scientifici al riguardo, l'applicazione della normalizzazione alla progettazione di basi di dati relazionali è ormai di uso abbastanza comune.

L'obiettivo di questo capitolo è quindi duplice. Per prima cosa, verrà introdotta la fase di progettazione logica, illustrando una metodologia per la traduzione di schemi ER in schemi relazionali. Verranno quindi brevemente discusse le anomalie che si possono riscontrare negli schemi relazionali generati, presentando infine le nozioni alla base della teoria della normalizzazione che permettono di eliminare o ridurre la presenza di tali anomalie.

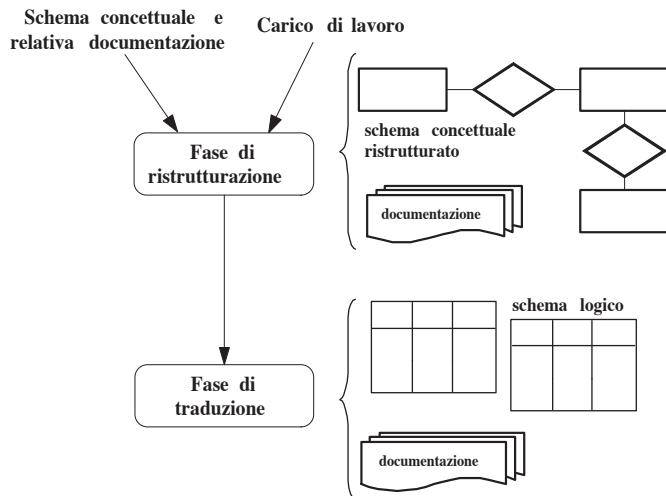


Figura 6.1: Fasi della progettazione logica di una base di dati

### 6.1 Progettazione logica

L’obiettivo principale della progettazione logica è tradurre uno schema ER, ottenuto come risultato della fase di progettazione concettuale, in uno schema relazionale equivalente, che rappresenti cioè le stesse informazioni, e che tenga anche conto di aspetti legati alle prestazioni delle operazioni che verranno eseguite sulla base di dati. La traduzione dello schema ER nello schema relazionale può essere vista come un’attività incrementale. Essa prende in input lo schema ER con la relativa documentazione ed il *carico di lavoro*, descritto in termini di dimensioni dei dati e caratteristiche delle operazioni da eseguire sulla base di dati, e genera uno schema relazionale equivalente allo schema ER di partenza ma ottimizzato rispetto al carico di lavoro, con la relativa documentazione.

La progettazione logica si articola in due fasi principali, la *ristrutturazione* e la *traduzione* dello schema ER, come illustrato nella Figura 6.1:

- **Fase di ristrutturazione dello schema ER.** Lo scopo principale di questa fase è generare uno schema ER semplificato, ma equivalente a quello di partenza, al fine di semplificarne la traduzione successiva. La fase di ristrutturazione prevede quindi l’eliminazione dallo schema ER di tutti quei costrutti non direttamente rappresentabili nel modello relazionale. Questa fase include inoltre ulteriori ristrutturazioni dello schema che tengano conto di aspetti prestazionali, identificati dall’analisi del carico di lavoro. Più precisamente, la fase di ristrutturazione è composta dalle seguenti sotto-fasi: (i) analisi della ridondanza; (ii) partizionamento/accorpamento di entità; (iii) eliminazione degli attributi composti e multi-valore; (iv) eliminazione delle

gerarchie di generalizzazione. Per ciascuna sotto-fase di ristrutturazione, possono esistere più alternative. La scelta di una soluzione specifica dipende dal carico di lavoro e da considerazioni relative alla conseguente realizzazione della base di dati che stiamo progettando.

- **Fase di traduzione dello schema ER.** In questa fase, lo schema ER restituito dalla fase di ristrutturazione (*schema ER ristrutturato*) viene tradotto in un equivalente schema relazionale. Tale traduzione avviene mediante l'applicazione di un insieme di *regole di trasformazione* che specificano come tradurre entità, attributi ed associazioni del modello ER in termini di relazioni ed attributi del modello relazionale. Anche in questo caso, la traduzione non è sempre univoca e la scelta di una delle soluzioni possibili dipende da considerazioni di carattere prestazionale.

L'output della fase di progettazione logica è costituito da uno schema logico e da documentazione di supporto. È opportuno osservare che, mentre la documentazione generata dalla fase di progettazione concettuale contiene anche informazioni relative ai vincoli di integrità e di autorizzazione, che non possono essere espresi nel modello ER (e che quindi non fanno parte dello schema concettuale), tali vincoli possono essere rappresentati, opportunamente tradotti, nello schema relazionale. Durante la creazione dello schema risultante nel DBMS prescelto, ogni vincolo potrà infatti essere tradotto in SQL come segue: utilizzando vincoli CHECK od asserzioni, se è un vincolo di integrità statico (vedi Capitolo 3); utilizzando trigger, se è un vincolo di transizione (vedi Capitolo 11); utilizzando opportuni comandi di GRANT, se è un vincolo di autorizzazione (vedi Capitolo 9). I vincoli rappresentano quindi una componente dello schema logico e non fanno parte della documentazione aggiuntiva. Un'analoga considerazione vale per i domini degli attributi, che possono essere rappresentati nello schema delle relazioni. La documentazione generata dalla progettazione logica conterrà quindi solo i dizionari per le relazioni presenti nello schema ed eventualmente dettagli sulle scelte progettuali effettuate.

Nei paragrafi successivi, illustreremo in dettaglio le varie fasi del processo di progettazione e le attività in esse coinvolte. Per chiarire al meglio i concetti introdotti, utilizzeremo sia esempi relativi al dominio della videoteca, modificando se necessario alcuni dei requisiti presentati nel Capitolo 5, sia esempi relativi ad altri domini, se ritenuti più rilevanti per illustrare la teoria esposta.

## 6.2 Fase di ristrutturazione

Lo scopo principale della fase di ristrutturazione è l'eliminazione dallo schema ER dei costrutti che non hanno una rappresentazione immediata nel modello relazionale. Tali costrutti sono: gli attributi composti e multi-valore e le gerarchie di generalizzazione. In questa fase vengono inoltre applicate ulteriori modifiche allo schema concettuale guidate da informazioni contenute nel carico di lavoro, come l'eliminazione della ridondanza ed il partizionamento/accorpamento di entità.

### 6.2.1 Analisi della ridondanza

Uno schema ER presenta delle *ridondanze* quando un'informazione viene rappresentata sia esplicitamente nello schema sia può essere derivata da altre informazioni (ad esempio attributi od associazioni) presenti nello schema. Sono esempi di ridondanza la presenza di cicli tra le associazioni o di attributi il cui valore può essere derivato da altri attributi ed/od associazioni.

L'eliminazione dei costrutti ridondanti permette di semplificare lo schema ER e quindi generare successivamente uno schema relazionale a sua volta non ridondante. Notiamo che un dato ridondante nello schema logico comporta sempre una maggiore occupazione di spazio ed un appesantimento delle procedure di aggiornamento, in quanto i dati ridondanti dovranno sempre essere aggiornati contemporaneamente, al fine di evitare la generazione di inconsistenze. D'altro canto, la presenza di un dato ridondante può avere il vantaggio di rendere più efficienti alcune interrogazioni descritte nel carico di lavoro. In linea generale possiamo quindi affermare che nei diagrammi ER la presenza di ridondanza dovrebbe essere limitata solo a quei casi in cui sia possibile ottenere un significativo beneficio in termini di tempo di esecuzione delle interrogazioni. Naturalmente, la decisione sul mantenere o meno una ridondanza dovrà anche tenere in considerazione la frequenza di esecuzione delle interrogazioni che beneficiano in termini di tempo di esecuzione della presenza del dato ridondante.

Benché una trattazione dettagliata delle problematiche relative alla ridondanza esuli dagli obiettivi di questo capitolo, è opportuno sottolineare come le valutazioni da effettuare per l'eventuale eliminazione di entità od associazioni ridondanti, considerando uno schema concettuale e non logico, saranno basate su stime piuttosto che su valori effettivi. In particolare, per stimare l'occupazione di memoria del dato ridondante è necessario poter disporre, derivandole dal documento di specifica o tramite successive interazioni con i committenti, di informazioni sul *volumen* dei dati, in termini di numero di istanze delle entità e delle associazioni che compongono il diagramma ER. Queste informazioni saranno anche utilizzate per stimare il costo delle operazioni in presenza oppure in assenza di ridondanza. La stima in questo caso avviene a partire dalla porzione di schema ER coinvolta nell'operazione, stimando il numero di letture e/o scritture necessarie per eseguire l'operazione in questione.

**Esempio 6.1** Consideriamo il diagramma ER della Figura 5.15. Se il diagramma contenesse un attributo `numNoleggi` associato all'entità `Cliente`, questo attributo sarebbe ridondante in quanto il numero totale di noleggi effettuati da un certo cliente può essere ottenuto mediante il conteggio delle istanze dell'associazione `Effettua`, che legano il cliente considerato ai noleggi da lui effettuati. In questo caso, il noleggio di un nuovo video da parte di un cliente comporterebbe, oltre ad un aggiornamento delle istanze dell'entità `Noleggio`, anche un aggiornamento del valore dell'attributo `numNoleggi` per il cliente considerato, appesantendo la procedura di aggiornamento. Tuttavia, se il carico di lavoro prevedesse un'operazione di stampa di un report contenente per ogni cliente il numero totale di

noleggi, la presenza dell'attributo `numNoleggi` renderebbe più veloce l'esecuzione di questa operazione. Nel caso in cui questa operazione venga eseguita molto frequentemente, potrebbe quindi essere ragionevole mantenere l'attributo ridondante, anche perché lo spazio aggiuntivo richiesto è comunque limitato: sulla base del volume dei dati presentato nel Paragrafo 5.4, i clienti sono circa 2'000 e quindi lo spazio necessario corrisponde a quello richiesto dalla memorizzazione di 2'000 valori interi (circa 8Mb).  $\square$

### 6.2.2 Partizionamento ed accorpamento di entità

Lo schema ER può essere ulteriormente ristrutturato partizionando od accorpan-  
do entità ed associazioni sulla base dell'analisi del carico di lavoro. In particolare,  
un'entità  $E$  può essere partizionata in due entità  $E_1$  ed  $E_2$ , una delle quali identifi-  
cata esternamente dall'altra, collegate mediante un'associazione uno a uno. Questa  
operazione può essere conveniente quando alcune operazioni frequenti coinvolgono  
solo un sottoinsieme degli attributi di  $E$ .

Viceversa, due entità  $E_1$  ed  $E_2$ , collegate da un'associazione uno a uno, possono  
essere accorpate in un'unica entità contenente gli attributi di  $E_1$  ed  $E_2$  nel caso in  
cui operazioni frequenti abbiano la necessità di accedere ad entrambi gli insiemi di  
attributi. L'accorpamento permette quindi di evitare la navigazione dell'associa-  
zione durante l'esecuzione di tali operazioni. Notiamo che l'accorpamento può  
generare attributi opzionali in caso di partecipazione opzionale all'associazione di  
almeno un'entità.

È opportuno ricordare che, benché le operazioni di partizionamento e di ac-  
corpamento possano già essere eseguite nella fase di ristrutturazione, spesso ven-  
gono rimandate alla fase di progettazione fisica, in cui saranno disponibili ulteriori  
informazioni relative all'esecuzione delle interrogazioni (vedi Capitolo 7).

**Esempio 6.2** Consideriamo il diagramma ER della Figura 6.2(a). L'entità `Cliente` è caratterizzata da dati anagrafici e dati relativi alla storia dei noleggi effettuati. Se le operazioni sulla base di dati richiedono alternativamente l'accesso a dati anagrafici o di noleggio, è possibile partizionare l'entità `Cliente`, ottenendo lo schema della Figura 6.2(b), che risulta più efficace per l'esecuzione di tali operazioni. Viceversa, lo schema della Figura 6.2(b) potrebbe essere accorpato ottenendo lo schema della Figura 6.2(a) più efficiente, nel caso in cui le operazioni non distinguano tra dati anagrafici e di noleggio.  $\square$

### 6.2.3 Eliminazione degli attributi composti e multi-valore

Il modello relazionale consente la specifica solo di attributi semplici e mono-valore.  
È quindi necessario ristrutturare lo schema ER generato dalla fase di proget-  
tazione concettuale per eliminare eventuali attributi composti e multi-valore in  
esso presenti.

Consideriamo in primo luogo gli attributi composti, cioè attributi costituiti da  
un certo numero di sotto-attributi. Consideriamo il caso in cui i sotto-attributi

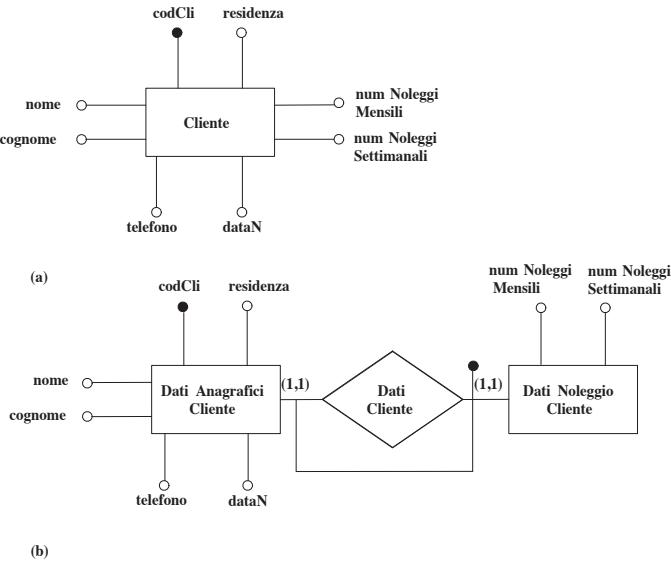


Figura 6.2: Partizionamento ed accorpamento di entità: (a) una singola entità; (b) due entità associate fra loro che rappresentano la stessa informazione

siano attributi semplici. L'eliminazione di un attributo composto  $A$  da un'entità  $E$  può avvenire in due modi: (i) eliminando i sotto-attributi di  $A$  e considerando l'attributo composto come un attributo semplice; (ii) considerando tutti i sotto-attributi di  $A$  come attributi di  $E$ . Quest'ultima soluzione richiede ovviamente una ridefinizione del dominio dell'attributo. Eventuali vincoli di cardinalità esistenti per l'attributo composto vengono associati a ciascuno dei nuovi attributi generati tramite la ristrutturazione. Notiamo che, adottando la prima soluzione, è compito dell'applicazione garantire che il nuovo attributo contenga valori coerenti con la semantica dell'attributo composto ristrutturato, mentre con la seconda soluzione si perde la relazione che esiste tra i sotto-attributi.

**Esempio 6.3** Consideriamo l'entità *Cliente* della Figura 6.3(a), contenente l'attributo composto *residenza* con dominio `string × string × string × integer`. La prima alternativa per l'eliminazione dell'attributo *residenza*, illustrata nella Figura 6.3(b), consiste nel considerare *residenza* come un attributo semplice di tipo `string`. In questo caso, le applicazioni dovranno garantire che le stringhe inserite come valore per *residenza* rappresentino effettivamente un indirizzo. La seconda alternativa consiste nella sostituzione dell'attributo *residenza* con gli attributi *città*, *via*, *no.*, di tipo `string`, e *cap*, di tipo `integer`. Tale soluzione è illustrata nella Figura 6.3(c).  $\square$

Se le componenti dell'attributo composto sono a loro volta attributi composti, la procedura illustrata in precedenza può essere applicata ricorsivamente alle

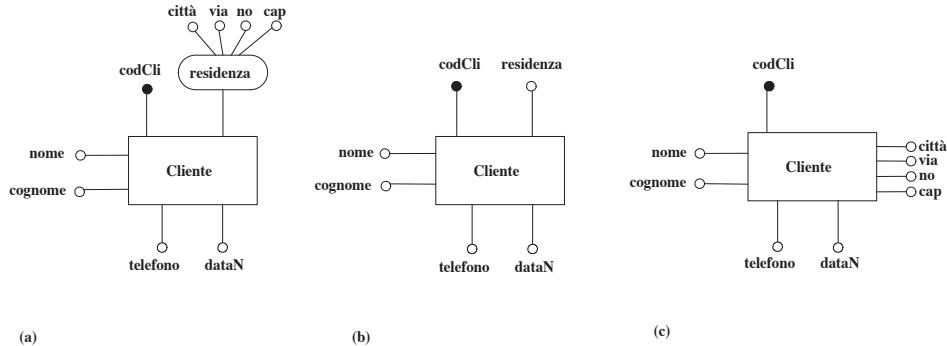


Figura 6.3: Eliminazione di un attributo composto: (a) schema iniziale; (b) attributo composto come attributo semplice; (c) attributo composto come insieme delle sue componenti

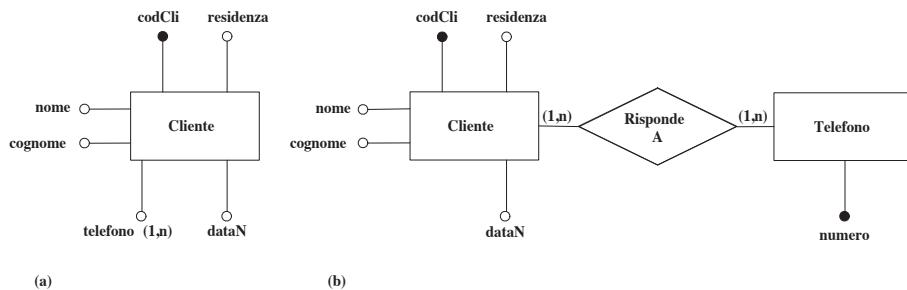


Figura 6.4: Eliminazione di un attributo multi-valore: (a) schema iniziale; (b) schema finale

componenti fino ad arrivare ad un attributo composto formato solo da attributi semplici.

La modellazione di attributi multi-valore mediante attributi a valore semplice richiede, invece, la definizione di una nuova entità, collegata all’entità di partenza tramite un’opportuna associazione, in cui l’attributo multi-valore è rappresentato mediante un attributo mono-valore che identifica l’entità. Il vincolo di cardinalità rispetto alla nuova associazione, per l’entità che conteneva prima della ristrutturazione l’attributo multi-valore, coinciderà con il vincolo di cardinalità dell’attributo multi-valore oggetto di ristrutturazione, mentre il vincolo di cardinalità per la nuova entità può essere in generale posto uguale a (1,n).

**Esempio 6.4** Consideriamo l’entità **Cliente** della Figura 6.4(a), contenente l’attributo multi-valore **telefono**. Nella ristrutturazione, tale entità viene scomposta in due entità: l’entità **Cliente**, contenente tutti gli attributi dell’entità di partenza ad eccezione dell’attributo multi-valore **telefono**, e l’entità **Telefono**, con un at-

tributo `numero` che rappresenta ciascun elemento dell'attributo composto iniziale, legata dall'associazione `RispondeA` all'entità `Cliente`. Il diagramma risultante è illustrato nella Figura 6.4(b).  $\square$

#### 6.2.4 Eliminazione delle gerarchie di generalizzazione

Il modello ER, a differenza del modello relazionale, consente la specifica di gerarchie di generalizzazione. Durante la fase di ristrutturazione, è quindi necessario rappresentare le gerarchie di generalizzazione presenti nello schema ER di partenza tramite costrutti equivalenti che abbiano una traduzione diretta nel modello relazionale.

Consideriamo un'entità  $E$  generalizzazione di un insieme di entità  $E_1, \dots, E_n$ . Per eliminare la gerarchia di generalizzazione che lega  $E$  ad  $E_1, \dots, E_n$  è inizialmente necessario recuperare informazioni relative al tipo di tale gerarchia (totale o parziale, esclusiva o condivisa) dalla documentazione di supporto generata dalla fase di progettazione concettuale. Possiamo quindi adottare tre alternative, descritte nel seguito:

1. **Eliminazione delle entità figlie.** Le entità  $E_1, \dots, E_n$  vengono eliminate ed i loro attributi vengono inseriti nell'entità padre come attributi opzionali. All'entità padre viene inoltre aggiunto un attributo che specifica da quale entità figlia nello schema originario proviene ciascuna istanza dell'entità padre nello schema ristrutturato. Nel caso di generalizzazioni totali, tale attributo non può mai assumere valore nullo, mentre nel caso di generalizzazioni parziali un valore nullo indica un'istanza dell'entità padre che, nello schema originario, non era istanza di alcuna delle entità figlie. Nel caso di generalizzazioni condivise, l'attributo sarà multi-valore in quanto un'istanza dell'entità padre potrebbe essere istanza di più entità figlie.<sup>1</sup> Per ogni altro attributo inserito nell'entità padre, è inoltre necessario aggiungere un vincolo di integrità che indichi quando tale attributo può assumere un valore nullo, sulla base del tipo dell'istanza considerata e sulla base del tipo di generalizzazione (totale o parziale, condivisa od esclusiva). Se la generalizzazione è totale, gli attributi di almeno un'entità figlia  $E_i$  dovranno essere obbligatori. Se è esclusiva, gli attributi di al più un'entità figlia  $E_i$  dovranno essere obbligatori. Infine, la partecipazione (obbligatoria od opzionale) di un'entità figlia ad un'associazione viene sostituita con la partecipazione opzionale dell'entità padre alla stessa associazione. Per ogni associazione, è inoltre necessario aggiungere un vincolo di integrità che indichi quali tipi di istanze dell'entità padre possono essere coinvolti nell'associazione.

**Esempio 6.5** Consideriamo la gerarchia di generalizzazione presentata nella Figura 6.5(a) e supponiamo che sia totale ed esclusiva. Supponiamo di

---

<sup>1</sup>L'attributo multi-valore può essere ristrutturato utilizzando la tecnica descritta nel Paragrafo 6.2.3.

ristrutturare questa gerarchia eliminando le entità figlie. Questo comporta l'eliminazione delle entità **Standard** e **VIP** e l'inserimento degli attributi opzionali **ptiMancanti** e **bonus** nell'entità **Cliente**. Inoltre, all'entità **Cliente** viene aggiunto un attributo mono-valore **tipo** per mantenere la distinzione tra le occorrenze dell'entità **Cliente** rappresentata dalla gerarchia di generalizzazione, con dominio {‘standard’,‘vip’}. Poiché la generalizzazione è totale, tale attributo è obbligatorio. Inoltre, poiché la generalizzazione è esclusiva, sono necessari alcuni vincoli di integrità per indicare che per ogni istanza di **Cliente**, esattamente uno tra gli attributi **ptiMancanti** e **bonus**, in relazione al tipo del cliente, assumerà valore nullo. Il diagramma risultante è illustrato nella Figura 6.5(b), insieme all'elenco dei vincoli di integrità generati.  $\square$

2. **Eliminazione dell'entità padre.** Questa soluzione, simmetrica rispetto alla precedente, consiste nell'eliminare l'entità padre  $E$  e nell'inserire i suoi attributi in ciascuna delle entità figlie. Chiaramente, tale procedimento è applicabile solo nel caso in cui la generalizzazione sia totale, altrimenti le istanze dell'entità padre che non sono istanze di alcuna entità figlia non verrebbero rappresentate nello schema ristrutturato. In caso di generalizzazione esclusiva, è necessario aggiungere un vincolo per indicare che, nello schema ristrutturato, non possono esistere istanze di due entità figlie distinte aventi lo stesso valore per gli identificatori. Ogni associazione a cui partecipava l'entità padre viene inoltre sostituita con  $n$  nuove associazioni, una per ogni entità figlia. Il vincolo di cardinalità di ciascuna entità figlia rispetto alla nuova associazione coinciderà con il vincolo di cardinalità dell'entità padre rispetto all'associazione eliminata. I vincoli di cardinalità delle altre entità diventeranno invece opzionali, in quanto non è noto a quale sottotentità erano associate prima della ristrutturazione. L'applicazione di questa soluzione alla gerarchia di generalizzazione della Figura 6.5(a) è illustrata nella Figura 6.5(c).
3. **Sostituzione della generalizzazione con associazioni.** Le entità coinvolte nella generalizzazione non vengono modificate, mentre la gerarchia di generalizzazione viene sostituita da  $n$  associazioni uno a uno, ognuna delle quali lega l'entità padre con una diversa entità figlia. Le entità figlie sono identificate esternamente dall'entità padre e partecipano obbligatoriamente alle associazioni create mentre la partecipazione dell'entità padre è opzionale. È inoltre necessario aggiungere dei vincoli. Infatti, se la generalizzazione è esclusiva, un'istanza dell'entità padre non può partecipare contemporaneamente a due o più associazioni. Inoltre, se la generalizzazione è totale, ogni istanza dell'entità padre deve partecipare obbligatoriamente ad almeno un'associazione. La ristrutturazione del diagramma della Figura 6.5(a) secondo questa soluzione è illustrata nella Figura 6.5(d).

La scelta della soluzione da adottare per l'eliminazione delle gerarchie di ge-

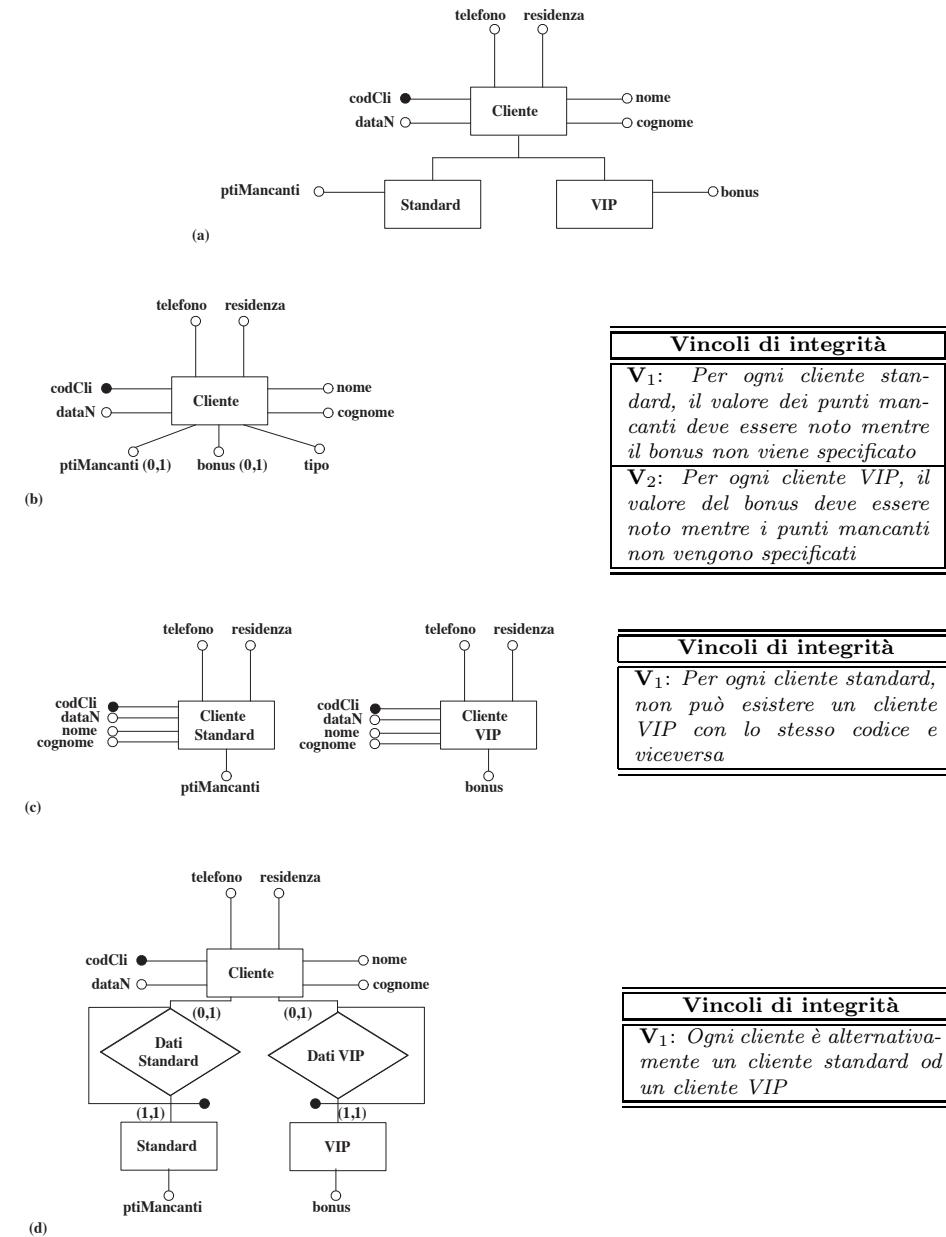


Figura 6.5: Eliminazione di una gerarchia di generalizzazione: (a) schema iniziale; (b) eliminazione delle entità figlie; (c) eliminazione dell'entità padre; (d) sostituzione della gerarchia di generalizzazione con associazioni

neralizzazione dipende dal tipo di operazioni da eseguire sulle entità, quindi dal carico di lavoro. In generale, la scelta di accorpate le entità figlie nell'entità padre comporta uno spreco di memoria per la presenza dei valori nulli. Tale scelta è quindi conveniente solo nel caso in cui le operazioni non fanno distinzione tra le varie sotto-entità. La soluzione di eliminare l'entità padre consente, invece, un risparmio di memoria rispetto alla soluzione di eliminare le entità figlie in quanto evita il problema dei valori nulli. Tale soluzione è conveniente soprattutto nel caso in cui esistano operazioni che si riferiscono alle istanze di una specifica entità figlia. Notiamo però come tale soluzione sia applicabile solo quando la generalizzazione è totale. Infine, la soluzione di sostituire la generalizzazione con delle associazioni è, analogamente alla soluzione di eliminare l'entità padre, preferibile alla soluzione di eliminare le entità figlie per quanto riguarda la quantità di memoria utilizzata. Tale soluzione risulta conveniente quando esistono delle operazioni che discriminano tra entità padre ed entità figlie in quanto consente, rispetto alla soluzione di eliminare l'entità padre, di accedere ad un numero minore di attributi. In alcune situazioni, può inoltre essere conveniente adottare soluzioni ibride, ottenute per combinazioni delle soluzioni elencate. Per esempio, possiamo decidere di eliminare solo un sottoinsieme delle entità figlie, mantenendo le altre nello schema. Notiamo infine che, in caso di generalizzazioni a più livelli, è possibile applicare le strategie illustrate partendo dalle foglie della gerarchia complessiva. Lo schema risultante dipenderà dal tipo della ristrutturazione applicata ad ogni livello.

### 6.3 Fase di traduzione

Lo scopo della fase di traduzione è generare, a partire dallo schema ER restituito dalla fase di ristrutturazione, un equivalente schema relazionale. La metodologia seguita è basata sull'applicazione di un insieme di regole di traduzione che consentono di mappare i costrutti del modello ER in costrutti equivalenti del modello relazionale. La fase di traduzione può essere suddivisa nelle seguenti sotto-fasi: (i) traduzione delle entità; (ii) traduzione delle associazioni; (iii) traduzione dei vincoli di integrità; (iv) ottimizzazioni finali.

Per quanto riguarda il punto (i), la traduzione delle entità nel modello relazionale avviene creando una relazione per ogni entità dello schema ER ristrutturato. La relazione contiene un attributo per ogni attributo dell'entità; contiene inoltre, come chiave esterna, una chiave per ogni relazione che rappresenta un'entità che identifica esternamente od in modo misto l'entità considerata. Tale chiave esterna rappresenta anche l'associazione attraverso cui l'entità viene identificata in modo esterno o misto. Queste associazioni non devono quindi essere ulteriormente tradotte. Le chiavi di ciascuna relazione risultante sono costituite dagli insiemi di attributi corrispondenti agli identificatori (interni, esterni e misti) dell'entità di partenza. Tra tutte le chiavi, sarà necessario individuarne una come chiave primaria.

Il metodo con cui le associazioni vengono tradotte (escluse, come abbiamo detto, le associazioni relative ad identificatori esterni o misti) dipende invece sia

dal numero di entità partecipanti all'associazione sia dai vincoli di cardinalità dell'associazione stessa. In generale, sono possibili due diverse alternative:

1. l'associazione viene modellata inserendo opportuni attributi come chiavi esterne in una delle relazioni rappresentanti le entità che partecipano all'associazione;
2. l'associazione viene realizzata creando una nuova relazione.

Ogni vincolo di integrità per lo schema ristrutturato deve essere tradotto in un vincolo equivalente per lo schema relazionale. Inoltre, poiché i domini di tipo intervallo e di tipo enumerazione relativi ad attributi dello schema concettuale non hanno una diretta rappresentazione nel modello relazionale (vedi Capitoli 2 e 3), è necessario aggiungere all'insieme dei vincoli quelli imposti da tali domini. Altri vincoli devono inoltre essere definiti per ogni entità che partecipa in modo obbligatorio ad una associazione. Ogni vincolo deve infatti specificare che ogni tupla della relazione corrispondente a tale entità deve comparire (tramite i suoi valori chiave) nella relazione che modella l'associazione.

Lo schema relazionale ottenuto applicando le regole introdotte in precedenza può poi essere ulteriormente ottimizzato, sulla base di considerazioni di efficienza, eliminando ad esempio eventuali ridondanze, sorte durante la traduzione.

Nei paragrafi successivi illustreremo le soluzioni da adottare per la traduzione delle entità e di ogni tipo di associazione. Alcuni esempi di ottimizzazioni finali e di traduzione dei vincoli di integrità verranno, invece, discussi nel contesto di un esempio complessivo, presentato nel Paragrafo 6.4. Per semplicità di presentazione, non discuteremo ulteriormente i vincoli derivati dalla partecipazione obbligatoria delle entità alle associazioni.

### 6.3.1 *Entità*

Ogni entità  $E_1$  dello schema ER ristrutturato viene tradotta in una corrispondente relazione  $R_1$ . Distinguiamo due casi, in relazione al fatto che  $E_1$  abbia o meno identificatori esterni o misti.

Se  $E_1$  non ha identificatori esterni o misti,  $R_1$  contiene un attributo per ogni attributo di  $E_1$ . Informazioni sul dominio di tali attributi possono essere recuperate dalla documentazione generata al termine della fase di progettazione concettuale. Per tutti gli attributi obbligatori deve inoltre essere specificato un corrispondente vincolo di obbligatorietà a livello logico (che tradurremo in un vincolo NOT NULL nella corrispondente specifica in SQL, vedi Capitolo 3). Ogni identificatore interno (semplice o composto) di  $E_1$  rappresenta una chiave per  $R_1$ .

Se  $E_1$  invece ha almeno un identificatore esterno o misto, oltre a quanto sopra specificato, è necessario rappresentare ciascuno di essi in  $R_1$  con una chiave della relazione che rappresenta l'entità che identifica  $E_1$ . Più precisamente, sia  $E_2$  un'entità che identifica, in modo esterno o misto, l'entità  $E_1$  attraverso l'associazione  $A$ . Sia  $R_2$  la relazione corrispondente ad  $E_2$ . Consideriamo i seguenti casi:

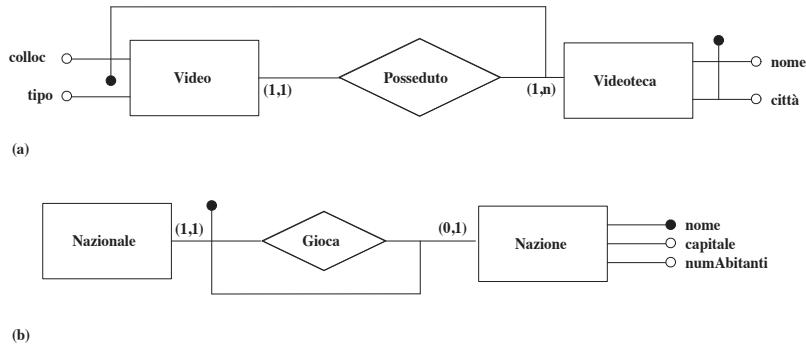


Figura 6.6: Entità con: (a) identificatore misto; (b) identificatore esterno

- **$E_2$  ha solo identificatori interni.** In questo caso, l'identificatore esterno o misto di  $E_1$  può essere rappresentato inserendo una chiave di  $R_2$  in  $R_1$ , mantenendo in  $R_1$  inalterati i vincoli che gli attributi corrispondenti hanno in  $R_2$ . Tali attributi, combinati con eventuali attributi di  $R_1$  nel caso di identificatori misti, diventano chiave per  $R_1$ . Definendo inoltre tali attributi come chiave esterna su  $R_2$ , l'associazione  $A$  viene automaticamente rappresentata nello schema relazionale così ottenuto.
- **$E_2$  ha almeno un identificatore esterno o misto.** In questo caso, la traduzione avviene in due passi: (i) traduzione degli identificatori esterni o misti di  $E_2$ , applicando il procedimento illustrato al punto precedente ricorsivamente all'entità  $E_2$ ; (ii) traduzione dell'identificatore esterno o misto di  $E_1$ , applicando la procedura illustrata al punto precedente.

**Esempio 6.6** Consideriamo il diagramma ER della Figura 6.6(a). L'entità **Video** ha un identificatore misto, composto dall'attributo **colloc** e dall'entità **Videoteca**, tramite l'associazione **Posseduto**. L'entità **Videoteca** ha invece un solo identificatore interno. Le entità **Video** e **Videoteca** sono tradotte come segue:

**Videoteca**(nome, città)  
**Video**(colloc, nome<sup>Videoteca</sup>, città<sup>Videoteca</sup>, tipo).

In modo simile, il diagramma ER della Figura 6.6(b) può essere tradotto come segue:

**Nazione**(nome, capitale, numAbitanti)  
**Nazionale**(nome<sup>Nazione</sup>). □

Al termine della traduzione sopra descritta, ciascuna relazione potrebbe essere caratterizzata da più di una chiave. In questa fase è quindi necessario selezionare una di queste chiavi come chiave primaria. I criteri di scelta sono molteplici e discendono da valutazioni di efficienza, in quanto la chiave primaria viene utilizzata

dai DBMS relazionali per il reperimento efficiente dei dati tramite la costruzione di strutture ausiliarie di accesso (vedi Capitolo 7). Questi criteri possono anche essere utilizzati per determinare quale chiave di una relazione  $R_2$  inserire come chiave esterna in una relazione  $R_1$ . Anche in questo caso, benché una qualunque chiave, primaria o alternativa, di  $R_2$  possa essere utilizzata per la definizione della chiave esterna in  $R_1$  (vedi Capitolo 2), per i motivi sopra citati, è preferibile definire una chiave esterna sulla base di una chiave primaria.

I criteri per la scelta della chiave primaria si possono sintetizzare come segue:

- Poiché gli attributi di una chiave primaria non possono mai contenere valori nulli, gli identificatori che contengono attributi opzionali non possono essere selezionati come chiave primaria.
- Identificatori composti da pochi attributi sono preferibili ad identificatori composti da molti attributi, al fine di ridurre le dimensioni delle strutture ausiliarie di accesso costruite automaticamente dal sistema. Poiché le operazioni di join vengono spesso eseguite su attributi che rappresentano chiavi primarie e chiavi esterne, ridurre il numero di attributi delle chiavi permette anche di eseguire le operazioni di join più efficientemente (vedi Capitolo 7).
- Gli identificatori che assumiamo vengano utilizzati da molte operazioni per accedere alle entità sono da preferire, in quanto l'accesso tramite chiave primaria viene ottimizzato dal sistema.

Nel caso in cui nessuno tra gli identificatori determinati durante la progettazione concettuale e la ristrutturazione soddisfi i criteri precedenti, è consigliabile aggiungere alla relazione un ulteriore attributo come chiave primaria ed assegnare a tale attributo valori speciali (codici) generati appositamente ai fini dell'identificazione.

**Esempio 6.7** Consideriamo il diagramma ER della Figura 6.7. L'entità **Cliente** è caratterizzata da due identificatori interni: un identificatore semplice, rappresentato dall'attributo **codCli**, ed un identificatore composto dagli attributi **nome**, **cognome** e **dataN**. In questo caso, sembra ragionevole scegliere come chiave primaria **codCli**, in quanto è formata da un numero inferiore di attributi. L'identificatore (**nome**, **cognome**, **dataN**) diventa invece una chiave alternativa per la relazione. L'entità **Cliente** è quindi tradotta in una relazione **Cliente** con schema:

**Cliente**(**codCli**, **nome**, **cognome**, **telefono**, **dataN**, **residenza**). □

### 6.3.2 Associazione binaria uno a uno

Un'associazione binaria uno a uno viene rappresentata all'interno di una delle due relazioni risultanti dalla traduzione delle entità coinvolte nell'associazione, inserendo come chiave esterna una chiave della relazione rappresentante l'altra

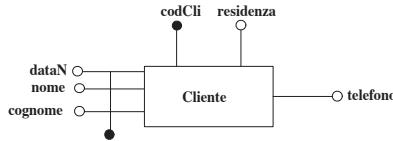


Figura 6.7: Entità con identificatori multipli

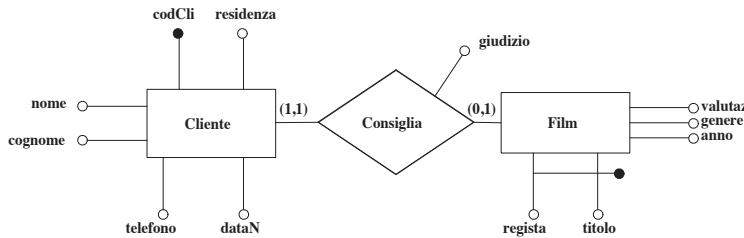


Figura 6.8: Associazione binaria uno a uno con partecipazione obbligatoria di una sola entità

entità partecipante all'associazione ed un attributo per ogni attributo dell'associazione.<sup>2</sup>

L'entità in cui inserire gli attributi viene scelta in relazione al fatto che la partecipazione all'associazione di una o di entrambe le entità sia opzionale ovvero obbligatoria. L'inserimento degli attributi nella relazione che rappresenta un'entità partecipante all'associazione in modo opzionale si traduce infatti nella presenza di valori nulli per le tuple che rappresentano istanze di entità non coinvolte nell'associazione. In alcuni casi, i valori nulli possono essere evitati scegliendo traduzioni specifiche, come discusso nel seguito.

**Partecipazione obbligatoria di una sola entità.** In questo caso, l'associazione si può rappresentare inserendo come chiave esterna, nella relazione che rappresenta l'entità la cui partecipazione all'associazione è obbligatoria, una chiave della relazione rappresentante l'entità la cui partecipazione all'associazione è opzionale. Gli attributi della chiave esterna in questo caso non potranno mai essere nulli. Inoltre, per ogni attributo dell'associazione, viene inserito nella stessa relazione un attributo con lo stesso nome.

**Esempio 6.8** Consideriamo il diagramma ER della Figura 6.8. L'associazione **Consiglia** associa ad ogni film il cliente che lo consiglia, insieme ad un giudizio su tale film. Supponiamo che ogni cliente possa consigliare un solo film ed un film possa essere consigliato da al più un cliente. L'associazione è pertanto obbligatoria per l'entità **Cliente** ed opzionale per l'entità **Film**. Le relazioni generate sono:

<sup>2</sup>Come abbiamo già ribadito, è preferibile definire una chiave esterna sulla base di una chiave primaria.

---

```

Film(titolo,regista,anno,genere,valutaz)
Cliente(codCli, nome, cognome, telefono, dataN, residenza, titoloFilm,
registaFilm, giudizio). □

```

**Partecipazione opzionale od obbligatoria di entrambe le entità.** A differenza dal caso precedente, poiché entrambe le entità partecipano allo stesso modo all'associazione, la relazione a cui aggiungere gli attributi può essere scelta indistintamente tra le due relazioni che modellano le entità partecipanti all'associazione.

Nel caso in cui la partecipazione di entrambe le entità sia opzionale, per evitare la presenza di valori nulli nelle istanze dello schema relazionale risultante, può a volte essere opportuno utilizzare una rappresentazione alternativa: l'associazione viene modellata tramite una relazione che contiene come attributi sia le chiavi delle relazioni rappresentanti le entità che partecipano all'associazione (che diventano chiavi esterne per la relazione), sia un attributo per ogni attributo dell'associazione. La chiave della relazione rappresentante l'associazione può essere scelta tra le chiavi delle relazioni che rappresentano le entità coinvolte nell'associazione. Questa rappresentazione ha il vantaggio di non generare mai valori nulli per le chiavi esterne, in quanto la nuova relazione conterrà solo le istanze dell'associazione. Lo svantaggio è invece quello di dover creare una relazione in più, con conseguente aumento della complessità dello schema relazionale risultante. Questa soluzione dovrebbe quindi essere adottata solo quando le istanze delle entità che partecipano all'associazione sono poche rispetto al numero totale delle istanze.

**Esempio 6.9** Consideriamo il diagramma ER della Figura 6.8 e supponiamo che il vincolo di cardinalità per l'entità **Cliente** nell'associazione **Consiglia** sia (0,1). Ciò corrisponde a supporre che un cliente possa non consigliare alcun film. Per modellare nello schema relazionale le entità e l'associazione coinvolte nell'esempio, sono possibili le seguenti alternative:

1. **Film(titolo,regista,anno,genere,valutaz)**  
**Cliente(codCli, nome, cognome, telefono, dataN, residenza, titolo<sup>Film</sup>,**  
**regista<sup>Film</sup>, giudizio.)**
2. **Film(titolo,regista,anno,genere,valutaz,codCli<sup>Cliente</sup>,giudizio.)**  
**Cliente(codCli, nome, cognome, telefono, dataN, residenza)**
3. **Film(titolo,regista,anno,genere,valutaz)**  
**Cliente(codCli, nome, cognome, telefono, dataN, residenza)**  
**Consiglia(titolo<sup>Film</sup>,regista<sup>Film</sup>,codCli<sup>Cliente</sup>,giudizio)** oppure  
**Consiglia(titolo<sup>Film</sup>,regista<sup>Film</sup>,codCli<sup>Cliente</sup>,giudizio).**

Come discusso in precedenza, la soluzione (3) è l'unica che non preveda attributi che possono assumere valori nulli. È però da adottare solo quando il numero dei film consigliati è significativamente inferiore al numero totale dei film ed al numero dei clienti. □

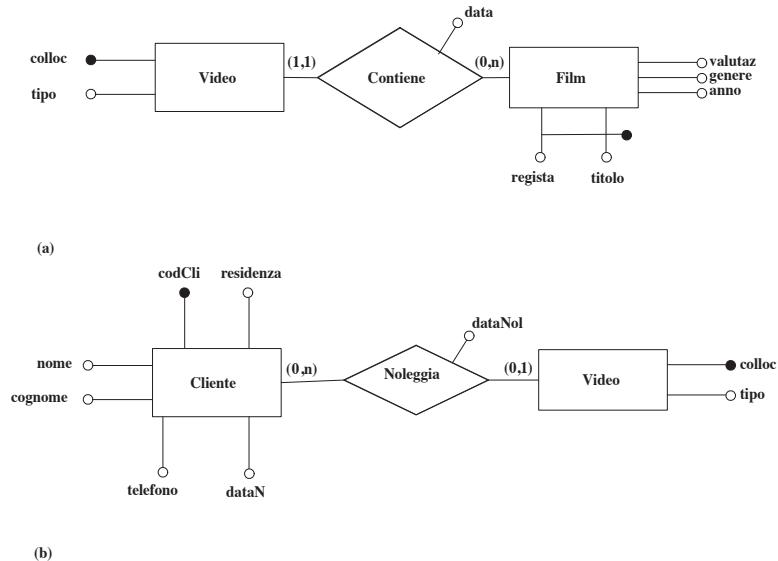


Figura 6.9: Associazioni binarie uno a molti: (a) con partecipazione obbligatoria dell'entità dal lato uno; (b) con partecipazione opzionale dell'entità dal lato uno

### 6.3.3 Associazione binaria uno a molti

Un'associazione binaria uno a molti viene rappresentata inserendo come chiave esterna, nella relazione che rappresenta l'entità con cardinalità massima uguale a uno rispetto all'associazione (entità dal lato uno), una chiave della relazione rappresentante l'altra entità che partecipa all'associazione ed un attributo per ogni attributo dell'associazione. Questa strategia di traduzione si applica indipendentemente dal fatto che l'entità dal lato uno della associazione vi partecipi obbligatoriamente od optionalmente.

Nel caso in cui l'entità dal lato uno abbia partecipazione opzionale, per ridurre la presenza dei valori nulli, è a volte conveniente utilizzare una rappresentazione alternativa, ovvero rappresentare l'associazione mediante una relazione distinta avente come attributi sia gli attributi dell'associazione sia le chiavi delle relazioni rappresentanti le entità che partecipano all'associazione (che diventano chiavi esterne per la relazione). La chiave della relazione rappresentante l'associazione è la chiave della relazione che rappresenta l'entità che compare dal lato uno dell'associazione.

**Esempio 6.10** Consideriamo il diagramma ER della Figura 6.9(a). L'associazione `Contiene` collega un video al film in esso contenuto ed è obbligatoria per l'entità `Video` mentre è opzionale per l'entità `Film` (supponendo che esistano film per i quali non è presente alcun video nella videoteca). Le relazioni generate per questo diagramma sono:

---

```

Film(titolo,regista,anno,genere,valutaz)
Video(colloc,titoloFilm,registaFilm,tipo,data).

```

Consideriamo ora il diagramma della Figura 6.9(b). L'associazione **Noleggia**, opzionale sia per l'entità **Cliente** sia per l'entità **Video**, lega un cliente ai video correntemente in noleggio, insieme alla data di inizio noleggio. Ovviamente, in ogni momento un video può essere noleggiato da al più un cliente. Sono possibili le seguenti rappresentazioni alternative:

1. Video(colloc,tipo,codCli<sup>Cliente</sup>,dataNol.)
   
 Cliente(codCli,nome,cognome,telefono,dataN,residenza)
2. Video(colloc,tipo)
   
 Cliente(codCli,nome,cognome,telefono,dataN,residenza)
   
 Noleggia(colloc,codCli<sup>Cliente</sup>,dataNol).

Come discusso in precedenza, la soluzione (2) è da adottare solo quando il numero dei video in noleggio è significativamente inferiore al numero totale dei video presenti nella videoteca, al fine di ridurre la presenza dei valori nulli. □

#### 6.3.4 Associazione binaria molti a molti

Nel caso in cui le entità siano legate da un'associazione binaria molti a molti, l'unico metodo possibile per la traduzione dell'associazione è quello di definire una relazione che rappresenta l'associazione stessa contenente un attributo per ogni attributo dell'associazione. Inoltre, tale relazione conterrà come chiavi esterne le chiavi di entrambe le relazioni che rappresentano le entità partecipanti all'associazione. La chiave della relazione che rappresenta l'associazione è costituita dalle chiavi delle relazioni corrispondenti alle entità partecipanti all'associazione.

**Esempio 6.11** Consideriamo il diagramma ER della Figura 6.10. L'associazione **Consiglia** associa ad ogni film i giudizi di un numero arbitrario di clienti. Ogni cliente può inoltre consigliare un numero arbitrario di film. Le relazioni corrispondenti sono:

```

Cliente(codCli,nome,cognome,telefono,dataN,residenza)
Film(titolo,regista,anno,genere,valutaz)
Consiglia(codCliCliente,titoloFilm,registaFilm,giudizio).

```

□

#### 6.3.5 Associazione unaria

La traduzione di un'associazione unaria è del tutto analoga a quella effettuata per le associazioni binarie a parte il fatto di dovere generare attributi distinti per i distinti ruoli giocati dall'entità nell'associazione. Nel caso in cui la relazione unaria sia uno a uno oppure uno a molti, l'associazione viene generalmente rappresentata inserendo gli attributi nella relazione che modella l'entità coinvolta nell'associazione. Nel caso in cui l'associazione unaria sia molti a molti è, invece, necessario

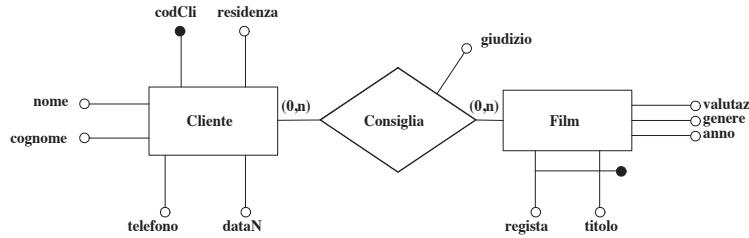


Figura 6.10: Associazione binaria molti a molti

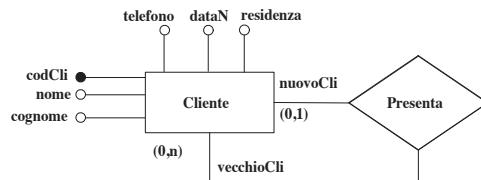


Figura 6.11: Associazione unaria

rappresentare l'associazione tramite una relazione distinta avente come attributi gli attributi dell'associazione e le chiavi corrispondenti ai ruoli che l'entità può giocare nell'associazione (che diventano chiavi esterne per la relazione). Tali attributi costituiscono la chiave della relazione. Analogamente a quanto visto per le associazioni binarie, rappresentare l'associazione tramite una relazione distinta può essere talvolta utile anche nel caso in cui l'associazione unaria sia uno a uno, oppure uno a molti, con partecipazione opzionale dell'entità dal lato uno, al fine di ridurre la presenza di valori nulli.

**Esempio 6.12** Consideriamo il diagramma ER della Figura 6.11. L'associazione **Presenta** associa ogni nuovo cliente ad al più un cliente esistente che lo ha presentato alla videoteca. Un cliente esistente può inoltre presentare un numero arbitrario di nuovi clienti. L'associazione è quindi uno a molti. La relazione corrispondente è:

```
Cliente(codCli, nome, cognome, telefono, dataN, residenza,  
vecchioCliCliente).
```

Nel caso in cui solo pochi clienti siano presentati da clienti esistenti, lo schema seguente riduce la presenza di valori nulli:

```
Cliente(codCli, nome, cognome, telefono, dataN, residenza)  
Presenta(nuovoCliCliente, vecchioCliCliente).
```

Supponiamo adesso che uno stesso cliente possa essere presentato da un numero arbitrario di clienti esistenti. In questo caso, l'associazione diventa molti a molti e può essere tradotta come segue:

---

```
Cliente(codCli,nome,cognome,telefono,dataN,residenza)
Presenta(nuovoCliCliente,vecchioCliCliente). □
```

### 6.3.6 Associazione n-aria

La traduzione di un'associazione *n*-aria è del tutto simile a quella vista per le associazioni binarie. Molto spesso, le associazioni *n*-arie sono associazioni molti a molti. L'associazione viene quindi modellata tramite una relazione che contiene un attributo per ogni attributo dell'associazione ed una chiave esterna per ogni relazione corrispondente ad un'entità che partecipa all'associazione.

In generale, la chiave della relazione che modella l'associazione *n*-aria è data dalle chiavi delle relazioni che corrispondono alle entità che partecipano all'associazione. Esistono però alcuni casi particolari in cui, applicando il metodo sopra esposto, otteniamo una super-chiave e non una chiave della relazione rappresentante l'associazione. In situazioni di questo tipo, la determinazione della chiave della relazione può avvenire analizzando un tipo particolare di vincoli di integrità, noti come *dipendenze funzionali*. Tali vincoli possono essere derivati a partire dalla documentazione generata nella fase di progettazione concettuale (vedi Paragrafo 6.5). Nel caso in cui, invece, almeno un'entità partecipi all'associazione *n*-aria con vincolo di cardinalità massima pari a uno, allora l'associazione si può modellare inserendo come chiavi esterne, nella relazione che rappresenta tale entità, le chiavi delle relazioni che rappresentano le entità coinvolte nell'associazione e gli attributi dell'associazione, in analogia con la traduzione delle associazioni binarie od unarie uno a molti.

**Esempio 6.13** Consideriamo il diagramma ER della Figura 6.12 che associa ogni cliente ai film da lui consigliati, per ogni attore. Le relazioni corrispondenti sono:

```
Cliente(codCli,nome,cognome,telefono,dataN,residenza)
Film(titolo,regista,anno,genere,valutaz)
Attore(codA,nome,cognome)
Consiglia(codCliCliente,titoloFilm,registaFilm,codAAttore,giudizio).
```

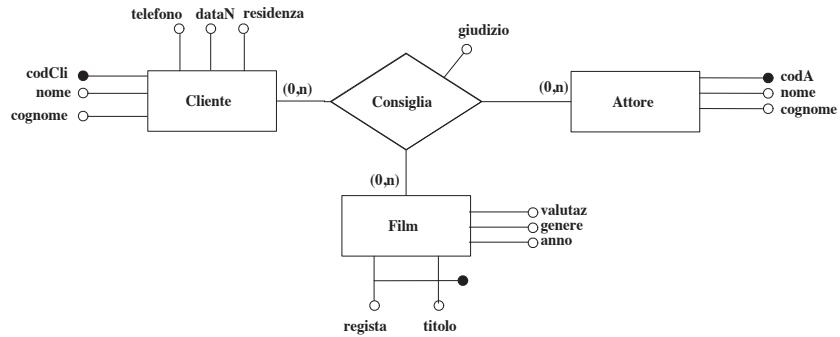
Supponiamo adesso che ogni cliente possa ancora consigliare un numero arbitrario di film ma al più uno per ogni attore.<sup>3</sup> L'associazione continua ad essere molti a molti. Tuttavia, in questo caso, la chiave della relazione *Consiglia* è costituita dagli attributi (*codCli*, *codA*), in quanto, dato un cliente ed un attore, esiste al più un film consigliato dal cliente in cui recita l'attore considerato.

Se infine il vincolo di cardinalità per *Cliente* fosse (0,1), quindi nel caso in cui ogni cliente possa fornire al più un consiglio, le relazioni generate sarebbero le seguenti:

```
Cliente(codCli,nome,cognome,telefono,dataN,residenza,titoloFilm,
```

---

<sup>3</sup>Come vedremo nel Paragrafo 6.5.2, questo vincolo può essere espresso tramite la dipendenza funzionale *codCli codA → titolo regista*, per la relazione *Consiglia*.

Figura 6.12: Associazione  $n$ -aria

$\text{regista}_{\text{Film}}^{\text{Film}}, \text{codA}_{\text{Attore}}^{\text{Attore}}, \text{giudizio}_0)$   
 $\text{Film}(\underline{\text{titolo}}, \underline{\text{regista}}, \text{anno}, \text{genere}, \text{valutaz})$   
 $\text{Attore}(\underline{\text{codA}}, \text{nome}, \text{cognome}).$

Per ridurre la presenza di valori nulli, lo schema precedente può essere modificato come segue:

$\text{Cliente}(\underline{\text{codCli}}, \text{nome}, \text{cognome}, \text{telefono}, \text{dataN}, \text{residenza})$   
 $\text{Film}(\underline{\text{titolo}}, \underline{\text{regista}}, \text{anno}, \text{genere}, \text{valutaz})$   
 $\text{Attore}(\underline{\text{codA}}, \text{nome}, \text{cognome})$   
 $\text{Consiglia}(\underline{\text{codCli}}^{\text{Cliente}}, \underline{\text{titolo}}^{\text{Film}}, \underline{\text{regista}}^{\text{Film}}, \text{codA}^{\text{Attore}}, \text{giudizio}). \square$

#### 6.4 Un esempio di progettazione logica

Per illustrare le varie fasi della progettazione logica, in questo paragrafo genereremo uno schema relazionale partendo dallo schema ER e dalla relativa documentazione per la base di dati relativa alla videoteca (vedi Figura 5.15 e Tabelle 5.2, 5.3, 5.4 e 5.5), unitamente al seguente carico di lavoro, rappresentato da un insieme di operazioni che vogliamo eseguire sulla base di dati e dalla loro frequenza:

- **Operazione 1.** Inserisce un nuovo video ed il corrispondente film, se non ancora presente (frequenza: 30 video/mese, 10 film/mese).
- **Operazione 2.** Inserisce un nuovo cliente, classificandolo come cliente standard, indicando tutti i suoi dati anagrafici (frequenza: 5 clienti/settimana).
- **Operazione 3.** Inserisce le informazioni relative ad un nuovo noleggio ed aggiorna i punti mancanti per accedere alla categoria VIP; quando i punti mancanti ad un cliente standard per accedere alla categoria VIP sono zero, il cliente cambia categoria e diventa un cliente VIP a cui viene assegnato un determinato bonus (frequenza: 200 noleggi/giorno).

- **Operazione 4.** Aggiorna i dati del noleggio al momento della restituzione del video da parte di un generico cliente (frequenza: 200 noleggi/giorno).
- **Operazione 5.** Stampa l'elenco dei nomi e delle residenze di tutti i clienti che hanno noleggiato almeno un video da più di 1 settimana e non l'hanno ancora restituito (frequenza: 1 stampa/giorno).
- **Operazione 6.** Stampa l'elenco dei nomi e delle residenze di tutti i clienti VIP, per inviare materiale informativo relativo alla situazione bonus (frequenza: 2 stampe/mese).

#### *6.4.1 Fase di ristrutturazione*

La ristrutturazione, il cui risultato è illustrato nella Figura 6.13 e nella Tabella 6.1, consiste nei passi di seguito illustrati. Notiamo che la fase di analisi della ridondanza non rileva alcun costrutto eliminabile e quindi non viene ulteriormente discussa nel seguito.

**Partizionamenti ed accorpamenti.** Poiché lo schema non contiene associazioni uno a uno, non è possibile effettuare accorpamenti. Dall'analisi del carico di lavoro risulta però che potrebbe essere opportuno partizionare i noleggi in noleggi conclusi e noleggi in corso. Tale partizionamento potrebbe migliorare le prestazioni delle operazioni 4 e 5, relative ai soli noleggi in corso. Benché tale partizionamento possa già essere eseguito a livello di ristrutturazione, spesso si preferisce effettuare questa scelta sullo schema logico, durante la fase di progettazione fisica (vedi Capitolo 7), quando saranno note ulteriori informazioni relative all'esecuzione delle interrogazioni. Supponiamo quindi di non eseguire il partizionamento, rimandando alla fase di progettazione fisica ogni decisione in merito.

**Eliminazione degli attributi composti e multi-valore.** Lo schema contiene un solo attributo composto, **residenza**, per l'entità **Cliente**. Poiché non ci sono operazioni che coinvolgono singoli sotto-attributi di **residenza**, conviene ristrutturare l'attributo sostituendolo con un attributo semplice che contiene l'intero indirizzo. L'entità **Cliente** ha inoltre un attributo multi-valore, **telefono**. È quindi necessario introdurre una nuova entità **Telefono** nello schema e collegarla all'entità **Cliente** con un'associazione molti a molti.

**Eliminazione delle generalizzazioni.** Lo schema contiene due generalizzazioni, entrambe totali ed esclusive (vedi Tabella 5.3). Consideriamo per prima la generalizzazione su **Cliente**. La ristrutturazione basata sull'eliminazione dell'entità padre non sembra ragionevole in quanto alcune delle operazioni identificate (operazione 5) non fanno distinzione tra il tipo di clienti e sono piuttosto frequenti. La ristrutturazione basata sull'eliminazione delle entità figlie risolverebbe questo problema ma, poiché possiamo ragionevolmente assumere che il numero dei clienti standard sia molto superiore al numero dei clienti VIP, richiederebbe di accedere

Vincoli di integrità
<b>V<sub>1</sub>:</b> <i>Ogni cliente non può noleggiare più di tre video contemporaneamente</i>
<b>V<sub>2</sub>:</b> <i>Un video non può essere noleggiato prima dell'uscita del film a cui è relativo</i>
<b>V<sub>3</sub>:</b> <i>La data di noleggio di un video non può essere successiva alla data di restituzione</i>
<b>V<sub>4</sub>:</b> <i>Uno stesso video non può essere noleggiato da due o più clienti diversi contemporaneamente</i>
<b>V<sub>5</sub>:</b> <i>Ogni cliente è alternativamente un cliente standard od un cliente VIP</i>

Tabella 6.1: Vincoli di integrità per lo schema ER ristrutturato

ad un numero di tuple molto elevato anche quanto l'operazione coinvolge solo clienti VIP (operazione 6). Ne consegue che la soluzione più opportuna è la sostituzione della generalizzazione con l'inserimento di nuove associazioni tra l'entità **Cliente** e le entità **Standard** e **VIP**. È inoltre necessario aggiungere opportuni vincoli di integrità che specifichino che la generalizzazione di partenza era totale ed esclusiva. Questo comporta l'aggiunta di un vincolo (vincolo **V<sub>5</sub>** nella Tabella 6.1) a quelli già identificati durante la progettazione concettuale. Consideriamo adesso la generalizzazione su **Video**. Poiché le sotto-entità non hanno attributi propri e le operazioni che coinvolgono i video sono solo operazioni di inserimento e non fanno mai distinzioni sul tipo (operazione 1), è conveniente eliminare le entità figlie. L'attributo **tipo** non potrà mai essere nullo in quanto la generalizzazione è totale. Il suo dominio è { 'd', 'v' }. Poiché le sotto-entità non hanno attributi propri, non è necessario aggiungere alcun nuovo vincolo di integrità.

#### 6.4.2 Fase di traduzione

La traduzione consiste nei seguenti passi.

**Traduzione delle entità.** Ogni entità viene tradotta in una relazione con lo stesso nome, avente un attributo per ogni attributo dell'entità. Ogni identificatore interno diventa una chiave per la relazione definita. Osserviamo che lo schema ristrutturato contiene inoltre quattro identificatori esterni o misti: uno, esterno, per ogni nuova associazione introdotta dall'eliminazione della gerarchia di generalizzazione su **Cliente** e due, misti, già presenti nello schema iniziale, per l'entità **Noleggio** (vedi Figura 6.13). Per ogni identificatore esterno o misto, è necessario inserire una chiave esterna nella relazione che rappresenta l'entità identificata in modo esterno o misto. Relativamente agli identificatori esterni delle entità **Standard** e **VIP**, è necessario scegliere uno tra i due identificatori interni dell'entità **Cliente** ed inserirlo come chiave esterna nelle relazioni corrispondenti. Seguendo le regole introdotte nel Paragrafo 6.3.1, sceglieremo l'identificatore **codCli**, in quanto contiene un numero inferiore di attributi. **codCli** diventerà

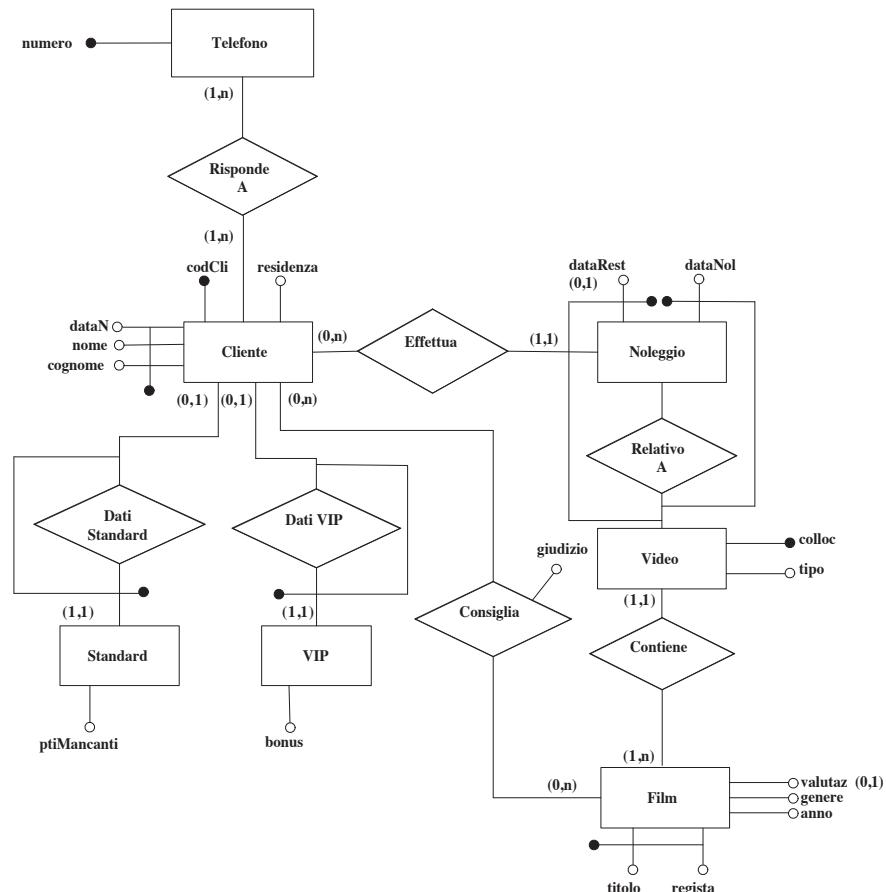


Figura 6.13: Schema ER ristrutturato, prima dell'eliminazione degli identificatori esterni e misti

anche chiave per le relazioni **Standard** e **VIP**. Per quanto riguarda gli identificatori misti di **Noleggio**, poiché l'entità **Video** ha un solo identificatore, **colloc**, tale attributo viene inserito come chiave esterna nella relazione corrispondente a **Noleggio**. (**colloc,dataNol**) e (**colloc,dataRest**) diventano inoltre chiavi per la relazione **Noleggio**. Tutti gli attributi delle relazioni ottenute, escluso **dataRest** in **Noleggio** e **valutaz** in **Film**, sono obbligatori.

Per tutte le relazioni create, deve inoltre essere scelta una chiave primaria. Le relazioni **Cliente** e **Noleggio** hanno più di una chiave e quindi è necessario effettuare una scelta. Per **Cliente**, seguendo le regole introdotte nel Paragrafo 6.3.1, sembra ragionevole scegliere l'attributo **codCli** in quanto l'altra chiave è composta da un numero maggiore di attributi. Per l'entità **Noleggio**, l'unica chiave che

può essere scelta come chiave primaria è (`colloc`,`dataNol`), in quanto l'attributo `dataRest` può essere nullo. Le chiavi non selezionate come chiavi primarie diventano chiavi alternative.

**Traduzione delle associazioni.** L'associazione `Effettua` viene tradotta inserendo l'identificatore di `Cliente` in `Noleggio`, poiché si tratta di un'associazione uno a molti con partecipazione obbligatoria dell'entità dal lato uno. Analogamente, l'associazione `Contiene` viene tradotta inserendo l'identificatore di `Film` in `Video`. Le associazioni `RispondeA` e `Consiglia` vengono tradotte con due nuove relazioni, in quanto entrambe sono di tipo molti a molti.

#### 6.4.3 Schema finale ed ottimizzazioni

Lo schema relazionale risultante è presentato nella Figura 6.14. Notiamo che, poiché solo l'attributo `dataRest` dell'entità `Cliente` e l'attributo `valutaz` dell'entità `Film` nello schema ER sono opzionali, tutti gli altri attributi sono considerati obbligatori (non potranno quindi assumere valori nulli).

Lo schema comprende anche otto vincoli di integrità, elencati nella Tabella 6.2 insieme alla modalità con cui possono essere rappresentati in SQL, sulla base della classificazione introdotta nel Capitolo 3.<sup>4</sup> I vincoli  $V_1, \dots, V_5$  sono quelli identificati nelle fasi di progettazione concettuale e ristrutturazione (vedi Tabella 6.1). I vincoli  $V_6, V_7$  e  $V_8$  rappresentano invece i vincoli relativi ai domini di tipo intervallo e di tipo enumerazione associati rispettivamente agli attributi `valutaz` dell'entità `Film`, `tipo` dell'entità `Video` e `giudizio` dell'associazione `Consiglia`, per i quali non esiste un dominio corrispondente nel modello relazionale.

Lo schema risultante può essere ulteriormente ottimizzato. Infatti, ogni numero di telefono che compare nella relazione `RispondeA` compare anche nella relazione `Telefono` in quanto la partecipazione dell'entità `Telefono` all'associazione `RispondeA` è obbligatoria. Poiché la relazione `Telefono` non contiene altri attributi può essere eliminata dallo schema senza perdita di informazioni. Inoltre, poiché la chiave primaria di `Film` è composta da due attributi, `titolo` e `regista`, e poiché due relazioni, precisamente `Video` e `Consiglia`, contengono una chiave esterna su `Film`, le informazioni sui titoli e sui registi sono replicate in tre relazioni distinte, generando una certa ridondanza. Per ovviare a questo problema, possiamo aggiungere un nuovo attributo `codF` alla relazione `Film`, contenente un codice identificativo per ogni film presente nella videoteca. La ridondanza può quindi essere eliminata definendo `codF` come chiave primaria e ridefinendo opportunamente le chiavi esterne. La coppia (`titolo`,`regista`) diventerà in questo caso chiave alternativa della relazione `Film`.

---

<sup>4</sup>La tabella non contiene i vincoli derivati dalla partecipazione obbligatoria delle entità alle associazioni.

```

Cliente(codCli, nome, cognome, dataN, residenza)
Standard(codCliCliente, ptiMancanti)
VIP(codCliCliente, bonus)
Telefono(numero)
Video(colloc, tipo, titoloFilm, registaFilm)
Film(titolo, regista, anno, genere, valutaz.)
Noleggio(collocVideo, dataNol, codCliCliente, dataRest.)
RispondeA(codCliCliente, numeroTelefono)
Consiglia(titoloFilm, registaFilm, codCliCliente, giudizio)

```

Figura 6.14: Schema relazionale generato dalla fase di traduzione

Vincoli di integrità	Traduzione in SQL
<b>V<sub>1</sub>:</b> Ogni cliente non può noleggiare più di tre video contemporaneamente	Asserzione
<b>V<sub>2</sub>:</b> Un video non può essere noleggiato prima dell'uscita del film a cui è relativo	Asserzione
<b>V<sub>3</sub>:</b> La data di noleggio di un video non può essere successiva alla data di restituzione	Vincolo su relazione
<b>V<sub>4</sub>:</b> Uno stesso video non può essere noleggiato da due o più clienti diversi contemporaneamente	Asserzione
<b>V<sub>5</sub>:</b> Ogni cliente è alternativamente un cliente standard od un cliente VIP	Asserzione
<b>V<sub>6</sub>:</b> Il tipo di un video è 'd', per i dvd, e 'v', per i vhs	Vincolo su colonna
<b>V<sub>7</sub>:</b> La valutazione di un film è un numero reale compreso tra 0 e 5	Vincolo su colonna
<b>V<sub>8</sub>:</b> Il giudizio espresso da un cliente su un film è un numero intero compreso tra 0 e 5	Vincolo su colonna

Tabella 6.2: Vincoli di integrità per lo schema relazionale

chiave esterna contiene un riferimento alla relazione stessa. Ad esempio, la relazione `Film` potrebbe contenere come chiave esterna su `Film` stessa una coppia di attributi (`titoloPre`, `registaPre`), contenente titolo e regista del film di cui il film è eventualmente il seguito. Come evidenziato dall’Esempio 2.6, inoltre, una relazione può contenere più chiavi esterne, eventualmente anche sulla stessa relazione. Rimarchiamo inoltre che le chiavi esterne, come del resto le chiavi, devono essere esplicitamente specificate in uno schema di relazione. Il fatto di avere attributi con lo stesso nome e domini compatibili in relazioni diverse non offre di per sé alcuna garanzia relativamente al mantenimento dell’integrità referenziale. Notiamo infine che, se non esplicitamente impedito mediante la specifica di un apposito vincolo, le chiavi esterne possono assumere valore nullo.

Per concludere la trattazione del modello dei dati relazionale, sottolineiamo che i vincoli di integrità che il modello permette di rappresentare direttamente, e che abbiamo discusso in questo paragrafo, non sono sufficienti a garantire che il contenuto della base di dati rispecchi in modo fedele le informazioni del dominio applicativo da rappresentare. Discuteremo più in dettaglio nel Capitolo 3 (vedi Paragrafo 3.4) altre categorie di vincoli, oltre ai vincoli di dominio, di obbligatorietà, di chiave e di chiave esterna qui discussi, cui il modello relazionale non offre diretto supporto.

## 2.2 Algebra relazionale

L’algebra relazionale è costituita da varie operazioni per la manipolazione delle relazioni. Più precisamente, l’algebra è composta da cinque operazioni di base: *proiezione*, *selezione*, *prodotto cartesiano*, *unione* e *differenza*. Queste operazioni definiscono completamente l’algebra relazionale. Ogni operazione ha come argomento una o due relazioni (a seconda dell’operazione) e restituisce come risultato una relazione; è pertanto possibile applicare un’operazione al risultato di un’altra operazione (proprietà detta di *chiusura*). Esistono operazioni addizionali, che possono essere espresse in termini delle cinque operazioni di base. Tali operazioni non estendono il potere espressivo dato dalle cinque operazioni di base, ma sono utili come abbreviazione. Tra le operazioni addizionali, l’operazione detta di *join* è la più rilevante. È da notare che la definizione delle varie operazioni è leggermente diversa a seconda che si consideri la definizione del modello relazionale per posizione o la definizione per nome (vedi Paragrafo 2.1); la differenza principale consiste nel modo in cui vengono denotate le componenti delle tuple in alcune delle operazioni che hanno una o più di tali componenti come ulteriore argomento. Nella trattazione seguente, considereremo la definizione delle operazioni in base alla notazione per nome. In riferimento a tale notazione, viene introdotta un’ulteriore operazione, di *ridenominazione*, che permette di modificare i nomi degli attributi.

### 2.2.1 Operazioni di base

In quanto segue descriviamo le operazioni di base dell'algebra evidenziando eventuali restrizioni sugli argomenti e lo schema della relazione risultato di ogni operazione. Poiché facciamo riferimento alla notazione con nome, introduciamo innanzitutto l'operazione di ridenominazione.

**Ridenominazione.** La *ridenominazione* di una relazione  $R$  rispetto ad una lista di coppie di nomi di attributi  $(A_1, B_1), (A_2, B_2), \dots, (A_m, B_m)$ , tale che  $A_i \in U_R$  è un nome di attributo di  $R$ , indicata con  $\rho_{A_1, A_2, \dots, A_m \leftarrow B_1, B_2, \dots, B_m}(R)$ , ridenomina l'attributo di nome  $A_i$  con il nome  $B_i$ ,  $i = 1, \dots, m$ . La ridenominazione è corretta se il nuovo schema di relazione per  $R$  ha attributi con nomi tutti distinti. La relazione ottenuta ha lo stesso grado della relazione  $R$  ed ha lo stesso contenuto. Gli attributi della relazione risultato sono  $U_R \setminus \{A_1, A_2, \dots, A_m\} \cup \{B_1, B_2, \dots, B_m\}$ .

**Esempio 2.8** Consideriamo la relazione `Noleggio` della Figura 2.2 e supponiamo di voler ridenominare i suoi attributi in `video`, `dataIn`, `cliente`, `dataFine`. L'operazione algebrica per eseguire tale ridenominazione è:

$$\rho_{\text{colloc}, \text{dataNol}, \text{codCli}, \text{dataRest} \leftarrow \text{video}, \text{dataIn}, \text{cliente}, \text{dataFine}}(\text{Noleggio})$$

il cui effetto è la modifica dello schema di `Noleggio` in `Noleggio(video, dataIn, cliente, dataFine)`, senza modificarne il contenuto.  $\square$

**Proiezione.** La *proiezione* di una relazione  $R$  su un insieme  $A = \{A_1, A_2, \dots, A_m\} \subseteq U_R$  di nomi di attributi di  $R$ , indicata con  $\Pi_{A_1, A_2, \dots, A_m}(R)$ , è una relazione di grado  $m$  le cui tuple hanno come attributi solo gli attributi specificati in  $A$ . Pertanto la proiezione genera un insieme  $T$  di tuple con  $m$  attributi. Sia  $t = [A_1 : v_1, A_2 : v_2, \dots, A_m : v_m]$  una tupla in  $T$ ;  $t$  è tale che esiste una tupla  $t'$  in  $R$  tale che  $t[A] = t'[A]$ . Nella relazione risultato gli attributi compaiono secondo l'ordine specificato in  $A$ ; è pertanto possibile specificare operazioni di proiezione che permangano gli attributi di una relazione. La proiezione permette di estrarre da una relazione solo alcune delle informazioni in essa contenute, eliminando gli attributi al cui valore non siamo interessati. La relazione risultato, oltre ad avere un grado inferiore a quello della relazione argomento, può anche avere cardinalità inferiore a quella della relazione argomento, poiché eliminando alcuni attributi possono venire generate delle tuple duplicate che compariranno una sola volta nella relazione risultato.

**Esempio 2.9** Consideriamo la relazione `Film` e supponiamo di essere interessati solo al titolo ed all'anno di uscita dei film presenti nella videoteca. L'espressione algebrica corrispondente è:

$$\Pi_{\text{titolo}, \text{anno}}(\text{Film})$$

il risultato della cui valutazione sulla relazione della Figura 2.1 è illustrato nella Figura 2.4(a). Supponiamo ora di essere interessati ai generi dei film presenti nella

titolo	anno	genere
underground	1995	drammatico
edward mani di forbice	1990	fantastico
nightmare before christmas	1993	animazione
ed wood	1994	fantascienza
mars attacks	1996	horror
il mistero di sleepy hollow	1999	commedia
big fish	2003	thriller
la sposa cadavere	2005	
la fabbrica di cioccolato	2005	
io non ho paura	2003	
nirvana	1997	
mediterraneo	1991	
pulp fiction	1994	
le iene	1992	

(a)  $\Pi_{\text{titolo}, \text{anno}}(\text{Film})$ (b)  $\Pi_{\text{genere}}(\text{Film})$ 

Figura 2.4: Proiezione

videoteca. L'espressione algebrica corrispondente è:

$$\Pi_{\text{genere}}(\text{Film})$$

il risultato della cui valutazione sulla relazione della Figura 2.1 è illustrato nella Figura 2.4(b).  $\square$

**Selezione.** La *selezione* su una relazione  $R$ , dato un predicato  $F$  su  $R$ , indicata con  $\sigma_F(R)$ , genera una relazione che contiene tutte le tuple di  $R$  che verificano  $F$ . Un predicato  $F$  su  $R$  è un predicato semplice su  $R$ , oppure una combinazione booleana, ottenuta mediante gli operatori booleani  $\wedge$ ,  $\vee$ ,  $\neg$ ,<sup>3</sup> di prediciati semplici su  $R$ . Un predicato semplice su  $R$  ha la forma  $A \text{ op } v$  oppure  $A \text{ op } A'$ , dove  $A$  ed  $A'$  sono nomi di attributi di  $R$  i cui domini devono essere compatibili;  $\text{op}$  è un *operatore relazionale di confronto* ed appartiene all'insieme  $\{<, =, >, \geq, \leq, \neq\}$ ,<sup>4</sup>  $v$  è un valore costante compatibile con il dominio di  $A$ .

**Esempio 2.10** I seguenti sono esempi di prediciati definiti sulla relazione `Noleggio` introdotta nella Figura 2.2: `codCli=6635`; `dataNol=dataRest`; `codCli=6635  $\vee$  dataNol=dataRest`. I primi due prediciati sono prediciati semplici.  $\square$

Se il grado della relazione operando  $R$  è  $k$ , la selezione  $\sigma_F(R)$  genera un insieme  $T$  di tuple di grado  $k$  (sottoinsieme di  $R$ ). Quindi, lo schema (ed il grado) della relazione risultato sono uguali a quelli della relazione operando. Sia  $t = [A_1 :$

<sup>3</sup>Tali operatori sono spesso denotati rispettivamente con AND, OR e NOT.

<sup>4</sup>Nei linguaggi come SQL sono presenti numerosi altri operatori di confronto (vedi Capitolo 3).

	colloc	dataNol	codCli	dataRest
(a) $\sigma_{\text{codCli}=6635}(\text{Noleggio})$	1111	01-Mar-2006	6635	02-Mar-2006
	1115	01-Mar-2006	6635	02-Mar-2006
	1117	02-Mar-2006	6635	06-Mar-2006
	1118	02-Mar-2006	6635	06-Mar-2006
	1119	08-Mar-2006	6635	10-Mar-2006
	1120	08-Mar-2006	6635	10-Mar-2006
	1121	15-Mar-2006	6635	18-Mar-2006
	1122	15-Mar-2006	6635	18-Mar-2006
	1113	15-Mar-2006	6635	18-Mar-2006
	1129	15-Mar-2006	6635	20-Mar-2006
	1127	22-Mar-2006	6635	?
	1125	22-Mar-2006	6635	?

(b) $\sigma_{\text{dataRest}=\text{dataNol}}(\text{Noleggio})$	colloc	dataNol	codCli	dataRest

Figura 2.5: Selezione

$v_1, A_2 : v_2, \dots, A_k : v_k]$  una tupla in  $T$ ;  $t$  è tale che  $F_{A_1/t[A_1], A_2/t[A_2], \dots, A_k/t[A_k]}$  è vera, dove la notazione  $A_i/t[A_i]$ ,  $i = 1, \dots, k$ , indica la sostituzione in  $F$  del nome di attributo  $A_i$  (se tale attributo compare in  $F$ ) con il valore dell'attributo di nome  $A_i$  della tupla  $t$ . Se nessuna tupla verifica  $F$ , il risultato è una relazione vuota. La selezione, quindi, permette di filtrare alcune tuple dalla relazione argomento mediante la specifica di opportune condizioni sui suoi attributi. Il risultato è una relazione la cui cardinalità è minore od uguale a quella della relazione argomento.

**Esempio 2.11** Consideriamo la relazione Noleggio di Figura 2.2 e supponiamo di essere interessati ai noleggi effettuati dal cliente con codice 6635. L'espressione algebrica corrispondente è:

$$\sigma_{\text{codCli}=6635}(\text{Noleggio})$$

il risultato della cui valutazione sulla relazione della Figura 2.2 è illustrato nella Figura 2.5(a). Supponiamo ora di essere interessati ai noleggi conclusi lo stesso giorno in cui sono iniziati. L'espressione algebrica corrispondente è:

$$\sigma_{\text{dataNol}=\text{dataRest}}(\text{Noleggio})$$

il risultato della cui valutazione sulla relazione della Figura 2.2 è illustrato nella Figura 2.5(b) e corrisponde ad una relazione vuota.  $\square$

**Prodotto cartesiano.** Il *prodotto cartesiano* di due relazioni  $R$  ed  $S$ , di grado rispettivamente  $k_1$  e  $k_2$ , indicato con  $R \times S$ , è una relazione di grado  $k_1 + k_2$  le cui tuple sono tutte le possibili tuple che hanno:

- come prime  $k_1$  componenti tuple di  $R$ ;

$\Pi_{\text{codCli}, \text{nome}}(\text{Cliente})$	$\Pi_{\text{titolo}, \text{anno}}(\sigma_{\text{regista}=\text{'quentin tarantino'}}(\text{Film}))$
$\begin{array}{cc} \text{codCli} & \text{nome} \\ \hline 6610 & \text{anna} \\ 6635 & \text{paola} \\ 6642 & \text{marco} \end{array}$	$\begin{array}{cc} \text{titolo} & \text{anno} \\ \hline \text{pulp fiction} & 1994 \\ \text{le iene} & 1992 \end{array}$
(a) $\Pi_{\text{codCli}, \text{nome}}(\text{Cliente})$	(b) $\Pi_{\text{titolo}, \text{anno}}(\sigma_{\text{regista}=\text{'quentin tarantino'}}(\text{Film}))$
	$\begin{array}{cccc} \text{codCli} & \text{nome} & \text{titolo} & \text{anno} \\ \hline 6610 & \text{anna} & \text{pulp fiction} & 1994 \\ 6635 & \text{paola} & \text{pulp fiction} & 1994 \\ 6642 & \text{marco} & \text{pulp fiction} & 1994 \\ 6610 & \text{anna} & \text{le iene} & 1992 \\ 6635 & \text{paola} & \text{le iene} & 1992 \\ 6642 & \text{marco} & \text{le iene} & 1992 \end{array}$
(c) $\Pi_{\text{codCli}, \text{nome}}(\text{Cliente}) \times \Pi_{\text{titolo}, \text{anno}}(\sigma_{\text{regista}=\text{'quentin tarantino'}}(\text{Film}))$	

Figura 2.6: Prodotto cartesiano

- come ultime  $k_2$  componenti tuple di  $S$ .

Il prodotto cartesiano può essere applicato solo se le due relazioni  $R$  ed  $S$  hanno schemi disgiunti.<sup>5</sup> Nella relazione risultato i primi  $k_1$  attributi sono gli attributi della relazione  $R$ , gli ultimi  $k_2$  attributi sono gli attributi della relazione  $S$ . Il prodotto cartesiano permette di costruire nuove tuple combinando le informazioni presenti nelle tuple delle relazioni argomento. Poiché ogni tupla di  $R$  viene combinata con ogni tupla di  $S$ , la cardinalità del risultato è il prodotto delle cardinalità degli argomenti.

**Esempio 2.12** Consideriamo la relazione  $\Pi_{\text{codCli}, \text{nome}}(\text{Cliente})$ , contenente i codici ed i nomi dei clienti della videoteca, e la relazione contenente il titolo e l'anno dei film di Quentin Tarantino, cioè  $\Pi_{\text{titolo}, \text{anno}}(\sigma_{\text{regista}=\text{'quentin tarantino'}}(\text{Film}))$ . Il prodotto cartesiano di queste due relazioni, espresso come:

$$\Pi_{\text{codCli}, \text{nome}}(\text{Cliente}) \times \Pi_{\text{titolo}, \text{anno}}(\sigma_{\text{regista}=\text{'quentin tarantino'}}(\text{Film}))$$

valutato sulle relazioni delle Figure 2.1 e 2.2 produce il risultato illustrato nella Figura 2.6 (c).  $\square$

**Unione.** L'*unione* delle relazioni  $R$  ed  $S$ , indicata con  $R \cup S$ , è l'insieme delle tuple che sono in  $R$  od in  $S$ . L'unione delle relazioni può essere eseguita solo se le relazioni hanno lo stesso schema (cioè stesso grado e, per ogni attributo,

<sup>5</sup>Se le due relazioni hanno attributi con lo stesso nome, è necessario ridenominare gli attributi in una delle due relazioni mediante l'operazione di ridenominazione  $\rho$ .

stesso nome<sup>6</sup> e domini compatibili). La relazione risultato ha lo stesso schema delle relazioni argomento. Essendo l'unione un'operazione tra insiemi (di tuple), le tuple duplicate vengono eliminate dal risultato.

**Esempio 2.13** Consideriamo  $R = \Pi_{\text{anno}, \text{genere}}(\sigma_{\text{regista}=\text{'tim burton'}}(\text{Film}))$  e  $S = \Pi_{\text{anno}, \text{genere}}(\sigma_{\text{regista}=\text{'gabriele salvatores'}}(\text{Film}))$ , relazioni contenenti anno e genere dei film di Tim Burton e Gabriele Salvatores, rispettivamente. L'unione  $R \cup S$  di queste due relazioni, valutata sulle relazioni delle Figure 2.1 e 2.2, produce il risultato illustrato nella Figura 2.7(a), che contiene l'anno di produzione ed il genere dei film di Tim Burton o Gabriele Salvatores. La Figura 2.7(d) mostra invece il risultato dell'espressione  $\Pi_{\text{genere}}(R) \cup \Pi_{\text{genere}}(S)$ , che restituisce i generi dei film girati da Tim Burton o Gabriele Salvatores.  $\square$

**Differenza.** La *differenza* delle relazioni  $R$  ed  $S$ , indicata con  $R - S$ , è l'insieme delle tuple che sono in  $R$  ma non in  $S$ . La differenza, come l'unione, di due relazioni può essere eseguita solo se le relazioni hanno lo stesso schema e produce una relazione con lo stesso schema. Se le relazioni hanno attributi con nomi diversi, si applica quanto già detto per l'unione.

**Esempio 2.14** Consideriamo nuovamente le relazioni  $R$  ed  $S$  dell'Esempio 2.13. La differenza  $R - S$  di queste due relazioni produce il risultato illustrato nella Figura 2.7(b). La Figura 2.7(e) mostra invece il risultato dell'espressione  $\Pi_{\text{genere}}(R) - \Pi_{\text{genere}}(S)$ , che restituisce i generi di cui Tim Burton ha girato un film ma Gabriele Salvatores non ne ha girato.  $\square$

### 2.2.2 Operazioni derivate

Tramite le operazioni di base è possibile definire altre operazioni, dette *operazioni derivate*. Alcune delle più rilevanti sono descritte in quanto segue.

**Intersezione.** L'*intersezione* di due relazioni  $R$  ed  $S$  è denotata con  $R \cap S$  ed è espressa come  $R - (R - S)$ . L'intersezione di  $R$  ed  $S$  restituisce le tuple che sono sia in  $R$  che in  $S$ . L'intersezione, come l'unione e la differenza di due relazioni, può essere eseguita solo se le relazioni hanno lo stesso schema e produce una relazione con lo stesso schema.

**Esempio 2.15** Consideriamo nuovamente le relazioni  $R$  ed  $S$  dell'Esempio 2.13. L'intersezione  $R \cap S$  di queste due relazioni produce il risultato illustrato nella Figura 2.7(c). La Figura 2.7(f) mostra invece il risultato dell'espressione  $\Pi_{\text{genere}}(R) \cap \Pi_{\text{genere}}(S)$ , che restituisce i generi di cui sia Tim Burton che Gabriele Salvatores hanno girato un film.  $\square$

---

<sup>6</sup>Se i nomi degli attributi nelle due relazioni sono diversi è sufficiente applicare l'operazione di ridenominazione  $\rho$ .

anno	genere
1990	fantastico
1993	animazione
1994	drammatico
1996	fantascienza
1999	horror
2003	fantastico
2005	animazione
2005	fantastico

*R*

anno	genere
2003	drammatico
1997	fantascienza
1991	commedia

*S*

anno	genere
1990	fantastico
1993	animazione
1994	drammatico
1996	fantascienza
1999	horror
2003	fantastico
2005	animazione
2005	fantastico
2003	drammatico
1997	fantascienza
1991	commedia

(a)  $R \cup S$ 

anno	genere
1990	fantastico
1993	animazione
1994	drammatico
1996	fantascienza
1999	horror
2003	fantastico
2005	animazione
2005	fantastico

(b)  $R - S$ 

anno	genere
1993	animazione
1994	drammatico
1996	fantascienza
1999	horror
2003	fantastico
2005	animazione
2005	fantastico

(c)  $R \cap S$ 

genere
fantastico
animazione
drammatico
fantascienza
horror

 $GR = \Pi_{\text{genere}}(R)$ 

genere
drammatico
fantascienza
commedia

 $GS = \Pi_{\text{genere}}(S)$ 

genere
fantastico
animazione
drammatico
fantascienza
horror
commedia

(d)  $GR \cup GS$ 

genere
fantastico
animazione
horror

(e)  $GR - GS$ 

genere
drammatico

(f)  $GR \cap GS$ 

Figura 2.7: Unione, differenza ed intersezione

---

codCli	nome	titolo	anno
6610	anna	pulp fiction	1994
6635	paola	pulp fiction	1994
6642	marco	pulp fiction	1994
6610	anna	le iene	1992

(a)  $\Pi_{\text{codCli}, \text{nome}}(\text{Cliente}) \bowtie_{\text{nome} < \text{titolo}} \Pi_{\text{titolo}, \text{anno}}(\sigma_{\text{regista} = 'quentin tarantino'}(\text{Film}))$

titolo	regista
la sposa cadavere	tim burton
la fabbrica di cioccolato	tim burton
big fish	tim burton
le iene	quentin tarantino

(b)  $\Pi_{\text{titolo}, \text{regista}}(\sigma_{\text{codCli} = 6635 \wedge \text{dataNol} = '15-Mar-2006'}(\text{Noleggio}) \bowtie_{\text{colloc} = c} \rho_{\text{colloc} \leftarrow c}(\text{Video}))$

codCli	nome	titolo	anno
6635	paola	le iene	1992
6635	paola	pulp fiction	1994
6642	marco	pulp fiction	1994

(c)  $\Pi_{\text{codCli}, \text{nome}, \text{titolo}, \text{anno}}(\text{Cliente} \bowtie \text{Noleggio} \bowtie \text{Video} \bowtie \sigma_{\text{regista} = 'quentin tarantino'}(\text{Film}))$

Figura 2.8: Theta-join, equi-join e join naturale

**Join.** Il *join* (detto anche *theta-join*) di due relazioni  $R$  ed  $S$  sugli attributi  $A$  di  $R$  ed  $A'$  di  $S$ , indicato con  $R \bowtie_{A\theta A'} S$ , dove  $\theta$  è un operatore relazionale di confronto, è definito dall'espressione algebrica  $\sigma_{A\theta A'}(R \times S)$ . Il join è pertanto un prodotto cartesiano seguito da una selezione; il predicato  $A\theta A'$  è detto *predicato di join*. Come per il prodotto cartesiano, gli schemi delle due relazioni argomento devono essere disgiunti e lo schema della relazione risultato è dato dalla loro unione.

**Esempio 2.16** Consideriamo le relazioni dell'Esempio 2.12. Se vogliamo collegare le informazioni di un cliente a quelle di un film solo se il nome del cliente precede in ordine alfabetico il titolo del film, dobbiamo formulare la seguente espressione:

$$\Pi_{\text{codCli}, \text{nome}}(\text{Cliente}) \bowtie_{\text{nome} < \text{titolo}} \Pi_{\text{titolo}, \text{anno}}(\sigma_{\text{regista} = 'quentin tarantino'}(\text{Film}))$$

che, valutata sulle relazioni delle Figure 2.1 e 2.2, produce il risultato illustrato nella Figura 2.8(a).  $\square$

Il join è un'operazione estremamente importante in quanto permette di collegare tuple di relazioni diverse e di attraversare le associazioni rappresentate nelle relazioni della base di dati mediante il meccanismo delle chiavi esterne. In particolare, il join prende il nome di *equi-join* quando l'operatore  $\theta$  usato nel predicato di join è l'operatore di uguaglianza ( $=$ ).

**Esempio 2.17** Consideriamo le relazioni `Video` e `Noleggio` della base di dati della videoteca. Se vogliamo determinare il titolo ed il regista dei film noleggiati il 15 Marzo 2006 dal cliente di codice 6635 dobbiamo formulare la seguente interrogazione:

$$\Pi_{\text{titolo}, \text{regista}}(\sigma_{\text{codCli}=6635 \wedge \text{dataNol}='15-\text{Mar}-2006'}(\text{Noleggio}) \bowtie_{\text{colloc}=\text{c}} \rho_{\text{colloc}=\text{c}}(\text{Video}))$$

che, valutata sulle relazioni delle Figure 2.1 e 2.2, produce il risultato illustrato nella Figura 2.8(b) Notiamo che è necessario applicare la ridenominazione poiché l'equi-join (così come il theta-join ed il prodotto cartesiano) richiede che gli schemi dei suoi argomenti siano disgiunti.  $\square$

**Join naturale.** L'operazione di *join naturale* rappresenta una “semplificazione” del join. È un'operazione che tuttavia, a differenza delle altre finora introdotte, ha senso solo nella notazione con nome del modello relazionale. Il join naturale di due relazioni  $R$  ed  $S$ , denotato come  $R \bowtie S$ , è definito come segue. Sia  $\{A_1, A_2, \dots, A_k\} = U_R \cap U_S$  l'insieme dei nomi di attributo presenti sia nello schema di  $R$  sia in quello di  $S$ . Sia inoltre  $\{I_1, I_2, \dots, I_m\} = U_R \cup U_S$  l'insieme dei nomi di attributo unione dell'insieme dei nomi degli attributi di  $R$  e dell'insieme dei nomi degli attributi di  $S$ . Siano infine  $\{B_1, B_2, \dots, B_k\}$  nomi di attributo non appartenenti né ad  $R$  né ad  $S$ . L'espressione che definisce il join naturale è:

$$R \bowtie S = \Pi_{I_1, I_2, \dots, I_m}(\sigma_C(R \times (\rho_{A_1, A_2, \dots, A_k \leftarrow B_1, B_2, \dots, B_k}(S))))$$

dove  $C$  è un predicato della forma  $A_1 = B_1 \wedge A_2 = B_2 \wedge \dots \wedge A_k = B_k$ . Pertanto il join naturale esegue un join eguagliando gli attributi con lo stesso nome delle due relazioni argomento dell'operazione e poi elimina gli attributi duplicati dalla relazione risultato. Si noti che, affinché l'operazione di join sia ben definita, gli attributi con nomi uguali nelle relazioni argomento dell'operazione devono avere domini compatibili. Notiamo che, nel caso particolare  $U_R = U_S$ , il join naturale degenera nell'intersezione, mentre, nel caso  $U_R \cap U_S = \emptyset$ , degenera nel prodotto cartesiano.

Il join naturale permette di navigare in modo molto più agevole, rispetto al join, attraverso le associazioni rappresentate tramite il meccanismo delle chiavi esterne, nel caso in cui per chiave e chiave esterna sia stato utilizzato lo stesso nome.

**Esempio 2.18** Consideriamo le relazioni dell'Esempio 2.12. Supponiamo di voler abbinare le informazioni del cliente a quelle del film solo se il cliente ha noleggiato un video di quel film, il che corrisponde a determinare, per ogni cliente che ha noleggiato un film di Quentin Tarantino, nome e codice del cliente e titolo ed anno del film. Dobbiamo formulare l'interrogazione seguente:

$$\Pi_{\text{codCli}, \text{nome}, \text{titolo}, \text{anno}}(\text{Cliente} \bowtie \text{Noleggio} \bowtie \text{Video} \bowtie \sigma_{\text{regista}='quentin tarantino'}(\text{Film}))$$

- Eliminazione di una colonna, specificata come:

```
DROP [COLUMN] <nome colonna> {RESTRICT | CASCADE}
```

dove **<nome colonna>** è il nome della colonna da eliminare e **RESTRICT** e **CASCADE** hanno il significato discusso per il comando **DROP TABLE**.

**Esempio 3.5** Consideriamo nuovamente lo schema della base di dati della videoteca definito nell’Esempio 3.3. Il comando:

```
ALTER TABLE Film ADD COLUMN studio VARCHAR(20);
```

ha l’effetto di aggiungere come sesta ed ultima colonna della relazione **Film** la colonna **studio**. Poiché non viene specificato un valore di default, a tutte le tuple di **Film** viene assegnato il valore **NULL** per tale colonna. Il comando:

```
ALTER TABLE Video ALTER COLUMN tipo SET DEFAULT 'v';
```

ha l’effetto di modificare il valore di default per la colonna **tipo** di **Video**, ponendolo uguale al valore ‘**v**’. □

### 3.2 Interrogazioni

Questo paragrafo descrive vari aspetti relativi alla specifica di interrogazioni in SQL. Per prima cosa introdurremo il formato di base di un’interrogazione SQL ed i principali operatori e funzioni utilizzabili nelle interrogazioni. Verranno poi discusse ulteriori caratteristiche del linguaggio di interrogazione, quali ordinamento, operazione di join, funzioni di gruppo, valori nulli e sotto-interrogazioni.

#### 3.2.1 Formato di base del comando SELECT

Le interrogazioni in SQL sono espresse tramite il comando **SELECT**. La forma base di un’interrogazione SQL ha la seguente struttura:<sup>8</sup>

```
SELECT {DISTINCT Ri1.C1, Ri2.C2, ..., Rin.Cn | *}  
FROM R1, R2, ..., Rk WHERE F;
```

dove:

- $R_i$  ( $i = 1, \dots, k$ ) è un nome di relazione. La clausola **FROM** specifica pertanto le relazioni oggetto dell’interrogazione.

---

<sup>8</sup>Come discuteremo nel seguito del paragrafo, la clausola **DISTINCT** del comando **SELECT** è in realtà opzionale.

- $R_{i_j}.C_j$  ( $j = 1, \dots, n$ ,  $i_j \in \{1, \dots, k\}$ ) indica che l'interrogazione deve restituire la colonna  $C_j$  della relazione  $R_{i_j}$ . La relazione  $R_{i_j}$  deve essere una delle relazioni specificate nella clausola **FROM** dell'interrogazione. Se la colonna  $C_j$  appare in una sola tra le relazioni specificate nella clausola **FROM**, allora possiamo semplicemente usare la notazione  $C_j$ , invece di  $R_{i_j}.C_j$ . La lista  $R_{i_1}.C_1, R_{i_2}.C_2, \dots, R_{i_n}.C_n$  costituisce la *clausola di proiezione* dell'interrogazione. Il simbolo '\*' indica che l'interrogazione deve restituire tutte le colonne delle relazioni che compaiono nella clausola **FROM**, non viene quindi effettuata alcuna proiezione.
- $F$  è la *clausola di qualificazione* dell'interrogazione in quanto qualifica, tramite predicati, le tuple di interesse. Tale clausola consiste in una combinazione booleana di predicati: in  $F$  possono essere usati i connettivi logici **AND**, **OR** e **NOT**. Le tuple che soddisfano  $F$  *verificano* l'interrogazione. Da tali tuple vengono estratte le colonne specificate nella clausola di proiezione dell'interrogazione.

Il significato di un'interrogazione SQL, specificata in accordo al formato precedente, è rappresentato tramite la seguente espressione dell'algebra relazionale:<sup>9</sup>

$$\Pi_{R_{i_1}.C_1, R_{i_2}.C_2, \dots, R_{i_n}.C_n}(\sigma_F(R_1 \times \dots \times R_k))$$

nel caso in cui la clausola di proiezione contenga una lista di colonne e tramite l'espressione:

$$\sigma_F(R_1 \times \dots \times R_k)$$

nel caso in cui la clausola di proiezione contenga '\*'. La valutazione dell'interrogazione inizia quindi dalla clausola **FROM**, viene poi applicata la clausola di qualificazione ed infine la clausola di proiezione.

L'ordine in cui le colonne sono elencate nella clausola di proiezione determina l'ordine da sinistra a destra secondo cui le colonne appaiono nella relazione risultato. Quando si usa '\*', l'ordine in cui le colonne appaiono nel risultato è dato dall'ordine in cui le relazioni compaiono nella clausola **FROM** e, per ogni relazione, dall'ordine specificato nella definizione della relazione (cioè nel comando **CREATE TABLE**).

La clausola di qualificazione può contenere i connettivi booleani **AND**, **OR** e **NOT** con l'usuale precedenza. Ordini di precedenza diversi tra questi connettivi possono essere specificati utilizzando le parentesi. I predicati semplici in SQL hanno in genere la forma  $e \ op \ e'$ , dove  $e$  ed  $e'$  sono espressioni che denotano valori mentre  $\op$  è un operatore relazionale di confronto. Molto spesso un'espressione che denota un valore è semplicemente un nome di colonna; in tal caso il valore denotato è il valore della colonna per la tupla considerata. È tuttavia possibile formulare predicati in cui i valori sono specificati tramite espressioni, uso di funzioni e sotto-interrogazioni, discussi nel prosieguo del paragrafo. È ovviamente possibile

---

<sup>9</sup>Notiamo che gli schemi di  $R_1, \dots, R_k$  sono resi disgiunti premettendo ad ogni nome di colonna il nome della relazione cui appartiene.

specificare predicati della forma  $C \ op \ C'$ , dove  $C$  e  $C'$  sono nomi di colonne; in questo caso il predicato esprime un confronto tra i valori di due colonne. Anche nella clausola di qualificazione al nome della colonna può (e deve, in caso di colonna appartenente a più relazioni nella clausola FROM) essere premesso il nome della relazione cui la colonna appartiene, con la notazione  $R.C$ . Oltre agli operatori di confronto già visti per l'algebra relazionale, SQL offre numerosi altri operatori di confronto, alcuni dei quali verranno discussi nel Paragrafo 3.2.2.

L'esempio successivo, così come gli altri presentati in questo capitolo, è basato sulle relazioni Film, Video, Cliente e Noleggio illustrate nelle Figure 2.1 e 2.2.

**Esempio 3.6** I seguenti sono esempi di interrogazioni espresse in SQL. Per ogni interrogazione, riportiamo anche la traduzione in algebra relazionale.

**Q1** Selezionare i film girati prima del 1999:

```
SELECT * FROM Film WHERE anno < 1999;
```

Traduzione in algebra relazionale:  $\sigma_{\text{anno} < 1999}(\text{Film})$ .

**Q2** Selezionare il titolo ed il regista dei film di fantascienza girati prima del 1999:

```
SELECT DISTINCT titolo, regista FROM Film
WHERE anno < 1999 AND genere = 'fantascienza';
```

Traduzione in algebra relazionale:

$\Pi_{\text{titolo}, \text{regista}}(\sigma_{\text{anno} < 1999 \wedge \text{genere} = \text{'fantascienza'}}(\text{Film}))$ .

**Q3** Selezionare il titolo dei film di Tim Burton di genere horror o fantascienza:

```
SELECT DISTINCT titolo FROM Film WHERE regista = 'tim burton'
AND (genere = 'horror' OR genere = 'fantascienza');
```

Traduzione in algebra relazionale:

$\Pi_{\text{titolo}}(\sigma_{\text{regista} = \text{'tim burton'} \wedge (\text{genere} = \text{'horror'} \vee \text{genere} = \text{'fantascienza'})}(\text{Film}))$ .

La Figura 3.1 riporta i risultati delle interrogazioni precedenti.  $\square$

La clausola DISTINCT del comando SELECT è in realtà opzionale. La sua inclusione ha l'effetto di richiedere l'eliminazione dei duplicati dal risultato. È importante notare che, se tale clausola è omessa, eventuali duplicati nel risultato di un'interrogazione non vengono automaticamente eliminati, perdendo quindi la corrispondenza con l'operazione di proiezione dell'algebra relazionale.

**Esempio 3.7** Consideriamo l'interrogazione per determinare i generi dei film. Con la formulazione:

```
SELECT DISTINCT genere FROM Film;
```

titolo	regista	anno	genere	valutaz			
underground	emir kusturica	1995	drammatico	3.20			
edward mani di forbice	tim burton	1990	fantastico	3.60			
nightmare before christmas	tim burton	1993	animazione	4.00			
ed wood	tim burton	1994	drammatico	4.00			
mars attacks	tim burton	1996	fantascienza	3.00			
nirvana	gabriele salvatores	1997	fantascienza	3.00			
mediterraneo	gabriele salvatores	1991	commedia	3.80			
pulp fiction	quentin tarantino	1994	thriller	3.50			
le iene	quentin tarantino	1992	thriller	4.00			
titolo	regista						
mars attacks	tim burton						
nirvana	gabriele salvatores						
titolo							
	mars attacks						
	il mistero di sleepy hollow						

Figura 3.1: Risultati delle interrogazioni dell’Esempio 3.6

ogni genere compare nel risultato un’unica volta, viceversa con la formulazione:

```
SELECT genere FROM Film;
```

il risultato può contenere duplicati. La Figura 3.2 riporta i risultati delle interrogazioni precedenti.  $\square$

Rimarchiamo infine che il risultato di un’interrogazione, applicato ad una o più relazioni, è sempre una relazione (proprietà di chiusura). Questa proprietà permette di usare interrogazioni all’interno di altre interrogazioni, come vedremo nel Paragrafo 3.2.7.

### 3.2.2 Operatori e funzioni

Come discusso in precedenza, SQL fornisce numerosi operatori di confronto e funzioni predefinite, che permettono di specificare un vasto insieme di espressioni e predicati. Nel seguito discuteremo brevemente i più comuni.

#### 3.2.2.1 Operatori di confronto

SQL fornisce operatori aggiuntivi, oltre agli usuali operatori relazionali di confronto già visti nel contesto dell’algebra relazionale. Alcuni di questi operatori sono in effetti ridondanti e sono stati introdotti per rendere più agevole l’uso del linguaggio. Altri operatori sono invece introdotti per fornire maggiori funzionalità nel trattare alcuni tipi di dato (come ad esempio le stringhe). Alcuni di questi operatori sono illustrati nel seguito. È da notare, tuttavia, che il linguaggio SQL è estremamente ricco e fornisce molti altri operatori, solo alcuni dei quali sono discussi nella trattazione successiva.

genere	genere
drammatico	drammatico
fantastico	fantastico
animazione	animazione
fantascienza	drammatico
horror	fantascienza
commedia	horror
thriller	fantastico
	animazione
	fantastico
	drammatico
	fantascienza
	commedia
	thriller
	thriller

Figura 3.2: Risultati delle interrogazioni dell’Esempio 3.7

**Condizioni su intervalli di valori.** L’operatore BETWEEN permette di ritrovare le tuple che contengono valori di una colonna in un intervallo specificato. Un predicato di confronto con l’operatore BETWEEN ha il formato  $e \text{ BETWEEN } v_1 \text{ AND } v_2$  dove:  $e$  è un’espressione che denota un valore, nel caso più semplice un nome di colonna;<sup>10</sup>  $v_1$  e  $v_2$  sono valori, compatibili con il tipo di  $e$ , detti rispettivamente estremo inferiore e superiore dell’intervallo. L’espressione  $e \text{ BETWEEN } v_1 \text{ AND } v_2$  è semplicemente un’abbreviazione per  $e \geq v_1 \text{ AND } e \leq v_2$ . L’operatore BETWEEN ha anche una forma negata; un predicato con la forma negata di tale operatore ha il formato  $e \text{ NOT BETWEEN } v_1 \text{ AND } v_2$ .

**Esempio 3.8** Per determinare tutte le informazioni relative ai film girati tra il 1995 e il 2000 formuliamo la seguente interrogazione:

```
SELECT * FROM Film WHERE anno BETWEEN 1995 AND 2000;
```

il cui risultato è illustrato nella Figura 3.3.  $\square$

**Ricerca di valori in un insieme.** L’operatore IN permette di determinare le tuple che contengono uno tra i valori di un insieme specificato. Un predicato di confronto con l’operatore IN ha il formato  $e \in (v_1, \dots, v_n)$  dove:  $e$  è un’espressione che denota un valore, mentre  $v_1, \dots, v_n$  sono valori il cui tipo è compatibile con il tipo di  $e$ . L’espressione  $e \in (v_1, \dots, v_n)$  è semplicemente un’abbreviazione per  $e = v_1 \text{ OR } \dots \text{ OR } e = v_n$ . Anche per l’operatore IN esiste la forma negata, che ha il formato  $e \notin (v_1, \dots, v_n)$ . L’operatore IN è particolarmente utile non tanto quando l’insieme di valori è esplicitamente enumerato, bensì quando è ottenuto tramite un’interrogazione alla base di dati. Esempi più significativi saranno pertanto discussi nella trattazione relativa alle sotto-interrogazioni.

<sup>10</sup>È da notare, per questo predicato come per i successivi discussi in questo paragrafo, che  $e$  potrebbe anche essere una sotto-interrogazione (vedi Paragrafo 3.2.7).

titolo	regista	anno	genere	valutaz
underground	emir kusturica	1995	drammatico	3.20
mars attacks	tim burton	1996	fantascienza	3.00
il mistero di sleepy hollow	tim burton	1999	horror	3.50
nirvana	gabriele salvatores	1997	fantascienza	3.00

titolo	regista	anno	genere	valutaz
mars attacks	tim burton	1996	fantascienza	3.00
il mistero di sleepy hollow	tim burton	1999	horror	3.50
nirvana	gabriele salvatores	1997	fantascienza	3.00

titolo	regista	anno	genere	valutaz
underground	emir kusturica	1995	drammatico	3.20
mediterraneo	gabriele salvatores	1991	commedia	3.80

Figura 3.3: Risultati delle interrogazioni degli Esempi 3.8, 3.9 e 3.10

**Esempio 3.9** Per determinare tutte le informazioni relative ai film il cui genere è horror o fantascienza formuliamo la seguente interrogazione:

```
SELECT * FROM Film WHERE genere IN ('horror', 'fantascienza');
```

il cui risultato è illustrato nella Figura 3.3. □

**Condizioni di confronto per stringhe di caratteri.** L'operatore LIKE permette di eseguire alcune semplici operazioni di *pattern matching* su colonne di tipo stringa. Un predicato di confronto espresso con l'operatore LIKE ha il formato: *e* LIKE *pattern*, dove *e* è un'espressione che denota un valore di tipo stringa e *pattern* è una stringa di caratteri che può contenere i caratteri speciali '%' e '\_'. Il simbolo '%' denota una sequenza di caratteri arbitrari di lunghezza qualsiasi (anche zero), mentre il simbolo '\_' indica esattamente un carattere.

**Esempio 3.10** Per determinare tutte le informazioni relative ai film che hanno il carattere 'd' come terza lettera del titolo, formuliamo la seguente interrogazione:

```
SELECT * FROM Film WHERE titolo LIKE '_d%';
```

il cui risultato è illustrato nella Figura 3.3. □

### 3.2.2.2 Espressioni e funzioni

Le espressioni sono di solito formulate applicando funzioni (aritmetiche, su stringhe, su date e tempi) ai valori delle colonne delle tuple. Oltre a poter essere usate nei predicati, le espressioni possono comparire nella clausola di proiezione delle interrogazioni, nelle espressioni di assegnamento usate nel comando UPDATE (vedi Paragrafo 3.3) e nella specifica di colonne derivate (vedi Paragrafo 3.5).

Un'espressione usata nella clausola di proiezione dà luogo ad una relazione con una colonna non presente nella relazione su cui è effettuata l'interrogazione. Tale colonna è detta *virtuale* in quanto è derivata dalle colonne, dette *di base*, della relazione oggetto dell'interrogazione. Una colonna virtuale non è fisicamente memorizzata nella relazione, ma è derivata dinamicamente da colonne della relazione. Ad una colonna virtuale, derivata tramite un'espressione, viene assegnato nell'eventuale presentazione del risultato un nome di colonna dato dall'espressione che definisce la colonna. È possibile, tuttavia, specificare nomi alternativi per tali colonne, facendo seguire l'espressione dalla clausola AS <nome colonna>. Poiché tuttavia la clausola di proiezione è l'ultima clausola dell'interrogazione ad essere valutata, tali nomi non possono essere utilizzati nelle altre clausole dell'interrogazione.

**Espressioni e funzioni aritmetiche.** Oltre agli usuali operatori aritmetici comunemente inclusi (+, -, \*, /) sono previste le seguenti funzioni:

- la funzione ABS( $n$ ) che ha come argomento un valore numerico  $n$  e ne calcola il valore assoluto;
- la funzione MOD( $n, b$ ) che ha come argomenti due valori interi  $n$  e  $b$ , e calcola il resto intero della divisione di  $n$  per  $b$ .

Sono inoltre previste funzioni logaritmiche, esponenziali, per il calcolo della radice quadrata, dell'elevamento a potenza, della parte intera superiore ed inferiore.

**Espressioni e funzioni per stringhe.** Il principale operatore tra stringhe è l'operatore di concatenazione di stringhe, denotato da '||'. Sono inoltre previste alcune funzioni per la manipolazione di stringhe, tra cui citiamo:

- la funzione LENGTH( $str$ ) che ha come argomento una stringa  $str$  e ne calcola la lunghezza, intesa come numero di caratteri;
- le funzioni UPPER( $str$ ) e LOWER( $str$ ) che trasformano la stringa  $str$  in caratteri tutti maiuscoli o tutti minuscoli, rispettivamente;
- la funzione SUBSTR( $str, m[, n]$ ) che ha come argomento una stringa  $str$  e due interi  $m$  ed  $n$ , ed estrae da  $str$  la sotto-stringa dal carattere di posizione  $m$  fino all'ultimo carattere o per una lunghezza  $n$  (se l'argomento  $n$  è specificato);
- la funzione TRIM [ $str_1$ ] FROM  $str_2$  che elimina dalla stringa  $str_2$  i caratteri in  $str_1$  (se  $str_1$  non è specificata elimina gli spazi).

**Espressioni e funzioni per date e tempi.** Data l'importanza delle informazioni temporali nelle più svariate applicazioni, SQL, oltre a fornire vari tipi di dato temporali, fornisce numerose funzioni per la manipolazione di date e tempi. Le funzioni più rilevanti sono:

- Le funzioni zerarie CURRENT\_DATE, CURRENT\_TIME, CURRENT\_TIMESTAMP che restituiscono, rispettivamente, la data, il tempo ed il timestamp attuale.
- EXTRACT ( $q$  FROM  $e$ ) estrae il campo corrispondente al qualificatore temporale  $q$  dall'espressione  $e$ . EXTRACT (DAY FROM DATE '08-Ott-1969'), ad esempio, restituisce 8.

Date e tempi possono essere usati in espressioni aritmetiche, per calcolare, tra gli altri, la differenza tra date, che è un intervallo temporale, come illustrato dagli esempi seguenti. In tali espressioni è inoltre possibile usare diverse *unità temporali*, tra cui: YEAR[S], MONTH[S], DAY[S], HOUR[S], MINUTE[S], SECOND[S].

Nell'esempio seguente la clausola di proiezione contiene un'espressione.

**Esempio 3.11** Per ritrovare la collocazione del video noleggiato e la durata (in giorni) di ogni noleggio del cliente di codice 6635 formuliamo la seguente interrogazione:

```
SELECT colloc, (dataRest - dataNol) DAY
FROM Noleggio WHERE codCli = 6635;
```

Notiamo che l'espressione (dataRest - dataNol) restituisce un valore di tipo intervallo e SQL richiede di specificare il qualificatore rispetto a cui esprimerlo, anche se in molte implementazioni questo non è in realtà necessario e l'interrogazione viene accettata anche senza l'indicazione di DAY. Il risultato dell'interrogazione è illustrato nella Figura 3.4. Notiamo che per i noleggi in corso la durata assume valore nullo. Come discuteremo in dettaglio nel Paragrafo 3.2.6, infatti, un'espressione in cui un argomento è nullo assume valore nullo.  $\square$

Le espressioni possono essere utilizzate sia nella clausola di proiezione sia nella clausola di qualificazione, come illustrato dal seguente esempio.

**Esempio 3.12** Vogliamo ora ritrovare la collocazione del video noleggiato e la durata (in giorni) di ogni noleggio di durata superiore a due giorni del cliente di codice 6635:

```
SELECT colloc, (dataRest - dataNol) DAY AS durata
FROM Noleggio
WHERE codCli = 6635 AND
      (dataRest - dataNol) DAY > INTERVAL '2' DAY;
```

colloc	(dataRest - dataNol) DAY	colloc	durata
1111	1		
1115	1		
1117	4	1117	4
1118	4	1118	4
1119	2	1121	3
1120	2	1122	3
1121	3	1113	3
1122	3	1129	5
1113	3		
1129	5		
1127	?		
1125	?		

Figura 3.4: Risultati delle interrogazioni degli Esempi 3.11 e 3.12

Nell’interrogazione viene specificato il nome **durata** per la colonna virtuale. L’espressione **(dataRest - dataNol) DAY** restituisce un intervallo, la cui durata viene confrontata con quella dell’intervallo **INTERVAL ‘2’ DAY**, cioè due giorni. Il risultato dell’interrogazione è illustrato nella Figura 3.4. Notiamo che tale risultato non contiene le tuple relative ai noleggi in corso. Come discuteremo nel Paragrafo 3.2.6, infatti, un’espressione che assume valore nullo non soddisfa predici di confronto con valori e non viene quindi restituita dall’interrogazione. □

Molto utile è infine la funzione di **CAST**, la cui sintassi è **CAST *e* AS *T***, che permette di convertire il tipo del valore restituito dall’espressione *e* al tipo specificato *T*. L’applicazione della funzione è soggetta ad un certo numero di restrizioni, che garantiscono che la conversione sia effettivamente possibile.

### 3.2.3 *Ordinamento del risultato di un’interrogazione*

Negli esempi visti finora, l’ordine delle tuple risultato di una interrogazione è determinato dal sistema. Di solito quest’ordine dipende, oltre che dalla memorizzazione fisica delle tuple, dalla strategia scelta dal DBMS per eseguire l’interrogazione (vedi Capitolo 7). È ovviamente importante per l’applicazione poter richiedere che le tuple risultato di un’interrogazione siano ordinate in base al valore di una o più colonne. Il comando **SELECT** prevede a tale scopo la clausola addizionale **ORDER BY**, illustrata dall’esempio seguente.

**Esempio 3.13** Vogliamo elencare la collocazione del video, la data di noleggio e la data di restituzione di tutti i noleggi del cliente 6635, ordinando le tuple in ordine crescente in base alla data di inizio del noleggio:

```
SELECT colloc, dataNol, dataRest FROM Noleggio
WHERE codCli = 6635
ORDER BY dataNol;
```

colloc	dataNol	dataRest	colloc	dataNol	dataRest
1111	01-Mar-2006	02-Mar-2006	1125	22-Mar-2006	?
1115	01-Mar-2006	02-Mar-2006	1127	22-Mar-2006	?
1117	02-Mar-2006	06-Mar-2006	1113	15-Mar-2006	18-Mar-2006
1118	02-Mar-2006	06-Mar-2006	1121	15-Mar-2006	18-Mar-2006
1119	08-Mar-2006	10-Mar-2006	1122	15-Mar-2006	18-Mar-2006
1120	08-Mar-2006	10-Mar-2006	1129	15-Mar-2006	20-Mar-2006
1121	15-Mar-2006	18-Mar-2006	1119	08-Mar-2006	10-Mar-2006
1122	15-Mar-2006	18-Mar-2006	1120	08-Mar-2006	10-Mar-2006
1113	15-Mar-2006	18-Mar-2006	1117	02-Mar-2006	06-Mar-2006
1129	15-Mar-2006	20-Mar-2006	1118	02-Mar-2006	06-Mar-2006
1127	22-Mar-2006	?	1111	01-Mar-2006	02-Mar-2006
1125	22-Mar-2006	?	1115	01-Mar-2006	02-Mar-2006

Figura 3.5: Risultati delle interrogazioni degli Esempi 3.13 e 3.14

Il risultato dell'interrogazione è illustrato nella Figura 3.5. □

L'ordinamento non è limitato ad una sola colonna, né ad un ordine crescente. Se più colonne sono specificate, le tuple sono ordinate prima in base ai valori della prima colonna specificata nella clausola `ORDER BY`, poi in base alla seconda colonna e così via per tutte le colonne. L'opzione `DESC` associata ad una colonna specifica l'ordinamento in base a valori decrescenti. L'altra opzione è l'opzione `ASC`, che è quella di default.

**Esempio 3.14** Vogliamo elencare la collocazione del video, la data di noleggio e la data di restituzione di tutti i noleggi del cliente 6635, ordinando le tuple in base alla data di inizio del noleggio in ordine decrescente ed alla collocazione in ordine crescente. Formuliamo l'interrogazione:

```
SELECT colloc, dataNol, dataRest FROM Noleggio
WHERE codCli = 6635
ORDER BY dataNol DESC, colloc;
```

il cui risultato è illustrato nella Figura 3.5. □

### 3.2.4 Operazione di join

Come abbiamo visto nel Capitolo 2, l'operazione di join rappresenta un'importante operazione in quanto permette di correlare dati memorizzati in relazioni diverse, quindi di “attraversare” le associazioni esistenti tra i dati. Tradizionalmente il join è espresso in SQL tramite un prodotto cartesiano a cui sono applicati uno o più *predicati di join*. I predicati di join in SQL sono del tutto simili ai predicati di selezione visti nella clausola di qualificazione ed esprimono la relazione che deve essere verificata dalle tuple risultato dell'interrogazione. SQL prevede però anche

diverse operazioni esplicite di join, che consentono formulazioni alternative delle interrogazioni. Tali operatori, poiché producono relazioni, vengono utilizzati nella clausola **FROM** di un'interrogazione, ed includono:

- **CROSS JOIN**, che è la forma di operatore di join più semplice e corrisponde al prodotto cartesiano. Ha sintassi **<nome relazione> CROSS JOIN <nome relazione>** ed equivale all'uso di **<nome relazione>, <nome relazione>** nella clausola **FROM**.
- **JOIN ON**, che corrisponde al theta-join ed ha sintassi **<nome relazione> JOIN <nome relazione> ON <predicato>**, con **<predicato>** predicato di join arbitrario.
- **JOIN USING**, in cui viene richiesta l'uguaglianza dei valori delle colonne specificate nella clausola **USING**, la cui sintassi è **<nome relazione> JOIN <nome relazione> USING (<lista nomi colonne>)**.
- **NATURAL JOIN**, che corrisponde al join naturale, in cui viene richiesta l'uguaglianza dei valori delle colonne che hanno lo stesso nome nelle due relazioni. La sintassi è **<nome relazione> NATURAL JOIN <nome relazione>**. L'operazione di **NATURAL JOIN** non corrisponde però completamente al join naturale algebrico (vedi Capitolo 2): non viene eseguita alcuna proiezione e lo schema risultante è quello del prodotto cartesiano.

**Esempio 3.15** Le seguenti sono tutte formulazioni alternative dell'interrogazione per ritrovare i titoli dei video noleggiati il 15 Marzo 2006 dal cliente con codice 6635:

```

SELECT titolo FROM Video, Noleggio
WHERE codCli = 6635 AND dataNol = DATE '15-Mar-2006',
      AND Video.colloc = Noleggio.colloc;

SELECT titolo FROM Video CROSS JOIN Noleggio
WHERE codCli = 6635 AND dataNol = DATE '15-Mar-2006',
      AND Video.colloc = Noleggio.colloc;

SELECT titolo
FROM Video JOIN Noleggio
          ON Video.colloc = Noleggio.colloc
WHERE codCli = 6635 AND dataNol = DATE '15-Mar-2006';

SELECT titolo FROM Video JOIN Noleggio USING (colloc)
WHERE codCli = 6635 AND dataNol = DATE '15-Mar-2006';

SELECT titolo FROM Video NATURAL JOIN Noleggio
WHERE codCli = 6635 AND dataNol = DATE '15-Mar-2006';

```

Il predicato di join è `Video.colloc = Noleggio.colloc`. Il risultato dell'interrogazione è illustrato nella Figura 3.6.  $\square$

Dato che il risultato di un'operazione di join è una relazione, è possibile richiedere che tale relazione sia ordinata, anche in base a valori di colonne di relazioni diverse, o che eventuali duplicati siano eliminati.

**Outer join.** In  $R \text{ JOIN } S$  non si ha traccia delle tuple di  $R$  che non corrispondono ad alcuna tupla di  $S$ . Questo non è sempre quello che si desidera. Per questo motivo SQL prevede un operatore di `OUTER JOIN` che aggiunge al risultato le tuple di  $R$  e/o  $S$  che non hanno partecipato al join, completandole con `NULL`. L'operatore di join originario, per contrasto, viene anche detto `INNER JOIN`. La variante `OUTER` può essere utilizzata sia per il join naturale sia per il theta-join.

Esistono diverse varianti dell'outer join, che vengono specificate premettendo il corrispondente qualificatore all'operatore `OUTER JOIN`. Consideriamo  $R \text{ OUTER JOIN } S$ :

- **FULL.** Sia le tuple di  $R$  sia quelle di  $S$  che non partecipano al join vengono completate ed inserite nel risultato.
- **LEFT.** Le tuple di  $R$  che non partecipano al join vengono completate ed inserite nel risultato.
- **RIGHT.** Le tuple di  $S$  che non partecipano al join vengono completate ed inserite nel risultato.

**Esempio 3.16** Supponiamo di voler visualizzare per ogni video contenente un film di Tim Burton di genere fantastico la sua collocazione, il titolo ed i codici dei clienti che l'hanno eventualmente noleggiato:

```
SELECT colloc, titolo, codCli
FROM Film NATURAL JOIN Video NATURAL LEFT OUTER JOIN Noleggio
WHERE regista = 'tim burton' AND genere = 'fantastico';
```

Senza l'utilizzo dell'outer join i video, quali il 1123, che non sono mai stati noleggiati non avrebbero fatto parte del risultato. Il risultato dell'interrogazione è illustrato nella Figura 3.6.  $\square$

### 3.2.5 Funzioni di gruppo e raggruppamento

Un'importante funzionalità che SQL aggiunge all'algebra relazionale è quella di poter estrarre dalle tuple di una relazione informazioni riassuntive, ottenute aggregando opportunamente i valori presenti in diverse tuple. I due principali costrutti che forniscono questa funzionalità sono le funzioni di gruppo e l'operatore di raggruppamento.

Il predicato di join è `Video.colloc = Noleggio.colloc`. Il risultato dell'interrogazione è illustrato nella Figura 3.6.  $\square$

Dato che il risultato di un'operazione di join è una relazione, è possibile richiedere che tale relazione sia ordinata, anche in base a valori di colonne di relazioni diverse, o che eventuali duplicati siano eliminati.

**Outer join.** In  $R \text{ JOIN } S$  non si ha traccia delle tuple di  $R$  che non corrispondono ad alcuna tupla di  $S$ . Questo non è sempre quello che si desidera. Per questo motivo SQL prevede un operatore di `OUTER JOIN` che aggiunge al risultato le tuple di  $R$  e/o  $S$  che non hanno partecipato al join, completandole con `NULL`. L'operatore di join originario, per contrasto, viene anche detto `INNER JOIN`. La variante `OUTER` può essere utilizzata sia per il join naturale sia per il theta-join.

Esistono diverse varianti dell'outer join, che vengono specificate premettendo il corrispondente qualificatore all'operatore `OUTER JOIN`. Consideriamo  $R \text{ OUTER JOIN } S$ :

- **FULL.** Sia le tuple di  $R$  sia quelle di  $S$  che non partecipano al join vengono completate ed inserite nel risultato.
- **LEFT.** Le tuple di  $R$  che non partecipano al join vengono completate ed inserite nel risultato.
- **RIGHT.** Le tuple di  $S$  che non partecipano al join vengono completate ed inserite nel risultato.

**Esempio 3.16** Supponiamo di voler visualizzare per ogni video contenente un film di Tim Burton di genere fantastico la sua collocazione, il titolo ed i codici dei clienti che l'hanno eventualmente noleggiato:

```
SELECT colloc, titolo, codCli
FROM Film NATURAL JOIN Video NATURAL LEFT OUTER JOIN Noleggio
WHERE regista = 'tim burton' AND genere = 'fantastico';
```

Senza l'utilizzo dell'outer join i video, quali il 1123, che non sono mai stati noleggiati non avrebbero fatto parte del risultato. Il risultato dell'interrogazione è illustrato nella Figura 3.6.  $\square$

### 3.2.5 Funzioni di gruppo e raggruppamento

Un'importante funzionalità che SQL aggiunge all'algebra relazionale è quella di poter estrarre dalle tuple di una relazione informazioni riassuntive, ottenute aggregando opportunamente i valori presenti in diverse tuple. I due principali costrutti che forniscono questa funzionalità sono le funzioni di gruppo e l'operatore di raggruppamento.

	colloc	titolo	codCli
<b>titolo</b>	1113	big fish	6635
<b>big fish</b>	1113	big fish	6642
la sposa cadavere	1114	big fish	6610
la fabbrica di cioccolato	1115	edward mani di forbice	6635
le iene	1115	edward mani di forbice	6610
	1122	la fabbrica di cioccolato	6635
	1122	la fabbrica di cioccolato	6642
	1123	la fabbrica di cioccolato	?

Figura 3.6: Risultati delle interrogazioni degli Esempi 3.15 e 3.16

### 3.2.5.1 Funzioni di gruppo

Le funzioni di gruppo, spesso chiamate anche *funzioni aggregate*, possono essere utilizzate nella clausola di proiezione di un'interrogazione e permettono di estrarre informazioni riassuntive da insiemi di valori. Una funzione di gruppo si applica all'insieme dei valori estratti dalle tuple che soddisfano la clausola di qualificazione dell'interrogazione. Le principali funzioni di gruppo previste da SQL sono:

- MAX determina il massimo in un insieme di valori;
- MIN determina il minimo in un insieme di valori;
- SUM esegue la somma dei valori in un insieme;
- AVG esegue la media dei valori in un insieme;
- COUNT determina la cardinalità di un insieme.

Oltre alle funzioni precedenti, in SQL:2003 sono previste funzioni per il calcolo della deviazione standard (funzione STDEV) e della varianza (funzione VAR). Il resto della discussione non tratterà queste funzioni addizionali.

Le funzioni SUM e AVG sono definite solo per insiemi di valori numerici, mentre le funzioni MAX, MIN e COUNT possono essere applicate anche ad altri insiemi di valori. Tutte le funzioni possono essere usate con il qualificatore DISTINCT; in tal caso, eventuali valori duplicati sono eliminati prima di applicare la funzione (notare che l'eliminazione dei duplicati è significativa solo per le funzioni SUM, AVG e COUNT). Ad eccezione della funzione COUNT, tutte le funzioni devono operare solo su insiemi di valori semplici (ad esempio insiemi di numeri o di stringhe) e non su insiemi di tuple. Nel caso più semplice tale insieme è denotato da un nome di colonna; possono essere tuttavia usate anche espressioni aritmetiche. La funzione COUNT può avere tre tipi di argomenti:

1. un nome di colonna: in tal caso la funzione restituisce il numero di valori non nulli presenti nella colonna;

COUNT(*)	COUNT(DISTINCT regista)	MIN(valutaz)	AVG(valutaz)	MAX(valutaz)
14	4	3.00	3.55	4.00
3	3	3.20	3.57	4.00

Figura 3.7: Risultati delle interrogazioni dell’Esempio 3.17

2. un nome di colonna preceduto dal qualificatore DISTINCT: in tal caso la funzione restituisce il numero di valori distinti e non nulli presenti nella colonna;
3. il carattere speciale ‘\*’: in tal caso la funzione restituisce il numero di tuple.

Come nel caso delle espressioni aritmetiche, le colonne della relazione risultato ottenute dall’applicazione di funzioni di gruppo sono colonne virtuali; ad una colonna virtuale, per default, viene assegnato il nome della funzione usata nel calcolo della colonna. Le funzioni di gruppo possono a loro volta essere usate in espressioni aritmetiche.

**Esempio 3.17** Consideriamo le seguenti interrogazioni.

**Q1** Vogliamo determinare il numero di film presenti in catalogo, il numero complessivo di registi che li hanno girati e la valutazione minima, media e massima di tali film:

```
SELECT COUNT(*), COUNT(DISTINCT regista),
       MIN(valutaz), AVG(valutaz), MAX(valutaz)
FROM Film;
```

**Q2** Vogliamo ora determinare il numero di film di genere drammatico presenti in catalogo, il numero complessivo di registi che li hanno girati e la valutazione minima, media e massima di tali film:

```
SELECT COUNT(*), COUNT(DISTINCT regista),
       MIN(valutaz), AVG(valutaz), MAX(valutaz)
FROM Film
WHERE genere = 'drammatico';
```

Il risultato delle interrogazioni, che contiene in entrambi i casi una sola tupla, è illustrato nella Figura 3.7.  $\square$

Un ultimo aspetto da puntualizzare riguarda quale valore viene restituito da una funzione di gruppo calcolata su un insieme vuoto. Questa situazione si verifica se non esiste alcuna tupla che soddisfa la condizione di ricerca dell’interrogazione. Il valore calcolato dipende dalla specifica funzione. La funzione COUNT restituisce il valore numerico 0, mentre tutte le altre funzioni restituiscono il valore nullo.

### 3.2.5.2 Raggruppamento

L'operatore di raggruppamento permette di suddividere le tuple di una relazione in base al valore di una o più colonne di tale relazione. Le colonne da usare nel partizionamento sono specificate in un'apposita clausola **GROUP BY** del comando **SELECT**. Le tuple su cui si esegue il partizionamento sono solo le tuple che verificano la clausola di qualificazione del comando **SELECT**. Le tuple che non verificano tale clausola non partecipano al partizionamento.

Il risultato di un comando **SELECT** che contiene la clausola di raggruppamento contiene tante tuple quanti sono i gruppi di tuple risultanti dal partizionamento. La clausola di proiezione di un'interrogazione contenente una clausola **GROUP BY** può includere solamente colonne che compaiono nella clausola **GROUP BY** oppure funzioni di gruppo. Non è invece possibile includere in tale clausola colonne delle relazioni che non compaiano nella clausola **GROUP BY**. Una volta raggruppate le tuple, infatti, si "perde" la possibilità di riferire il valore di tali colonne delle tuple originarie: il risultato contiene una sola tupla per ogni gruppo mentre le singole colonne delle tuple nel gruppo possono contenere diversi valori.

**Esempio 3.18** Per ogni regista vogliamo determinare quanti suoi film sono presenti in catalogo, di quanti generi diversi e la valutazione minima, media e massima di tali film:

```
SELECT regista, COUNT(*) AS numF,
       COUNT(DISTINCT genere) AS numG,
       MIN(valutaz) AS minV, AVG(valutaz) AS avgV,
       MAX(valutaz) AS maxV
  FROM Film
 GROUP BY regista;
```

Il risultato dell'interrogazione è illustrato nella Figura 3.8. Essendo quattro i gruppi ottenuti dal partizionamento in base ai valori della colonna **regista**, il risultato contiene quattro tuple. Notiamo che sono stati assegnati dei nomi alle colonne virtuali i cui valori sono calcolati mediante funzioni di gruppo. □

Il raggruppamento può essere applicato a relazioni ottenute come risultato di operazioni di join. In particolare, la clausola **GROUP BY** può avere come argomenti colonne di relazioni diverse, tra quelle che partecipano all'operazione di join.

**Esempio 3.19** Vogliamo partizionare i noleggi in base al cliente che li ha effettuati ed al regista del film noleggiato. Per ogni gruppo, vogliamo determinare il numero di noleggi e la durata massima (in giorni) di tali noleggi:

```
SELECT codCli, regista, COUNT(*) AS NumN,
       MAX((dataRest-dataNol) DAY) AS durataM
  FROM Noleggio NATURAL JOIN Video
 GROUP BY codCli, regista;
```

---

regista	numF	numG	minV	avgV	maxV
emir kusturica	1	1	3.20	3.20	3.20
tim burton	8	5	3.00	3.58	4.00
gabriele salvatores	3	3	3.00	3.43	3.80
quentin tarantino	2	1	3.50	3.75	4.00
codCli	regista	numN	durataM		
6635	emir kusturica	1	1		
6635	tim burton	8	4		
6635	gabriele salvatores	1	?		
6635	quentin tarantino	2	5		
6642	emir kusturica	1	1		
6642	tim burton	5	1		
6642	gabriele salvatores	1	1		
6642	quentin tarantino	1	2		
6610	emir kusturica	1	2		
6610	tim burton	4	1		
6610	gabriele salvatores	2	1		
regista	numF	numG	minV	avgV	maxV
tim burton	5	5	3.00	3.62	4.00
gabriele salvatores	2	2	3.00	3.40	3.80
quentin tarantino	2	1	3.50	3.75	4.00

Figura 3.8: Risultati delle interrogazioni degli Esempi 3.18, 3.19 e 3.20

Il risultato dell'interrogazione è illustrato nella Figura 3.8. □

SQL prevede inoltre la possibilità di specificare condizioni di ricerca su gruppi di tuple. Queste condizioni di ricerca permettono di scartare alcuni dei gruppi ricavati dal partizionamento ottenuto in base alla clausola **GROUP BY**. Condizioni di ricerca su gruppi di tuple sono specificate nell'apposita clausola **HAVING**. La condizione di ricerca nella clausola **HAVING** può essere una combinazione booleana di più predicati. Tali predicati, tuttavia, possono essere solo predicati che coinvolgono funzioni di gruppo. La clausola **HAVING**, infatti, non può contenere condizioni sui valori delle colonne delle singole tuple.<sup>11</sup> Notiamo infine che non vi è alcuna relazione tra le funzioni di gruppo eventualmente usate nella clausola **SELECT** e quelle usate nella clausola **HAVING**.

**Esempio 3.20** Vogliamo eseguire un'interrogazione simile all'interrogazione dell'Esempio 3.19, ma siamo interessati solo ai film girati prima del 2000 ed ai gruppi che contengono almeno due tuple. L'interrogazione, che restituisce quanti sono tali film, di quanti generi diversi e la valutazione minima, media e massima di tali film, è la seguente:

<sup>11</sup>In realtà SQL:2003 prevede due utili funzioni su insiemi, **EVERY** e **ANY**, che permettono di condizionare l'inclusione di un gruppo nel risultato al fatto che una condizione su singola tupla sia soddisfatta, rispettivamente, da tutte le tuple o da almeno una tupla del gruppo. Tali funzioni non vengono però considerate nella trattazione per mancanza di supporto nei DBMS commerciali.

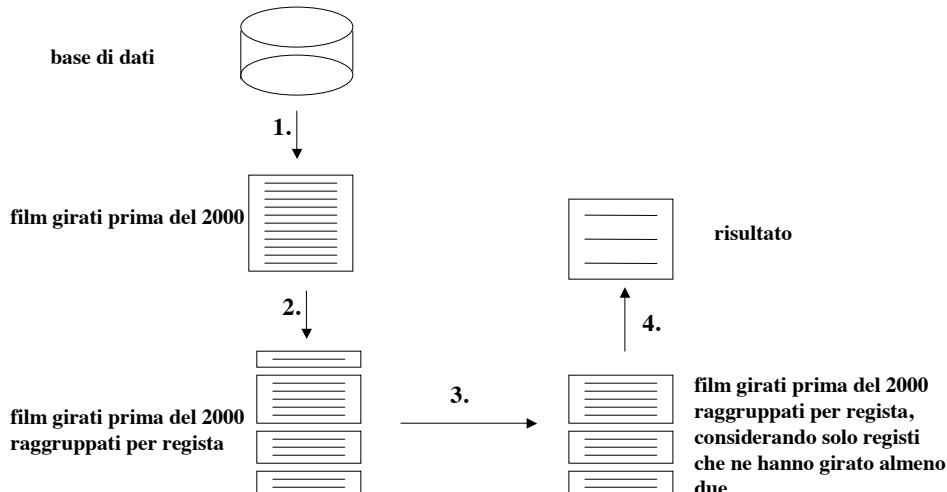


Figura 3.9: Esecuzione di interrogazioni con raggruppamento e clausola HAVING

```

SELECT regista, COUNT(*) AS numF,
       COUNT(DISTINCT genere) AS numG,
       MIN(valutaz) AS minV, AVG(valutaz) AS avgV,
       MAX(valutaz) AS maxV
FROM Film
WHERE anno < 2000
GROUP BY regista
HAVING COUNT(*) >= 2;
    
```

Il risultato dell’interrogazione è illustrato nella Figura 3.8. □

Per chiarire bene l’uso del meccanismo di partizionamento possiamo considerare il seguente semplice modello di “esecuzione” di interrogazioni, illustrato graficamente nella Figura 3.9 relativamente all’esecuzione dell’interrogazione dell’Esempio 3.20:

1. Si applica la condizione di ricerca specificata nella clausola WHERE a tutte le tuple delle relazioni oggetto dell’interrogazione. La valutazione avviene sulle singole tuple, quindi non è ovviamente possibile utilizzare funzioni di gruppo nella clausola WHERE.
2. Alle tuple ottenute al passo precedente si applica il partizionamento specificato dalla clausola GROUP BY.
3. Ad ogni gruppo di tuple ottenuto al passo precedente si applica la condizione di ricerca per i gruppi, specificata dalla clausola HAVING. Notiamo che la valutazione di tale condizione implica il calcolo di funzioni di gruppo.

---

AND	T	F	?
<b>T</b>	T	F	?
<b>F</b>	F	F	F
?	?	F	?

OR	T	F	?
<b>T</b>	T	T	T
<b>F</b>	T	F	?
?	T	?	?

NOT	T
<b>T</b>	F
<b>F</b>	T
?	?

Figura 3.10: Tabelle di verità per la valutazione di combinazioni booleane di predicati

4. I gruppi ottenuti al passo precedente sono i gruppi che verificano l’interrogazione. Per ognuno di tali gruppi, vengono calcolate le funzioni di gruppo specificate nella clausola di proiezione dell’interrogazione. I valori restituiti da tali funzioni costituiscono il risultato dell’interrogazione.

### 3.2.6 Valori nulli

Un aspetto importante nella valutazione delle interrogazioni, di cui bisogna tener conto per una loro corretta formulazione, riguarda i valori nulli. Come già discusso, è possibile che una o più tuple in una relazione abbiano valori nulli per alcune colonne. Una prima questione riguarda qual è il risultato della valutazione di una condizione di ricerca applicata a tuple le cui colonne abbiano valori nulli. A tale riguardo, SQL usa una logica booleana a tre valori; i valori sono:

TRUE (T), FALSE (F), UNKNOWN (?).

Tale logica, rispetto alla classica algebra di Boole, introduce un terzo valore di verità, UNKNOWN, il quale semplicemente indica che il valore di verità di una condizione di ricerca applicata ad una data tupla non è determinabile (ovvero è sconosciuto). Il valore di verità di una condizione di ricerca è determinato in base ai seguenti, semplici principi:

- un predicato semplice valutato su una colonna di una tupla che ha valore nullo dà come risultato della valutazione il valore di verità UNKNOWN; notiamo anche che nel caso di un predicato della forma  $C = C'$ , con  $C$  e  $C'$  colonne aventi entrambe valore nullo, la valutazione restituisce il valore di verità UNKNOWN;
- il valore di verità di una combinazione booleana di predicati viene calcolato in base alle tabelle di verità riportate nella Figura 3.10.

Le tuple per cui il valore di verità della condizione di ricerca è FALSE o UNKNOWN non verificano la condizione di ricerca e non vengono, pertanto, restituite dall’interrogazione.

**Esempio 3.21** Consideriamo le seguenti interrogazioni:

---

**Q1**      `SELECT colloc FROM Noleggio  
WHERE codCli = 6635 AND dataRest > DATE '15-Mar-2006';`

L'interrogazione non restituisce i noleggi in corso, cioè quelli per cui `dataRest` ha valore NULL. I numeri di collocazione restituiti sono quindi 1121, 1122, 1113 e 1129.

**Q2**      `SELECT colloc FROM Noleggio  
WHERE codCli = 6635 AND (dataNol > DATE '20-Mar-2006'  
                  OR dataRest > DATE '20-Mar-2006');`

L'interrogazione restituisce anche noleggi in corso, cioè quelli per cui `dataRest` ha valore NULL, purché siano iniziati dopo il 20 Marzo. I numeri di collocazione restituiti sono 1127 e 1125.

**Q3**      `SELECT colloc FROM Noleggio  
WHERE codCli = 6635 AND NOT dataRest < DATE '15-Mar-2006';`

L'interrogazione non restituisce i noleggi in corso, cioè quelli per cui `dataRest` ha valore NULL. I numeri di collocazione restituiti sono 1121, 1122, 1113 e 1129.

È interessante infine notare come interrogazioni quali le seguenti:

```
SELECT colloc FROM Noleggio  
WHERE dataRest = CURRENT_DATE OR NOT dataRest = CURRENT_DATE;  
  
SELECT colloc FROM Noleggio  
WHERE dataRest = dataRest;
```

non restituiscano tutti i noleggi, come potrebbe sembrare a prima vista, ma solo i noleggi terminati, cioè quelli per cui `dataRest` ha valore non nullo. □

Dato che le tuple con colonne aventi valore nullo non vengono restituite da prediciati su tali colonne, un problema riguarda come ritrovare queste tuple. A tale scopo SQL fornisce il predicato speciale `IS NULL`, di cui esiste la forma negata `IS NOT NULL`, utile per eliminare dal risultato dell'interrogazione tuple con valori nulli. La valutazione del predicato `IS NULL`, applicato ad una data colonna di una tupla, restituisce il valore di verità `TRUE` se la tupla ha valore nullo per tale colonna. Viceversa, la valutazione del predicato `IS NOT NULL` restituisce `TRUE` se la tupla ha un valore diverso dal valore nullo.

**Esempio 3.22** Per determinare le collocazioni dei video attualmente in noleggio al cliente 6635 formuliamo la seguente interrogazione:

```
SELECT colloc FROM Noleggio  
WHERE codCli = 6635 AND dataRest IS NULL;
```

che restituisce i numeri di collocazione 1127 e 1125. Analogamente, l'interrogazione:

---

```
SELECT colloc FROM Noleggio
WHERE codCli = 6635 AND dataRest IS NOT NULL;
```

permette di determinare le collocazioni dei video che sono stati noleggiati e restituiti dal cliente 6635.  $\square$

Nelle espressioni (ad esempio aritmetiche), se un argomento è `NULL`, il valore dell'intera espressione è `NULL`. Per questo motivo, negli esempi dei paragrafi precedenti, le tuple relative a noleggi correnti hanno durata (cioè valore dell'espressione `dataRest - dataNol`) indeterminata, cioè `NULL`. Nel calcolo di una funzione di gruppo, invece, vengono escluse le tuple che hanno valore nullo nella colonna su cui la funzione è calcolata. Questo ha come conseguenza che, ad esempio, in presenza di valori nulli, l'espressione `SUM(e1 + e2)` può dare un risultato diverso da `SUM(e1) + SUM(e2)`. Una funzione di gruppo restituisce infine `NULL` se applicata ad un insieme vuoto o ad un insieme contenente il solo valore `NULL`.

Come visto nell'esempio precedente, se due espressioni  $e_1$  ed  $e_2$  hanno valore `NULL`,  $e_1 = e_2$  non è vero (è `UNKNOWN`). Il vincolo `UNIQUE` viene definito basandosi sull'operatore di uguaglianza e pertanto, poiché due valori nulli non sono considerati uguali, la presenza di due tuple distinte con valori nulli non viene impedita dalla specifica di un vincolo `UNIQUE`. Questo è il motivo per cui, ad esempio, la presenza di due tuple nella relazione `Noleggio` con lo stesso valore per `colloc` e valore nullo per `dataRest` non porta ad una violazione del vincolo `UNIQUE(colloc,dataRest)` definito per la relazione `Noleggio` nell'Esempio 3.2.

Due espressioni con valore nullo non sono quindi valutate uguali, ma non sono distinte, cioè vengono considerate duplicati. Ciò vuol dire che, nel caso di `SELECT DISTINCT`, si ha al più un `NULL` nel risultato e che, nel caso di `GROUP BY`, si ha al più un gruppo per il valore `NULL`. Menzioniamo infine che esiste un predicato SQL `IS DISTINCT FROM` che coincide con `<>` tranne che per il valore nullo, cioè se  $e_1$  ed  $e_2$  sono `NULL`,  $e_1 <> e_2$  restituisce `UNKNOWN`, mentre  $e_1 \text{ IS DISTINCT FROM } e_2$  restituisce `FALSE`.

### 3.2.7 Sotto-interrogazioni

Uno dei motivi che rendono SQL un linguaggio potente è la possibilità di esprimere interrogazioni complesse in termini di interrogazioni più semplici. Come abbiamo già visto, la maggioranza dei predicati in SQL esegue confronti tra una colonna ed un valore costante. Molto spesso può essere necessario ottenere tali valori dalla base di dati, tramite un'opportuna interrogazione. Per agevolare la formulazione di tali interrogazioni, SQL fornisce il meccanismo delle *sotto-interrogazioni*. In particolare, la clausola `WHERE` di un'interrogazione, detta *interrogazione esterna*, può contenere un'altra interrogazione, detta *sotto-interrogazione* (subquery). Un primo semplice esempio di uso di sotto-interrogazioni è il seguente.

**Esempio 3.23** Vogliamo determinare il titolo di tutti i film che hanno la stessa valutazione di “Le iene”. Formuliamo la seguente interrogazione:

---

```
SELECT titolo FROM Film
WHERE valutaz = (SELECT valutaz FROM Film
                  WHERE titolo = 'le iene');
```

che restituisce i valori `nightmare before christmas`, `ed wood`, `la fabbrica di cioccolato` e `le iene`. La sotto-interrogazione:

```
SELECT valutaz FROM Film WHERE titolo = 'le iene';
```

restituisce come valore 4.00, usato poi nel predicato dell'interrogazione esterna per determinare le tuple che soddisfano la condizione di ricerca.  $\square$

Notiamo che un approccio in cui si determinasse con una interrogazione separata la valutazione di “Le iene” e poi si usasse tale valore direttamente in un’interrogazione per determinare i film con tale valutazione (quindi senza l’uso delle sotto-interrogazioni) avrebbe una serie di inconvenienti. Prima di tutto, richiederebbe due interrogazioni diverse, invece di una. Lo svantaggio maggiore sarebbe però che una modifica alla valutazione richiederebbe di riscrivere la seconda interrogazione.

Tramite il meccanismo delle sotto-interrogazioni è possibile formulare richieste più complesse del semplice esempio visto in precedenza.

**Esempio 3.24** Vogliamo determinare i film la cui valutazione è superiore alla media:

```
SELECT * FROM Film
WHERE valutaz > (SELECT AVG(valutaz) FROM Film);
```

La sotto-interrogazione restituisce il valore 3.55, quindi solo i film con valutazione superiore a tale valore vengono selezionati. Il risultato dell’interrogazione è illustrato nella Figura 3.11.  $\square$

Negli esempi visti finora ogni sotto-interrogazione restituisce un solo valore. Tali sotto-interrogazioni sono note come *sotto-interrogazioni scalari*. Se una sotto-interrogazione scalare restituisce più di una tupla, si genera un errore a run-time. Una particolarità delle sotto-interrogazioni scalari è che, se nessuna tupla verifica la sotto-interrogazione, viene restituito il valore `NULL`. Se, invece, la sotto-interrogazione restituisce un insieme di valori (*table subquery*) è necessario specificare come tali valori devono essere usati nel predicato. A tale scopo vengono introdotti i *quantificatori ANY* ed *ALL* che sono inseriti tra l’operatore di confronto e la sotto-interrogazione. *ANY* (come il suo sinonimo *SOME*) restituisce il valore di verità `TRUE` quando la valutazione dell’operatore di confronto restituisce `TRUE` su almeno una delle tuple ottenute dalla valutazione della sotto-interrogazione. In particolare, restituisce `FALSE` se la sotto-interrogazione non restituisce tuple. *ALL*, invece, restituisce il valore di verità `TRUE` quando la valutazione dell’operatore di confronto restituisce `TRUE` su tutte le tuple ottenute dalla valutazione della sotto-interrogazione. In particolare, restituisce `TRUE` se la sotto-interrogazione non restituisce tuple.

---

titolo	regista	anno	genere	valutaz
edward mani di forbice	tim burton	1990	fantastico	3.60
nightmare before christmas	tim burton	1993	animazione	4.00
ed wood	tim burton	1994	drammatico	4.00
la fabbrica di cioccolato	tim burton	2005	fantastico	4.00
mediterraneo	gabriele salvatores	1991	commedia	3.80
le iene	quentin tarantino	1992	thriller	4.00

titolo	regista	anno
edward mani di forbice	tim burton	1990
nightmare before christmas	tim burton	1993
mediterraneo	gabriele salvatores	1991
le iene	quentin tarantino	1992

titolo	regista	anno
edward mani di forbice	tim burton	1990
mediterraneo	gabriele salvatores	1991

Figura 3.11: Risultati delle interrogazioni degli Esempi 3.24, 3.25 e 3.26

**Esempio 3.25** Vogliamo determinare titolo, regista ed anno dei film più vecchi di *almeno un* film di Quentin Tarantino. Formuliamo la seguente interrogazione:<sup>12</sup>

```
SELECT titolo, regista, anno
FROM Film
WHERE anno < ANY (SELECT anno FROM Film
                    WHERE regista = 'quentin tarantino');
```

il cui risultato è illustrato nella Figura 3.11. □

**Esempio 3.26** Vogliamo ora determinare titolo, regista ed anno dei film precedenti a *tutti* i film di Quentin Tarantino. Formuliamo la seguente interrogazione:

```
SELECT titolo, regista, anno
FROM Film
WHERE anno < ALL (SELECT anno FROM Film
                    WHERE regista = 'quentin tarantino');
```

il cui risultato è illustrato nella Figura 3.11. □

Sono definite le seguenti abbreviazioni per ANY ed ALL:

- IN equivalente ad = ANY;
- NOT IN equivalente ad  $\neq$  ALL.

<sup>12</sup>Notiamo che sono possibili formulazione alternative di questa interrogazione e della successiva, tramite l'uso, rispettivamente, delle funzioni di gruppo MIN e MAX.

È inoltre possibile selezionare più di una colonna tramite una sotto-interrogazione. In tal caso è necessario apporre delle parentesi alla lista delle colonne a sinistra dell'operatore di confronto.

**Esempio 3.27** Consideriamo le seguenti interrogazioni.

**Q1** Elencare il titolo dei film di Quentin Tarantino presenti nella videoteca, usciti nello stesso anno di un film di Tim Burton:

```
SELECT titolo FROM Film
WHERE regista = 'quentin tarantino'
      AND anno IN (SELECT anno FROM Film
                    WHERE regista = 'tim burton');
```

L'interrogazione restituisce **pulp fiction**.

**Q2** Elencare il titolo dei film di Quentin Tarantino presenti nella videoteca, usciti in un anno in cui non sono usciti film di Tim Burton presenti nella videoteca:

```
SELECT titolo FROM Film
WHERE regista = 'quentin tarantino'
      AND anno NOT IN (SELECT anno FROM Film
                        WHERE regista = 'tim burton');
```

L'interrogazione restituisce **le iene**.

**Q3** Elencare il titolo dei film presenti nella videoteca con lo stesso anno di uscita e genere di un film di Tim Burton:

```
SELECT titolo FROM Film
WHERE regista <> 'tim burton' AND
      (anno,genere) IN (SELECT (anno,genere) FROM Film
                        WHERE regista = 'tim burton');
```

L'interrogazione non restituisce alcuna tupla. Notiamo che l'interrogazione principale contiene la condizione **regista <> 'tim burton'** per evitare di restituire i film di Tim Burton stessi. □

Una sotto-interrogazione può includere nella propria condizione di ricerca un'altra sotto-interrogazione, condizioni di join e tutti i predicati visti finora. La clausola di qualificazione di un'interrogazione può contenere una qualsiasi combinazione di condizioni normali e condizioni con sotto-interrogazioni. Le sotto-interrogazioni possono essere usate anche all'interno dei comandi di modifica previsti da SQL (vedi Paragrafo 3.3).

che, valutata sulle relazioni delle Figure 2.1 e 2.2, produce il risultato illustrato nella Figura 2.8(c).  $\square$

**Divisione.** L’operazione di *divisione*, date due relazioni  $R$  ed  $S$ , con insiemi di attributi  $U_R$  ed  $U_S$ , rispettivamente, tali che  $U_S \subset U_R$ , è denotata da  $R \div S$ . Tale operazione permette di determinare le tuple  $t$  di schema  $D = U_R - U_S$  tali che in  $R$  appaiano tutte le tuple che si ottengono combinando  $t$  con ogni tupla di  $S$ . Una tupla  $t$  viene cioè restituita se, per ogni tupla  $s$  in  $S$ , in  $R$  è presente una tupla  $r$  tale che  $r[D] = t[D]$  e  $r[U_S] = s[U_S]$ . Una formulazione alternativa della divisione, che giustifica il suo nome come operazione “inversa” del prodotto, è la seguente:  $t \in R \div S$  se e solo se  $\{t\} \times S \subseteq R$ . L’idea intuitiva è che l’operazione di divisione è utile per determinare le tuple per cui in una relazione c’è una corrispondenza con *tutte* le tuple di un’altra relazione.

**Esempio 2.19** Vogliamo determinare i codici dei clienti che hanno noleggiato *tutti* i film di Tim Burton. Iniziamo a determinare i film di Tim Burton, mediante l’espressione:

$$TB = \Pi_{\text{titolo}, \text{regista}}(\sigma_{\text{regista}='tim burton'}(\text{Film}))$$

il cui risultato è illustrato nella Figura 2.9(a). Determiniamo poi i film (identificati da titolo e regista) noleggiati dai clienti, con l’espressione:

$$NC = \Pi_{\text{codCli}, \text{titolo}, \text{regista}}(\text{Noleggio} \bowtie \text{Video})$$

il cui risultato è illustrato nella Figura 2.9(b). L’interrogazione iniziale corrisponde alla divisione tra  $NC$  e  $TB$ , è cioè formulata tramite la seguente espressione algebrica:

$$\Pi_{\text{codCli}, \text{titolo}, \text{regista}}(\text{Noleggio} \bowtie \text{Video}) \div \Pi_{\text{titolo}, \text{regista}}(\sigma_{\text{regista}='tim burton'}(\text{Film})).$$

Il risultato dell’interrogazione è dato, pertanto, da tutti quei codici cliente che appaiono nella relazione  $NC$  abbinati ad ognuno dei film in  $TB$ . Come evidenziato nella Figura 2.9(c), solo il cliente di codice 6635 verifica l’interrogazione.

Per una formulazione corretta dell’interrogazione, è stato necessario mantenere anche il regista nelle relazioni argomento della divisione poiché `titolo` non è chiave di `Film`, quindi potrebbe esistere un film, di altro regista, con lo stesso titolo di un film di Tim Burton. Notiamo anche che la divisione viene applicata ai clienti ed ai film che hanno noleggiato e non ai clienti ed ai video che hanno noleggiato perché possono in generale essere presenti più video dello stesso film e vogliamo richiedere che il cliente abbia noleggiato, per ogni film di Tim Burton, almeno un video che lo contiene (e non tutti i video che lo contengono).  $\square$

La divisione è un’operazione derivata dell’algebra relazionale. La divisione di  $R$  per  $S$  può infatti essere espressa come segue:

$$R \div S = \Pi_D(R) - \Pi_D((\Pi_D(R) \times S) - R).$$

titolo	regista
edward mani di forbice	tim burton
nightmare before christmas	tim burton
ed wood	tim burton
mars attacks	tim burton
il mistero di sleepy hollow	tim burton
big fish	tim burton
la sposa cadavere	tim burton
la fabbrica di cioccolato	tim burton

(a)  $\Pi_{\text{titolo}, \text{regista}}(\sigma_{\text{regista}=\text{'tim burton'}}(\text{Film}))$

codCli	titolo	regista
6635	underground	emir kusturica
6635	edward mani di forbice	tim burton
6635	nightmare before christmas	tim burton
6635	ed wood	tim burton
6642	underground	emir kusturica
6635	mars attacks	tim burton
6635	il mistero di sleepy hollow	tim burton
6642	nightmare before christmas	tim burton
6642	ed wood	tim burton
6635	la sposa cadavere	tim burton
6635	la fabbrica di cioccolato	tim burton
6635	big fish	tim burton
6635	le iene	quentin tarantino
6642	mars attacks	tim burton
6610	mediterraneo	gabriele salvatores
6610	underground	emir kusturica
6610	big fish	tim burton
6642	pulp fiction	quentin tarantino
6610	io non ho paura	gabriele salvatores
6610	edward mani di forbice	tim burton
6642	io non ho paura	gabriele salvatores
6610	nightmare before christmas	tim burton
6635	pulp fiction	quentin tarantino
6635	nirvana	gabriele salvatores
6642	la fabbrica di cioccolato	tim burton
6642	big fish	tim burton

(b)  $\Pi_{\text{codCli}, \text{titolo}, \text{regista}}(\text{Noleggio} \bowtie \text{Video})$

codCli  
6635

(c)  $\Pi_{\text{codCli}, \text{titolo}, \text{regista}}(\text{Noleggio} \bowtie \text{Video}) \div$   
 $\Pi_{\text{titolo}, \text{regista}}(\sigma_{\text{regista}=\text{'tim burton'}}(\text{Film}))$

Figura 2.9: Divisione

codCli	titolo	regista
6635	edward mani di forbice	tim burton
6635	nightmare before christmas	tim burton
6635	ed wood	tim burton
6635	mars attacks	tim burton
6635	il mistero di sleepy hollow	tim burton
6635	big fish	tim burton
6635	la sposa cadavere	tim burton
6635	la fabbrica di cioccolato	tim burton
6642	edward mani di forbice	tim burton
6642	nightmare before christmas	tim burton
6642	ed wood	tim burton
6642	mars attacks	tim burton
6642	il mistero di sleepy hollow	tim burton
6642	big fish	tim burton
6642	la sposa cadavere	tim burton
6642	la fabbrica di cioccolato	tim burton
6610	edward mani di forbice	tim burton
6610	nightmare before christmas	tim burton
6610	ed wood	tim burton
6610	mars attacks	tim burton
6610	il mistero di sleepy hollow	tim burton
6610	big fish	tim burton
6610	la sposa cadavere	tim burton
6610	la fabbrica di cioccolato	tim burton

(a)  $\Pi_{\text{codCli}}(NC) \times TB$

codCli	titolo	regista
6642	edward mani di forbice	tim burton
6642	il mistero di sleepy hollow	tim burton
6642	la sposa cadavere	tim burton
6610	ed wood	tim burton
6610	mars attacks	tim burton
6610	il mistero di sleepy hollow	tim burton
6610	la sposa cadavere	tim burton
6610	la fabbrica di cioccolato	tim burton

(b)  $(\Pi_{\text{codCli}}(NC) \times TB) - NC$

Figura 2.10: Divisione come operazione derivata

Op.	Funzionalità	Cond.	Semantica
$\Pi_A$	$\mathcal{R}(U) \rightarrow \mathcal{R}(A)$		$\Pi_A(R) = \{t[A] \mid t \in R\}$
$\sigma_F$	$\mathcal{R}(U) \rightarrow \mathcal{R}(U)$	$A(F) \subseteq U$	$\sigma_F(R) = \{t \mid t \in R \wedge F(t)\}$
$\times$	$\mathcal{R}(U) \times \mathcal{R}(V) \rightarrow \mathcal{R}(U \cup V)$	$U \cap V = \emptyset$	$R_1 \times R_2 = \{t_1 \cdot t_2 \mid t_1 \in R_1 \wedge t_2 \in R_2\}$
$\cup$	$\mathcal{R}(U) \times \mathcal{R}(U) \rightarrow \mathcal{R}(U)$		$R_1 \cup R_2 = \{t \mid t \in R_1 \vee t \in R_2\}$
$-$	$\mathcal{R}(U) \times \mathcal{R}(U) \rightarrow \mathcal{R}(U)$		$R_1 - R_2 = \{t \mid t \in R_1 \wedge t \notin R_2\}$
$\cap$	$\mathcal{R}(U) \times \mathcal{R}(U) \rightarrow \mathcal{R}(U)$		$R_1 \cap R_2 = \{t \mid t \in R_1 \wedge t \in R_2\}$
$\bowtie_F$	$\mathcal{R}(U) \times \mathcal{R}(V) \rightarrow \mathcal{R}(U \cup V)$	$U \cap V = \emptyset$	$R_1 \bowtie_F R_2 = \{t_1 \cdot t_2 \mid t_1 \in R_1 \wedge t_2 \in R_2 \wedge F(t_1 \cdot t_2)\}$
$\bowtie$	$\mathcal{R}(U) \times \mathcal{R}(V) \rightarrow \mathcal{R}(U \cup V)$		$R_1 \bowtie R_2 = \{t \mid t[U] \in R_1 \wedge t[V] \in R_2\}$
$\div$	$\mathcal{R}(U) \times \mathcal{R}(V) \rightarrow \mathcal{R}(U \setminus V)$	$V \subset U$	$R_1 \div R_2 = \{t \mid \forall t_2 \in R_2 \exists t_1 \in R_1 \text{ t.c. } t_1[U \setminus V] = t, t_1[V] = t_2\}$

Tabella 2.1: Operazioni dell’algebra relazionale

Il seguente esempio illustra il significato di tale espressione nel contesto dell’interrogazione dell’Esempio 2.19.

**Esempio 2.20** L’espressione  $\Pi_D(R) \times S$  corrisponde a “costruire” una relazione in cui le tuple  $t$  in  $\Pi_D(R)$  sono abbinate a tutte le tuple  $s$  in  $S$ . Tale relazione contiene quindi tutte le possibili combinazioni tra tuple  $t$  e tuple  $s$  che potrebbero essere in  $R$ . In riferimento all’Esempio 2.19, l’espressione  $\Pi_{\text{codcli}}(NC) \times TB$  produce la relazione, mostrata nella Figura 2.10(a), in cui tutti i clienti sono abbinati a tutti i film di Tim Burton.

Sottraendo da tale relazione  $R$ , otteniamo le tuple “mancanti” in  $R$ , cioè le tuple, combinazione di tuple  $t$  ed  $s$ , che non compaiono in  $R$ . In riferimento all’Esempio 2.19, l’espressione  $(\Pi_{\text{codcli}}(NC) \times TB) - NC$  produce una relazione, mostrata nella Figura 2.10(b), che contiene i clienti abbinati ai film di Tim Burton che non hanno mai noleggiato.

Sottraendo infine dall’insieme di tutte le tuple  $t$  quelle restituite dall’espressione  $\Pi_D((\Pi_D(R) \times S) - R)$  otteniamo il risultato richiesto. In riferimento all’Esempio 2.19, l’espressione  $\Pi_{\text{codcli}}(NC) - \Pi_{\text{codcli}}((\Pi_{\text{codcli}}(NC) \times TB) - NC)$ , elimina dai codici dei clienti che hanno effettuato almeno un noleggio (cioè 6635, 6642, 6610) quelli che compaiono nella relazione di Figura 2.10(b), corrispondenti ai clienti per cui esiste un film di Tim Burton che non hanno mai noleggiato (cioè 6642 e 6610). Il risultato è quindi costituito dal solo codice cliente 6635.  $\square$

La Tabella 2.1 riassume la funzionalità, le condizioni di applicabilità e la semantica delle operazioni dell’algebra relazionale. Nella tabella, indichiamo con  $U, V, A$  generici schemi di relazione, con  $\mathcal{R}(U)$  l’insieme delle relazioni di schema  $U$  e con  $A(F)$  gli attributi che compaiono nel predicato  $F$ . Date due tuple  $t_1$  e  $t_2$ , inoltre,  $t_1 \cdot t_2$  indica la concatenazione di tali tuple, cioè la tupla che contiene tutte le componenti di  $t_1$  seguite da tutte le componenti di  $t_2$ .

### 2.2.3 Esempi di interrogazioni

In questo paragrafo presentiamo alcuni esempi di interrogazioni complesse ottenute applicando gli operatori dell'algebra alle relazioni della base di dati della videoteca.

**Esempio 2.21** Consideriamo la base di dati della videoteca contenuta nelle Figure 2.1 e 2.2 e le seguenti interrogazioni.

**Q1** Selezionare i codici dei clienti che hanno noleggiato sia film drammatici che film horror:

$$\Pi_{codCli}(\text{Noleggio} \bowtie \text{Video} \bowtie \sigma_{\text{genere}='drammatico'}(\text{Film})) \cap \\ \Pi_{codCli}(\text{Noleggio} \bowtie \text{Video} \bowtie \sigma_{\text{genere}='horror'}(\text{Film})).$$

Notiamo come in questo caso debba essere utilizzata un'intersezione in quanto la condizione richiede di verificare la presenza nelle relazioni di due tuple distinte: una relativa al film drammatico ed al suo noleggio, l'altra relativa al film horror ed al suo noleggio.

**Q2** Selezionare i codici dei clienti che non hanno mai noleggiato film horror:

$$\Pi_{codCli}(\text{Cliente}) \setminus \Pi_{codCli}(\text{Noleggio} \bowtie \text{Video} \bowtie \sigma_{\text{genere}='horror'}(\text{Film})).$$

Notiamo come in questo caso debba essere utilizzata una differenza in quanto la condizione non può essere verificata su una singola tupla (con una condizione del tipo  $\text{genere} \neq 'horror'$ ): il fatto che un cliente abbia noleggiato un film di genere non horror non esclude che abbia noleggiato (anche) film horror.

**Q3** Selezionare i codici dei clienti che hanno noleggiato film di almeno due generi diversi:

$$\Pi_{codCli}(\sigma_{codCli=c \wedge \text{genere} \neq g}(\Pi_{codCli, \text{genere}}(\text{Noleggio} \bowtie \text{Video} \bowtie \text{Film}) \times \\ \rho_{codCli, \text{genere} \leftarrow c, g}(\Pi_{codCli, \text{genere}}(\text{Noleggio} \bowtie \text{Video} \bowtie \text{Film}))).$$

Notiamo come questa interrogazione richieda di effettuare un join tra una relazione e se stessa, per verificare la presenza di tuple che soddisfano la condizione. Poiché il join non sarà naturale (deve imporre che i generi delle due tuple siano diversi) è necessario ridenominare gli attributi in una delle due "copie" della relazione. Le condizioni di applicabilità del theta-join, infatti, come quelle del prodotto cartesiano, richiedono che gli schemi degli operandi siano disgiunti.

**Q4** Selezionare titolo e regista del film con la valutazione più alta:

$$\Pi_{titolo, \text{regista}}(\text{Film}) \setminus \\ \Pi_{titolo, \text{regista}}(\text{Film} \bowtie_{\text{valutaz} < v} \rho_{titolo, \text{regista}, \text{anno}, \text{genere}, \text{valutaz} \leftarrow t, r, a, g, v}(\text{Film})).$$

Anche in questo caso è necessario effettuare un join della relazione con se stessa (previa ridenominazione in una delle due “copie” per rendere gli schemi disgiunti). L’idea per determinare il massimo è quella di sottrarre da tutti i film quelli per cui ne esiste uno con valutazione maggiore.  $\square$

### 2.3 Calcolo relazionale

L’algebra relazionale è un linguaggio che richiede la specifica delle operazioni da eseguire per ottenere il risultato di una data interrogazione; è quindi un linguaggio procedurale. Il calcolo relazionale differisce rispetto all’algebra in quanto nel calcolo viene semplicemente data una descrizione formale del risultato senza specificare come ottenerlo; è quindi un linguaggio dichiarativo. Esistono due varianti del calcolo, la cui principale differenza riguarda gli oggetti denotati dalle variabili del linguaggio: il *calcolo relazionale orientato alla tupla* (*TRC – Tuple Relational Calculus*), in cui le variabili denotano tuple, ed il *calcolo relazionale orientato ai domini* (*DRC – Domain Relational Calculus*), in cui le variabili rappresentano valori di domini. Il DRC ha lo svantaggio di richiedere l’utilizzo di numerose variabili nella formulazione di interrogazioni, spesso una per ciascun attributo di ogni relazione coinvolta, e conseguentemente di numerosi quantificatori per tali variabili. Le uniche realizzazioni pratiche di linguaggi almeno in parte basate sul DRC, che vanno sotto il nome di *QBE* (*Query-By-Example*), utilizzano infatti un’interfaccia grafica per facilitare la specifica di interrogazioni. Nel seguito del paragrafo illustreremo brevemente il TRC, che ha una corrispondenza diretta con SQL, e non considereremo ulteriormente il DRC. Nel Paragrafo 2.3.3 confronteremo poi il TRC con l’algebra relazionale, discutendone l’equivalenza.

#### 2.3.1 Definizione del calcolo relazionale orientato alla tupla

In quanto segue introduciamo la definizione del TRC, definendo gli atomi, quindi le formule ed infine le espressioni.

**Atomi.** Gli atomi del TRC hanno una delle seguenti forme:

1.  $s \in R$  dove:  $R$  è un nome di relazione ed  $s$  è una variabile. Questo atomo asserisce che la tupla denotata da  $s$  appartiene alla relazione  $R$ .
2.  $s[A]\theta u[A']$  dove:  $s$  ed  $u$  sono variabili;  $\theta$  è un operatore relazionale di confronto (vedi Paragrafo 2.2.1);  $A$  ed  $A'$  sono nomi di attributi. Questo atomo asserisce che il valore dell’attributo di nome  $A$  della tupla denotata da  $s$  è in relazione  $\theta$  con il valore dell’attributo di nome  $A'$  della tupla denotata da  $u$ .
3.  $s[A]\theta a$  dove:  $s$  è una variabile;  $\theta$  è un operatore relazionale di confronto;  $A$  è un nome di attributo;  $a$  è una costante. Questo atomo asserisce che il valore dell’attributo di nome  $A$  della tupla denotata da  $s$  è in relazione  $\theta$  con il valore  $a$ .

### 3.2.8 Sotto-interrogazioni correlate

Negli esempi visti nel paragrafo precedente, ogni sotto-interrogazione viene eseguita una volta per tutte, rispetto all'esecuzione dell'interrogazione esterna, ed il valore (o l'insieme di valori) da essa restituito è usato nella clausola `WHERE` dell'interrogazione esterna. È possibile in SQL formulare interrogazioni più sofisticate in cui le sotto-interrogazioni sono eseguite ripetutamente per ogni *tupla candidata* considerata nella valutazione dell'interrogazione esterna.

**Esempio 3.28** Supponiamo di voler determinare titolo, regista ed anno dei film la cui valutazione è superiore alla media delle valutazioni dei film dello stesso regista. Per risolvere tale interrogazione è necessaria un'interrogazione esterna che selezioni i film in base ad un predicato sulla valutazione. Tale interrogazione avrà la forma:

```
SELECT titolo, regista, anno FROM Film
WHERE valutaz > (media della valutazione dei film
                   del regista del film candidato);
```

È inoltre necessaria una sotto-interrogazione che calcoli la valutazione media dei film del regista di ogni tupla candidata della relazione `Film`. Tale sotto-interrogazione avrà la forma:

```
(SELECT AVG(valutaz) FROM Film
 WHERE regista = valore di regista nella tupla candidata);
```

□

Come evidenziato dall'esempio precedente, ogni volta che l'interrogazione esterna considera una tupla candidata, deve invocare la sotto-interrogazione e “passare” il regista del film in esame. La sotto-interrogazione calcola quindi la valutazione media dei film del regista che è stato passato e restituisce tale valore all'interrogazione esterna. L'interrogazione esterna può quindi confrontare la valutazione del film in esame con il valore restituito dalla sotto-interrogazione.<sup>13</sup> Questo tipo di interrogazioni è detto *correlato* in quanto ogni esecuzione della sotto-interrogazione è correlato ai valori di una o più colonne delle tuple candidate nell'interrogazione esterna. Per fare riferimento alle colonne delle tuple candidate nell'interrogazione esterna si utilizzano gli *alias di relazione*. Un alias di relazione è definito nell'interrogazione esterna e viene utilizzato nella sotto-interrogazione correlata. L'alias viene definito nella clausola `FROM` facendo seguire il nome della relazione da un identificatore. Alternativamente, è possibile usare, sempre nella clausola `FROM`, la sintassi equivalente `<nome relazione> AS <alias>`.

**Esempio 3.29** La seguente interrogazione è la formulazione in SQL dell'interrogazione discussa nell'Esempio 3.28:

<sup>13</sup>Notiamo che quanto descritto è una strategia naïve per l'esecuzione dell'interrogazione dell'esempio. Lo scopo della breve descrizione è di illustrarne il significato. I DBMS, ovviamente, adottano strategie molto più efficienti per l'esecuzione di tali interrogazioni.

titolo	regista	anno
edward mani di forbice	tim burton	1990
nightmare before christmas	tim burton	1993
ed wood	tim burton	1994
la fabbrica di cioccolato	tim burton	2005
io non ho paura	gabriele salvatores	2003
mediterraneo	gabriele salvatores	1991
le iene	quentin tarantino	1992

Figura 3.12: Risultato dell’interrogazione dell’Esempio 3.29

```
SELECT titolo, regista, anno FROM Film X
WHERE valutaz > (SELECT AVG(valutaz) FROM Film
                   WHERE regista = X.regista);
```

Nell’interrogazione, *X* è un alias per la relazione *Film*. Il risultato dell’interrogazione è illustrato nella Figura 3.12.  $\square$

I concetti alla base della correlazione sono i seguenti:

- Una sotto-interrogazione correlata fa riferimento ad una o più colonne di tuple (dette tuple candidate) selezionate da una relazione *R* dall’interrogazione esterna.
- Nell’interrogazione esterna viene definito un alias per tale relazione *R*.
- La sotto-interrogazione utilizza l’alias per denotare i valori delle colonne nelle tuple (di *R*) candidate nell’interrogazione esterna.

Una sotto-interrogazione può pertanto essere vista come una “procedura” invocata ogni volta che una tupla candidata viene esaminata dall’interrogazione esterna e a cui vengono “passati” come “argomenti” uno o più valori di colonne di tale tupla.

Gli alias di relazione possono essere utilizzati anche in interrogazioni senza sotto-interrogazioni. In particolare, sono utili quando si vuole fare riferimento a due diverse tuple della stessa relazione, come illustrato dal seguente esempio.

**Esempio 3.30** Vogliamo determinare i registi di cui sono usciti (almeno) due film diversi lo stesso anno:

```
SELECT DISTINCT X.regista FROM Film X, Film Y
WHERE X.anno = Y.anno AND X.regista = Y.regista
      AND X.titolo <> Y.titolo;
```

La seguente:

```
SELECT regista FROM Film
GROUP BY regista, anno
HAVING COUNT(*) >= 2;
```

è una formulazione alternativa della stessa interrogazione basata sul raggruppamento. Il risultato dell'interrogazione contiene la sola tupla `tim burton`.  $\square$

**Operatori EXISTS e NOT EXISTS.** Le sotto-interrogazioni correlate sono spesso usate in combinazione con gli operatori `EXISTS` e `NOT EXISTS` il cui significato è il seguente. Sia *sq* una sotto-interrogazione:

- Il predicato `EXISTS(sq)` restituisce il valore booleano `TRUE` se la sotto-interrogazione *sq* restituisce almeno una tupla (ha cioè risultato non vuoto); restituisce il valore booleano `FALSE` altrimenti.
- Il predicato `NOT EXISTS(sq)` restituisce il valore booleano `TRUE` se la sotto-interrogazione *sq* non restituisce alcuna tupla (ha cioè risultato vuoto); restituisce il valore booleano `FALSE` altrimenti.

La valutazione di questi prediciati non restituisce mai il valore di verità `UNKNOWN`.

**Esempio 3.31** Una formulazione alternativa dell'interrogazione dell'Esempio 3.30, basata sull'uso dell'operatore `EXISTS`, è la seguente:

```
SELECT DISTINCT regista FROM Film X
WHERE EXISTS (SELECT * FROM Film
               WHERE regista = X.regista AND anno = X.anno
                     AND titolo <> X.titolo);
```

Supponiamo ora invece di volere determinare i registi di cui non sono mai usciti due film lo stesso anno. Non è sufficiente sostituire `EXISTS` con `NOT EXISTS` nell'interrogazione precedente, perché questo corrisponderebbe a richiedere i registi di cui, in almeno un anno, non sono usciti due film. Tutti i registi della nostra base di dati soddisfano quindi tale interrogazione. La formulazione corretta, basata sull'uso di `NOT EXISTS`, è la seguente:<sup>14</sup>

```
SELECT DISTINCT regista FROM Film X
WHERE NOT EXISTS (SELECT * FROM Film Y, Film Z
                     WHERE Y.regista = Z.regista AND Y.anno = Z.anno
                           AND Y.titolo <> Z.titolo
                           AND Y.regista = X.regista);
```

in cui l'interrogazione interna, cui viene richiesto di restituire un insieme vuoto di tuple, corrisponde all'interrogazione dell'Esempio 3.30. Il risultato dell'interrogazione è costituito dai valori `emir kusturica`, `gabriele salvatores` e `quentin tarantino`.  $\square$

---

<sup>14</sup>Le interrogazioni di questo esempio, come le interrogazioni di divisione discusse nel seguito, possono essere espresse più agevolmente per mezzo dei quantificatori `FOR SOME` e `FOR ALL` presenti in SQL:2003. Poiché però tali operatori non sono forniti dalla maggioranza dei DBMS, non li consideriamo nella trattazione.

**Divisione.** Un uso importante dell'operatore `NOT EXISTS` è nella formulazione dell'operazione algebrica di divisione per cui in SQL non esiste un operatore apposito. Consideriamo il seguente esempio.

**Esempio 3.32** Vogliamo determinare i codici dei clienti che hanno noleggiato tutti i film di Tim Burton. Ricordiamo che in algebra relazionale questa interrogazione è formulata tramite la seguente espressione algebrica (vedi Capitolo 2, Esempio 2.19):

$$\Pi_{\text{codCli}, \text{titolo}, \text{regista}}(\text{Noleggio} \bowtie \text{Video}) \div \Pi_{\text{titolo}, \text{regista}}(\sigma_{\text{regista} = 'tim burton'}(\text{Film})).$$

L'interrogazione in SQL richiede di ragionare in base al concetto di controesempio.<sup>15</sup> In altre parole, un cliente verifica l'interrogazione se non è possibile determinare un film di Tim Burton che il cliente non ha mai noleggiato, come richiesto dall'interrogazione:

```
SELECT DISTINCT codCli FROM Noleggio X
WHERE NOT EXISTS (SELECT * FROM Film F
                    WHERE regista = 'tim burton' AND
                      NOT EXISTS (SELECT *
                                FROM Noleggio NATURAL JOIN Video
                                WHERE codCli = X.codCli
                                  AND titolo = F.titolo
                                  AND regista = F.regista));
```

L'interrogazione è costituita da un'interrogazione esterna e due sotto-interrogazioni annidate. Le due sotto-interrogazioni sono entrambe correlate, la più esterna verifica la non esistenza del film di Tim Burton e la più interna la non esistenza di un noleggio del cliente in esame per tale film. Il risultato dell'interrogazione è 6635.  $\square$

Una formulazione alternativa delle interrogazioni di divisione, illustrata nell'esempio seguente, è basata sull'utilizzo della funzione `COUNT`.

**Esempio 3.33** Per determinare i clienti che hanno noleggiato tutti i film di Tim Burton confrontiamo il numero di film di Tim Burton con il numero dei film di Tim Burton noleggiati dal cliente:

```
SELECT codCli FROM Noleggio NATURAL JOIN Video
WHERE regista = 'tim burton'
GROUP BY codCli
HAVING COUNT(DISTINCT titolo) = (SELECT COUNT(DISTINCT titolo)
                                    FROM Film
                                    WHERE regista = 'tim burton');
```

---

<sup>15</sup>La specifica della divisione in SQL si basa sulla tautologia:  $\forall z(\exists y p(z,y)) \Leftrightarrow \neg\exists z(\neg\exists y p(z,y))$ .

Con riferimento al modello di esecuzione nella Figura 3.9, la valutazione del passo 2 dell’interrogazione esterna produce tre gruppi, uno per ogni cliente. La sotto-interrogazione restituisce il valore 8. Per il cliente 6635, la clausola HAVING è soddisfatta ( $8 = 8$ ), mentre la condizione non è verificata per i clienti 6642 ( $5 = 8$ ) e 6610 ( $3 = 8$ ). Il risultato dell’interrogazione è quindi 6635.  $\square$

### 3.2.9 Operazioni insiemistiche

Le operazioni insiemistiche di unione, differenza, intersezione dell’algebra relazionale (vedi Capitolo 2) hanno dato luogo ad appositi operatori in SQL. Un’interrogazione (o sotto-interrogazione) può essere costituita da una o più sotto-interrogazioni connesse dall’operatore UNION. Tale operatore restituisce tutte le tuple distinte restituite da almeno una delle sotto-interrogazioni a cui è applicata.

**Esempio 3.34** Supponiamo di voler determinare i nomi ed i cognomi di registi o di clienti della nostra videoteca. Formuliamo la seguente interrogazione:

```
SELECT regista FROM Film
UNION
SELECT nome || ' ' || cognome FROM Cliente;
```

il cui risultato è illustrato nella Figura 3.13.  $\square$

Analogamente a quanto avviene in algebra relazionale, l’operatore UNION impone alcune restrizioni sulle sotto-interrogazioni su cui opera. Le interrogazioni devono restituire lo stesso numero di colonne e le colonne corrispondenti devono avere lo stesso dominio (non è necessario che abbiano la stessa lunghezza) o domini compatibili. A differenza di quanto avviene in algebra, il nome delle colonne può essere differente e le colonne nel risultato assumono il nome delle colonne nel primo argomento. Se si usa una clausola ORDER BY, tale clausola deve essere usata una sola volta alla fine dell’interrogazione e non alla fine di ogni SELECT. Quando si specificano le colonne su cui eseguire l’ordinamento non si possono usare i nomi di colonna, poiché questi potrebbero essere differenti nelle varie relazioni. Occorre pertanto indicarle specificandone la *posizione relativa* all’interno della clausola di proiezione dell’interrogazione.

**Esempio 3.35** Vogliamo che il risultato dell’interrogazione presentata nell’Esempio 3.34 sia ordinato:

```
(SELECT regista FROM Film
UNION
SELECT nome || ' ' || cognome FROM Cliente)
ORDER BY 1;
```

$\square$

regista	anno
emir kusturica	1990
tim burton	1993
gabriele salvatores	1996
quentin tarantino	1999
anna rossi	2003
paola bianchi	2005
marco verdi	2005

Figura 3.13: Risultati delle interrogazioni degli Esempi 3.34 e 3.36

L'operatore **UNION** corrisponde all'unione insiemistica e pertanto elimina i duplicati dal risultato. Esiste un operatore **UNION ALL** che corrisponde all'unione senza eliminazione dei duplicati. La motivazione per l'introduzione di tale operatore è che in molti casi l'utente sa che non ci saranno duplicati nel risultato; in tal caso la computazione effettuata per eliminare i duplicati causa un inutile peggioramento delle prestazioni.

Altri operatori insiemistici sono gli operatori **INTERSECT** ed **EXCEPT** (od il suo sinonimo **MINUS**) che eseguono rispettivamente l'intersezione e la differenza di due insiemi di tuple. Per questi operatori valgono le stesse condizioni di applicabilità discusse per l'operatore **UNION**.

**Esempio 3.36** Consideriamo le seguenti interrogazioni.

**Q1** Determinare gli anni in cui sono usciti sia film di Tim Burton sia film di Quentin Tarantino:

```
SELECT anno FROM Film WHERE regista = 'tim burton'
INTERSECT
SELECT anno FROM Film WHERE regista = 'quentin tarantino';
```

**Q2** Determinare gli anni in cui sono usciti film di Tim Burton ma non film di Quentin Tarantino:

```
SELECT anno FROM Film WHERE regista = 'tim burton'
EXCEPT
SELECT anno FROM Film WHERE regista = 'quentin tarantino';
```

Il risultato di **Q1** è 1994, quello di **Q2** è illustrato nella Figura 3.13. □

Le operazioni di intersezione e differenza possono essere comunque eseguite in SQL tramite l'uso degli operatori **EXISTS** e **NOT EXISTS**, come illustrato dall'esempio seguente.

**Esempio 3.37** I seguenti comandi **SELECT** esprimono, rispettivamente, le interrogazioni **Q1** e **Q2** dell'Esempio 3.36:

---

```

SELECT DISTINCT anno FROM Film F
WHERE regista = 'tim burton'
    AND EXISTS (SELECT * FROM Film
                 WHERE regista = 'quentin tarantino'
                     AND anno = F.anno);

SELECT DISTINCT anno FROM Film F
WHERE regista = 'tim burton'
    AND NOT EXISTS (SELECT * FROM Film
                      WHERE regista = 'quentin tarantino'
                          AND anno = F.anno);
```

□

### 3.3 Operazioni di aggiornamento

Come evidenziato dalla Tabella 3.1, SQL fornisce tre comandi per l'esecuzione di inserimenti, modifiche e cancellazioni di tuple di relazioni.<sup>16</sup> Un aspetto importante dei comandi di aggiornamento è che permettono di determinare le tuple da modificare tramite ricerche in base al contenuto delle stesse, mantenendo quindi lo stesso potere espressivo del linguaggio di interrogazione anche nella ricerca delle tuple da modificare.

#### 3.3.1 Inserimento

Il comando `INSERT` permette l'inserimento di nuove tuple in una relazione data. Ha due formati di base, che differiscono rispetto alla modalità con cui i valori delle colonne delle tuple da inserire sono specificati. La prima modalità prevede che i valori siano specificati esplicitamente come una lista di valori nel comando; la seconda modalità prevede che i valori siano estratti dalla base di dati tramite una sotto-interrogazione. Il formato generale del comando `INSERT` è il seguente:

```
INSERT INTO R [(C1, ..., Cn)]

{VALUES (v1, ..., vn) | sq};
```

dove:

- $R$  è il nome della relazione su cui si esegue l'inserimento.
- $C_1, \dots, C_n$  è la lista delle colonne della nuova tupla, o delle nuove tuple, a cui si assegnano valori. Tutte le colonne non esplicitamente elencate ricevono il valore specificato come default nella definizione di  $R$  od il valore nullo, nel caso non sia stato specificato alcun valore di default. La mancata specifica di una lista di colonne è equivalente alla specifica di una lista in cui sono presenti tutte le colonne della relazione nell'ordine dato dal comando `CREATE TABLE`.

---

<sup>16</sup>SQL:2003 ha introdotto un quarto comando di aggiornamento, il comando `MERGE`, che permette di combinare operazioni di inserimento e di modifica in un unico comando.

## Capitolo 3

# Linguaggio SQL

Il linguaggio SQL, abbreviazione di *Structured Query Language*, è il linguaggio di definizione e manipolazione dei dati supportato dalla totalità dei DBMS relazionali oggi disponibili. SQL ha rappresentato una tappa fondamentale nello sviluppo della tecnologia delle basi di dati, in quanto è stato il primo linguaggio con caratteristiche dichiarative progettato specificatamente per l'accesso e la manipolazione di collezioni di dati. SQL è basato sui linguaggi presentati nel Capitolo 2 per l'interrogazione di basi di dati relazionali, cioè l'algebra ed il calcolo relazionale. Le operazioni dell'algebra relazionale, in particolare, sono fondamentali per comprendere i tipi di interrogazioni che possono essere poste ad una base di dati relazionale. Nell'algebra relazionale, tuttavia, un'interrogazione è formulata come una sequenza di operazioni che, una volta eseguite, producono il risultato richiesto. L'utente deve quindi specificare *in quale ordine* le operazioni devono essere eseguite. In SQL, al contrario, l'utente specifica solo *quale* deve essere il risultato, caratterizzandolo in maniera dichiarativa. Questo permette al DBMS di determinare strategie efficienti per l'esecuzione delle operazioni di accesso e manipolazione dei dati, mediante un processo detto di *ottimizzazione*, che verrà discusso nel Capitolo 7. SQL include inoltre alcune operazioni dell'algebra (in particolare, le operazioni insiemistiche di unione, differenza, intersezione) ma si basa in misura maggiore sul calcolo relazionale, anche se la sintassi di SQL è più semplice da utilizzare.

SQL differisce quindi dai linguaggi di programmazione, quali C, C++, Java, per il fatto di essere orientato alla manipolazione di collezioni di dati e di permettere all'utente di specificare quale deve essere il risultato di una data operazione, senza dover indicare come eseguire l'operazione stessa. La specifica del risultato di un'operazione è espressa tramite condizioni sul contenuto dei dati. La disponibilità di un linguaggio ad alto livello, dichiarativo, specializzato per la manipolazione dei dati, è molto importante dal punto di vista dello sviluppo delle applicazioni. SQL manca, però, dei costrutti di controllo propri dei linguaggi di programmazione di uso generale e da solo non è quindi sufficiente alla programmazione di gran parte delle applicazioni reali. Pertanto, oltre a poter essere usato in modalità stand-alone (tipicamente tramite un'interfaccia interattiva), SQL è “combinato” con linguaggi di programmazione secondo diversi approcci, che verranno discussi in dettaglio nel Capitolo 4.

Data la sua rapida diffusione come linguaggio per la gestione dei dati, SQL è stato oggetto di numerose estensioni e standardizzazioni. Il linguaggio è diventato standard ufficiale nel 1986 ed ha subito significative revisioni nel 1992 (SQL2, noto anche come SQL-92) e nel 1999 (SQL:1999, noto anche come SQL3). SQL:1999 ha introdotto numerose estensioni ad SQL tra cui i trigger (non standardizzati in SQL2) ed alcuni costrutti del paradigma ad oggetti, tanto che il modello dei dati corrispondente non è più un modello relazionale puro, ma un modello relazionale ad oggetti. Tali caratteristiche verranno discusse in dettaglio nei Capitoli 10 e 11. Lo standard più recente è lo standard SQL:2003. Nel seguito della trattazione, faremo implicitamente riferimento a tale versione dello standard. Indicheremo esplicitamente la versione solo quando vorremo evidenziare una differenza significativa rispetto ad altre versioni.

Uno dei motivi del successo sul mercato delle basi di dati relazionali è sicuramente legato all'esistenza del linguaggio standard SQL. Avere a disposizione un linguaggio standard, infatti, permette agli utenti di non doversi preoccupare troppo della migrazione delle loro applicazioni da un sistema relazionale ad un altro, nel caso non siano soddisfatti del particolare prodotto. Tale conversione non dovrebbe essere troppo costosa nel caso in cui entrambi i sistemi siano conformi allo standard. Analogamente, possiamo scrivere programmi applicativi che contengono comandi SQL, come verrà discusso nel Capitolo 4, che accedono ai dati memorizzati in due o più DBMS relazionali utilizzando gli stessi comandi di manipolazione dei dati. In realtà, del linguaggio SQL esistono diverse implementazioni e vi sono differenze anche significative tra i DBMS relazionali commerciali. Tipicamente, le funzionalità di base del linguaggio sono le stesse in ogni implementazione; esistono, invece, differenze nelle funzionalità più avanzate. A partire da SQL:1999, lo standard SQL definisce un vasto insieme di caratteristiche (*SQL Core*) che devono essere fornite da un DBMS per potersi dichiarare conforme allo standard. Nella trattazione, presenteremo le principali funzionalità di tale nucleo del linguaggio, che dovrebbero quindi essere fornite in tutte le implementazioni, evidenziando esplicitamente eventuali caratteristiche che ancora non sono diffusamente supportate nelle implementazioni.

Il documento che specifica lo standard è strutturato in nove diverse parti. Tra queste, il materiale coperto in questo capitolo è contenuto in *SQL/Foundation*, che è la parte più estesa e più importante, che specifica il “nucleo” del linguaggio (SQL Core), corrispondente al minimo livello di conformità. Nel Capitolo 4 verrà invece trattata la parte *SQL/PSM (Persistent Stored Module)*, che specifica costrutti procedurali simili a quelli che si trovano nei comuni linguaggi di programmazione.

SQL comprende istruzioni per la definizione, l'interrogazione e l'aggiornamento dei dati. Quindi, rispetto alla terminologia introdotta nel Capitolo 1, è sia un DDL che un DML. Tutte le implementazioni di SQL forniscono inoltre comandi SDL per la definizione di strutture di memorizzazione e di strutture ausiliarie di accesso (vedi Capitolo 7), ma tali comandi non sono stati standardizzati in SQL.

In questo capitolo, introdurremo dapprima il linguaggio di definizione dei dati, in seguito presenteremo il linguaggio di interrogazione e le altre componenti del

Operazione	DDL	DML
Creazione	CREATE	INSERT
Modifica	ALTER	UPDATE
Cancellazione	DROP	DELETE

Tabella 3.1: Comandi SQL di aggiornamento a livello di schema e di istanza

linguaggio di manipolazione dei dati. La specifica di vincoli di integrità e viste completerà la trattazione.

Nel modello relazionale abbiamo usato i termini di “relazione” ed “attributo”; il linguaggio SQL differisce rispetto a tale terminologia in quanto usa i termini di “tabella” (table) e “colonna” (column). Nel seguito di questo capitolo continueremo ad usare il termine “relazione” come equivalente del termine “tabella” ed useremo il termine “colonna” come equivalente del termine “attributo”. Nel presentare la sintassi del linguaggio, come usuale, verranno utilizzate la parentesi quadra per racchiudere componenti la cui specifica non è obbligatoria e parentesi graffe per racchiudere componenti alternative, di cui una deve essere obbligatoriamente specificata, separate dal simbolo '|'. Un asterisco ('\*') indicherà la ripetibilità di una componente.

### 3.1 Linguaggio di definizione dei dati

Le relazioni possono essere definite tramite il comando **CREATE**, modificate tramite il comando **ALTER** e cancellate tramite il comando **DROP**. Tali comandi costituiscono i comandi principali del DDL. La Tabella 3.1 riassume i principali comandi offerti da SQL, a livello di schema e di istanza, rispettivamente, per aggiornare lo schema ed i dati. In questo paragrafo discuteremo i comandi del DDL, mentre i comandi di aggiornamento del DML saranno discussi nel Paragrafo 3.3, dopo aver presentato il linguaggio di interrogazione.

Nel seguito del paragrafo, introdurremo innanzitutto i tipi di dato forniti da SQL, che corrispondono ai domini del modello relazionale, specificando quindi i possibili contenuti delle colonne delle relazioni. Presenteremo poi i comandi per la creazione e successivamente quelli per la cancellazione e la modifica di relazioni.

#### 3.1.1 Tipi di dato

In questo paragrafo discuteremo i tipi di dato *predefiniti* di SQL. SQL prevede la possibilità di utilizzare anche tipi *definiti dall'utente*, che verranno trattati nell'ambito delle caratteristiche relazionali ad oggetti di SQL, discusse nel Capitolo 10. I tipi predefiniti sono suddivisi in tre categorie principali: *tipi numerici*, *tipi carattere* e *tipi temporali*. Ognuna di queste categorie include, a sua volta, delle sotto-categorie di seguito discusse.

**Tipi numerici.** I tipi numerici sono classificati in *tipi numerici esatti*, che permettono di rappresentare valori interi e valori decimali in virgola fissa, e *tipi numerici approssimati*, che permettono di rappresentare valori numerici approssimati con rappresentazione in virgola mobile, cioè tramite mantissa ed esponente. I tipi numerici esatti includono i seguenti tipi:

- **INTEGER.** Rappresenta i valori interi. La *precisione* (numero totale di cifre) di questo tipo di dato è espressa in numero di bit o cifre, a seconda della specifica implementazione di SQL.
- **SIMALLINT.** Anche questo tipo di dato rappresenta valori interi. I valori di questo tipo sono usati per eventuali ottimizzazioni in quanto richiedono minore spazio di memorizzazione. L'unico requisito è che la precisione di questo tipo di dato sia non maggiore della precisione del tipo di dato **INTEGER**.
- **BIGINT.** Anche questo tipo di dato rappresenta valori interi. L'unico requisito è che la precisione di questo tipo di dato sia non minore della precisione del tipo di dato **INTEGER**.<sup>1</sup>
- **NUMERIC.** Questo tipo è caratterizzato da una precisione (numero totale di cifre) ed una *scala* (numero di cifre dopo la virgola decimale). La specifica di questo tipo di dato ha la forma **NUMERIC[(p[,s])]**, dove *p* è la precisione che viene richiesta ed *s* è la scala. Il valore di default per la precisione è 1, mentre il valore di default per la scala è 0.
- **DECIMAL.** È simile al tipo di dato **NUMERIC**. Analogamente a quest'ultimo, la specifica di questo tipo di dato ha la forma **DECIMAL[(p[,s])]**, dove *p* è la precisione che viene richiesta ed *s* è la scala. La differenza tra **NUMERIC** e **DECIMAL** è che nel primo la precisione con cui i valori sono implementati deve essere esattamente la precisione richiesta dalla specifica del tipo di dato (cioè *p*), mentre nel secondo caso la precisione con cui i valori sono implementati deve essere almeno uguale alla precisione richiesta (ma può anche essere maggiore).

I tipi numerici approssimati includono i seguenti tipi:

- **REAL.** Rappresenta valori a singola precisione in virgola mobile. La precisione dipende dalla specifica implementazione di SQL.
- **DOUBLE PRECISION.** Anche questo tipo di dato rappresenta valori in virgola mobile, di solito a doppia precisione, e la sua precisione dipende dalla specifica implementazione di SQL. La differenza tra il tipo di dato **REAL** e il tipo di dato **DOUBLE PRECISION** è che la precisione di quest'ultimo deve essere maggiore della precisione del tipo di dato **REAL**.

---

<sup>1</sup>Anche se non specificato nello standard, usualmente 64 bit invece dei 32 bit di **INTEGER**.

- **FLOAT.** Questo tipo di dato, a differenza dei due precedenti, permette di richiedere la precisione desiderata. Pertanto, la specifica di questo tipo di dato ha la forma `FLOAT[(p)]`, dove  $p$  è la precisione che viene eventualmente richiesta. La precisione minima che può essere specificata è 1, mentre la precisione di default e quella massima dipendono dalle specifiche implementazioni di SQL.

I valori dei tipi numerici esatti sono rappresentati dalle cifre corrispondenti, con eventualmente prefisso il segno e l'utilizzo del punto decimale (ad esempio, `4.0`, `-6`). I valori dei tipi numerici approssimati sono rappresentati in notazione mantissa ed esponente (ad esempio, `1256E-4`).

**Tipi carattere.** I tipi carattere includono i seguenti tipi:

- **CHARACTER.** Questo tipo di dato (spesso abbreviato in `CHAR`) permette di definire stringhe di caratteri di lunghezza predefinita. La specifica di questo tipo di dato ha la forma `CHAR[(n)]` dove  $n$  è la lunghezza delle stringhe. È ovviamente possibile utilizzare una stringa di lunghezza inferiore ad  $n$  come valore di tipo `CHAR(n)`, ma la stringa sarà completata con degli spazi fino a raggiungere la lunghezza  $n$ . Se non viene specificata alcuna lunghezza, il default è 1.
- **CHARACTER VARYING.** Questo tipo di dato (spesso abbreviato in `VARCHAR`) permette di definire stringhe di caratteri di una lunghezza massima predefinita. La specifica di questo tipo ha la forma `VARCHAR(n)` dove  $n$  è la lunghezza massima delle stringhe. Notiamo che in questo caso la specifica della lunghezza massima è obbligatoria e può variare tra 1 ed un valore massimo che dipende dalla specifica implementazione di SQL. La differenza tra il tipo di dato `CHAR` e il tipo di dato `VARCHAR` è che nel primo per ogni stringa viene comunque allocato uno spazio la cui lunghezza in numero di caratteri è sempre pari alla lunghezza specificata  $n$ . Nel secondo tipo di dato si adottano invece strategie di memorizzazione delle stringhe differenti, che evitano sprechi di spazio.

I valori dei tipi carattere vanno racchiusi tra singoli apici, come in '*pulp fiction*'. In tali valori viene effettuata una distinzione tra caratteri maiuscoli e minuscoli.

**Tipi temporali.** I tipi temporali includono i seguenti tipi:

- **DATE.** Rappresenta le date espresse come anno (4 cifre), mese (2 cifre comprese tra 1 e 12), giorno (2 cifre comprese tra 1 e 31 ed ulteriori restrizioni a seconda del mese). Un valore di questo tipo viene rappresentato dalla parola chiave `DATE` seguita dalla stringa che rappresenta la data nel formato scelto. Sono possibili diversi formati di rappresentazione, in questo testo utilizzeremo il formato giorno - prime tre lettere del mese - anno. Un esempio di data in questo formato è `DATE '08-Ott-1969'`.

- **TIME.** Rappresenta i tempi espressi come ora (2 cifre comprese tra 0 e 23), minuto (2 cifre comprese tra 0 e 59) e secondo (2 cifre comprese tra 0 e 61).<sup>2</sup> La specifica ha la forma **TIME**[(*p*)], dove *p* è l'eventuale numero di cifre frazionarie cui si è interessati, fino al microsecondo (6 cifre). Se non viene specificato *p*, il valore di default è 0. Un valore di questo tipo viene rappresentato dalla parola chiave **TIME** seguita dalla stringa che rappresenta il tempo, ad esempio **TIME** '21:56:32.5'.
- **TIMESTAMP.** Rappresenta una “concatenazione” dei due tipi di dato precedenti. Pertanto permette di rappresentare timestamp che consistono di: anno, mese, giorno, ora, minuto, secondo ed eventualmente microsecondo (6 cifre). Come per il tipo **TIME**, la specifica è **TIMESTAMP**[(*p*)], dove *p* è l'eventuale numero di cifre frazionarie cui si è interessati. Un valore di questo tipo viene rappresentato dalla parola chiave **TIMESTAMP** seguita dalla stringa che rappresenta data e tempo, nei formati introdotti precedentemente, ad esempio **TIMESTAMP** '08-Ott-1969 21:56:32.5'.
- **INTERVAL.** Rappresenta una durata temporale in riferimento ad uno o più dei qualificatori tra **YEAR**, **MONTH**, **DAY**, **HOUR**, **MINUTE** e **SECOND**. I valori di questo tipo sono rappresentati dalla parola chiave **INTERVAL** seguita da una stringa che caratterizza la durata in termini di uno o due qualificatori. Se sono presenti due qualificatori, il primo è più ampio del secondo ed i due sono separati dalla parola chiave **TO**. In realtà, a seconda dei qualificatori utilizzati, distinguiamo tra intervalli *year-month* ed intervalli *day-time*. Ad esempio, **INTERVAL** '3' **YEAR** rappresenta un intervallo di durata 3 anni, **INTERVAL** '3-11' **YEAR TO MONTH** rappresenta un intervallo di durata 3 anni ed 11 mesi, mentre **INTERVAL** '36 22:30' **DAY TO MINUTE** rappresenta un intervallo di durata 36 giorni, 22 ore e 30 minuti.

I tipi predefiniti includono inoltre il *tipo booleano* **BOOLEAN**, i cui valori sono **TRUE**, **FALSE** ed **UNKNOWN**,<sup>3</sup> ed i tipi **BLOB** (Binary Large Object) e **CLOB** (Character Large Object), per la memorizzazione di stringhe di bit o di caratteri, rispettivamente, di dimensione elevata. Tali tipi sono utili, ad esempio, nella memorizzazione di immagini e testi.

Un’ultima importante questione riguarda la compatibilità tra i vari tipi di dato. Spesso, ad esempio, può essere necessario confrontare valori di colonne diverse e ovviamente i confronti possono essere eseguiti solo se le colonne hanno tipi di dato *compatibili*. Normalmente le specifiche compatibilità dipendono dal sistema e sono contenute nei manuali d’uso, insieme ad eventuali regole di conversione. In generale si può dire che tutti i tipi di dato numerici sono compatibili e tutti i tipi di dato carattere sono compatibili. Inoltre, il tipo di dato **DATE** è compatibile con i tipi di dato carattere e lo stesso vale per i tipi di dato **TIME** e **TIMESTAMP**.

---

<sup>2</sup>I secondi possono arrivare fino a 61 e non fino a 59 come ci aspetteremmo per il fenomeno noto come *leap second*: occasionalmente vengono aggiunti uno o due secondi ad un minuto per tenere il tempo “ufficiale” sincronizzato con il tempo siderale.

<sup>3</sup>Il valore **UNKNOWN** verrà discusso in dettaglio nel Paragrafo 3.2.6.

### 3.1.2 Creazione di relazioni

La creazione di relazioni avviene in SQL tramite l'uso del comando `CREATE TABLE`, che permette di specificare lo schema della relazione ed alcuni importanti vincoli di integrità, come discusso nei paragrafi seguenti.

#### 3.1.2.1 Specifica dello schema di una relazione

La sintassi base del comando `CREATE TABLE`, che permette di creare una relazione specificandone lo schema, è la seguente:<sup>4</sup>

```
CREATE TABLE <nome relazione>
    (<specifica colonna> [,<specifica colonna>]*);
```

dove:

- `<nome relazione>` è il nome della relazione che viene creata;
- `<specifica colonna>` è una specifica di colonna, il cui formato è il seguente:

```
<nome colonna> <dominio> [DEFAULT <valore default>]
```

dove:

- `<nome colonna>` è il nome della colonna (che deve essere distinto da tutti gli altri nomi di colonna della stessa relazione);
- `<dominio>` è il dominio della colonna e deve essere uno dei tipi di dato di SQL;
- la clausola `DEFAULT` specifica un valore di default per la colonna. Tale valore è quello assegnato ad una nuova tupla se nessun valore è specificato per la colonna al momento dell'inserimento della tupla. Il valore di default è un qualunque valore appartenente al dominio.

**Esempio 3.1** La relazione `Video` della Figura 2.1 può essere definita in SQL come segue:

```
CREATE TABLE Video
    (colloc  DECIMAL(4),
     titolo  VARCHAR(30),
     regista VARCHAR(20),
     tipo     CHAR DEFAULT 'd');
```

□

Ad ogni relazione è associato un tipo, chiamato *tipo riga* (*row type*), che corrisponde al tipo delle tuple di tale relazione.

---

<sup>4</sup>In questo testo, per convenzione, useremo i caratteri maiuscoli per i comandi SQL. In realtà nelle parole riservate SQL è indifferente l'uso di caratteri maiuscoli o minuscoli; il linguaggio distingue tra caratteri maiuscoli e minuscoli solo nei valori di tipo stringa.

### 3.1.2.2 Vincoli di obbligatorietà, chiavi e chiavi esterne

Un aspetto importante nella definizione di uno schema di basi di dati riguarda la definizione dei vincoli di integrità. SQL prevede la specifica di vincoli di integrità, come parte del comando `CREATE TABLE`, e la verifica automatica di tali vincoli. Oltre alla lista delle specifiche di colonne, quindi, un comando di definizione di relazione spesso contiene anche la definizione di importanti vincoli, discussi nel seguito di questo paragrafo: obbligatorietà di colonne e specifica di chiavi e chiavi esterne. È anche possibile specificare vincoli di integrità aggiuntivi, relativi alle tuple della relazione od alle singole colonne di tali tuple, al momento della definizione di una relazione, ma, poiché tale specifica richiede l'utilizzo del linguaggio di interrogazione, rimandiamo la trattazione di questo aspetto al Paragrafo 3.4.

Un'ulteriore possibilità fornita da SQL è quella di definire in una relazione colonne derivate, il cui valore è cioè ottenuto a partire dai valori di altre colonne, e di derivare interamente il contenuto di una relazione a partire da altri dati attraverso un'interrogazione. I meccanismi di derivazione dei dati verranno discussi nel Paragrafo 3.5.

**Obbligatorietà di colonne.** Per la specifica dell'obbligatorietà di una colonna è sufficiente aggiungere `NOT NULL` alla specifica della colonna. In tal modo impomiamo che ogni tupla della relazione abbia necessariamente un valore diverso dal valore nullo per la colonna.

**Chiavi.** La specifica delle chiavi si effettua in SQL mediante le parole chiave `UNIQUE` e `PRIMARY KEY`. Come discusso nel Capitolo 2, una relazione può avere una chiave primaria (`PRIMARY KEY`) ed una o più chiavi alternative (chiavi `UNIQUE`). La parola chiave `UNIQUE` garantisce che non esistano due tuple che condividono gli stessi valori non nulli per le colonne (le colonne `UNIQUE` possono contenere valori nulli). La parola chiave `PRIMARY KEY` impone invece che per ogni tupla i valori delle colonne specificati siano non nulli e diversi da quelli di ogni altra tupla. Le colonne specificate come `PRIMARY KEY` non possono quindi assumere valori nulli per alcuna tupla della relazione. In una tabella è possibile specificare più chiavi `UNIQUE` ma una sola `PRIMARY KEY`. Per specificare una chiave composta da una sola colonna è sufficiente far seguire la specifica della colonna da `UNIQUE` o `PRIMARY KEY`, alternativamente si può far seguire la definizione della tabella dalla clausola `PRIMARY KEY (<lista nomi colonne>)` o `UNIQUE(<lista nomi colonne>)`. Questa è la sola sintassi utilizzabile nel caso di chiavi composte da più colonne.

**Esempio 3.2** Le relazioni `Video` e `Noleggio` della base di dati della videoteca possono essere definite come segue, includendo la specifica di chiavi e vincoli di obbligatorietà. Anticipiamo che, come verrà discusso nel Paragrafo 3.2.2, la funzione `CURRENT_DATE` restituisce la data corrente.

```
CREATE TABLE Video
  (colloc      DECIMAL(4) PRIMARY KEY,
```

```

titolo      VARCHAR(30) NOT NULL,
regista    VARCHAR(20) NOT NULL,
tipo        CHAR NOT NULL DEFAULT 'd');

CREATE TABLE Noleggio
(colloc      DECIMAL(4),
dataNol     DATE DEFAULT CURRENT_DATE,
codCli      DECIMAL(4) NOT NULL,
dataRest    DATE,
PRIMARY KEY (colloc,dataNol),
UNIQUE (colloc,dataRest));

```

Poiché `dataRest` può assumere valori nulli ed i valori nulli non sono distinti tra di loro (vedi Paragrafo 3.2.6), il vincolo `UNIQUE` non impedisce in realtà che esistano due noleggi correnti per lo stesso video. Per evitare questa situazione dovrà essere definita un'apposita asserzione (vedi Paragrafo 3.4).  $\square$

È importante evidenziare che una relazione SQL si differenzia dalla relazione del modello relazionale così come definita nel Capitolo 2 in quanto una relazione SQL può contenere tuple duplicate. Per questo motivo può avere senso, in SQL, definire chiavi costituite da tutte le colonne di una relazione.

**Chiavi esterne.** La specifica di chiavi esterne avviene mediante la clausola `FOREIGN KEY` del comando `CREATE TABLE`. Tale clausola è opzionale e ripetibile ed ha la seguente sintassi:<sup>5</sup>

```

FOREIGN KEY (<lista nomi colonne>)
  REFERENCES <nome relazione>
  [ON DELETE {NO ACTION | CASCADE | SET NULL | SET DEFAULT}]
  [ON UPDATE {NO ACTION | CASCADE | SET NULL | SET DEFAULT}]

```

La clausola specifica:

- Una lista di una o più colonne che costituiscono una chiave esterna della relazione  $R$  (relazione *referente*) che stiamo definendo.
- La relazione *riferita*  $R'$  (specificata dalla clausola `REFERENCES`) di cui le suddette colonne sono chiave primaria.<sup>6</sup>
- L'azione da eseguire (specificata nella clausola `ON DELETE`) se una tupla della relazione riferita è cancellata ed esistono tuple in  $R$  che fanno riferimento a tale tupla. Le opzioni possibili sono:

<sup>5</sup>In realtà la clausola prevede anche una clausola opzionale `MATCH` per la specifica del *tipo di match* tra i valori degli attributi chiave e quelli degli attributi chiave esterna. Tale clausola è significativa nel caso di chiavi esterne costituite da più di un attributo ed in presenza di valori nulli e non viene considerata nella trattazione.

<sup>6</sup>In realtà SQL prevede la possibilità che anche chiavi `UNIQUE` possano essere riferite. Non consideriamo tale possibilità nel presente capitolo per non complicare la trattazione.

- **NO ACTION.** La cancellazione di una tupla dalla relazione riferita è eseguita solo se non esiste alcuna tupla in  $R$  che fa riferimento a tale tupla.
- **CASCADE.** La cancellazione della tupla della relazione riferita implica la cancellazione di tutte le tuple di  $R$  che fanno riferimento a tale tupla.
- **SET NULL.** La cancellazione della tupla della relazione riferita implica che, in tutte le tuple di  $R$  che fanno riferimento a tale tupla, il valore della chiave esterna viene posto uguale al valore nullo.
- **SET DEFAULT.** La cancellazione della tupla della relazione riferita implica che, in tutte le tuple di  $R$  che fanno riferimento a tale tupla, il valore della chiave esterna viene posto uguale al valore di default specificato per le colonne che costituiscono la chiave esterna.

L'opzione di default è **NO ACTION**.

- L'azione da eseguire (specificata nella clausola **ON UPDATE**) se la chiave primaria di una tupla della relazione riferita viene modificata ed esistono tuple in  $R$  che fanno riferimento a tale tupla. Le opzioni possibili sono le stesse ed hanno lo stesso significato di quelle che abbiamo visto per la gestione della cancellazione. L'unica differenza riguarda l'opzione **CASCADE**, la quale ha l'effetto di assegnare, come valori della chiave esterna delle tuple che fanno riferimento alla tupla la cui chiave primaria è modificata, il nuovo valore della chiave primaria di quest'ultima. L'opzione di default è, anche in questo caso, **NO ACTION**.

È importante notare che la possibilità di dichiarare chiavi esterne e di poter, inoltre, specificare le azioni che il sistema deve eseguire se una tupla appartenente ad una relazione riferita è cancellata o la sua chiave modificata, permette una gestione semplice ed efficace dell'integrità referenziale. Si noti, infine, che l'integrità referenziale potrebbe essere compromessa anche da inserimenti e modifiche del valore di chiave esterna di tuple della relazione referente. In questo caso, se la modifica porta ad una violazione dell'integrità referenziale viene semplicemente rifiutata dal sistema e non è possibile specificare azioni di riparazione alternative.

Una relazione può avere più chiavi esterne; la sua definizione contiene in tal caso tante clausole **FOREIGN KEY** quante sono le chiavi esterne. Nel caso di chiave esterna costituita da un solo attributo si può far seguire la specifica della colonna da **REFERENCES** e dalla specifica delle informazioni relative alla chiave esterna.

**Esempio 3.3** La definizione delle relazioni **Film**, **Video**, **Cliente** e **Noleggio** della base di dati della videoteca, con la specifica delle chiavi esterne, è la seguente.

```
CREATE TABLE Film
(titolo  VARCHAR(30),
regista VARCHAR(20),
anno    DECIMAL(4) NOT NULL,
```

```

genere  CHAR(15) NOT NULL,
valutaz NUMERIC(3,2),
PRIMARY KEY (titolo,regista));

CREATE TABLE Cliente
(codCli    DECIMAL(4) PRIMARY KEY,
nome      VARCHAR(20) NOT NULL,
cognome   VARCHAR(20) NOT NULL,
telefono  CHAR(15) NOT NULL,
dataN     DATE NOT NULL,
residenza VARCHAR(30) NOT NULL,
UNIQUE (nome,cognome,dataN));

CREATE TABLE Video
(colloc      DECIMAL(4) PRIMARY KEY,
titolo      VARCHAR(30) NOT NULL,
regista    VARCHAR(20) NOT NULL,
tipo        CHAR NOT NULL DEFAULT 'd',
FOREIGN KEY (titolo,regista) REFERENCES Film
                                ON DELETE RESTRICT);

CREATE TABLE Noleggio
(colloc    DECIMAL(4) REFERENCES Video
                        ON DELETE CASCADE ON UPDATE CASCADE,
dataNol   DATE DEFAULT CURRENT_DATE,
codCli    DECIMAL(4) NOT NULL REFERENCES Cliente
                        ON DELETE CASCADE ON UPDATE CASCADE,
dataRest  DATE,
PRIMARY KEY (colloc,dataNol),
UNIQUE (colloc,dataRest));

```

Per quanto riguarda le violazioni dell'integrità referenziale:

- se viene cancellato un cliente vengono cancellati anche tutti i noleggi da lui effettuati;
- se viene cancellato un video vengono cancellati anche tutti i noleggi di tale video;
- non viene permessa la cancellazione di informazioni relative a film se vi sono video contenenti quel film;
- le modifiche di codice cliente e collocazione video vengono propagate ai relativi noleggi. □

### 3.1.3 Cancellazione e modifica di relazioni

La cancellazione di una relazione è eseguita tramite il comando:

```
DROP TABLE <nome relazione> {RESTRICT | CASCADE};
```

dove **<nome relazione>** è il nome della relazione da cancellare. La cancellazione di una relazione comporta la cancellazione di tutte le tuple in essa contenute. Nel richiedere la cancellazione di una relazione, è necessario specificare una tra le opzioni **RESTRICT** e **CASCADE**. Se viene specificata l'opzione **RESTRICT** la relazione viene cancellata solo se non è riferita da altri elementi dello schema della base di dati. Al contrario, se viene specificata l'opzione **CASCADE** la relazione e tutti gli elementi dello schema che la riferiscono vengono cancellati.

**Esempio 3.4** Consideriamo lo schema della base di dati della videoteca definito nell'Esempio 3.3. Poiché la relazione **Film** è riferita dalla relazione **Video**, che è a sua volta riferita dalla relazione **Noleggio**, il comando:

```
DROP TABLE Film RESTRICT;
```

non altera lo schema della base di dati, mentre il comando:

```
DROP TABLE Film CASCADE;
```

causa la cancellazione delle tre relazioni. □

Uno schema di relazione può essere modificato mediante il comando **ALTER TABLE**, la cui sintassi è la seguente:

```
ALTER TABLE <nome relazione> <modifica>;
```

dove **<nome relazione>** è il nome della relazione da modificare e **<modifica>** è la specifica della modifica da effettuare. Le modifiche possibili sono:<sup>7</sup>

- Aggiunta di una nuova colonna, specificata come:

```
ADD [COLUMN] <specifica colonna>
```

dove **<specifica colonna>** è una specifica di colonna definita in accordo al formato discusso per il comando **CREATE TABLE**.

- Modifica di una colonna, specificata come:

```
ALTER [COLUMN] {SET DEFAULT <valore default> | DROP DEFAULT}
```

in cui **<valore default>** è il nuovo valore di default per la colonna. Tale modifica permette di aggiungere o modificare il valore di default per la colonna (mediante **SET DEFAULT**) oppure di rimuovere il valore di default (mediante **DROP DEFAULT**).

---

<sup>7</sup>Come vedremo nel Paragrafo 3.4, in realtà tra le modifiche possibili vi sono anche **ADD CONSTRAINT** e **DROP CONSTRAINT** per l'aggiunta e la rimozione di un vincolo, rispettivamente.

- Eliminazione di una colonna, specificata come:

```
DROP [COLUMN] <nome colonna> {RESTRICT | CASCADE}
```

dove **<nome colonna>** è il nome della colonna da eliminare e **RESTRICT** e **CASCADE** hanno il significato discusso per il comando **DROP TABLE**.

**Esempio 3.5** Consideriamo nuovamente lo schema della base di dati della videoteca definito nell’Esempio 3.3. Il comando:

```
ALTER TABLE Film ADD COLUMN studio VARCHAR(20);
```

ha l’effetto di aggiungere come sesta ed ultima colonna della relazione **Film** la colonna **studio**. Poiché non viene specificato un valore di default, a tutte le tuple di **Film** viene assegnato il valore **NULL** per tale colonna. Il comando:

```
ALTER TABLE Video ALTER COLUMN tipo SET DEFAULT 'v';
```

ha l’effetto di modificare il valore di default per la colonna **tipo** di **Video**, ponendolo uguale al valore ‘**v**’. □

### 3.2 Interrogazioni

Questo paragrafo descrive vari aspetti relativi alla specifica di interrogazioni in SQL. Per prima cosa introdurremo il formato di base di un’interrogazione SQL ed i principali operatori e funzioni utilizzabili nelle interrogazioni. Verranno poi discusse ulteriori caratteristiche del linguaggio di interrogazione, quali ordinamento, operazione di join, funzioni di gruppo, valori nulli e sotto-interrogazioni.

#### 3.2.1 Formato di base del comando SELECT

Le interrogazioni in SQL sono espresse tramite il comando **SELECT**. La forma base di un’interrogazione SQL ha la seguente struttura:<sup>8</sup>

```
SELECT {DISTINCT Ri1.C1, Ri2.C2, ..., Rin.Cn | *}  
FROM R1, R2, ..., Rk WHERE F;
```

dove:

- $R_i$  ( $i = 1, \dots, k$ ) è un nome di relazione. La clausola **FROM** specifica pertanto le relazioni oggetto dell’interrogazione.

---

<sup>8</sup>Come discuteremo nel seguito del paragrafo, la clausola **DISTINCT** del comando **SELECT** è in realtà opzionale.

---

```

SELECT DISTINCT anno FROM Film F
WHERE regista = 'tim burton'
    AND EXISTS (SELECT * FROM Film
                 WHERE regista = 'quentin tarantino'
                     AND anno = F.anno);

SELECT DISTINCT anno FROM Film F
WHERE regista = 'tim burton'
    AND NOT EXISTS (SELECT * FROM Film
                      WHERE regista = 'quentin tarantino'
                          AND anno = F.anno);
```

□

### 3.3 Operazioni di aggiornamento

Come evidenziato dalla Tabella 3.1, SQL fornisce tre comandi per l'esecuzione di inserimenti, modifiche e cancellazioni di tuple di relazioni.<sup>16</sup> Un aspetto importante dei comandi di aggiornamento è che permettono di determinare le tuple da modificare tramite ricerche in base al contenuto delle stesse, mantenendo quindi lo stesso potere espressivo del linguaggio di interrogazione anche nella ricerca delle tuple da modificare.

#### 3.3.1 Inserimento

Il comando `INSERT` permette l'inserimento di nuove tuple in una relazione data. Ha due formati di base, che differiscono rispetto alla modalità con cui i valori delle colonne delle tuple da inserire sono specificati. La prima modalità prevede che i valori siano specificati esplicitamente come una lista di valori nel comando; la seconda modalità prevede che i valori siano estratti dalla base di dati tramite una sotto-interrogazione. Il formato generale del comando `INSERT` è il seguente:

```
INSERT INTO R [(C1, ..., Cn)]

{VALUES (v1, ..., vn) | sq};
```

dove:

- $R$  è il nome della relazione su cui si esegue l'inserimento.
- $C_1, \dots, C_n$  è la lista delle colonne della nuova tupla, o delle nuove tuple, a cui si assegnano valori. Tutte le colonne non esplicitamente elencate ricevono il valore specificato come default nella definizione di  $R$  od il valore nullo, nel caso non sia stato specificato alcun valore di default. La mancata specifica di una lista di colonne è equivalente alla specifica di una lista in cui sono presenti tutte le colonne della relazione nell'ordine dato dal comando `CREATE TABLE`.

---

<sup>16</sup>SQL:2003 ha introdotto un quarto comando di aggiornamento, il comando `MERGE`, che permette di combinare operazioni di inserimento e di modifica in un unico comando.

- $v_1, \dots, v_n$  è la lista di valori da assegnare alla nuova tupla. Tale lista di valori costituisce l'argomento della clausola **VALUES** la cui presenza indica l'inserimento di una tupla i cui valori sono passati esplicitamente al comando. I valori vengono assegnati alle colonne in base ad una corrispondenza posizionale; pertanto  $e_i$  viene assegnato alla colonna  $C_i$ ,  $i = 1, \dots, n$ . Tale lista può anche contenere come valore la parola chiave **NULL**, per assegnare alla colonna corrispondente il valore nullo, e la parola chiave **DEFAULT**, per assegnare alla colonna corrispondente il valore di default.
- $sq$  è una sotto-interrogazione. La presenza di tale sotto-interrogazione, mutuamente esclusiva rispetto alla presenza della clausola **VALUES**, indica che i valori da assegnare alla tupla od alle tuple da inserire sono ottenuti dalla base di dati tramite un'interrogazione. La clausola di proiezione della sotto-interrogazione deve includere colonne (o più in generale espressioni) di tipo compatibile con le colonne della tupla o delle tuple da inserire. Il tipo della colonna  $C_i$  deve essere pertanto compatibile con il tipo della colonna (o espressione)  $i$ -esima presente nella clausola di proiezione della sotto-interrogazione,  $i = 1, \dots, n$ .

Quando usiamo una sotto-interrogazione all'interno di un comando **INSERT** viene inserito un numero di tuple uguale alla cardinalità del risultato della sotto-interrogazione (quindi possiamo inserire molteplici tuple); viceversa, se usiamo la clausola **VALUES** viene inserita una sola tupla.

**Esempio 3.38** Vogliamo inserire un nuovo film, il titolo è “La tigre e la neve”, il regista è Roberto Benigni, l’anno di produzione è 2005 e il genere è commedia. Non essendo specificata una valutazione, questa assume valore nullo:

```
INSERT INTO Film(titolo,regista,anno,genere)
VALUES ('la tigre e la neve','roberto benigni',2005,'commedia');
```

Vogliamo ora inserire il fatto che, in data odierna, il cliente 6635 noleggia tutti i video contenenti film di Gabriele Salvatores che non ha mai noleggiato. Tali informazioni devono essere estratte dalle relazioni **Video** e **Noleggio**.

```
INSERT INTO Noleggio(colloc,codCli)
SELECT colloc, 6635 FROM Video
WHERE regista = 'gabriele salvatores' AND (colloc,6635)
NOT IN (SELECT colloc, codCli FROM Noleggio);
```

Questo comando ha l’effetto di inserire nella relazione **Noleggio** le tuple  $(1124, CURRENT\_DATE, 6635, NULL)$  e  $(1126, CURRENT\_DATE, 6635, NULL)$ . Notiamo che non sono stati specificati valori per gli attributi **dataNol** e **dataRest**. Il primo assume il valore di default **CURRENT\_DATE** mentre il secondo, per cui non è stato specificato nello schema un valore di default, assume valore **NULL** (vedi Esempio 3.3).  $\square$

### 3.3.2 Cancellazione

Il comando **DELETE** permette la cancellazione di tuple da una data relazione. Le tuple da cancellare sono specificate tramite una condizione di ricerca; se non è specificata alcuna condizione di ricerca, vengono cancellate tutte le tuple presenti nella relazione oggetto del comando. Notiamo tuttavia che questo non implica che la relazione sia cancellata; la relazione rimane semplicemente vuota e nuove tuple vi possono essere inserite successivamente. Il comando **DELETE** ha il seguente formato:

```
DELETE FROM R [<alias>] [WHERE F];
```

dove:

- $R$  è il nome della relazione da cui si esegue la cancellazione. Il nome della relazione può essere seguito da un alias di relazione se è necessario riferire tale relazione in qualche sotto-interrogazione specificata come parte della clausola di qualificazione  $F$ .
- $F$  è la clausola di qualificazione che seleziona le tuple da cancellare.

**Esempio 3.39** Vogliamo cancellare il film “La tigre e la neve”, inserito con il primo comando **INSERT** dell’Esempio 3.38:

```
DELETE FROM Film
WHERE titolo = 'la tigre e la neve'
AND regista = 'roberto benigni';
```

Il comando seguente illustra l’uso delle sotto-interrogazioni nel comando **DELETE**. Vogliamo cancellare tutti i clienti che non hanno effettuato noleggi nell’ultimo anno (notiamo che, per come è stata definita la relazione **Noleggio**, ciò causerà anche la cancellazione di tutti i noleggi effettuati da tali clienti).

```
DELETE FROM Cliente
WHERE codCli NOT IN (SELECT codCli FROM Noleggio
                      WHERE dataNol > (CURRENT_DATE - 1 YEAR)); □
```

### 3.3.3 Modifica

Il comando **UPDATE** permette l’esecuzione di modifiche ad una o più colonne delle tuple di una relazione. Le tuple da modificare sono specificate tramite una condizione di ricerca; tale condizione di ricerca può includere tutti i predicati visti ed anche sotto-interrogazioni ed operazioni di join. Un altro aspetto interessante di questo comando è che le sotto-interrogazioni possono essere usate non solo per determinare le tuple da modificare ma anche per determinare i valori da assegnare alle colonne da modificare. Il comando **UPDATE** ha il seguente formato:

---

```
UPDATE R [alias]
SET C1 = {e1 | NULL}, ..., Cn = {en | NULL}
[WHERE F];
```

dove:

- $R$  è il nome della relazione su cui si esegue la modifica; il nome della relazione può essere seguito da un alias di relazione se è necessario riferire tale relazione in qualche sotto-interrogazione specificata come parte della clausola di qualificazione  $F$ .
- $C_i = \{e_i | \text{NULL}\}$ ,  $i = 1, \dots, n$ , è un'espressione di assegnamento che specifica che alla colonna  $C_i$  deve essere assegnato il valore denotato dall'espressione  $e_i$ , il cui tipo deve essere compatibile con il dominio di  $C_i$ . L'espressione  $e_i$  può essere un valore costante, oppure un'espressione, spesso funzione dei valori correnti delle tuple da modificare, o una sotto-interrogazione; alternativamente, tramite la parola chiave `NULL`, possiamo richiedere che alla colonna venga assegnato il valore nullo.
- $F$  è la clausola di qualificazione che seleziona le tuple a cui vanno apportate le modifiche.

**Esempio 3.40** Supponiamo di aver modificato la scala di valutazione dei film da 0-5 a 0-10; vogliamo ora aggiornare le singole valutazioni, moltiplicando per due le valutazioni correnti di tutti i film:

```
UPDATE Film SET valutaz = valutaz * 2;
```

Supponiamo che il cliente 6635 restituisca tutti i video che ha attualmente in noleggio e segnali che il noleggio è iniziato ieri (e non come erroneamente diversamente registrato):

```
UPDATE Noleggio
SET dataRest = CURRENT_DATE, dataNol = (CURRENT_DATE - 1 DAY)
WHERE codCli = 6635 AND dataRest IS NULL; □
```

L'esempio seguente illustra l'uso delle sotto-interrogazioni all'interno del comando `UPDATE`, rispettivamente all'interno della clausola di qualificazione e della clausola di assegnamento (denotata dalla parola chiave `SET`).

**Esempio 3.41** Vogliamo aumentare del 10% la valutazione dei film horror usciti dopo il 1999 che sono stati noleggiati da almeno due clienti:

```
UPDATE Film
SET valutaz = valutaz * 1.1
WHERE genere = 'horror' AND anno > 1999 AND (titolo, regista)
IN (SELECT titolo, regista
    FROM Noleggio NATURAL JOIN Video
    GROUP BY titolo, regista
    HAVING COUNT(DISTINCT codCli) >= 2);
```

Vogliamo ora assegnare ad ogni film che è stato noleggiato almeno una volta nell'ultimo mese, una valutazione pari al 110% della valutazione media dei film dello stesso anno e genere:

```
UPDATE Film X
SET valutaz = (SELECT 1.1 * AVG(valutaz)
                 FROM Film
                 WHERE anno = X.anno AND genere = X.genere)
WHERE EXISTS (SELECT * FROM Noleggia NATURAL JOIN Video
                  WHERE dataNol > CURRENT_DATE - 1 MONTH
                  AND titolo = X.titolo AND regista = X.regista); □
```

Le modifiche in SQL sono eseguite in modo *set-oriented*: la clausola di qualificazione ed il valore dell'espressione di assegnamento vengono valutati un'unica volta, poi gli aggiornamenti vengono effettuati su tutte le tuple qualificate "contemporaneamente". L'effetto del comando precedente, quindi, sarà di assegnare la stessa valutazione a tutti i film dello stesso anno e genere (noleggiati almeno una volta nell'ultimo mese).

### 3.4 Vincoli di integrità

Come abbiamo visto nel Capitolo 1, un aspetto molto importante nella gestione di una base di dati riguarda la correttezza semantica dei dati, cioè il fatto che i dati rappresentino in modo corretto una determinata realtà applicativa. SQL permette la specifica di alcuni *vincoli di integrità semantica*, dove con vincolo intendiamo una proprietà che un insieme di dati deve verificare. È possibile innanzitutto distinguere tra vincoli *statici* e vincoli *di transizione*. Un vincolo statico riguarda uno stato della base di dati (ad esempio: "la valutazione relativa ad un film deve essere compresa tra 0 e 5") mentre un vincolo di transizione mette in relazione stati diversi della base di dati (ed esempio: "non è possibile modificare la data di restituzione di un video assegnandogli una data precedente a quella memorizzata"). In questo paragrafo consideremo solo vincoli statici; i vincoli di transizione possono essere modellati mediante l'uso di trigger (vedi Capitolo 11).

I vincoli possono essere ulteriormente classificati a seconda degli oggetti cui si riferiscono. Una possibile (parziale) classificazione è la seguente:

- **Vincoli su singola relazione.** Sono i vincoli che riguardano una singola relazione; a loro volta possono essere classificati in:
  - **Vincoli su singola tupla.** Vincoli di questo tipo coinvolgono gli attributi di una singola tupla e sono suddivisi in:
    - \* *Vincoli su singolo attributo.* Vincoli di questo tipo coinvolgono un singolo attributo. Un esempio di vincolo di questo tipo è il vincolo NOT NULL già introdotto.

Vogliamo ora assegnare ad ogni film che è stato noleggiato almeno una volta nell'ultimo mese, una valutazione pari al 110% della valutazione media dei film dello stesso anno e genere:

```
UPDATE Film X
SET valutaz = (SELECT 1.1 * AVG(valutaz)
                 FROM Film
                 WHERE anno = X.anno AND genere = X.genere)
WHERE EXISTS (SELECT * FROM Noleggia NATURAL JOIN Video
                  WHERE dataNol > CURRENT_DATE - 1 MONTH
                  AND titolo = X.titolo AND regista = X.regista); □
```

Le modifiche in SQL sono eseguite in modo *set-oriented*: la clausola di qualificazione ed il valore dell'espressione di assegnamento vengono valutati un'unica volta, poi gli aggiornamenti vengono effettuati su tutte le tuple qualificate "contemporaneamente". L'effetto del comando precedente, quindi, sarà di assegnare la stessa valutazione a tutti i film dello stesso anno e genere (noleggiati almeno una volta nell'ultimo mese).

### 3.4 Vincoli di integrità

Come abbiamo visto nel Capitolo 1, un aspetto molto importante nella gestione di una base di dati riguarda la correttezza semantica dei dati, cioè il fatto che i dati rappresentino in modo corretto una determinata realtà applicativa. SQL permette la specifica di alcuni *vincoli di integrità semantica*, dove con vincolo intendiamo una proprietà che un insieme di dati deve verificare. È possibile innanzitutto distinguere tra vincoli *statici* e vincoli *di transizione*. Un vincolo statico riguarda uno stato della base di dati (ad esempio: "la valutazione relativa ad un film deve essere compresa tra 0 e 5") mentre un vincolo di transizione mette in relazione stati diversi della base di dati (ed esempio: "non è possibile modificare la data di restituzione di un video assegnandogli una data precedente a quella memorizzata"). In questo paragrafo consideremo solo vincoli statici; i vincoli di transizione possono essere modellati mediante l'uso di trigger (vedi Capitolo 11).

I vincoli possono essere ulteriormente classificati a seconda degli oggetti cui si riferiscono. Una possibile (parziale) classificazione è la seguente:

- **Vincoli su singola relazione.** Sono i vincoli che riguardano una singola relazione; a loro volta possono essere classificati in:
  - **Vincoli su singola tupla.** Vincoli di questo tipo coinvolgono gli attributi di una singola tupla e sono suddivisi in:
    - \* *Vincoli su singolo attributo.* Vincoli di questo tipo coinvolgono un singolo attributo. Un esempio di vincolo di questo tipo è il vincolo NOT NULL già introdotto.

Tipo di vincolo	Esempio	Rappresentazione in SQL
Su singolo attributo	La valutazione di un film deve essere compresa tra 0 e 5	Vincolo CHECK su colonna
Su attributi multipli	La data di restituzione non deve essere precedente alla data di noleggio	Vincolo CHECK su relazione
Di aggregazione	Un cliente non può noleggiare più di tre video contemporaneamente	Asserzione
Su relazioni multiple	Un video non può essere noleggiato prima che il film contenuto nel video sia uscito	Asserzione

Tabella 3.2: Rappresentazione in SQL per alcuni tipi di vincoli di integrità

- \* *Vincoli su attributi multipli.* Vincoli di questo tipo coinvolgono attributi diversi della stessa tupla.
- **Vincoli su tuple multiple di una stessa relazione.** Sono vincoli che coinvolgono tuple diverse della stessa relazione ed includono, tra gli altri:
  - \* *Dipendenze funzionali.* Permettono di modellare correlazioni tra i valori di determinati attributi in tuple diverse e di esprimere che un attributo od un insieme di attributi siano chiave di una relazione; sono trattati in dettaglio nel Capitolo 6.
  - \* *Vincoli di aggregazione.* Sono vincoli che richiedono l'uso di funzioni di aggregazione, quali somma o media. Impongono che una determinata funzione aggregata calcolata su un insieme di tuple verifichi una data relazione di confronto rispetto ad un valore dato.
- **Vincoli su relazioni multiple.** Sono vincoli che coinvolgono tuple di relazioni diverse. Anche questo tipo di vincoli ha una classificazione articolata, a seconda, ad esempio, del fatto che si considerino funzioni aggregate. Un importante tipo di vincolo in questa classe è l'integrità referenziale, discussa in precedenza.

La Tabella 3.2, per alcuni dei tipi di vincoli fin qui discussi, presenta la rappresentazione in SQL ed alcuni esempi dal dominio applicativo della videoteca.

In SQL, oltre ai vincoli predefiniti di chiave (UNIQUE e PRIMARY KEY), obbligatorietà di colonne (NOT NULL) e chiavi esterne (FOREIGN KEY) discussi nel Paragrafo 3.1.2.2, è possibile, nel comando CREATE TABLE, definire *vincoli CHECK* arbitrari *su colonna* o *su relazione*, che corrispondono a vincoli su singola tupla della classificazione precedente. Poiché, come discusso, molte condizioni di integrità semantica dei dati non sono relative a singole tuple di una relazione, SQL fornisce anche la nozione di *asserzione* che permette di specificare condizioni che coinvolgono più tuple e/o relazioni. È da notare, tuttavia, che non tutti i DBMS prevedono tutti i tipi di vincoli. In particolare, le asserzioni non sono generalmente supportate. Quasi tutti i DBMS si limitano a gestire i vincoli che possono essere verificati

esaminando una sola tupla alla volta, poiché di tali vincoli è possibile garantire una valutazione efficiente.

Nel seguito del paragrafo discuteremo innanzitutto la specifica di vincoli **CHECK** ed asserzioni. Preciseremo poi quando avviene la verifica di vincoli di integrità espressi tramite tali costrutti e che effetti comporta la loro violazione.

### 3.4.1 *Vincoli CHECK*

Per definire un vincolo **CHECK** su colonna, alla specifica della colonna viene affiancata la parola chiave **CHECK** (**<condizione>**). Per definire un vincolo **CHECK** su una relazione, alla definizione della relazione viene aggiunta la parola chiave **CHECK** (**<condizione>**). La condizione è, in entrambi i casi, un predicato oppure una combinazione booleana di prediciati. Qualsiasi condizione possa comparire nella clausola di qualificazione di un'interrogazione SQL può essere utilizzata nella definizione di un vincolo,<sup>17</sup> quindi anche sotto-interrogazioni che fanno riferimento ad altre relazioni. In un comando **CREATE TABLE**, quindi, la clausola **CHECK** può comparire di seguito alla definizione di una colonna (vincoli **CHECK** su colonna) oppure come clausola separata all'interno della definizione della relazione (vincoli **CHECK** su relazione), come illustrato dal seguente esempio.

**Esempio 3.42** Vogliamo imporre che la valutazione di un film sia compresa tra 0 e 5. La specifica della colonna **valutaz** della relazione **Film** viene modificata come segue:

```
CREATE TABLE Film ( ...
    valutaz DECIMAL(3,2) CHECK (valutaz BETWEEN 0.00 AND 5.00),
    ...);
```

Vogliamo imporre che il tipo di un video sia **d** (per dvd) o **v** (per videocassetta). La specifica della colonna **tipo** della relazione **Video** viene modificata come segue:

```
CREATE TABLE Video ( ...
    tipo CHAR NOT NULL DEFAULT 'd' CHECK (tipo IN ('d','v')),
    ...);
```

Vogliamo imporre che un noleggio non possa terminare prima di essere iniziato. La specifica della relazione **Noleggio** viene modificata come segue:

```
CREATE TABLE Noleggio
(
    ...
    CHECK (dataRest >= dataNol));
```

□

Specificando un vincolo **CHECK** su relazione richiediamo che *ogni tupla* nella relazione soddisfi la condizione associata al vincolo. Una conseguenza importante

---

<sup>17</sup>L'unica restrizione è che i vincoli non possono contenere condizioni la cui valutazione può dare risultati differenti, sullo stesso stato della base di dati, a seconda del momento in cui è valutata, quali riferimenti al tempo di sistema, mediante ad esempio la funzione **CURRENT\_TIME**.

di questa semantica è che la relazione vuota soddisfa sempre tutti i vincoli CHECK. Per questo motivo, non è possibile utilizzare vincoli CHECK per richiedere che una relazione contenga almeno un certo numero di tuple che soddisfano una condizione.

Più in generale, anche se è possibile, mediante l'uso di sotto-interrogazioni, esprimere vincoli CHECK che verificano condizioni arbitrarie, è consigliabile, sia per migliorare la comprensibilità dello schema che l'efficienza nella verifica dei vincoli, esprimere tramite vincoli CHECK solo le condizioni che devono essere verificate da ogni singola tupla della relazione cui associamo il vincolo. Condizioni che richiedano di esaminare più tuple della relazione o tuple di relazioni diverse andrebbero invece espresse tramite asserzioni.

Una possibilità prevista da SQL per tutti i vincoli (anche quelli predefiniti) associati alle definizioni di relazioni è quella di assegnare un nome al vincolo premettendo alla specifica del vincolo la clausola `CONSTRAINT <nome vincolo>`. Assegnare un nome ad un vincolo è particolarmente utile per potervisi riferire in seguito, ad esempio per poterlo modificare od eliminare.<sup>18</sup> Il nome del vincolo deve essere unico rispetto ad altri nomi di vincoli definiti per la stessa relazione. L'aggiunta di un vincolo ad una relazione è possibile solo se tutte le istanze correnti della relazione soddisfano il vincolo, altrimenti l'aggiunta viene rifiutata.

**Esempio 3.43** La definizione della relazione `Video` in cui vengono assegnati i nomi ai vincoli in essa definiti diventa:

```
CREATE TABLE Video
  (colloc DECIMAL(4) CONSTRAINT PKey PRIMARY KEY,
   titolo VARCHAR(30) CONSTRAINT Tnn NOT NULL,
   regista VARCHAR(20) CONSTRAINT Rnn NOT NULL,
   tipo    CHAR CONSTRAINT Snn NOT NULL DEFAULT 'd'
           CONSTRAINT Tok CHECK (tipo IN ('d','v'))
   CONSTRAINT FK FOREIGN KEY (titolo,regista) REFERENCES Film
                           ON DELETE RESTRICT);
```

È a questo punto possibile, ad esempio, modificare il vincolo `Tok` e rimuovere il vincolo `Rnn` mediante i seguenti comandi del DDL:

```
ALTER TABLE Video DROP CONSTRAINT Tok;
ALTER TABLE Video ADD CONSTRAINT Tok
  CHECK (tipo IN ('d','v','x'));
ALTER TABLE Video DROP CONSTRAINT Rnn;
```

□

### 3.4.2 Asserzioni

SQL prevede il meccanismo delle *asserzioni* per specificare vincoli su più tuple o relazioni. Ogni vincolo su singola tupla è, come abbiamo visto, esprimibile

---

<sup>18</sup>Il comando `ALTER TABLE` visto nel Paragrafo 3.3 prevede apposite clausole `ADD CONSTRAINT` e `DROP CONSTRAINT` per l'aggiunta e l'eliminazione di vincoli a/da una relazione.

come vincolo CHECK su colonna o come vincolo CHECK su relazione. Può però ovviamente essere espresso anche tramite un'asserzione. Condizioni relative al numero minimo di tuple (che soddisfano una certa condizione) contenute in una tabella possono invece essere espresse solo tramite asserzioni. Più in generale, è opportuno esprimere tramite asserzioni tutti i vincoli su tuple multiple di una stessa relazione ed i vincoli su relazioni multiple.

Le asserzioni sono elementi dello schema e come tali vengono manipolate da appositi comandi del DDL. In particolare, il comando per la definizione di un'asserzione ha la sintassi:

```
CREATE ASSERTION <nome asserzione>
    CHECK (<condizione>);
```

dove <nome asserzione> è il nome assegnato all'asserzione e <condizione> è, come nel caso dei vincoli CHECK, qualsiasi condizione possa comparire nella clausola di qualificazione di un'interrogazione SQL.

**Esempio 3.44** La seguente asserzione garantisce che uno stesso video non possa essere noleggiato da due clienti diversi contemporaneamente:

```
CREATE ASSERTION SoloUno CHECK (NOT EXISTS
    (SELECT * FROM Noleggio
        WHERE dataRest IS NULL
        GROUP BY colloc
        HAVING COUNT(*) > 1));
```

La seguente asserzione garantisce che un cliente non possa avere più di tre video in noleggio contemporaneamente:

```
CREATE ASSERTION Max3 CHECK (NOT EXISTS
    (SELECT * FROM Noleggio
        WHERE dataRest IS NULL
        GROUP BY codCli
        HAVING COUNT(*) > 3));
```

La seguente asserzione garantisce, invece, che un video non possa essere noleggiato prima dell'uscita del film che lo contiene:

```
CREATE ASSERTION DateOk CHECK (NOT EXISTS
    (SELECT * FROM Noleggio NATURAL JOIN Video NATURAL JOIN Film
        WHERE YEAR(dataNol) > anno));
```

Tali asserzioni controllano condizioni che coinvolgono più tuple. Nei primi due casi l'asserzione coinvolge più tuple della stessa relazione, nel terzo tuple di relazioni diverse. Benché sia sintatticamente possibile specificare tali vincoli come vincoli CHECK con sotto-interrogazioni, associati ad esempio alla relazione *Noleggio*, è concettualmente più corretto, ed, in generale, porta ad una verifica più efficiente, definire tali vincoli di integrità come asserzioni. □

Un'asserzione può essere rimossa mediante il comando `DROP ASSERTION <nome asserzione>`, che causa la cancellazione dell'asserzione <nome asserzione>.

### 3.4.3 Controllo di vincoli di integrità

Ad ogni vincolo di integrità SQL, definito come vincolo `CHECK` oppure asserzione,<sup>19</sup> è associata una *modalità di controllo* che specifica quando deve esserne effettuata la verifica. Come vedremo meglio nel Capitolo 8, dedicato alla gestione delle transazioni, SQL consente di specificare sia vincoli d'integrità con *valutazione immediata*, cioè valutati dopo ogni comando di manipolazione dei dati, sia vincoli la cui *valutazione* è *differita* al termine dell'esecuzione di una sequenza di operazioni di manipolazione dei dati, che costituiscono una *transazione*.

Quando i vincoli sono immediatamente verificati, una violazione del vincolo causa la non esecuzione del comando che ne ha causato la violazione (eventuali modifiche parziali vengono annullate). Tutti i vincoli predefiniti, quali quelli di obbligatorietà, di chiave e di chiave esterna discussi nel Paragrafo 3.1.2.2, sono di default verificati in modo immediato. Nel caso in cui i vincoli siano valutati alla fine della transazione, invece, la violazione del vincolo comporta l'abort della transazione (vedi Capitolo 8). Tutte le operazioni nella transazione vengono cioè annullate, poiché non vi è modo di stabilire quale operazione ha causato la violazione del vincolo.

Nella definizione di un vincolo è possibile specificare le opzioni, mutuamente esclusive, `DEFERRABLE` e `NOT DEFERRABLE`. Se non viene specificata alcuna opzione, il vincolo è, per default, di tipo `DEFERRABLE`. Un vincolo definito con l'opzione `NOT DEFERRABLE` sarà sempre valutato dopo ogni singola operazione SQL, viceversa la valutazione di un vincolo definito come `DEFERRABLE` potrà essere differita al termine dell'esecuzione della transazione. In particolare, un vincolo `DEFERRABLE` può essere specificato con le opzioni `INITIALLY IMMEDIATE` (che costituisce il default) o `INITIALLY DEFERRED`. Nel primo caso, la verifica avviene dopo ogni istruzione SQL; nel secondo, alla fine della transazione.

La modalità di controllo di un vincolo può essere cambiata dinamicamente da `DEFERRED` ad `IMMEDIATE` e viceversa, mediante il comando `SET CONSTRAINTS`, la cui sintassi è la seguente:

```
SET CONSTRAINTS {<lista nomi vincoli> | ALL}
    {DEFERRED | IMMEDIATE};
```

il cui effetto è modificare la modalità di controllo dei vincoli inclusi in `<lista nomi vincoli>` (o di tutti i vincoli `DEFERRABLE` se si usa l'opzione `ALL`) per la transazione all'interno della quale viene eseguito il comando.

Precisiamo infine che un vincolo di integrità è *violato* se la valutazione della condizione di controllo restituisce `FALSE` come valore booleano di verità; viceversa, un vincolo di integrità *non è violato* se la valutazione della condizione di controllo restituisce `TRUE` o `UNKNOWN` (a causa di valori nulli) come valore booleano di verità.

---

<sup>19</sup>La trattazione in questo paragrafo è relativa ad entrambi i tipi di vincoli, a cui ci riferiremo per semplicità come vincoli o vincoli di integrità.

**Esempio 3.45** Consideriamo il vincolo che un noleggio non possa terminare prima di essere iniziato, formulato nell’Esempio 3.42 come `CHECK (dataRest >= dataNol)` sulla relazione `Noleggio`. Per i noleggi in corso, `dataRest` è nullo, quindi la valutazione della condizione produce il valore `UNKNOWN`, che non causa la violazione del vincolo.  $\square$

Un’ultima osservazione da fare riguardo all’integrità semantica è che né le asserzioni né i vincoli `CHECK` permettono la specifica di azioni di *riparazione* del vincolo. Se un vincolo di integrità è violato, le modifiche che hanno causato la violazione vengono semplicemente annullate. Un’alternativa molto più utile in numerose circostanze è quella di eseguire ulteriori azioni che permettano di ripristinare la correttezza semantica dei dati. La specifica di tali azioni è possibile utilizzando i trigger (vedi Capitolo 11) per implementare i vincoli di integrità. Questo è uno dei motivi per cui i DBMS commerciali hanno optato per il supporto di trigger piuttosto che di vincoli generali ed asserzioni.

### 3.5 Dati derivati e viste

SQL fornisce diversi meccanismi per derivare dati da altri dati presenti nella base di dati. Il meccanismo più semplice è quello delle colonne derivate, mentre il più significativo, che permette di definire viste alternative della base di dati e di realizzare il livello esterno nella rappresentazione dei dati (vedi Capitolo 1), è quello delle viste.

#### 3.5.1 Colonne derivate

Le relazioni SQL possono contenere un numero arbitrario di colonne derivate (o *generate*), ognuna delle quali è associata ad un’espressione scalare nella definizione della relazione. I valori per la colonna derivata sono calcolati ed assegnati automaticamente ogni volta che una nuova tupla è inserita nella relazione e mantenuti aggiornati in seguito ad aggiornamenti a tuple della relazione. Nei comandi `INSERT` ed `UPDATE`, alle colonne derivate può essere assegnato solo il valore speciale `DEFAULT`. La specifica di una colonna derivata, nel comando `CREATE TABLE` o nella clausola `ADD COLUMN` del comando `ALTER TABLE`, è la seguente:

```
<nome colonna> [<dominio>] GENERATED ALWAYS AS <espressione>
```

dove tutte le colonne che compaiono nell’espressione scalare devono essere colonne della relazione in cui viene definita la colonna ed il dominio, se specificato, deve essere compatibile con il tipo dell’espressione.

**Esempio 3.46** Alla relazione `Noleggio` può essere aggiunta una colonna derivata `durata` che consente di mantenere esplicitamente la durata dei noleggi, mediante il seguente comando:

```
ALTER TABLE Noleggio ADD COLUMN  
durata GENERATED ALWAYS AS ((dataRest-dataNol) DAY);  $\square$ 
```

**Esempio 3.45** Consideriamo il vincolo che un noleggio non possa terminare prima di essere iniziato, formulato nell’Esempio 3.42 come `CHECK (dataRest >= dataNol)` sulla relazione `Noleggio`. Per i noleggi in corso, `dataRest` è nullo, quindi la valutazione della condizione produce il valore `UNKNOWN`, che non causa la violazione del vincolo.  $\square$

Un’ultima osservazione da fare riguardo all’integrità semantica è che né le asserzioni né i vincoli `CHECK` permettono la specifica di azioni di *riparazione* del vincolo. Se un vincolo di integrità è violato, le modifiche che hanno causato la violazione vengono semplicemente annullate. Un’alternativa molto più utile in numerose circostanze è quella di eseguire ulteriori azioni che permettano di ripristinare la correttezza semantica dei dati. La specifica di tali azioni è possibile utilizzando i trigger (vedi Capitolo 11) per implementare i vincoli di integrità. Questo è uno dei motivi per cui i DBMS commerciali hanno optato per il supporto di trigger piuttosto che di vincoli generali ed asserzioni.

### 3.5 Dati derivati e viste

SQL fornisce diversi meccanismi per derivare dati da altri dati presenti nella base di dati. Il meccanismo più semplice è quello delle colonne derivate, mentre il più significativo, che permette di definire viste alternative della base di dati e di realizzare il livello esterno nella rappresentazione dei dati (vedi Capitolo 1), è quello delle viste.

#### 3.5.1 Colonne derivate

Le relazioni SQL possono contenere un numero arbitrario di colonne derivate (o *generate*), ognuna delle quali è associata ad un’espressione scalare nella definizione della relazione. I valori per la colonna derivata sono calcolati ed assegnati automaticamente ogni volta che una nuova tupla è inserita nella relazione e mantenuti aggiornati in seguito ad aggiornamenti a tuple della relazione. Nei comandi `INSERT` ed `UPDATE`, alle colonne derivate può essere assegnato solo il valore speciale `DEFAULT`. La specifica di una colonna derivata, nel comando `CREATE TABLE` o nella clausola `ADD COLUMN` del comando `ALTER TABLE`, è la seguente:

```
<nome colonna> [<dominio>] GENERATED ALWAYS AS <espressione>
```

dove tutte le colonne che compaiono nell’espressione scalare devono essere colonne della relazione in cui viene definita la colonna ed il dominio, se specificato, deve essere compatibile con il tipo dell’espressione.

**Esempio 3.46** Alla relazione `Noleggio` può essere aggiunta una colonna derivata `durata` che consente di mantenere esplicitamente la durata dei noleggi, mediante il seguente comando:

```
ALTER TABLE Noleggio ADD COLUMN  
durata GENERATED ALWAYS AS ((dataRest-dataNol) DAY);  $\square$ 
```

### 3.5.2 Derivazione di relazioni

SQL prevede la possibilità di definire una nuova relazione basandosi su relazioni già definite. A livello di schema è possibile, mediante la clausola opzionale `LIKE` del comando `CREATE TABLE`, copiare la struttura completa (cioè i nomi delle colonne, i domini ed i vincoli di obbligatorietà) di una o più relazioni esistenti nella definizione di una nuova relazione.

**Esempio 3.47** Il seguente comando SQL definisce una nuova relazione `ClienteV` con lo stesso schema di `Cliente`, a cui viene aggiunta una colonna `bonus`, di tipo `NUMERIC(4,2)`:

```
CREATE TABLE ClienteV
  (LIKE Cliente,
   bonus NUMERIC(4,2));
```

La relazione `ClienteV` conterrà quindi, oltre alla colonna `bonus`, le colonne `codCli`, `nome`, `cognome`, `telefono`, `dataN` e `residenza`, con gli stessi domini e vincoli di obbligatorietà specificati in `Cliente`. □

SQL permette inoltre, mediante la clausola opzionale `AS` del comando `CREATE TABLE`, di creare una relazione con la stessa struttura di un'interrogazione. Questo permette di copiare solo una parte della struttura di una o più relazioni esistenti, ridenominandone eventualmente le colonne. Se viene specificata la clausola `WITH DATA`, la relazione così creata è popolata con le tuple risultato della valutazione dell'interrogazione, altrimenti viene creata una relazione vuota. A differenza di quanto avviene con il meccanismo delle viste discusso nel paragrafo successivo, però, l'interrogazione viene utilizzata solo per la creazione ed eventualmente per la popolazione iniziale della relazione, ma una volta creata la relazione è modificata direttamente ed il suo contenuto non è più legato all'interrogazione utilizzata per definirla.

**Esempio 3.48** Decidiamo di voler memorizzare in due relazioni separate i noleggi conclusi ed i noleggi in corso. Le due nuove relazioni `NoleggioA` e `NoleggioC` possono essere definite come segue:

```
CREATE TABLE NoleggioA WITH DATA AS
  SELECT colloc, dataNol, codCli FROM Noleggio
  WHERE dataRest IS NULL;

CREATE TABLE NoleggioC WITH DATA AS
  SELECT * FROM Noleggio
  WHERE dataRest IS NOT NULL;
```

Nella relazione `NoleggioA` non è stata inclusa la colonna `dataRest`, che avrebbe sempre avuto valore nullo. □

### 3.5.3 Viste

In SQL è possibile definire viste alternative degli stessi dati. Una *vista* (view) è una relazione virtuale attraverso cui è possibile “vedere” i dati memorizzati nelle relazioni “reali” (dette di base). Una vista non contiene tuple, ma può essere usata a quasi tutti gli effetti come una relazione di base. Una vista è definita tramite un’interrogazione su una o più relazioni di base o su altre viste ed è *materializzata* eseguendo l’interrogazione che la definisce. Il meccanismo delle viste è utile per semplificare l’accesso ai dati, assicurare l’indipendenza logica alle applicazioni (vedi Capitolo 1) e garantire la protezione dei dati (vedi Capitolo 9).

Il comando di definizione di una vista ha la seguente sintassi:

```
CREATE VIEW <nome vista> [(<lista nomi colonne>)]
AS <interrogazione>
[WITH [<LOCAL | CASCDED>] CHECK OPTION];
```

dove:

- <nome vista> è il nome della vista che viene creata. Tale nome deve essere unico rispetto a tutte le relazioni e viste create dallo stesso utente che definisce la vista.
- <interrogazione> è l’*interrogazione di definizione* della vista. In particolare, una vista ha un numero di colonne pari alle colonne (di base o virtuali) specificate nella clausola di proiezione di tale interrogazione.
- <lista nomi colonne> è una lista di nomi da assegnare alle colonne della vista. La specifica di tale lista non è obbligatoria, tranne nel caso in cui l’interrogazione contenga nella clausola di proiezione funzioni di gruppo o espressioni cui non è assegnato un nome con la clausola AS. Se non viene specificata alcuna lista di nomi, i nomi delle colonne della vista sono gli stessi delle colonne corrispondenti, presenti nella clausola di proiezione dell’interrogazione.
- La clausola WITH CHECK OPTION verrà discussa nel Paragrafo 3.5.3.3.

**Esempio 3.49** Vogliamo creare una vista che restituisca il codice cliente, la data di inizio noleggio e la collocazione dei video in noleggio da più di tre giorni:

```
CREATE VIEW Nol3gg AS
SELECT codCli, dataNol, colloc
FROM Noleggio
WHERE dataRest IS NULL AND
(CURRENT_DATE - dataNol) DAY > INTERVAL '3' DAY; □
```

Nella definizione di viste è possibile usare tutte le funzionalità del linguaggio di interrogazione; l’interrogazione di definizione di una vista può ad esempio contenere operazioni di join e fare uso di funzioni di gruppo ed espressioni.

**Esempio 3.50** Vogliamo creare una vista che restituisca il numero di telefono del cliente, la data di inizio noleggio, il titolo del film e l'importo dovuto per i video in noleggio da più di tre giorni (supponendo che la tariffa giornaliera di noleggio sia 5 Euro):

```
CREATE VIEW InfoNol3gg AS
    SELECT telefono, dataNol, titolo,
           5 * ((CURRENT_DATE - dataNol) DAY) AS importo
      FROM Cliente NATURAL JOIN Noleggio NATURAL JOIN Video
     WHERE dataRest IS NULL AND
           (CURRENT_DATE - dataNol) DAY > INTERVAL '3' DAY;
```

Vogliamo ora creare una vista che per ogni cliente visualizzi il nome, la residenza, l'anno di nascita, il numero di noleggi effettuati e la durata massima di tali noleggi:

```
CREATE VIEW InfoCli (nome, residenza, annoN, numNol, durataM)
AS SELECT nome, residenza, YEAR(dataN), COUNT(*),
          MAX((dataRest - dataNol) DAY)
     FROM Cliente NATURAL JOIN Noleggio
    GROUP BY codCli, nome, residenza, dataN;
```

Nella definizione della vista `InfoCli` è stata inclusa la lista con i nomi da assegnare alle colonne di tale vista, in quanto la vista contiene colonne che sono derivate tramite espressioni o funzioni dalle colonne della relazione di base, cui non è stato assegnato un nome nella clausola di proiezione, come è invece stato fatto per la colonna derivata `importo` della vista `InfoNol3gg`. Notiamo infine che nella definizione della vista `InfoCli` è stato necessario raggruppare anche per `nome`, `residenza`, `dataN` per poter restituire tali attributi nella clausola `SELECT`. L'aggiunta di tali attributi all'attributo `codCli` nella clausola `GROUP BY` non cambia i gruppi che vengono costruiti, perché ogni cliente ha un unico nome, un'unica residenza ed un'unica data di nascita. □

Una vista può essere cancellata tramite il comando:

```
DROP VIEW <nome vista> {RESTRICT | CASCADE};
```

dove `<nome vista>` è il nome della vista da cancellare e le opzioni `RESTRICT` e `CASCADE` hanno lo stesso significato discusso relativamente alla cancellazione di relazioni nel Paragrafo 3.1.3.

Sulle viste è possibile eseguire (con alcune importanti restrizioni) sia interrogazioni sia aggiornamenti, come discusso in quanto segue.

#### 3.5.3.1 Interrogazioni su viste

Una volta definita, una vista è parte dello schema e può essere utilizzata nelle interrogazioni come una relazione di base. È possibile ad esempio effettuare sia

proiezioni sia specificare condizioni di ricerca, come pure effettuare dei join con altre relazioni o viste, come illustrato dal seguente esempio.

**Esempio 3.51** Vogliamo determinare dalla vista `InfoCli`, definita nell’Esempio 3.50, le informazioni (nome e dati sui noleggi) relative ai clienti di Genova<sup>20</sup> nati tra il 1970 ed il 1976:

```
SELECT nome, NumNol, durataM FROM InfoCli
WHERE annoN BETWEEN 1970 AND 1976 AND residenza LIKE '%genova';
```

Vogliamo determinare il numero di telefono del cliente che è più in ritardo con la restituzione dei video (cioè che ha in prestito un video da più tempo). Tale interrogazione può essere formulata come segue, utilizzando la vista `InfoNol3gg`, definita nell’Esempio 3.50:

```
SELECT telefono FROM InfoNol3gg
WHERE dataNol <= ALL (SELECT dataNol FROM InfoNol3gg);
```

o anche come segue, utilizzando la vista `Nol3gg` definita nell’Esempio 3.49:

```
SELECT telefono FROM Nol3gg NATURAL JOIN Cliente
WHERE dataNol <= ALL (SELECT dataNol FROM Nol3gg);
```

Vogliamo infine determinare il numero di telefono del cliente che ha il maggior debito con la videoteca, relativamente ai video in prestito da più di tre giorni:

```
SELECT telefono FROM InfoNol3gg
GROUP BY telefono
HAVING SUM(importo) >= ALL (SELECT SUM(importo) FROM InfoNol3gg
                                GROUP BY telefono);
```

In tale interrogazione sono stati aggregati i debiti di clienti diversi che abbiano dato lo stesso recapito telefonico alla videoteca, coerentemente con l’intuizione che tali clienti fanno probabilmente parte dello stesso nucleo familiare. □

È inoltre possibile definire una vista su un’altra vista. In questo caso, l’interrogazione di definizione della vista contiene delle viste nella clausola `FROM`.

**Esempio 3.52** Una definizione alternativa della vista `InfoNol3gg`, basata sulla vista `Nol3gg`, è la seguente:

```
CREATE VIEW InfoNol3ggB AS
SELECT telefono, dataNol, titolo,
       5 * ((CURRENT_DATE - dataNol) DAY) AS importo
  FROM Nol3gg NATURAL JOIN Cliente NATURAL JOIN Video;
```

---

<sup>20</sup>Notiamo che, poiché la residenza contiene le informazioni relative a via, numero civico e cap, oltre alla città, la condizione è `residenza LIKE '%genova'` e non `residenza = 'genova'`.

Una delle restrizioni da tener presente nella formulazione di interrogazioni su viste è che non è ammesso l'uso di funzioni di gruppo su colonne di viste che sono a loro volta definite tramite funzioni di gruppo. In questo senso, l'uso delle viste non è completamente trasparente. Non è ad esempio possibile calcolare sulla vista `InfoCli` dell'Esempio 3.50 la media o la somma della colonna `durataM`. Il motivo di questa restrizione è che la valutazione di un'interrogazione su una vista può essere effettuata *componendo* l'interrogazione formulata dall'utente con l'interrogazione di definizione della vista. Poiché in SQL non è possibile applicare funzioni di gruppo in cascata, tale composizione non sarebbe possibile se si ammettesse l'uso di funzioni di gruppo su colonne di viste che sono a loro volta definite tramite funzioni di gruppo.

### 3.5.3.2 Aggiornamenti su viste

Un ulteriore aspetto da tenere in considerazione riguarda le operazioni di aggiornamento eseguite attraverso le viste. L'esecuzione di un'operazione di aggiornamento su una vista viene propagata alla relazione su cui la vista è definita. In alcuni casi, però, non è possibile realizzare l'operazione richiesta attraverso un'operazione sulle relazioni di base. Ad esempio, non è possibile realizzare un inserimento attraverso una vista se la vista non contiene una colonna della relazione di base su cui è specificato un vincolo `NOT NULL` e per il quale non è specificato nello schema un valore di default. Siccome la vista non contiene la colonna, l'operazione non specificherà un valore per tale colonna. L'inserimento sulla relazione di base, tuttavia, essendo la colonna obbligatoria e senza valore di default, verrà rifiutato in mancanza di tale valore. In altri casi, inoltre, non è possibile stabilire in modo univoco un aggiornamento sulle relazioni di base il cui effetto sulla vista sia l'operazione richiesta. Per questo motivo, non essendo in questi casi possibile interpretare in maniera univoca la richiesta dell'utente, l'aggiornamento non viene permesso. Ad esempio, la cancellazione di una tupla da una vista definita come join di più relazioni di base può essere ottenuta mediante cancellazione da una o più delle relazioni di base o ponendo a `NULL` il valore dell'attributo di join in una o più relazioni.

In SQL, quindi, non tutti i tipi di operazioni di aggiornamento sono possibili su ogni vista. Data una vista  $V$  con interrogazione di definizione  $Q$ , le operazioni di aggiornamento possibili su  $V$  sono stabilite in accordo alle seguenti condizioni:

1. È possibile eseguire il comando `DELETE` su  $V$  se  $Q$ :
  - è un'interrogazione su una singola relazione  $R$ ;
  - non contiene la clausola `GROUP BY`, l'opzione `DISTINCT`, operatori insiemistici, né alcuna funzione di gruppo;
  - eventuali sotto-interrogazioni presenti nella clausola di selezione non fanno riferimento, né mediante correlazione né elencandola esplicitamente nella clausola `FROM`, alla relazione  $R$ .
2. È possibile eseguire il comando `UPDATE` su una colonna  $C$  di  $V$  se  $Q$ :

- soddisfa tutte le condizioni di cui al punto precedente;
  - $C$  non è definita tramite un'espressione o funzione.
3. È possibile eseguire il comando `INSERT` su  $V$  se  $Q$ :
- soddisfa tutte le condizioni di cui ai punti precedenti;
  - qualsiasi colonna per cui valga il vincolo `NOT NULL` è inclusa nella vista.

Le restrizioni precedenti assicurano che le operazioni di aggiornamento attraverso la vista possano essere realizzate in maniera univoca mediante aggiornamenti sulla relazione di base. In realtà, a partire da SQL:1999, vengono ammessi anche aggiornamenti su viste la cui interrogazione di definizione contiene più di una relazione, purché sia possibile stabile una corrispondenza uno a uno tra le tuple della vista e le tuple di una relazione di base. I DBMS commerciali, tuttavia, adottano a questo proposito strategie più restrittive di quelle previste dallo standard e si limitano comunque ad ammettere, al più, aggiornamenti su viste contententi un'unica relazione nell'interrogazione di definizione.

**Esempio 3.53** In riferimento alle viste definite negli Esempi 3.49 e 3.50, su `No13gg` possono essere effettuati sia inserimenti sia cancellazioni e modifiche, mentre su `InfoNo13gg` e `InfoCli` non può essere effettuata alcuna operazione di aggiornamento.  $\square$

#### 3.5.3.3 Check option

Una vista può contenere una condizione sul contenuto delle tuple appartenenti alle relazioni su cui la vista è definita; solo le tuple che verificano tale condizione appartengono alla vista. Un problema riguarda gli inserimenti nella vista di tuple che non verificano la condizione specificata nell'interrogazione di definizione della vista stessa.

**Esempio 3.54** Supponiamo di voler inserire nella vista `No13gg` dell'Esempio 3.50 la tupla `(1128, CURRENT_DATE, 6635)`. Tale tupla non verifica la condizione di ricerca specificata nell'interrogazione di definizione di `No13gg`. La tupla viene pertanto inserita ma non è ritrovata da alcuna interrogazione sulla vista.  $\square$

Per assicurare quindi che le tuple inserite tramite una vista (o modificate tramite una vista) siano accettate solo se verificano la condizione dell'interrogazione di definizione della vista, si deve specificare la clausola `CHECK OPTION` nel comando di definizione della vista. Pertanto, se la vista `No13gg` è definita con `CHECK OPTION`, l'inserimento della tupla `(1128, CURRENT_DATE, 6635)` non viene eseguita.

La situazione si complica ulteriormente nel caso di viste definite in termini di altre viste, perché ognuna di tali viste potrebbe o meno essere definita con `CHECK OPTION`. Per tale motivo, la `CHECK OPTION` può essere specificata con due possibili alternative: `LOCAL` e `CASCDED`. La differenza tra le due alternative è rilevante solo nei casi in cui una vista sia definita in termini di un'altra vista. La clausola `WITH`

**LOCAL CHECK OPTION** significa che solo la clausola **WHERE** della vista in cui compare l'opzione viene verificata. Con **CASCDED CHECK OPTION**, invece, le clausole **WHERE** della vista in cui compare l'opzione e di tutte le viste su cui si basa eventualmente la sua definizione vengono verificate, indipendentemente dal fatto che queste siano definite con **CHECK OPTION** o meno. Se non viene specificato né **LOCAL** né **CASCDED** viene comunque tenuto quest'ultimo comportamento, perché **CASCDED** è l'opzione di default.

### Note conclusive

Il linguaggio SQL è un linguaggio estremamente potente e ricco di funzionalità specifiche per le applicazioni di gestione dati. In questo capitolo abbiamo presentato le principali funzionalità del linguaggio ma la trattazione è inevitabilmente incompleta. Alcune caratteristiche, quali **SEQUENCE**, utili per generare codici progressivi, e *domini*, cioè tipi cui è possibile assegnare un nome ed associare valori di default e vincoli, non sono state considerate per limitazioni di spazio. Alcune funzionalità del linguaggio di interrogazione non sono state considerate. In particolare, non abbiamo considerato la possibilità offerta da SQL di utilizzare sotto-interrogazioni nella clausola **FROM**. L'uso delle sotto-interrogazioni nella clausola **FROM**, infatti, sebbene sia previsto da SQL, dovrebbe a nostro parere essere evitato perché produce interrogazioni difficili da capire e da verificare. Alcune utili funzionalità del linguaggio di interrogazione, quali i predicati quantificati **FOR ANY** e **FOR SOME** e le funzioni su insiemi **EVERY** e **ANY**, non sono state introdotte per mancanza di supporto nei DBMS commerciali. Analogamente non abbiamo presentato il comando di aggiornamento **MERGE**.

Nella trattazione, abbiamo cercato di limitarci a quelle funzionalità supportate in modo conforme allo standard nelle varie implementazioni. Purtroppo, però, anche relativamente alle funzionalità di base discusse in questo capitolo, vi sono alcuni aspetti relativamente ai quali le implementazioni si discostano dallo standard. In tal caso è bene fare riferimento alla documentazione dello specifico DBMS che si intende utilizzare. Esempi di tali differenze sono relative al supporto per i tipi temporali e relative funzioni, al meccanismo di cast, alle modifiche di schema in generale (comandi del DDL), alle operazioni insiemistiche, alle sotto-interrogazioni che restituiscono più di una colonna, a vincoli ed asserzioni, alle operazioni sulle viste.

Le più significative funzionalità relative al modello dei dati relazionale ad oggetti, quali i tipi definiti dagli utenti, verranno presentate nel Capitolo 10. Le funzionalità relative alla specifica di comportamento reattivo mediante trigger verranno invece trattate nel Capitolo 11. Alcune funzionalità relative alle strutture di memorizzazione dei dati, alle transazioni ed al controllo dell'accesso verranno introdotte, rispettivamente, nei Capitoli 7, 8 e 9. Non verranno invece trattate nel testo le pur significative funzionalità legate alla possibilità di formulare interrogazioni ricorsive e le caratteristiche relative alla materializzazione delle viste e funzionalità di analisi OLAP, discusse brevemente nelle note conclusive alla fine del testo.

## Capitolo 7

# Memorizzazione dei dati ed elaborazione delle interrogazioni

Finora abbiamo considerato modelli dei dati ad alto livello, cioè a livello *logico*. Tale livello è quello corretto per gli utenti della base di dati. Tuttavia, un fattore importante nell'accettazione di un DBMS da parte dell'utente è dato dalle sue prestazioni. Le prestazioni del DBMS dipendono dall'efficienza delle strutture dati e dall'efficienza del sistema nell'operare su tali strutture. Nei capitoli precedenti abbiamo introdotto le nozioni di tupla e relazione e i linguaggi per la loro manipolazione. A livello fisico, tali tuple saranno memorizzate in record di file su memoria secondaria ed una manipolazione efficiente verrà garantita dall'uso di opportune tecniche di elaborazione delle interrogazioni. In particolare, per velocizzare la ricerca dei dati vengono in genere utilizzate particolari strutture di accesso, dette *indici*, che consentono di accedere direttamente ai record corrispondenti alle tuple con un certo valore per un attributo, senza scandire l'intero contenuto del file. La scelta delle strutture di memorizzazione e di indicizzazione più efficienti dipende dal tipo di accessi che si eseguono sui dati. Normalmente, ogni DBMS ha le proprie strategie di implementazione di un modello dei dati; tuttavia, l'utente può influenzare le scelte fatte dal sistema. Le scelte dell'utente a questo riguardo costituiscono la *progettazione fisica* della base di dati e si concretizzano mediante l'utilizzo di opportuni comandi forniti dai DBMS. Una volta determinate le strutture di memorizzazione dei dati ed eventuali strutture ausiliarie di accesso, è compito del DBMS determinare, per ogni operazione di manipolazione dei dati, la strategia più efficiente per eseguirla, date le strutture disponibili. Benché tale processo riguardi tutte le operazioni di manipolazione dei dati, gran parte del processo di ottimizzazione è relativo alle operazioni di interrogazione, poiché il costo delle operazioni è dominato dal costo per il ritrovamento delle tuple, che, essendo specificato mediante condizioni dichiarative, offre notevoli margini di ottimizzazione. Per tale motivo ci concentreremo principalmente sull'elaborazione e l'ottimizzazione delle interrogazioni.

In questo capitolo, dopo aver brevemente introdotto l'architettura generale di un sistema di gestione dati, discuteremo le tecniche utilizzate per la memorizzazione fisica e l'indicizzazione dei dati. Illustreremo poi le principali problematiche nell'elaborazione di interrogazioni e descriveremo brevemente l'attività di progettazione fisica.

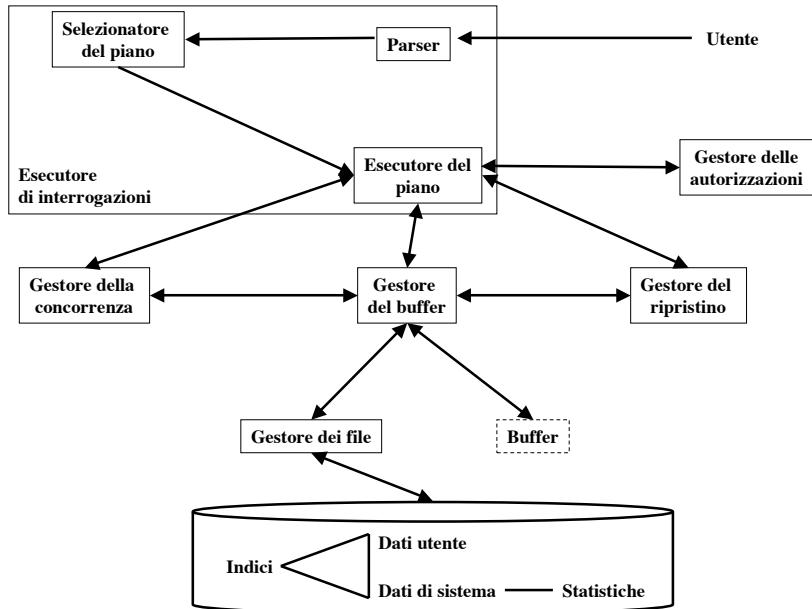


Figura 7.1: Architettura di un DBMS

### 7.1 Architettura di un DBMS

Le basi di dati memorizzano in modo persistente grosse quantità di dati. La maggior parte delle basi di dati sono memorizzate in maniera permanente su supporti di memorizzazione secondaria (cui ci riferiremo anche come memoria di massa), tipicamente su dischi magnetici. I dati in memoria secondaria non possono essere elaborati direttamente dalla CPU, ma devono essere prima copiati in memoria principale, in un'opportuna area di *buffer*.

Un DBMS deve garantire una gestione efficiente, concorrente, affidabile e sicura dei dati, preservandone inoltre l'integrità. Ciascuno degli aspetti precedenti è supportato nel DBMS da specifiche componenti, o sottosistemi, che complessivamente rappresentano l'architettura del sistema, illustrata graficamente nella Figura 7.1. Nella Figura 7.1, le componenti funzionali sono indicate da rettangoli, mentre il disco (memoria secondaria) è indicato con un cilindro e le aree in memoria principale con un rettangolo tratteggiato. Un DBMS è quindi costituito da diverse componenti funzionali che includono:

- **gestore dei file**, che gestisce l'allocazione dello spazio su disco e le strutture dati usate per rappresentare le informazioni memorizzate su disco;

- **gestore del buffer**, responsabile del trasferimento delle informazioni tra disco e memoria principale;
- **esecutore di interrogazioni**, responsabile dell'esecuzione delle richieste utente e costituito da:
  - **parser**, che traduce i comandi del DDL e del DML in un formato interno (*parse tree*);
  - **selezionatore del piano**, che stabilisce il modo più efficiente di processare una richiesta utente;
  - **esecutore del piano**, che processa le richieste utente in accordo al piano di esecuzione selezionato;
- **gestore delle autorizzazioni**, che controlla che gli utenti abbiano gli opportuni diritti di accesso ai dati;
- **gestore del ripristino**, che assicura che la base di dati rimanga in uno stato consistente a fronte di cadute o malfunzionamenti del sistema;
- **gestore della concorrenza**, che assicura che le esecuzioni concorrenti di processi procedano senza conflitti.

Come evidenziato nella Figura 7.1, il DBMS memorizza, oltre ai dati utente, anche dati di sistema (quali informazioni sullo schema della base di dati e sulle autorizzazioni), strutture ausiliarie di accesso a tali dati (indici) e dati statistici (quali il numero di tuple in una relazione), utilizzati dal selezionatore del piano per determinare la migliore strategia di esecuzione. Tutte le diverse componenti accedono a dati utente e/o di sistema nello svolgimento delle loro funzioni (ad esempio, il selezionatore del piano accede ai dati statistici ed il gestore delle autorizzazioni accede alle autorizzazioni), interagendo quindi con il gestore del buffer, anche se, per semplicità, nella Figura 7.1 alcune frecce sono state omesse.

In questo capitolo, introdurremo innanzitutto brevemente la gestione dei dati di sistema. Ci concentreremo poi sulle problematiche inerenti alla memorizzazione dei dati e l'ottimizzazione di interrogazioni, che concorrono alla realizzazione di una gestione efficiente dei dati, introducendo, quindi, il gestore dei file, il gestore del buffer e l'esecutore di interrogazioni. Il gestore delle autorizzazioni, responsabile dell'integrità e della riservatezza dei dati, ha come componente principale un meccanismo di controllo dell'accesso quali quelli che verranno discussi nel Capitolo 9. I gestori di concorrenza e ripristino, garanti, rispettivamente, dell'accesso concorrente ai dati e della loro affidabilità, saranno discussi nel Capitolo 8.

**Cataloghi di sistema.** Un DBMS descrive i dati che gestisce, incluse le informazioni sullo schema della base di dati e gli indici, tramite *meta-dati*. Tali meta-dati sono memorizzati in relazioni speciali, dette *cataloghi* di sistema, e sono utilizzati dalle diverse componenti del DBMS. In tali cataloghi vengono innanzitutto memorizzate informazioni di schema. Ad esempio, per ogni relazione vengono

nomeA	nomeR	pos	tipo
nomeA	Attributi	1	VARCHAR(20)
nomeR	Attributi	2	VARCHAR(20)
pos	Attributi	3	INTEGER
tipo	Attributi	4	VARCHAR(10)
titolo	Film	1	VARCHAR(30)
regista	Film	2	VARCHAR(20)
anno	Film	3	DECIMAL(4)
genere	Film	4	CHAR(15)
valutaz	Film	5	NUMERIC(3,2)
codCli	Cliente	1	DECIMAL(4)
...	...	...	...
dataRest	Noleggio	4	DATE

Figura 7.2: Catalogo relativo allo schema della videoteca

mantenuti il nome della relazione, il nome e il tipo di ciascuno degli attributi, il nome di ciascun indice definito sulla relazione, gli eventuali vincoli definiti sulla relazione. Analogamente, per ogni indice vengono memorizzati il nome, il tipo, gli attributi su cui è definito; per ogni vista vengono memorizzati il nome e la definizione. Nei cataloghi vengono inoltre memorizzate statistiche sulle relazioni e sugli indici (vedi Paragrafo 7.4.2), oltre ad informazioni sugli utenti del DBMS e sui loro diritti di accesso (vedi Capitolo 9).

Un aspetto elegante di un DBMS relazionale è che i cataloghi di sistema sono essi stessi relazioni. Ad esempio, le informazioni sulle relazioni e sulle viste saranno contenute in un catalogo avente come attributi: il nome della relazione, il nome dell'utente creatore, il tipo (relazione/vista) ed il numero di attributi. In un altro catalogo verranno ad esempio memorizzate le informazioni sugli attributi delle relazioni e delle viste. Tale relazione conterrà una tupla per ogni attributo di ogni relazione o vista, con attributi: il nome dell'attributo, il nome della relazione o vista a cui l'attributo appartiene, la posizione ordinale dell'attributo tra gli attributi della stessa relazione o vista, il tipo dell'attributo. Un esempio di tale relazione, che supponiamo essere di schema **Attributi**(*nomeA, nomeR, pos, tipo*), relativamente alla base di dati della videoteca, è presentato nella Figura 7.2. Notiamo come la relazione contenga anche le informazioni relative agli attributi della relazione stessa.

Il fatto che i cataloghi di sistema siano relazioni permette di interrogarli come tutte le altre relazioni e di applicare ad essi tutte le tecniche per l'implementazione e la gestione di relazioni. I DBMS esistenti utilizzano schemi di catalogo differenti, ma le informazioni contenute sono pressapoco le stesse.

## 7.2 Memorizzazione dei dati e gestione del buffer

In questo paragrafo introdurremo le nozioni alla base della memorizzazione fisica dei dati su disco e discuteremo due aspetti cruciali nel minimizzare il costo di

esecuzione dei comandi di manipolazione dei dati: l'organizzazione in cluster e la gestione del buffer.

### 7.2.1 File, record e blocchi

I dati sono trasferiti tra il disco e la memoria principale in unità chiamate *blocchi* (*o pagine*); un blocco è una sequenza di byte contigui su disco. La dimensione del blocco dipende dal sistema operativo e varia tipicamente tra 4 e 16 KB. Poiché il costo di I/O di un blocco domina il costo delle operazioni tipiche delle basi di dati, le strategie di memorizzazione dei dati e di ottimizzazione delle interrogazioni hanno come scopo principale la minimizzazione dei blocchi trasferiti. Evidenziamo inoltre che trasferire blocchi contigui ha un costo decisamente inferiore che trasferire gli stessi blocchi in ordine casuale.

I dati sono generalmente memorizzati in forma di *record*. Ogni record è costituito da un insieme di valori collegati, dove ogni valore è formato da uno o più byte e corrisponde ad un particolare *campo* del record. I record corrispondono in generale a tuple delle relazioni ed i campi del record ai loro attributi. Una collezione di nomi di campi a cui sono associati i tipi corrispondenti costituisce un *tipo di record*. Il tipo associato ad un campo specifica quali valori il campo può assumere; il tipo di un campo è generalmente uno dei tipi predefiniti. Il numero di byte necessari per la memorizzazione di un valore di un certo tipo è fissato per ogni sistema.

**Esempio 7.1** Consideriamo ad esempio un sistema in cui un numero intero può essere memorizzato in 4 byte, un numero reale in 4, una data in 4, una stringa di lunghezza  $k$  in  $k$ . Una tupla della relazione *Film* della Figura 2.1 potrebbe corrispondere ad un record di tipo `struct Film {char titolo[30]; char regista[20]; int anno; char genere[15]; real valutaz}` per la cui memorizzazione sarebbero quindi necessari 73 byte.  $\square$

Ciascun record di un file ha un identificatore unico chiamato *record id* o *RID*, tramite cui è possibile identificare l'indirizzo su disco del blocco che contiene il record. Generalmente il RID è costituito da un identificatore di blocco associato all'identificatore del record all'interno del blocco.

Un *file* è una sequenza di record. In molti casi, tutti i record memorizzati in un file sono dello stesso tipo. Se tutti i record memorizzati in un file hanno la stessa dimensione (in byte), parliamo di *file con record a lunghezza fissa*. Se, al contrario, nel file sono memorizzati record di dimensioni diverse, abbiamo un *file con record a lunghezza variabile*. Un file può contenere record a lunghezza variabile per varie ragioni. Il file può innanzitutto contenere record di tipi differenti e quindi di dimensioni differenti. Questo può succedere se record contenenti informazioni collegate ma di tipi differenti sono memorizzati in posizione contigua sullo stesso blocco di disco. Ad esempio, ciò accade se vogliamo memorizzare le informazioni relative ai noleggi effettuati da un cliente vicino a quelle del cliente (vedi Paragrafo 7.2.2). Il file può inoltre contenere record tutti dello stesso tipo, ma uno o più campi

di tali record possono avere dimensione variabile, essere opzionali o, nel caso di modelli dei dati quale quello relazionale ad oggetti (vedi Capitolo 10), possono assumere più di un valore.

In un file con record a lunghezza fissa, ogni record ha gli stessi campi, e la lunghezza dei campi è fissa, quindi si può identificare la posizione di partenza di ogni campo rispetto alla posizione di partenza del record. Questo facilita l'accesso ai campi del record. Viceversa, nel caso di file con record a lunghezza variabile, sono possibili diverse rappresentazioni: una prima possibilità è l'aggiunta di un simbolo speciale *end-of-record* che indica la fine di ogni record. Alternativamente, si può rappresentare un file con record a lunghezza variabile mediante file con record a lunghezza fissa. La scelta del tipo di rappresentazione dipende dalle caratteristiche dei record contenuti nel file.

A seconda dell'organizzazione primaria dei dati scelta (vedi Paragrafo 7.3), il file che contiene i record dei dati può essere:

- un *file heap* (o file seriale), in cui i record dei dati vengono memorizzati uno dopo l'altro in ordine di inserimento;
- un *file ordinato* (o file sequenziale), in cui i record dei dati sono memorizzati mantenendo l'ordinamento su uno o più campi, in questo caso al file è associato un indice ad albero (vedi Paragrafo 7.3.2) su tali campi;
- un *file hash*, in cui i record dei dati sono memorizzati in una posizione nel file che dipende dal valore ottenuto dall'applicazione di una funzione hash ad uno o più campi del record (vedi Paragrafo 7.3.3);
- un file indice ad albero *integrato*, in cui i record dati sono memorizzati all'interno delle entrate di un indice ad albero (vedi Paragrafo 7.3.2) costruito su uno o più campi di tali record.

Un file può essere visto come una collezione di record. Tuttavia, poiché i dati sono trasferiti in blocchi da memoria secondaria a memoria principale, è importante assegnare i record ai blocchi in modo tale che uno stesso blocco contenga record tra loro correlati. Memorizzando sullo stesso blocco record che sono spesso richiesti insieme si risparmiano infatti accessi a disco. Questo motiva la definizione delle tecniche di organizzazione in cluster discusse nel paragrafo seguente.

### 7.2.2 Organizzazione in cluster

Una strategia di memorizzazione efficiente per alcune interrogazioni è basata sull'organizzazione in *cluster* (*clustering*, o *raggruppamento*) delle tuple che hanno lo stesso valore di uno o più attributi, che prendono il nome di *chiave*<sup>1</sup> del cluster. Clusterizzare una relazione sul valore di uno o più attributi significa organizzare la memorizzazione fisica delle tuple di tale relazione in base al valore di tali attributi,

---

<sup>1</sup>Il termine chiave utilizzato in questo capitolo è diverso da quello introdotto nel Capitolo 2, cui faremo riferimento in questo capitolo come *chiave primaria*.

6610	anna	rossi	01055664433	05-Ott-1979	via scribanti 16	16131 genova
1126	15-Mar-2006	6610	16-Mar-2006			
1112	16-Mar-2006	6610	18-Mar-2006			
1114	16-Mar-2006	6610	17-Mar-2006			
1124	20-Mar-2006	6610	21-Mar-2006			
1115	20-Mar-2006	6610	21-Mar-2006			
1116	21-Mar-2006	6610	?			
1117	21-Mar-2006	6610	?			
6635	paola	bianchi	0104647992	12-Apr-1976	via dodecaneso 35	16146 genova
1111	01-Mar-2006	6635	02-Mar-2006			
1115	01-Mar-2006	6635	02-Mar-2006			
1117	02-Mar-2006	6635	06-Mar-2006			
1118	02-Mar-2006	6635	06-Mar-2006			
1119	08-Mar-2006	6635	10-Mar-2006			
1120	08-Mar-2006	6635	10-Mar-2006			
1121	15-Mar-2006	6635	18-Mar-2006			
1122	15-Mar-2006	6635	18-Mar-2006			
1113	15-Mar-2006	6635	18-Mar-2006			
1129	15-Mar-2006	6635	20-Mar-2006			
1127	22-Mar-2006	6635	?			
1125	22-Mar-2006	6635	?			
6642	marco	verdi	3336745383	16-Ott-1972	via lagustena 35	16131 genova
1111	04-Mar-2006	6642	05-Mar-2006			
1116	08-Mar-2006	6642	09-Mar-2006			
1118	10-Mar-2006	6642	11-Mar-2006			
1119	15-Mar-2006	6642	16-Mar-2006			
1128	18-Mar-2006	6642	20-Mar-2006			
1124	21-Mar-2006	6642	22-Mar-2006			
1122	22-Mar-2006	6642	?			
1113	22-Mar-2006	6642	?			

Figura 7.3: Co-clustering delle relazioni Noleggio e Cliente sull'attributo codCli

quindi le tuple della stessa relazione con lo stesso valore per tali attributi saranno memorizzate fisicamente contigue, nello stesso blocco o su blocchi adiacenti. Tale organizzazione in cluster è associata all'uso di tecniche di indice (ad albero o hash), come organizzazione primaria dei dati, come discusso nel Paragrafo 7.3. Ad esempio, se la memorizzazione fisica dei record corrispondenti alle tuple della relazione Noleggio è quella della Figura 2.2, la relazione è clusterizzata sull'attributo dataNol. Con tale strategia di memorizzazione dei dati possiamo ritrovare in maniera efficiente tutti i noleggi effettuati in un certo giorno, o in un certo periodo, perché i record corrispondenti sono memorizzati nello stesso blocco o in blocchi adiacenti. Viceversa, un clustering su codCli sarebbe più efficiente per ritrovare tutti i noleggi effettuati da un certo cliente e un clustering su colloc per ritrovare tutti i noleggi di un certo video.

È anche possibile organizzare in cluster due o più relazioni, parleremo in questo caso di *co-clustering*. Nel co-clustering vengono memorizzate fisicamente contigue le tuple delle due relazioni che hanno lo stesso valore per gli attributi associati alla chiave del cluster, tipicamente gli attributi che sono chiave esterna di una relazione sull'altra, che vengono utilizzati per effettuarne il join.

**Esempio 7.2** Consideriamo le relazioni Noleggio e Cliente, contenenti le tuple della Figura 2.2, e l'interrogazione SQL che ritrova nome, cognome e data di

nascita dei clienti che hanno effettuato dei noleggi, oltre alla collocazione del video noleggiato ed alla data di noleggio:

```
SELECT nome, cognome, dataN, colloc, dataNol
FROM Noleggio NATURAL JOIN Cliente;
```

Una strategia di memorizzazione efficiente per questo tipo di interrogazione è basata sul co-clustering delle relazioni sull'attributo di join (`codCli`), come illustrato nella Figura 7.3. Il co-clustering può rendere tuttavia inefficiente l'esecuzione di altre interrogazioni. Ad esempio, l'esecuzione dell'interrogazione:

```
SELECT * FROM Cliente;
```

richiede l'accesso ad un numero di blocchi maggiore rispetto ad una strategia di memorizzazione in cui si usa un file separato per ogni relazione. Inoltre, per poter ritrovare le tuple della relazione `Cliente` può essere necessario collegarle con dei puntatori.  $\square$

Un cluster è quindi una struttura di memorizzazione che contiene dati di una o più relazioni. Ogni cluster ha una chiave, costituita da uno o più attributi. Nell'inserire una relazione in un cluster alcuni attributi della relazione vengono associati alla chiave del cluster. Le tuple delle relazioni inserite nel cluster che hanno lo stesso valore per gli attributi associati alla chiave del cluster vengono memorizzate fisicamente contigue. La scelta degli attributi su cui clusterizzare una relazione, così come la scelta di effettuare il co-clustering di relazioni, dipende dalle operazioni da eseguire su di esse e viene effettuata durante la fase di progettazione fisica della base di dati (vedi Paragrafo 7.5). Nel Paragrafo 7.3.4 presenteremo i comandi SQL per organizzare in cluster una o più relazioni.

### 7.2.3 Gestione del buffer

L'obiettivo principale delle strategie di memorizzazione è minimizzare gli accessi a disco. Un altro modo, oltre a quello discusso nel paragrafo precedente, per ottenere lo stesso risultato è mantenere più blocchi possibile in memoria principale. A questo scopo viene utilizzato un *buffer* che permette di tenere in memoria principale copia di alcuni blocchi di disco. Il gestore del buffer di un DBMS usa politiche di gestione che sono più sofisticate delle politiche usualmente utilizzate dai sistemi operativi. In particolare, per motivi legati alla gestione del ripristino (vedi Capitolo 8), in alcuni casi un blocco non può essere trasferito su disco, mentre in altri casi è necessario forzare la copiatura di un blocco su disco, anche se il suo spazio non è stato reclamato. Inoltre, un sistema operativo usa tipicamente politiche di tipo *least recently used* (LRU) per gestire il buffer. In accordo a tali politiche, il blocco contenente dati a cui si è acceduto meno recentemente viene copiato su disco ed eliminato dal buffer. Una politica di questo tipo è adeguata quando non si sa predire il pattern degli accessi. Un DBMS è invece spesso in grado di predire meglio il tipo dei riferimenti futuri, come discusso dall'esempio seguente.

**Esempio 7.3** Consideriamo l'operazione di join:

**Noleggio  $\bowtie$  Cliente**

e supponiamo che le relazioni **Noleggio** e **Cliente** siano memorizzate su file diversi. Per eseguire tale join, una possibilità è considerare ogni tupla di **Noleggio** e confrontarla con ogni tupla di **Cliente** (vedi Paragrafo 7.4.3.4, iterazione orientata ai blocchi). In tal caso, una volta che una tupla della relazione **Noleggio** è stata usata, non è più necessaria. Quindi, non appena le tuple di un blocco sono state esaminate, il blocco non serve più; il gestore del buffer deve pertanto liberare tale blocco (strategia *toss immediate*). Per quanto riguarda invece i blocchi della relazione **Cliente**, il blocco a cui si è acceduto più recentemente sarà utilizzato di nuovo solo dopo che tutti gli altri blocchi saranno stati esaminati. Pertanto, la strategia migliore per i blocchi del file **Cliente** è quella di rimuovere l'ultimo blocco esaminato (*most recently used* - MRU). Affinché tale strategia funzioni correttamente, però, è necessario tenere in memoria il blocco correntemente esaminato fino a che non ne sono state considerate tutte le tuple.  $\square$

### 7.3 Strutture ausiliarie di accesso

Spesso le interrogazioni accedono solo ad un piccolo sottoinsieme dei dati. Per risolvere efficientemente le interrogazioni può quindi essere utile utilizzare delle strutture ausiliarie che permettano di determinare direttamente i record che verificano una data condizione, senza dover accedere a tutti i dati. In tale contesto, il termine *chiave di ricerca* indica un attributo, od un insieme di attributi, usati per la ricerca.<sup>2</sup> Per migliorare l'accesso ai dati vengono utilizzati in genere due tipi di organizzazioni:

- **Organizzazioni primarie.** Tali organizzazioni impongono un criterio di memorizzazione dei dati, possono essere organizzazioni ad albero o basate su tecniche hash.
- **Organizzazioni secondarie.** Tali organizzazioni fanno ricorso ad indici (separati dal file dei dati) che sono normalmente organizzati ad albero (ma sono anche possibili indici hash).

Poiché un'organizzazione primaria impone un criterio di allocazione dei dati, mentre un'organizzazione secondaria no, è possibile avere per gli stessi dati un'organizzazione primaria ed una o più organizzazioni secondarie. In generale, esistono quindi più modalità (cammini) di accesso ai dati. La Figura 7.4 illustra una possibile struttura di memorizzazione per la relazione **Film**. Tale struttura utilizza un'organizzazione primaria, basata su struttura hash sull'attributo **regista**, e due organizzazioni secondarie, una ad albero sull'attributo **anno** e l'altra hash

---

<sup>2</sup>Ricordiamo nuovamente che questo concetto è diverso dal concetto di *chiave primaria* introdotto relativamente all'identificazione delle tuple nel Capitolo 2.

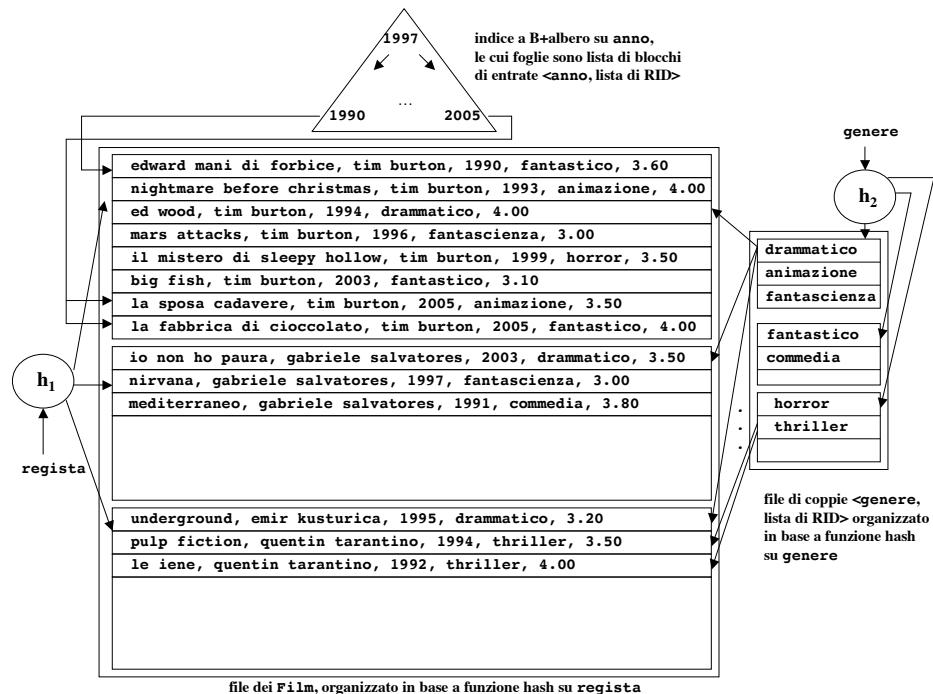


Figura 7.4: Organizzazioni primarie e secondarie

sull'attributo **genere**. L'uso di più indici rende l'esecuzione delle interrogazioni più efficiente, ma rende in generale più costosi gli aggiornamenti. Quando si esegue l'inserimento o la cancellazione di un record è necessario infatti modificare tutti gli indici allocati sul file.

Un indice è costituito da un insieme di *entrate*, ciascuna corrispondente ad un valore  $k_i$  della *chiave* dell'indice, ovvero dell'attributo (degli attributi) su cui è definito l'indice. Le diverse tecniche di indicizzazione (alberi o hash) differiscono essenzialmente nel modo in cui organizzano l'insieme di entrate. Un'entrata dell'indice contiene abbastanza informazioni per localizzare uno o più record di dati che hanno valore  $k_i$  della chiave. Ci sono tre diverse alternative rispetto a che cosa memorizzare nell'entrata dell'indice relativa ad un valore di chiave  $k_i$ :

1. l'entrata contiene un record dati con valore di chiave  $k_i$ , parliamo in questo caso di *indice integrato*;
2. l'entrata contiene una coppia  $(k_i, r_i)$  dove  $r_i$  è il RID del record (eventualmente il solo) con valore di chiave  $k_i$ ;
3. l'entrata contiene una coppia  $(k_i, l_i)$  dove  $l_i$  è una lista di RID di record con valore di chiave  $k_i$ .

L'alternativa (1) corrisponde ad un'organizzazione primaria dei dati, mentre le alternative (2) e (3) corrispondono ad organizzazioni secondarie. Gli indici

<b>Contenuto delle entrate dell'indice</b>	<i>Indice integrato</i> indice le cui entrate contengono i record dei dati (alternativa (1))	<i>Indice separato</i> indice le cui entrate contengono riferimenti ai record dei dati (alternative (2) e (3))
<b>Unicità dei valori di chiave</b>	<i>Indice su chiave primaria</i> indice la cui chiave è chiave primaria per la relazione	<i>Indice su chiave secondaria</i> indice la cui chiave non è chiave primaria per la relazione
<b>Corrispondenza tra posizione entrate dell'indice e record dei dati</b>	<i>Indice clusterizzato</i> indice in cui vi è corrispondenza tra le posizioni delle entrate dell'indice e quelle dei record dati	<i>Indice non clusterizzato</i> indice in cui non vi è corrispondenza tra le posizioni delle entrate dell'indice e quelle dei record dati
<b>Numero di entrate nell'indice</b>	<i>Indice denso</i> indice il cui numero di entrate è pari al numero di valori di $k_i$	<i>Indice sparso</i> indice il cui numero di entrate è minore del numero di valori di $k_i$
<b>Numero di livelli</b>	<i>Indice a singolo livello</i> indice organizzato su un singolo livello	<i>Indice multi-livello</i> indice organizzato su più livelli
<b>Numero di attributi nella chiave</b>	<i>Indice su singolo attributo</i> indice la cui chiave è un singolo attributo	<i>Indice multi-attributo</i> indice la cui chiave è costituita da due o più attributi

Tabella 7.1: Classificazione dei vari tipi di indice

che utilizzano le alternative (2) e (3) vengono a volte anche indicati come *indici separati*. In riferimento alla Figura 7.4, entrambe le organizzazioni secondarie (su **anno** e **genere**) utilizzano l'alternativa (3). Se vogliamo costruire più di un indice su una collezione di record, come discusso precedentemente, uno solo tra gli indici userà l'alternativa (1), per evitare di memorizzare i dati più di una volta. Il vantaggio nell'uso di un indice di tipo (2) o (3) nasce dal fatto che la chiave è solo parte dell'informazione contenuta in un record. Pertanto, l'indice occupa meno spazio del file dei dati. Nel seguito di questo paragrafo introdurremo innanzitutto una classificazione delle varie tipologie di indici per poi discutere le due tecniche di indice più comunemente utilizzate: gli indici ad albero e gli indici hash.

### 7.3.1 Tipologie di indici

La Tabella 7.1 riassume la classificazione degli indici in base alle loro caratteristiche principali, che verrano discusse in quanto segue.

**Indici su chiave primaria e su chiave secondaria.** Un indice *su chiave primaria* ha come chiave un insieme di attributi che include la chiave primaria della relazione su cui è definito. Esisterà quindi al più un record con tale valore per ogni possibile valore di chiave. Un indice è invece *su chiave secondaria* se la chiave dell'indice non include la chiave primaria della relazione su cui è definito l'indice

e possono quindi in generale esistere più record con lo stesso valore di chiave. Ad esempio, in riferimento alla relazione `Noleggio`, l'indice su `(colloc,dataNol)` è un indice su chiave primaria, mentre l'indice su `codCli` è un indice su chiave secondaria. Nel caso di indice su chiave primaria possono essere utilizzate le alternative (1) o (2) discusse nel paragrafo precedente, mentre non ha senso utilizzare l'alternativa (3), poiché esiste un unico record per ogni valore di chiave. Per gli indici su chiave secondaria, al contrario, può essere usata l'alternativa (3). Possono però essere anche usate le alternative (1) o (2), nelle seguenti modalità: (i) con entrate *duplicate*, cioè con lo stesso valore di chiave, che quindi compare in più entrate dell'indice; (ii) inserendo un ulteriore livello di indirezione: l'entrata per la chiave  $k_i$  dell'indice contiene il riferimento ad un blocco in cui sono memorizzati i record dei dati oppure i RID dei record dei dati (a seconda che si tratti di alternativa (1) o (2)) aventi valore di chiave  $k_i$ . Il vantaggio rispetto all'alternativa (3) è che le entrate dell'indice hanno tutte la stessa dimensione. Il vantaggio di (ii) rispetto ad (i) è invece che non devono essere modificati gli algoritmi di ricerca ed inserimento nell'indice. Anche se la soluzione (ii) ha lo svantaggio evidente del livello di indirezione aggiuntivo introdotto, risulta comunque la più utilizzata.

**Indici clusterizzati e non clusterizzati.** Quando un file è organizzato in modo tale che vi è una corrispondenza tra la posizione delle entrate dell'indice e quelle dei record dati corrispondenti, l'indice è detto *clusterizzato* (o *primario*,<sup>3</sup> od *organizzato in cluster*); altrimenti è un indice *non clusterizzato* (o *secondario*, o *non organizzato in cluster*). Un indice che usa l'alternativa (1), cioè un indice integrato, è organizzato in cluster per definizione. Un indice che usa l'alternativa (2) o (3) è clusterizzato se è un indice ad albero ed i record dei dati sono ordinati sui campi chiave dell'indice. Poiché le entrate di un indice ad albero sono ordinate in base al valore della chiave, se i record dei dati sono ordinati sui campi corrispondenti si ha anche in questo caso una corrispondenza tra la posizione delle entrate dell'indice e quelle dei record dei dati corrispondenti.

La Figura 7.5 illustra un esempio di indice non clusterizzato. In particolare, illustra un indice sull'attributo `colloc` della relazione `Noleggio`, non clusterizzato, denso, su chiave secondaria che usa l'alternativa (3). La Figura 7.6 illustra invece un esempio di indice clusterizzato. In particolare, illustra un indice sull'attributo `dataNol` della relazione `Noleggio`, clusterizzato, sparso, su chiave secondaria che usa l'alternativa (3).

**Indici densi e sparsi.** Relativamente al numero di entrate dell'indice possiamo distinguere tra indici densi ed indici sparsi. Un *indice denso* contiene un'entrata per ogni valore della chiave di ricerca nel file; al contrario, in un *indice sparso* le entrate dell'indice sono create solo per alcuni valori della chiave. La Figura 7.6 illustra un indice sparso su `dataNol`. Per localizzare un record, utilizzando

---

<sup>3</sup>Alcuni testi usano il termine indice primario per indicare indici su chiave primaria. Per evitare ambiguità, eviteremo il termine indice primario ed useremo invece i termini indice su chiave primaria ed indice clusterizzato.

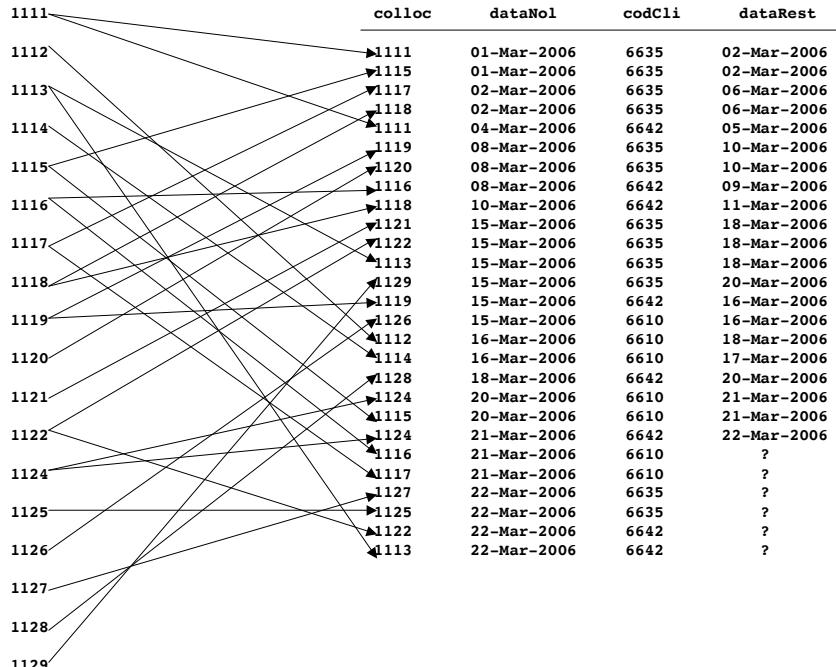


Figura 7.5: Indice non clusterizzato

un indice sparso, viene eseguita una scansione fino a trovare il record con il più alto valore della chiave che sia minore o uguale al valore cercato. Viene quindi effettuata una ricerca sequenziale nel file dei dati, a partire da tale record, fino a trovare il record cercato. Gli indici non clusterizzati sono indici densi anziché sparsi; il file dei dati, infatti, non è ordinato in base alla chiave dell'indice. In riferimento all'indice su `colloc` illustrato nella Figura 7.5, infatti, se l'entrata per 1126 non fosse presente, l'indice non sarebbe di alcuna utilità per ritrovare i noleggi di tale video. Un indice denso consente una ricerca più veloce, ma impone maggiori costi di aggiornamento. Un indice sparso è viceversa meno efficiente, ma impone minori costi di aggiornamento. Poiché la strategia è di minimizzare il numero di blocchi trasferiti, un compromesso spesso adottato consiste nell'avere un'entrata nell'indice per ogni blocco.

**Indici multi-livello.** Un indice a singolo livello è un indice le cui entrate sono costruite sui record dei dati. Un indice multi-livello, viceversa, è un indice le cui entrate sono costruite su un altro indice. Molto spesso, infatti, un indice, anche se sparso, può essere di dimensioni notevoli. Ad esempio, un file di 100'000 record, con 10 record per blocco, richiede un indice con 10'000 entrate; assumendo che un blocco contenga 100 entrate dell'indice, sono necessari 100 blocchi. Se

	<b>colloc</b>	<b>dataNol</b>	<b>codCli</b>	<b>dataRest</b>
01-Mar-2006	1111	01-Mar-2006	6635	02-Mar-2006
	1115	01-Mar-2006	6635	02-Mar-2006
	1117	02-Mar-2006	6635	06-Mar-2006
	1118	02-Mar-2006	6635	06-Mar-2006
	1111	04-Mar-2006	6642	05-Mar-2006
	1119	08-Mar-2006	6635	10-Mar-2006
	1120	08-Mar-2006	6635	10-Mar-2006
	1116	08-Mar-2006	6642	09-Mar-2006
10-Mar-2006	1118	10-Mar-2006	6642	11-Mar-2006
	1121	15-Mar-2006	6635	18-Mar-2006
	1122	15-Mar-2006	6635	18-Mar-2006
	1113	15-Mar-2006	6635	18-Mar-2006
	1129	15-Mar-2006	6635	20-Mar-2006
	1119	15-Mar-2006	6642	16-Mar-2006
	1126	15-Mar-2006	6610	16-Mar-2006
	1112	16-Mar-2006	6610	18-Mar-2006
	1114	16-Mar-2006	6610	17-Mar-2006
	1128	18-Mar-2006	6642	20-Mar-2006
20-Mar-2006	1124	20-Mar-2006	6610	21-Mar-2006
	1115	20-Mar-2006	6610	21-Mar-2006
	1124	21-Mar-2006	6642	22-Mar-2006
	1116	21-Mar-2006	6610	?
	1117	21-Mar-2006	6610	?
	1127	22-Mar-2006	6635	?
	1125	22-Mar-2006	6635	?
	1122	22-Mar-2006	6642	?
	1113	22-Mar-2006	6642	?

Figura 7.6: Indice sparso (clusterizzato)

l'indice è piccolo, può essere tenuto in memoria principale. Molto spesso, però, è necessario tenerlo su disco e quindi la scansione dell'indice può richiedere parecchi trasferimenti di blocchi (se l'indice occupa un numero  $b$  di blocchi e si usa una ricerca binaria, è necessario accedere ad un numero di blocchi pari a  $1 + \log_2 b$ , circa 7 nel nostro esempio). È quindi necessario trattare l'indice come un file ed allocare un indice sparso sull'indice stesso. Si parla in questo caso di *indice sparso a due livelli*. Se l'indice esterno è mantenuto in memoria principale, nell'esempio precedente è necessario accedere ad un solo blocco dell'indice.

**Indici multi-attributo.** Un indice multi-attributo è un indice la cui chiave è costituita da più di un attributo. Viceversa, un indice la cui chiave è costituita da un solo attributo è detto indice su singolo attributo. Ad esempio, l'indice su  $(\text{colloc}, \text{dataNol})$  per la relazione *Noleggio* è un indice multi-attributo, mentre l'indice su  $\text{codCli}$  è un indice su singolo attributo. L'ordinamento tra i valori di chiave è dato dall'ordinamento lessicografico. L'indice multi-attributo è definito su una *lista* di attributi, cioè diversi ordinamenti degli attributi chiave danno luogo ad indici differenti. Un indice su  $(\text{colloc}, \text{dataNol})$  è cioè differente, ad esempio, da un indice su  $(\text{dataNol}, \text{colloc})$ . Un indice multi-attributo permette di determinare efficientemente le tuple che soddisfano condizioni di uguaglianza o di intervallo (nel caso di indici ad albero) su tutti gli attributi nella chiave o su un *prefisso* della lista di attributi. L'indice su  $(\text{dataNol}, \text{colloc})$ , ad esempio, è utile per condizioni quali  $\text{dataNol} = \text{DATE } '15-\text{Mar-2006}' \text{ AND colloc} = 1111$ ,  $\text{dataNol}$

= DATE '15-Mar-2006' oppure `dataNol BETWEEN DATE '15-Mar-2006' AND DATE '18-Mar-2006'`. Viceversa, l'indice non aiuta, ad esempio, a determinare tutti i noleggi per il video 1120.

Un indice multi-attributo può supportare una più vasta gamma di interrogazioni rispetto ad un indice a singolo attributo. Inoltre, poiché le entrate dell'indice contengono più informazioni sul record dei dati, gli indici multi-attributo offrono maggiori opportunità di poter valutare interrogazioni accedendo solo all'indice e non al file dei dati (come vedremo nel Paragrafo 7.4, una valutazione basata soltanto sull'indice non necessita di accedere ai dati, ma trova tutti i valori degli attributi richiesti nelle entrate dell'indice). D'altra parte, un indice multi-attributo deve essere aggiornato più frequentemente ed è di dimensione maggiore rispetto ad un indice su singolo attributo.

### 7.3.2 Indici ad albero

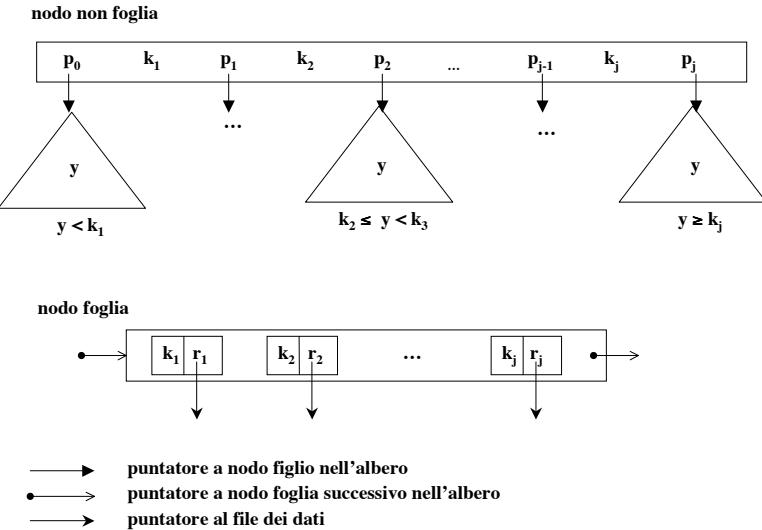
Gli indici ad albero sono alberi binari di ricerca bilanciati per memoria secondaria che rispondono ai seguenti requisiti fondamentali:

- **bilanciamento**, l'indice è bilanciato rispetto ai blocchi e non ai singoli nodi, in quanto è il numero di blocchi acceduti a determinare il costo I/O di una ricerca;
- **occupazione minima**, per evitare un sotto-utilizzo della memoria viene stabilito un limite inferiore all'utilizzazione dei blocchi;
- **efficienza di aggiornamento**, il costo delle operazioni di aggiornamento è comunque limitato.

In un indice ad albero, quindi, ogni nodo corrisponde ad un blocco, memorizza molti valori di chiave e tipicamente ha centinaia di figli. Il costo delle operazioni (ricerca, inserimento e cancellazione) in tali strutture è lineare nell'altezza dell'albero<sup>4</sup> e logaritmico nel numero di elementi memorizzati nell'indice. La struttura impone un limite minimo (oltre all'ovvio limite massimo corrispondente alla dimensione del nodo) al numero di elementi contenuti in un nodo. Il numero di figli di un nodo è dato dal numero di elementi contenuti nel nodo più uno. Se ogni nodo non foglia avesse  $n$  figli, un albero di altezza  $h$  avrebbe  $n^h$  nodi foglia. Nella pratica, i nodi non hanno lo stesso numero di figli, ma usando il valore medio  $m$ ,  $m^h$  è una buona approssimazione del numero dei nodi foglia. Sempre nella pratica,  $m$  è almeno 100, quindi un albero di altezza quattro contiene 100 milioni di nodi foglia.

Gli indici ad albero più utilizzati sono i B<sup>+</sup>-alberi, il formato dei cui nodi è illustrato nella Figura 7.7. Nella figura, entrambi i tipi di nodi, sia quelli interni sia i nodi foglia, hanno  $j$  elementi,  $k_1, \dots, k_j$  sono i valori di chiave,  $p_0, \dots, p_j$  sono i

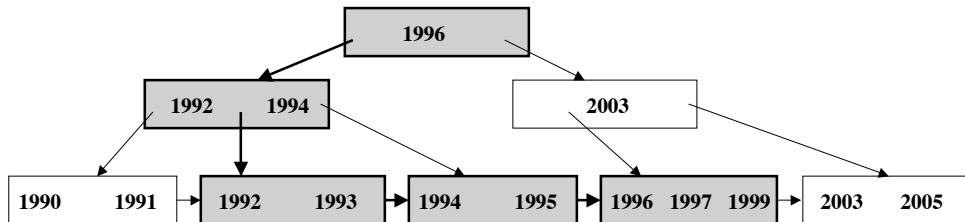
<sup>4</sup>Con altezza dell'albero indichiamo la lunghezza di un cammino dalla radice ad una foglia. La proprietà di bilanciamento dell'albero assicura che tutti i cammini radice-foglia abbiano la stessa lunghezza.

Figura 7.7: Formato di un nodo di un B<sup>+</sup>-albero

puntatori ai nodi figli e  $r_1, \dots, r_j$  sono RID (facciamo riferimento ad un albero che utilizza l'alternativa (2) discussa in precedenza). Come evidenziato dalla figura, in un B<sup>+</sup>-albero le entrate dell'indice sono contenute nelle foglie, mentre i nodi interni costituiscono una “mappa” che consente una rapida localizzazione delle chiavi e memorizzano dei separatori. La funzione dei separatori è determinare il giusto cammino nella ricerca di una chiave. Il sotto-albero sinistro di un separatore contiene valori di chiave minori del separatore. Il sotto-albero destro contiene valori di chiave maggiori od uguali al separatore.<sup>5</sup> Nel caso di chiavi alfanumeriche è possibile utilizzare separatori di lunghezza ridotta risparmiando spazio e, eventualmente, riducendo l'altezza dell'albero. I nodi foglia sono inoltre collegati a lista.

**Esempio 7.4** Consideriamo l'indice a B<sup>+</sup>-albero sull'attributo **anno** della relazione **Film**, introdotto nella Figura 7.4. Un B<sup>+</sup>-albero contenente i valori di tale attributo è mostrato nella Figura 7.8. In realtà, per illustrare le caratteristiche dell'albero, abbiamo utilizzato nodi che contengono al massimo 3 elementi, quindi con capacità decisamente inferiore a quella dei nodi utilizzati in pratica. Per semplicità, inoltre, non abbiamo evidenziato nella figura i riferimenti al file dei dati. L'albero nella figura contiene 11 elementi (cioè 11 entrate dell'indice), ha 5 foglie ed altezza 3. Ricordiamo che, rispetto alla classificazione nella Tabella 7.1, tale indice è un indice su chiave secondaria, denso, su singolo attributo (non

<sup>5</sup>È possibile anche adottare la convenzione che i valori uguali siano contenuti nel sotto-albero sinistro.

Figura 7.8: B<sup>+</sup>-albero sull'attributo anno della relazione Film

specifichiamo se clusterizzato o meno perché non sappiamo come è organizzato il file dei dati). □

La ricerca di una chiave avviene nel B<sup>+</sup>-albero fino ad individuare una foglia, nella quale si trovano le entrate dell'indice, cioè le chiavi ed i corrispondenti riferimenti ai dati. Una volta trasferita la radice in memoria, si esegue la ricerca tra le chiavi contenute nel nodo in esame fino a determinare la presenza o l'assenza nel nodo della chiave cercata. Se la chiave non viene trovata, si continua la ricerca nell'unico sotto-albero del nodo corrente che può contenere l'elemento. Se invece il nodo è foglia, significa che la chiave non è presente nell'albero. Il collegamento a lista dei nodi foglia consente di risolvere in modo efficiente anche ricerche su intervalli: viene ricercato nell'albero l'estremo sinistro dell'intervalle arrivando ad una foglia, la lista dei nodi foglia viene quindi attraversata fino a raggiungere l'estremo destro dell'intervalle.

**Esempio 7.5** In riferimento al B<sup>+</sup>-albero nella Figura 7.8, se vogliamo ritrovare tutti i film usciti tra il 1992 e il 1998, si raggiungerà la foglia dell'indice corrispondente al valore di chiave 1992 e ci si sposterà poi nelle foglie dell'indice fino a trovare il valore di chiave 1999. Il cammino seguito nell'albero per effettuare la ricerca è evidenziato nella figura marcando gli archi attraversati in grassetto ed ombreggiando i nodi visitati. □

Le operazioni di inserimento e cancellazione richiedono innanzitutto una ricerca dell'elemento, ma possono richiedere ulteriori aggiustamenti dell'albero per mantenere la proprietà di bilanciamento ed i vincoli sul numero minimo e massimo di elementi contenuti in un nodo. L'idea chiave su cui si basano gli algoritmi per l'inserimento e la cancellazione è che le modifiche partono sempre dalle foglie e l'albero cresce o si accorcia verso l'alto. Ad esempio, nel caso dell'inserimento, non si creano nuovi figli dalle foglie, ma, se necessario, si crea una nuova foglia allo stesso livello delle altre e si propaga un valore di chiave (separatore) verso l'alto. I nodi ai livelli superiori non sono necessariamente pieni e quindi possono "assorbire" le informazioni che si propagano dalle foglie. La propagazione degli

effetti sino alla radice può provocare l'aumento dell'altezza dell'albero. Il costo delle operazioni è comunque lineare nell'altezza dell'albero.

### 7.3.3 Indici hash

L'uso di indici ad albero ha lo svantaggio di richiedere la scansione di una struttura dati per localizzare i dati; questo perché nelle tecniche di tipo "tabellare" l'associazione (chiave, indirizzo) è mantenuta in forma esplicita. Un'organizzazione hash, invece, utilizza una *funzione hash*  $h$  che trasforma ogni valore di chiave in un indirizzo. In queste organizzazioni, i record di un file sono raggruppati in  $M$  *bucket*, dove un bucket consiste in un blocco primario ed eventuali blocchi addizionali organizzati in una lista. La funzione hash, data una chiave, restituisce l'indirizzo (blocco o bucket di blocchi) da cui partire per cercare i record con quel valore di chiave (in relazione al fatto che vengano utilizzate le alternative (1), (2) o (3) discusse nel Paragrafo 7.3). Ad esempio, nella Figura 7.4, la funzione hash  $h_1$ , dato un **regista**, restituisce l'indirizzo del bucket contenente i record dati dei **Film** di tale regista (organizzazione primaria), viceversa, la funzione hash  $h_2$ , dato un **genere**, restituisce l'indirizzo del bucket contenente la lista dei RID dei **Film** di quel genere (organizzazione secondaria). Le organizzazioni hash sono usate principalmente per l'organizzazione primaria dei dati.

Una funzione hash  $h$  è una funzione dall'insieme dei possibili valori per la chiave all'insieme  $0, \dots, M - 1$  dei possibili indirizzi che deve verificare le seguenti proprietà:

- **distribuzione uniforme delle chiavi nello spazio degli indirizzi:** ogni indirizzo deve essere generato con la stessa probabilità;
- **distribuzione casuale delle chiavi:** eventuali correlazioni tra i valori delle chiavi non devono tradursi in correlazioni tra gli indirizzi generati.

Poiché tali proprietà dipendono dall'insieme delle chiavi su cui si opera non esiste una funzione universale ottima. Una delle funzioni hash che si comporta meglio, e quindi tra le più utilizzate in pratica, è quella basata sul metodo della divisione: la chiave numerica viene divisa per  $M$  e l'indirizzo è ottenuto considerando il resto. La funzione  $h$  è quindi definita come  $h(k) = k \bmod M$  dove, come usuale,  $\bmod$  indica il resto della divisione intera. Affinché  $h$  distribuisca bene è però necessario che  $M$  sia un numero primo oppure un numero non primo con nessun fattore primo minore di 20.

Per effettuare la ricerca, dato un valore di chiave  $k$ , calcoliamo semplicemente  $h(k)$ . Ogni indirizzo generato dalla funzione hash individua una pagina logica o bucket. Il costo delle operazioni è costante, se la struttura è ben progettata, ed ogni indirizzo corrisponde ad un singolo blocco. Il numero di elementi che possono essere allocati nello stesso bucket determina la *capacità*  $c$  dei bucket. Se una chiave viene assegnata ad un bucket che già contiene  $c$  chiavi si verifica un *trabocco* (*overflow*). La presenza di overflow può richiedere, dipendentemente dalla specifica organizzazione, l'uso di un'area di memoria separata, detta *area*

*di overflow.* L'area di memoria costituita dai bucket indirizzabili dalla funzione hash è detta *area primaria*. Alle pagine dell'area primaria si accede direttamente, mentre alle pagine dei trabocchi si accede mediante riferimenti memorizzati nelle pagine dell'area primaria.

Una funzione hash genera  $M$  indirizzi, tanti quanti sono i bucket dell'area primaria. Se il valore di  $M$  per una data organizzazione è costante, l'organizzazione è detta *statica*. In questo caso il dimensionamento dell'area primaria è parte integrante del progetto dell'organizzazione. Se l'area primaria può espandersi e contrarsi, per meglio adattarsi al volume effettivo dei dati da gestire, allora l'organizzazione è detta *dinamica*. In questo caso è necessario l'utilizzo di più funzioni hash.

**Confronto tra indici ad albero ed hash.** L'uso di un indice ad albero piuttosto che hash dipende dal tipo di interrogazioni. Se ad esempio la maggior parte delle interrogazioni che coinvolgono la chiave  $A_i$  ha la forma:

```
SELECT A1,A2,...,An FROM R WHERE Ai=C;
```

la tecnica hash è preferibile. La scansione di un indice ad albero ha infatti un costo proporzionale al logaritmo del numero di valori in  $R$  per  $A_i$  mentre in una struttura hash il tempo di ricerca è indipendente dalla dimensione della relazione.

Le strutture ad albero sono invece preferibili se le interrogazioni che coinvolgono la chiave  $A_i$  usano condizioni di intervallo come:

```
SELECT A1,A2,...,An FROM R WHERE Ai BETWEEN C1 AND C2;
```

perché è difficile determinare funzioni hash che mantengano l'ordine. In generale, è difficile prevedere a priori il tipo di interrogazioni per cui in pratica sono maggiormente utilizzate strutture di indici ad albero.

#### 7.3.4 Definizione di cluster ed indici in SQL

La maggior parte dei DBMS relazionali fornisce varie primitive che permettono al progettista della base di dati di definire la configurazione fisica dei dati. Queste primitive sono rese disponibili all'utente come comandi del linguaggio SQL. Lo standard SQL:2003, tuttavia, non include alcun comando per la definizione di strutture di memorizzazione o di indici. In realtà, lo standard non richiede neppure che le implementazioni di SQL supportino gli indici. Ovviamente, nella pratica, ogni DBMS relazionale in commercio supporta uno o più tipi di indice ed, in genere, alloca automaticamente indici, ad esempio, sugli attributi chiave primaria di relazione. Poiché esistono però numerose differenze tra i vari DBMS per quanto riguarda gli aspetti di organizzazione fisica, i comandi per la configurazione fisica dei dati differiscono notevolmente tra i vari sistemi. I comandi più importanti sono il comando per la creazione di cluster ed il comando per la creazione di indici, oltre

all'inserimento di una relazione in un cluster. Nel seguito faremo riferimento ad una versione semplificata della sintassi Oracle.

#### 7.3.4.1 Definizione di cluster

Il comando per la creazione di un cluster ha il seguente formato:

```
CREATE CLUSTER <nome cluster>
(<nome colonna> <dominio> [,<nome colonna> <dominio>]*)
[{:INDEX | HASHKEYS <intero> [HASH IS <espressione>]}];
```

dove:

- <nome cluster> è il nome del cluster che viene definito.
- La lista di coppie <nome colonna> <dominio> è la specifica delle colonne del cluster, cioè della chiave del cluster.

Come possiamo inoltre notare dal formato del comando, ogni cluster ha sempre associata una struttura di accesso ausiliaria. Le scelte possibili sono:

- INDEX. Viene allocato un indice di tipo B<sup>+</sup>-albero. Tale scelta, che è quella di default, conviene se si hanno frequenti interrogazioni di tipo intervallo sulla chiave del cluster o se le relazioni possono aumentare di dimensione in modo impredicibile. Un cluster su cui è allocato un indice è detto *cluster di tipo index*.
- HASH. Viene usata una struttura di accesso di tipo hash. Un cluster su cui viene utilizzata un'organizzazione hash è detto *cluster di tipo hash*.

Se il cluster è di tipo index, prima di poter manipolare le relazioni nel cluster è necessario creare un indice sul cluster tramite il comando **CREATE INDEX**, discusso nel Paragrafo 7.3.4.2.

La clausola **HASHKEYS** richiede l'uso dell'hash come struttura di accesso e specifica il numero di valori della funzione hash. Questo valore se non è un numero primo viene arrotondato dal sistema al primo numero primo maggiore. Tale intero viene usato come argomento dell'operazione usata dal sistema per generare i valori della funzione hash; sia  $M$  l'intero dato (od il suo arrotondamento), i valori generati sono compresi tra 0 ed  $M - 1$ . Nei cluster di tipo hash, la clausola **HASH IS** permette la specifica della funzione hash da utilizzare. Il DBMS fornisce sempre una funzione hash interna che viene usata come default. È tuttavia possibile non usare questo default e specificare, tramite l'opzione **HASH IS**, come valori della funzione hash i valori dell'espressione indicata in questa opzione. L'espressione deve però assumere come valori solo interi positivi e deve fare riferimento ad una o più tra le colonne del cluster.

**Esempio 7.6** I seguenti comandi definiscono, rispettivamente, un cluster NolCli di tipo index ed un cluster NolCliH di tipo hash. Entrambi i cluster hanno come chiave un'unica colonna cod, di tipo DECIMAL(4).

```
CREATE CLUSTER NolCli (cod DECIMAL(4));

CREATE CLUSTER NolCliH (cod DECIMAL(4)) HASHKEYS 10;
```

Dato che l'opzione HASHKEYS ha valore 10, il numero di valori generati dalla funzione hash è 11. Come funzione hash viene utilizzata quella di default fornita dal sistema.  $\square$

#### 7.3.4.2 Definizione di indici

Il comando per la creazione di un indice ha il seguente formato:

```
CREATE INDEX <nome indice>
ON {<nome relazione>(<lista nomi colonne>) |
    CLUSTER <nome cluster>}
[ {ASC | DESC}];
```

dove:

- <nome indice> è il nome dell'indice che viene creato.
- La clausola ON specifica l'oggetto su cui è allocato l'indice. Tale oggetto può essere: una relazione, ed in tal caso devono essere specificati i nomi delle colonne su cui l'indice è allocato, oppure un cluster, ed in tal caso non è necessario specificare alcuna colonna in quanto l'indice viene allocato automaticamente sulle colonne chiave del cluster. L'indice può essere allocato su più colonne; i valori della chiave su cui l'indice è costruito sono ottenuti come concatenazione di tutti i valori delle colonne su cui l'indice è allocato. Normalmente esiste un limite al numero di colonne su cui un singolo indice può essere allocato (in Oracle ad esempio tale limite è 16).
- Le opzioni (mutuamente esclusive) ASC e DESC specificano rispettivamente se i valori della chiave dell'indice devono essere ordinati in modo crescente o decrescente. ASC è l'opzione di default.

**Esempio 7.7** I seguenti comandi definiscono un indice idxCliente sull'attributo codCli della relazione Noleggio ed un indice idxNol sul cluster NolCli definito nell'Esempio 7.6.

```
CREATE INDEX idxCliente ON Noleggio (codCli);

CREATE INDEX idxNol ON CLUSTER NolCli;
```

Supponiamo che la relazione `Noleggio` sia inserita nel cluster `NolCli`, come verrà mostrato nell'Esempio 7.8. Rispetto alla classificazione introdotta nella Tabella 7.1, per la relazione `Noleggio` entrambi gli indici sono esempi di indici su singolo attributo e su chiave secondaria. L'indice `idxCliente` sarà clusterizzato (perché definito sul cluster `NolCli`), mentre l'indice `idxCliente no` (o non necessariamente). La scelta se mantenere tali indici come densi o come sparsi viene presa dal sistema, anche se, come discusso, gli indici non clusterizzati sono sicuramente densi e quelli clusterizzati generalmente sparsi.  $\square$

#### 7.3.4.3 Inserimento di relazioni in un cluster

Un cluster può includere una o più relazioni. Nel caso di una singola relazione, il cluster è usato per raggruppare le tuple della relazione aventi lo stesso valore per le colonne chiave del cluster. Nel caso di più relazioni, il cluster viene usato per effettuarne il co-clustering, cioè raggruppare le tuple di tutte le relazioni aventi lo stesso valore per la chiave del cluster, permettendo pertanto esecuzioni efficienti per operazioni di join sulle colonne che sono parte della chiave del cluster (vedi Paragrafo 7.2.2). Una relazione deve essere inserita nel cluster al momento della creazione; pertanto il comando `CREATE TABLE` include un'ulteriore clausola `CLUSTER` che permette di specificare il cluster in cui inserire la relazione.

**Esempio 7.8** Supponiamo di voler inserire nel cluster `NolCli` definito nell'Esempio 7.6 le relazioni `Noleggio` e `Cliente`. Alla chiave `cod` del cluster vogliamo far corrispondere, in entrambe le relazioni, la colonna `codCli`, il cui dominio, `DECIMAL(4)`, coincide con il dominio della chiave del cluster. I comandi per la definizione delle due relazioni sono i seguenti (per brevità, nelle definizioni delle relazioni abbiamo omesso gran parte degli attributi):

```
CREATE TABLE Noleggio (colloc DECIMAL(4) NOT NULL,
                      codCli DECIMAL(4),
                      ...)
CLUSTER NolCli (codCli);

CREATE TABLE Cliente (codCli DECIMAL(4) PRIMARY KEY,
                     ...)
CLUSTER NolCli (codCli);  $\square$ 
```

Come possiamo notare dall'esempio precedente, i nomi delle colonne delle relazioni su cui si esegue il clustering non devono necessariamente avere lo stesso nome della colonna del cluster. Alla colonna del cluster, che ha nome `cod`, vengono fatte corrispondere le colonne di nome `codCli` nelle relazioni `Noleggio` e `Cliente`. Le colonne del cluster e quelle delle relazioni devono essere però lo stesso numero ed i domini corrispondenti devono essere compatibili.

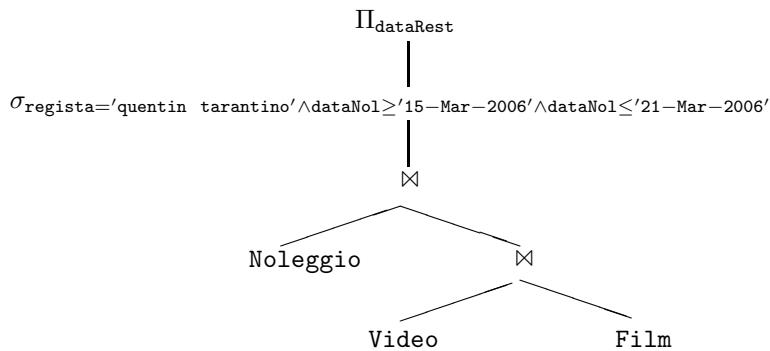


Figura 7.9: Rappresentazione ad albero di espressione algebrica (piano logico di esecuzione)

#### 7.4 Elaborazione di interrogazioni

Abbiamo visto finora come organizzare i dati in una base di dati. Le decisioni sulle strutture da allocare sono determinate durante la progettazione fisica della base di dati. La modifica di tali strutture in seguito può essere costosa. Quando un'interrogazione è presentata al sistema, occorre determinare il modo più efficiente per eseguirla usando le strutture disponibili. Un'interrogazione viene trasformata in un *piano* o *strategia di esecuzione* per l'interrogazione. Per interrogazioni complesse esistono più strategie possibili. Anche se il costo di determinare la strategia ottima può essere alto, il vantaggio in termini di efficienza che se ne ricava è tale che in genere conviene eseguire l'ottimizzazione.

**Esempio 7.9** Consideriamo la seguente interrogazione:

*Determinare la data di restituzione dei noleggi di film di Quentin Tarantino iniziati nella settimana tra il 15 ed il 21 Marzo 2006.*

e la riscrittura algebrica della corrispondente interrogazione SQL, rappresentata ad albero nella Figura 7.9.<sup>6</sup> Tale espressione corrisponde alla costruzione di un'enorme relazione intermedia, ottenuta come join delle tre relazioni, da cui poi vengono estratte le tuple che soddisfano la condizione di selezione. Se invece filtrassimo le tuple di *Film* corrispondenti ai film di Quentin Tarantino e recuperassimo solo le informazioni relative ai noleggi di tali film, eviteremmo l'accesso a molte tuple che di fatto non sono utilizzate per produrre il risultato. Se sapessimo inoltre di avere a disposizione, ad esempio, un indice sull'attributo *colloc* nella relazione *Noleggio*, una volta determinate le collocazioni dei video contenenti film di Quentin Tarantino, potremmo fare uso di tale indice per determinare

<sup>6</sup>La rappresentazione interna delle interrogazioni è proprio un albero in cui i nodi interni sono operatori dell'algebra relazionale ed i nodi foglia sono le relazioni.

direttamente i noleggi di tali video, senza scandire l'intera relazione `Noleggio`, che è di dimensioni considerevoli.  $\square$

In questo paragrafo esamineremo innanzitutto le fasi nell'elaborazione di un'interrogazione. Introdurremo poi le statistiche utilizzate per selezionare la strategia di esecuzione ed accenneremo alle principali tecniche di implementazione degli operatori relazionali, per discutere infine brevemente le fasi di ottimizzazione logica ed ottimizzazione fisica.

#### 7.4.1 Fasi nell'elaborazione

Le fasi nell'elaborazione di un'interrogazione possono essere riassunte come segue:

- **Parsing.** Viene controllata la correttezza sintattica dell'interrogazione e ne viene generata una rappresentazione interna, il parse tree, che è l'albero sintattico dell'interrogazione.
- **Ottimizzazione logica.** L'interrogazione, rappresentata mediante un albero che corrisponde ad un'espressione algebrica, viene trasformata in un'interrogazione equivalente ma più efficiente da eseguire, sfruttando le proprietà dell'algebra relazionale.
- **Ottimizzazione fisica.** Viene determinato in modo preciso come l'interrogazione sarà eseguita (per esempio che indici si useranno). La scelta della strategia è effettuata principalmente in modo da minimizzare il numero di accessi a disco.
- **Esecuzione.** L'interrogazione viene eseguita in accordo alla strategia scelta nella fase precedente.

Nel caso di utilizzo di SQL da linguaggio di programmazione, discusso nel Capitolo 4, per i comandi preparati è possibile separare le prime tre fasi (che costituiscono la *preparazione* del comando) dalla quarta (che ne costituisce l'*esecuzione*).

Le fasi nell'elaborazione di un'interrogazione sono illustrate graficamente nella Figura 7.10. L'interrogazione viene sottoposta al *parser* ne viene prodotta una rappresentazione interna. Dal parse tree così ottenuto, nella fase di *conversione*, viene prodotto un *piano logico di esecuzione* (*Logical Query Plan – LQP*) che consiste in un'espressione algebrica corrispondente all'interrogazione. In realtà si tratta di un'espressione di un'algebra estesa in cui sono presenti anche operatori corrispondenti all'ordinamento e al raggruppamento. Tale espressione è rappresentata internamente come un albero, come quello illustrato nella Figura 7.9. Su questo piano logico viene eseguita la fase di *riscrittura algebrica*, che produce un piano logico ottimizzato. Le fasi successive (ottimizzazione fisica) hanno lo scopo di determinare un piano fisico di esecuzione per tale piano logico. Un *piano fisico di esecuzione* di un'interrogazione consiste in un ordine di esecuzione delle varie operazioni relazionali e di una strategia per l'esecuzione di ogni operazione. È cioè

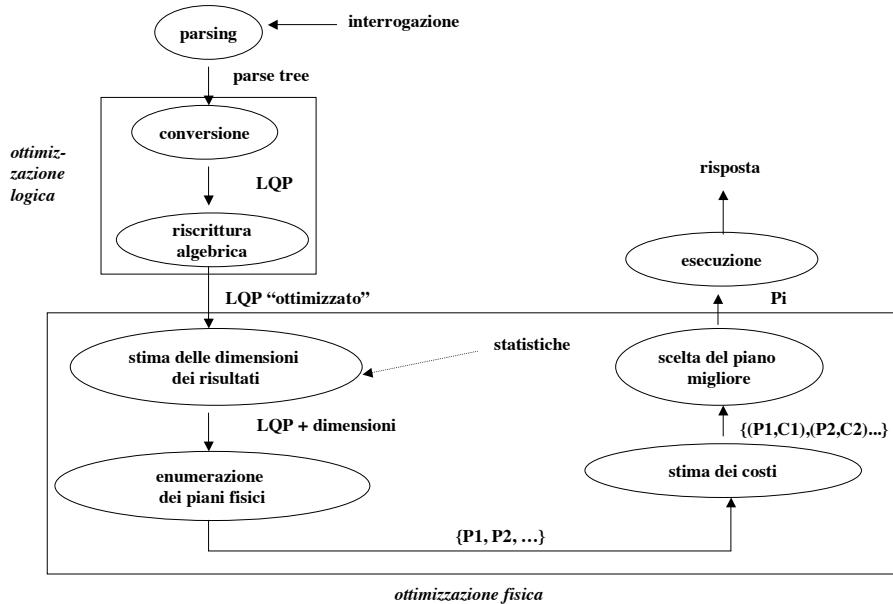


Figura 7.10: Fasi nell'elaborazione di un'interrogazione

l'albero corrispondente al LQP in cui, oltre all'ordine di esecuzione delle varie operazioni, vengono precisati (mediante opportuna etichettatura dei nodi) l'algoritmo per la valutazione di ogni operazione, la modalità di accesso ad ogni relazione utilizzata, eventuali fasi di ordinamento inserite e come i risultati delle operazioni intermedie sono passate alle operazioni successive (materializzazione piuttosto che pipelining, discusse nel seguito della trattazione). La determinazione del piano fisico corrispondente al piano logico selezionato avviene mediante l'*enumerazione* e la *stima del costo* dei possibili piani fisici (rappresentati nella Figura 7.10 come  $P_1, P_2, \dots$ , ciascuno di costo  $C_i$ ,  $i = 1, 2, \dots$ ). Tale stima si basa sulla *stima delle dimensioni dei risultati* delle varie operazioni, elaborata sulla base di *statistiche* opportunamente mantenute dal DBMS.

Quando un'interrogazione è composta da diversi operatori, il risultato di un operatore può essere *passato in cascata* (o in *pipelining*) all'operatore successivo, senza creare una relazione temporanea in cui scrivere il risultato intermedio. Se invece l'output di un operatore viene salvato in una relazione temporanea si dice che tale risultato è *materializzato*. Il pipelining del risultato di un operatore all'operatore successivo permette di risparmiare il costo di scrivere il risultato intermedio e di rileggerlo successivamente. Poiché tale costo può essere significativo, il pipelining è preferibile alla materializzazione se l'algoritmo per l'operatore lo permette. Se, ad esempio, si deve effettuare il join naturale di tre relazioni  $(A \bowtie B) \bowtie C$

Statistica	Significato
$T(R)$	numero di tuple nella relazione $R$
$B(R)$	numero di blocchi della relazione $R$
$S(R)$	dimensione di una tupla della relazione $R$ in byte (per tuple a lunghezza fissa, altrimenti valori medi)
$S(A, R)$	dimensione dell'attributo $A$ nella relazione $R$
$V(A, R)$	numero di valori distinti per l'attributo $A$ nella relazione $R$
$\text{Max}(A, R)$ e $\text{Min}(A, R)$	valori minimo e massimo dell'attributo $A$ nella relazione $R$
$K(I)$	numero di entrate (valori di chiave) dell'indice $I$
$L(I)$	numero di blocchi foglia dell'indice $I$
$H(I)$	altezza dell'indice $I$

Tabella 7.2: Principali statistiche

si può effettuare il pipelining del risultato del primo join con il secondo se non appena si ottiene una tupla di  $A \bowtie B$  la si utilizza per effettuare il join con  $C$ . Tale approccio è possibile solo con alcune tra le tecniche che vedremo nel Paragrafo 7.4.3.4. Il pipelining ha il grande vantaggio di non richiedere la scrittura del risultato di  $A \bowtie B$  in un file temporaneo perché le tuple di questa relazione vengono prodotte e direttamente consumate un blocco per volta.

#### 7.4.2 Statistiche

La strategia di esecuzione scelta dipende dalla dimensione di ogni relazione e dalla distribuzione dei valori nei vari attributi. Per selezionare la strategia si utilizzano stime del costo di esecuzione, basate su *profili* delle relazioni. Il DBMS mantiene quindi nei cataloghi di sistema alcune informazioni statistiche sui dati contenuti nelle relazioni. La Tabella 7.2 riassume le principali statistiche memorizzate per ogni relazione  $R$  ed indice  $I$ . Le statistiche sono aggiornate in seguito al popolamento iniziale delle relazioni od alla creazione di un indice (ma l'utente può richiedere esplicitamente il loro aggiornamento mediante un opportuno comando, ad esempio `ANALYZE` in Oracle, `UPDATE STATISTICS` in Informix). Da tali profili si ottengono i profili delle relazioni temporanee ottenute applicando operazioni algebriche alle relazioni di base. Ad esempio, il prodotto cartesiano  $R \times S$  contiene  $T(R) * T(S)$  tuple di dimensione  $S(R) + S(S)$ .

#### 7.4.3 Realizzazione degli operatori relazionali

Essendo gli operatori dell'algebra relazionale degli operatori ad alto livello, di tali operatori sono in generale possibili diverse realizzazioni. La scelta di quale realizzazione utilizzare dipende dalle caratteristiche dei dati e dall'espressione complessiva in cui l'operatore è contenuto. Viene ovviamente utilizzata la realizzazione che comporta il minor costo di esecuzione per l'interrogazione. Il costo delle varie operazioni viene tipicamente stimato in termini di operazioni di I/O effettuate

(lettura e scritture di blocco), senza considerare il costo di scrittura dell'output. Questo perché, in primo luogo, essendo la dimensione del risultato indipendente dalla strategia utilizzata per realizzare l'operatore, la scrittura del risultato ha lo stesso costo per tutte le tecniche. Inoltre, non è detto che il risultato venga effettivamente scritto su disco, perché, come discusso in precedenza, potrebbe essere passato in cascata all'operatore successivo.

Le tecniche che si utilizzano per sviluppare gli algoritmi per implementare i vari operatori sono essenzialmente di tre tipi:

- **Iterazione.** Le tuple della relazione di input vengono esaminate sequenzialmente (scansione sequenziale).
- **Uso di indici.** Se è specificata una selezione oppure una condizione di join, mediante un indice<sup>7</sup> vengono ritrovate direttamente ed esaminate solo le tuple che soddisfano la condizione.
- **Partizionamento.** Partizionando le tuple in base ad una chiave, un'operazione può essere decomposta in operazioni meno costose sulle partizioni. Tecniche basate sul partizionamento sono ad esempio quelle che fanno uso di *ordinamento* o *hashing* delle tuple.

Un *cammino di accesso* ci permette di descrivere un modo di ritrovare le tuple di una relazione. Un cammino di accesso è (*i*) una scansione sequenziale oppure (*ii*) un indice ed una condizione di selezione (detta *predicato di ricerca*) corrispondente, cioè tali che il predicato coinvolge attributi appartenenti alla chiave dell'indice e l'indice permette di determinare direttamente le tuple che soddisfano la condizione (ed è quindi utile per ridurre il costo della verifica della condizione). Ad esempio, se la condizione è  $C + D = 100$  oppure  $D \neq 20$ , indici su  $C$  e/o  $D$  non aiutano a limitare il numero di tuple a cui accedere. La scansione sequenziale è sempre un possibile cammino di accesso. Se l'operazione è una selezione o un join, ed esistono gli indici corrispondenti, si possono avere diversi cammini di accesso. Il costo di un cammino di accesso è il numero di blocchi visitati (sia dell'indice che dei dati) se viene utilizzato tale cammino di accesso per ritrovare le tuple desiderate.

In quanto segue, discutiamo brevemente l'ordinamento dei dati su disco e le tecniche per realizzare gli operatori di selezione, proiezione e join. Per quanto riguarda le altre operazioni, l'intersezione ed il prodotto cartesiano sono casi speciali di join. L'unione e la differenza sono simili e vengono realizzate mediante tecniche di partizionamento analoghe a quelle per l'eliminazione dei duplicati nella proiezione, che discuteremo nel seguito della trattazione. Tecniche analoghe sono utilizzate per l'operazione di raggruppamento, combinata con un calcolo incrementale delle funzioni di gruppo.

---

<sup>7</sup>Se non esplicitamente indicato, nel discutere l'elaborazione di interrogazioni faremo riferimento ad indici ad albero non integrati, quali quelli della Figura 7.7.

#### 7.4.3.1 *Ordinamento di dati su disco*

Prima di discutere le principali tecniche per realizzare gli operatori relazionali consideriamo innanzitutto il problema di ordinare dati memorizzati su disco (*external sorting*). È spesso infatti necessario ordinare i dati, sia perché l'interrogazione da eseguire contiene una clausola **ORDER BY** sia perché avere i dati ordinati permette di eseguire altre operazioni in modo più efficiente. Ad esempio, la tecnica di merge join per l'esecuzione del join (vedi Paragrafo 7.4.3.4) è molto efficiente, ma richiede che le relazioni siano ordinate sull'attributo di join. Gli algoritmi di ordinamento classici non sono utilizzabili perché i dati da ordinare sono troppi per risiedere in memoria principale. I due approcci principalmente utilizzati sono il *merge sort esterno a due fasi* e l'*uso di indici ad albero*. Nel merge sort esterno a due fasi, si ordinano separatamente porzioni di dati di dimensione tale da essere contenute in memoria (prima fase); tali dati vengono salvati su disco in sotto-liste ordinate di dimensioni pari a quella della memoria principale. Si effettua quindi la fusione di tali sotto-liste (seconda fase), eventualmente in più passi se il numero di sotto-liste ottenuto non consente di fonderle tutte contemporaneamente. Nel caso, molto frequente in pratica, in cui bastino due fasi, il costo dell'ordinamento è pari a tre operazioni di I/O per ogni blocco che contiene la relazione  $R$  da ordinare, cioè  $3 * B(R)$ . Se sulla relazione da ordinare abbiamo un indice a  $B^+$ -albero sull'attributo su cui vogliamo ordinare, possiamo pensare di effettuare l'ordinamento attraversando i nodi foglia dell'indice. Se l'indice è clusterizzato, questa è una buona tecnica, che ha costo pari a  $L(I) + B(R)$ . Viceversa, se l'indice non è clusterizzato, può essere una pessima idea, poiché, attraverso l'indice, può succedere di tornare allo stesso blocco dati più volte, effettuando, nel caso peggiore,  $T(R)$  accessi.

#### 7.4.3.2 *Selezione*

Consideriamo una semplice selezione della forma  $\sigma_{A \ op \ v}(R)$ . Una possibile strategia per la sua esecuzione è la scansione sequenziale: viene scandita l'intera relazione, controllando la condizione su ogni tupla ed aggiungendo la tupla al risultato se la condizione è soddisfatta. Il costo è  $B(R)$  operazioni di I/O. Se non abbiamo indici ed il file non è ordinato, la scansione sequenziale è l'unica strategia a disposizione. Se invece abbiamo un indice sull'attributo  $A$  di selezione, allora possiamo utilizzare l'indice per trovare le entrate dell'indice che soddisfano la selezione e, attraverso tali entrate, accedere solo ai blocchi dati corrispondenti. Se l'indice è di tipo hash, allora può essere utilizzato solo se  $op$  è l'operatore di uguaglianza. Il costo dipende dal numero di tuple che soddisfano la condizione di selezione (vedremo nel Paragrafo 7.4.5.2 come può essere stimato) e dal fatto che l'indice sia o meno clusterizzato. In ogni caso, il costo è pari al costo di determinare le entrate dell'indice che soddisfano la condizione più il costo di accedere ai corrispondenti blocchi dei dati (tipicamente maggiore). Esattamente come nel caso dell'ordinamento, mentre per un indice clusterizzato ogni blocco dei dati viene visitato al più una volta, per gli indici non clusterizzati si può dover tornare sullo

stesso blocco più volte. È però possibile (ed importante) evitare di visitare lo stesso blocco più volte per lo stesso valore della chiave di ricerca. Vengono quindi ritrovate le entrate dell'indice che soddisfano la condizione ed i RID dei record dei dati da reperire vengono ordinati, in modo che i RID di record nello stesso blocco siano vicini. I record corrispondenti vengono quindi ritrovati in ordine.

Nel caso più generale di selezione occorre portare il predicato di selezione in forma normale congiuntiva (**AND** di **OR**) e determinare i congiunti che non contengono **OR** (detti *fattori booleani*). Gli elementi di questi congiunti se sono falsi rendono falsa tutta l'interrogazione. È quindi possibile accedere solo alle tuple della relazione che soddisfano tale condizione, senza dover esaminare le altre tuple.

**Esempio 7.10** Consideriamo l'interrogazione dell'Esempio 7.9. Tutti e tre i congiunti di tale interrogazione sono fattori booleani. Tutte le tuple del risultato, infatti, devono necessariamente essere realtive a film di Quentin Tarantino e a noleggi effettuati dopo il 15 Marzo 2006 e prima del 21 Marzo 2006. Consideriamo invece la stessa interrogazione in cui la condizione sul regista viene modificata richiedendo che il regista del film noleggiato sia Quentin Tarantino oppure il genere sia horror. La condizione di selezione sarebbe quindi:

$$(\text{regista} = 'quentin tarantino' \vee \text{genere} = 'horror') \wedge \\ \text{dataNol} \geq '15 - Mar - 2006' \wedge \text{dataNol} \leq '21 - Mar - 2006'.$$

Potranno ora esistere tuple del risultato corrispondenti a noleggi di film (horror) non di Quentin Tarantino, quindi il primo congiunto non è un fattore booleano per l'interrogazione.  $\square$

Nel caso ci siano più indici applicabili, sono possibili due approcci. È possibile decidere di utilizzare comunque un solo indice. In tal caso, si determina il costo dell'accesso utilizzando i vari indici separatamente (vedi Paragrafo 7.4.5.2), si sceglie l'indice con costo minimo, si accede alle tuple della relazione attraverso tale indice e sulle tuple si verificano le altre condizioni. Altrimenti, è possibile utilizzare tutti gli indici disponibili. In tal caso, si estraggono dalle foglie dell'indice i RID dei record che soddisfano la condizione, si effettua l'intersezione di tali insiemi di RID, a questo punto si ritrovano i record e si verifica la parte rimanente della condizione.

#### 7.4.3.3 Proiezione

Per implementare la proiezione bisogna rimuovere gli attributi che non compaiono nella proiezione ed eliminare i duplicati. Il secondo passo è quello più difficile e costoso e viene realizzato mediante tecniche di partizionamento (ordinamento o hashing). Una prima tecnica è basata sull'ordinamento: si accede sequenzialmente alla relazione per ottenere un insieme di tuple che contengono solo gli attributi specificati nella proiezione; si ordina questo insieme di tuple con uno degli algoritmi visti usando tutti gli attributi come chiave per l'ordinamento; si scandisce infine

il risultato ordinato, eliminando le tuple duplicate (che sono adiacenti). L'approccio può essere migliorato integrando l'operazione di eliminazione dei duplicati nell'algoritmo di ordinamento.

Un'altra possibile tecnica è basata sull'hashing. Supponiamo di avere  $B$  blocchi di buffer. In una prima fase, di partizionamento, la relazione viene letta usando un blocco di input. Per ogni tupla, vengono eliminati gli attributi non inclusi nella proiezione e viene applicata una funzione hash  $h_1$  su tutti gli attributi rimasti per scegliere una delle  $B - 1$  partizioni, memorizzate nei rimanenti  $B - 1$  blocchi di buffer. Si ottengono così  $B - 1$  partizioni di tuple contenenti solo gli attributi richiesti, tali che due tuple in partizioni diverse sono sicuramente distinte. Nella seconda fase, di eliminazione dei duplicati, si legge ogni partizione e si costruisce una tabella hash in memoria, usando una funzione  $h_2$  (diversa da  $h_1$ ) su tutti gli attributi. Se una tupla va a finire nello stesso bucket di una tupla esistente, si controlla che non sia un duplicato e, se lo è, la si elimina. Lo scopo di utilizzare  $h_2$  è di distribuire le tuple in una partizione in bucket diversi, in modo da minimizzare le collisioni. Se una partizione non sta in memoria, si può applicare ricorsivamente l'algoritmo alla partizione.

L'approccio basato sull'ordinamento si comporta meglio di quello basato sull'hashing se ci sono molti duplicati o se la distribuzione della funzione hash non è molto uniforme. Inoltre ha il side-effect di ordinare la relazione, che può essere utile per operazioni successive, ed è quindi in generale il più utilizzato. Se si deve proiettare solo su un attributo su cui si ha un indice (denso), si può accedere solo alle entrate dell'indice invece che al file dei dati, applicando così una strategia basata unicamente sulla scansione dell'indice, detta *index-only*.

#### 7.4.3.4 Join

Per effettuare il join  $R \bowtie S$  sono possibili diverse tecniche, che possono essere classificate come segue (supponiamo per semplicità che  $R$  ed  $S$  condividano un solo attributo  $A$ ):

- **Iterazione semplice.** Si accede ad una tupla di  $R$  (relazione *outer*) e la si confronta con ogni tupla di  $S$  (relazione *inner*); tale strategia richiede  $B(R) + T(R)*B(S)$  accessi, a meno che la relazione inner non sia abbastanza piccola da poter essere mantenuta in memoria principale per tutta l'esecuzione del join, nel qual caso il numero di accessi si riduce a  $B(R) + B(S)$ .
- **Iterazione orientata ai blocchi.** Si confronta ogni tupla del blocco correntemente in esame di  $R$  con tutte le tuple del blocco correntemente in esame di  $S$ . Tale strategia richiede  $B(R) + B(R)*B(S)$  accessi. Elaborando le relazione sulla base dei blocchi e non delle tuple è quindi possibile ridurre considerevolmente il numero di operazioni di I/O, se  $B(R) \ll T(R)$ , cioè la relazione outer è memorizzata in modo tale che uno stesso blocco contiene molte tuple della relazione.

- **Uso di indici.** Se una delle due relazioni ha un indice sull'attributo di join conviene renderla inner e sfruttare l'indice per determinare direttamente le tuple della relazione che corrispondono alla tupla della relazione outer in esame. Il costo in questo caso è, supponendo  $R$  la relazione outer,  $B(R) + V(A, R) * C(I_A, S)$  dove  $C(I_A, S)$  è il costo di accedere alle tuple della relazione  $S$  con un certo valore per l'attributo  $A$ , attraverso l'indice  $I_A$  sull'attributo  $A$  (vedi Paragrafo 7.4.5.2).
- **Merge join.** Se entrambe le relazioni sono ordinate sull'attributo di join e le tuple delle relazioni con lo stesso valore per l'attributo di join possono stare contemporaneamente in memoria, è possibile eseguire il join “scorrendo” contemporaneamente le due relazioni, accedendo un'unica volta ad ogni blocco delle due relazioni. Tale strategia richiede  $B(R) + B(S)$  accessi ed è quindi ottima. Poiché tale strategia è molto efficiente, se le relazioni non sono già ordinate, la fase di fusione dell'algoritmo di ordinamento può essere combinata con la strategia di join: si ottengono cioè le sotto-liste ordinate per entrambe le relazioni e si carica in memoria il primo blocco di ogni sotto-lista di ogni relazione. La fusione delle sotto-liste ed il controllo della condizione di join vengono effettuati contemporaneamente.
- **Hash join.** Nella prima fase, di *partizionamento*, si partizionano sia  $R$  che  $S$  usando una funzione hash  $h_1$ , in modo tale che le tuple di  $R$  nella partizione  $i$  possano corrispondere solo a tuple di  $S$  nella partizione  $i$ . Nella successiva fase, di *match*, si legge una partizione di  $R$  e ad ogni elemento si applica una funzione hash  $h_2$  (diversa da  $h_1$ ). Si scandisce quindi la corrispondente partizione di  $S$  cercando le tuple corrispondenti. Il costo totale (se non ci sono overflow delle partizioni) è  $3 * (B(R) + B(S))$ .

**Esempio 7.11** Consideriamo il join `Noleggio`  $\bowtie$  `Cliente`, supponendo di non aver effettuato il co-clustering delle relazioni discusso nell'Esempio 7.6. Supponiamo inoltre di avere le seguenti statistiche  $T(\text{Cliente}) = 2'000$ ,  $T(\text{Noleggio}) = 70'000$  ( $\simeq 200$  noleggi al giorno mantenendo i noleggi dell'ultimo anno) con le relazioni memorizzate in modo tale che  $B(\text{Cliente}) = 100$ ,  $B(\text{Noleggio}) = 3'500$ . Supponendo che le relazioni non siano ordinate e che quindi il merge join (che avrebbe costo 3'600) non sia direttamente applicabile, una possibilità è verificare l'esistenza di un indice (preferibilmente clusterizzato) su `Noleggio.codCli`, o (meno vantaggioso) su `Cliente.codCli`. Se tali indici non sono disponibili, le ulteriori possibilità sono : (i) iterazione orientata ai blocchi con `Cliente` come outer, di costo  $100 + 35'000 = 35'100$ ; (ii) iterazione orientata ai blocchi con `Noleggio` come outer, di costo  $3'500 + 35'000 = 38'300$  (da cui evinciamo che conviene scegliere la relazione di dimensione minore come outer); (iii) uso di hash, con costo  $3*3'600 \simeq 10'000$  accessi.  $\square$

Notiamo che, tra le tecniche viste, le tecniche di tipo iterativo e quelle basate sull'uso di indice ammettono il passaggio in cascata (pipelining) (vedi Paragrafo 7.4.1) delle tuple della relazione outer, che, nel caso corrisponda ad un'espressione,

$\sigma_{P_1 \wedge P_2}(e) \equiv \sigma_{P_2 \wedge P_1}(e) \equiv \sigma_{P_1}(\sigma_{P_2}(e))$	$a(P) \subseteq \{A_1, \dots, A_n\}$
$\Pi_{A_1, \dots, A_n}(\Pi_{B_1, \dots, B_m}(e)) \equiv \Pi_{A_1, \dots, A_n}(e)$	$a(P) \subseteq \{D_1, \dots, D_p\},$ $\{A_1, \dots, A_n\} \subseteq \{D_1, \dots, D_p\}$
$\Pi_{A_1, \dots, A_n}(\sigma_P(e)) \equiv \sigma_P(\Pi_{A_1, \dots, A_n}(e))$	$a(P) \subseteq a(e_1)$
$\Pi_{A_1, \dots, A_n}(\sigma_P(e)) \equiv \Pi_{A_1, \dots, A_n}(\sigma_P(\Pi_{D_1, \dots, D_p}(e)))$	$\{B_1, \dots, B_m\} = \{A_1, \dots, A_n\} \cap a(e_1)$ $\{C_1, \dots, C_k\} = \{A_1, \dots, A_n\} \cap a(e_2)$
$\sigma_P(e_1 \times e_2) \equiv \sigma_P(e_1) \times e_2$	
$\Pi_{A_1, \dots, A_n}(e_1 \times e_2) \equiv \Pi_{B_1, \dots, B_m}(e_1) \times \Pi_{C_1, \dots, C_k}(e_2)$	
$\sigma_P(e_1 \cup e_2) \equiv \sigma_P(e_1) \cup \sigma_P(e_2)$	
$\Pi_{A_1, \dots, A_n}(e_1 \cup e_2) \equiv \Pi_{A_1, \dots, A_n}(e_1) \cup \Pi_{A_1, \dots, A_n}(e_2)$	

Figura 7.11: Alcune equivalenze algebriche

non deve quindi necessariamente essere materializzata. Viceversa, merge join e hash join richiedono di avere a disposizione le intere relazioni argomento.

#### 7.4.4 Ottimizzazione logica

L'ottimizzazione logica è la fase dell'ottimizzazione in cui il piano logico di esecuzione *LQP* viene ottimizzato, producendo un nuovo piano che si presume più efficiente, senza considerare informazioni relative all'organizzazione fisica dei dati. Tale fase è basata sull'utilizzo di regole di equivalenza algebrica quali quelle nella Figura 7.11. Nella figura,  $a(P)$  ed  $a(e)$  denotano gli attributi nel predicato  $P$  e nell'espressione  $e$ , rispettivamente. Tali regole vengono applicate per trasformare, in base a delle euristiche, il piano logico di esecuzione dell'interrogazione in un piano logico ottimizzato che presumiamo sia più efficiente da eseguire. Un'alternativa sarebbe quella di considerare diversi piani logici equivalenti e stimare il costo dei diversi piani fisici associati ai vari piani logici considerati. In genere questa alternativa non viene utilizzata per limitare il numero di piani da valutare. L'ordine di esecuzione dei join, invece, viene deciso nella fase successiva, sulla base delle informazioni relative alla dimensione delle relazioni ed alla valutazione del costo dei diversi ordini di esecuzione.

Le principali euristiche su cui si basa l'ottimizzazione logica sono basate sul principio di ridurre la dimensione dei risultati intermedi. In particolare, si possono evidenziare le seguenti euristiche: *(i)* eseguire le operazioni di selezione il più presto possibile; *(ii)* trasformare espressioni della forma  $\sigma_{P_1 \wedge P_2}(e)$  in espressioni della forma  $\sigma_{P_1}(\sigma_{P_2}(e))$  in modo da poter poi anticipare le selezioni su  $P_1$  e  $P_2$ ; *(iii)* eseguire le operazioni di proiezione il più presto possibile; *(iv)* introdurre eventualmente ulteriori proiezioni nell'espressione, proiettando ad ogni passo solo sugli attributi che appaiono nel risultato dell'interrogazione o sono necessari in operazioni successive.

Trattandosi di euristiche, alcune interrogazioni potrebbero essere eseguite più efficientemente con strategie diverse. Consideriamo, ad esempio, l'espressione algebrica  $\Pi_A(\sigma_{B=v}(R))$ . L'eistica *(iv)* la trasformerebbe in  $\Pi_A(\sigma_{B=v}(\Pi_{A,B}(R)))$ . Se però è presente un indice su  $B$  per  $R$ , applicare la proiezione a tutta la re-

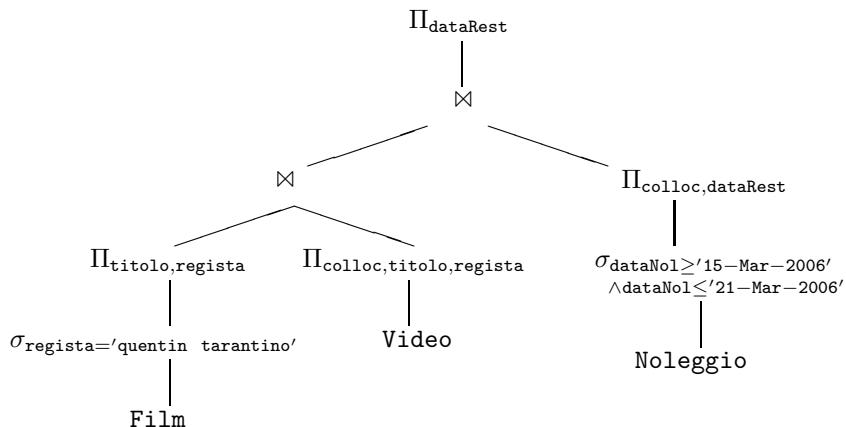


Figura 7.12: Piano logico di esecuzione ottimizzato

lazione costituisce un peggioramento rispetto al ritrovare tramite l'indice le tuple che soddisfano la condizione  $B = v$  e poi applicare solo a queste la proiezione.

**Esempio 7.12** Consideriamo l'interrogazione dell'Esempio 7.9. Un'espressione algebrica ottimizzata per tale interrogazione è presentata nella Figura 7.12. Notiamo che, in accordo all'euristica (iv) discussa in precedenza, sono state introdotte delle proiezioni intermedie, mantenendo ad ogni passo solo gli attributi che appaiono nel risultato dell'interrogazione o sono necessari in operazioni successive. Tali proiezioni, per le considerazioni fatte in precedenza, in effetti potrebbero non essere utili per ridurre il costo di elaborazione dell'interrogazione. Ricordiamo inoltre che, come discusso in precedenza, non è stato ancora fissato un ordine di esecuzione per i join.  $\square$

#### 7.4.5 Ottimizzazione fisica

Lo scopo di questa fase è trasformare il piano logico di esecuzione, selezionato nella fase di ottimizzazione logica, in un piano fisico. Questo in genere avviene considerando diversi possibili piani fisici che realizzano il piano logico scelto, valutando il costo di ognuno di essi e scegliendo il piano fisico di minor costo. Per stimare il costo di un piano di esecuzione, per ogni nodo nell'albero si stima il costo di effettuare l'operazione corrispondente, si stima la dimensione del risultato (che sarà l'input di operatori successivi) e si determina se è ordinato.

Ricordiamo che, dato un piano logico, i corrispondenti piani fisici sono ottenuti fissando, per ogni piano logico:

- un ordine di esecuzione per le operazioni associative e commutative (join, unione, intersezione);

$$\begin{aligned}
 F(A = v) &= 1/V(A, R) \\
 F(A > v) &= (Max(A, R) - v)/(Max(A, R) - Min(A, R)) \\
 F(A < v) &= (v - Min(A, R))/(Max(A, R) - Min(A, R)) \\
 F(A \text{ IN } (v_1, \dots, v_n)) &= n * F(A) \\
 F(A \text{ BETWEEN } v_1 \text{ AND } v_2) &= (v_2 - v_1)/(Max(A, R) - Min(A, R)) \\
 F(A_1 = A_2) &= 1/max(V(A_1, R), V(A_2, R)) \\
 F(C_1 \wedge C_2) &= F(C_1) * F(C_2) \\
 F(C_1 \vee C_2) &= F(C_1) + F(C_2) - F(C_1) * F(C_2) \\
 F(\neg C) &= 1 - F(C)
 \end{aligned}$$

Figura 7.13: Stima della selettività di un predicato

- una tecnica di realizzazione (algoritmo) per ogni operatore nel piano logico;
- operazioni addizionali (ad esempio, ordinamento) che non sono indicate nel piano logico;
- il modo in cui i risultati intermedi sono passati da un operatore al successivo (cioè materializzati o in pipelining).

Nel seguito vedremo innanzitutto come è possibile stimare la dimensione del risultato per poi discutere come i piani fisici vengono enumerati e selezionati.

#### 7.4.5.1 Stima della dimensione del risultato

Poiché il costo delle operazioni dipende dalla dimensione delle relazioni in input, per stimare il costo di un piano di esecuzione è necessario stimare le dimensioni delle relazioni intermedie, quindi stimare la dimensione del risultato di un'espresione algebrica. Tale stima viene effettuata, a seconda dell'operatore applicato, come segue.

**Selezione.** Data una selezione  $\sigma_P(R)$ , il rapporto tra le tuple di  $R$  che soddisfano  $P$  e tutte le tuple di  $R$  viene chiamato *fattore di selettività* del predicato  $P$  ed indicato con  $F(P)$ . È possibile stimare il fattore di selettività assumendo l'uniformità di distribuzione dei valori di ogni attributo, cioè sotto l'ipotesi che ogni valore appaia con la stessa probabilità. Dato  $F(P)$ , stimiamo che  $\sigma_P(R)$  seleziona un numero di tuple pari a  $T(R) * F(P)$ .

Nell'ipotesi di equiprobabilità dei valori, possono essere effettuate le stime riportate nella Figura 7.13. Tali stime si basano, oltre che sull'assunzione che tutti i valori siano egualmente probabili, anche sull'ipotesi che non vi siano correlazioni tra i valori di attributi diversi. In alcune situazioni non è però realistico assumere l'equiprobabilità dei valori. Ad esempio, nei fine settimana vengono effettuati più noleggi che nei giorni lavorativi. Quindi, per l'attributo `dataNol` della relazione `Noleggio` i valori corrispondenti a date il cui giorno della settimana è sabato sono

più probabili di valori corrispondenti a date il cui giorno della settimana è, ad esempio, lunedì. Allo stesso modo, è probabile che un video che contiene un film appena uscito sia noleggiato più frequentemente. Tuttavia, entrambe le ipotesi sono buone approssimazioni in molti casi e quindi vengono utilizzate nella pratica. Solo recentemente sono state sviluppate tecniche più sofisticate basate sul mantenimento di statistiche più dettagliate quali *istogrammi* dei valori di un attributo. Per approssimare in modo più accurato la distribuzione dei valori di un attributo vengono mantenute più informazioni che non il numero di valori assunti, il valore minimo ed il valore massimo. In particolare, l'intervallo dei valori assunti dall'attributo viene diviso in sotto-intervalli, detti bucket, e per ogni bucket si memorizza quante sono le tuple della relazione con valori dell'attributo in quel bucket. La distribuzione dei valori all'interno del bucket viene poi assunta uniforme. Sono possibili due diversi modi di determinare i bucket: *equi-width* (i sotto-intervalli contengono tutti lo stesso numero di valori dell'attributo) e *equi-depth* (i sotto-intervalli contengono tutti lo stesso numero di tuple), che consentono in genere stime più accurate. A volte vengono anche mantenuti i valori più frequenti ed il loro numero di occorrenze.

**Join.** Il prodotto cartesiano  $R \times S$  ha dimensione  $T(R)*T(S)*(S(R)+S(S))$ . La dimensione del risultato di un join viene stimata considerandolo come un prodotto cartesiano seguito da una selezione. Per quanto riguarda, invece, il join naturale, se  $U_R \cap U_S$ <sup>8</sup> è una chiave per  $S$ , allora  $T(R \bowtie S) \leq T(R)$ . Altrimenti, supponendo  $U_R \cap U_S = \{A\}$ ,  $T(R \bowtie S) = T(R) * T(S) / \max(V(A, R), V(A, S))$  che si generalizza al caso di più attributi di join come  $T(R \bowtie S) = T(R) * T(S) / \prod_{A_i \in U(R) \cap U(S)} \max(V(A_i, R), V(A_i, S))$ .

**Proiezione.** La proiezione  $\Pi_{A_1, \dots, A_n}(R)$  produce tuple di dimensione  $S(A_1, R) + \dots + S(A_n, R)$ . Il numero di tuple restituite è al massimo  $T(R)$  (se ad esempio gli attributi di proiezione includono la chiave) e dipende dal numero di duplicati. Il numero di tuple restituito viene stimato come il minimo tra  $T(R)/2$  e  $\prod_{i=1}^n V(A_i, R)$ .

**Altre operazioni.** Per le altre operazioni non è facile stimare la cardinalità del risultato. Per l'unione, ad esempio, si considera in genere la media tra  $T(R) + T(S)$ , corrispondente al caso  $R$  ed  $S$  disgiunte, e  $\max(T(R), T(S))$ , corrispondente al caso  $R \subseteq S$ , o viceversa. Analogamente per l'intersezione e la differenza si considerano, rispettivamente, la media tra 0 e  $\min(T(R), T(S))$  e la media tra  $T(R)$  e  $T(R) - T(S)$ .

**Esempio 7.13** Date le seguenti statistiche:  $T(\text{Film}) = 1'000$ ,  $T(\text{Video}) = 3'000$ ,  $T(\text{Noleggio}) = 70'000$ ,  $V(\text{regista}, \text{Film}) = 300$ ,  $\max(\text{dataNol}, \text{Noleggio}) - \min(\text{dataNol}, \text{Noleggio}) = 350$ , l'interrogazione dell'Esempio 7.9 sarebbe stimata

---

<sup>8</sup>Ricordiamo che  $U_R$  denota l'insieme degli attributi della relazione  $R$  (vedi Capitolo 2).

contenere  $T(\text{Noleggio}) * F(\text{regista} = \text{'quentin tarantino'}, \text{Film}) * F(\text{dataNol} \geq \text{'15-Mar-2006'} \wedge \text{dataNol} \leq \text{'21-Mar-2006'}, \text{Noleggio}) = 700'000 * (1/300) * (7/350) \simeq 5 \text{ tuple.}$   $\square$

#### 7.4.5.2 Enumerazione e scelta dei piani

Data un'interrogazione, nella fase di ottimizzazione fisica dovrebbero essere considerati tutti i possibili piani per eseguirla, valutando il costo di ogni piano, per poi selezionare quello di costo minimo. Abbiamo già detto che, in realtà, in fase di ottimizzazione logica si utilizzano alcune euristiche per non dover valutare tutti i possibili piani logici. In quanto segue discutiamo brevemente quali piani vengono valutati nella fase di ottimizzazione fisica e come viene determinato il loro costo, esaminando dapprima il caso di accesso a singola relazione e poi quello del join.

**Accesso a singola relazione.** Nel caso di accesso a singola relazione, bisogna scegliere il cammino di accesso alla relazione. Un piano possibile è la scansione sequenziale, che ha sempre costo  $B(R)$ . Abbiamo visto che altri possibili cammini di accesso sono attraverso predicati di ricerca che siano fattori booleani della condizione di selezione. Se supponiamo di utilizzare un solo indice per volta, per selezionare i cammini di accesso si determinano, dalla condizione di selezione, i predicati di ricerca che sono anche fattori booleani e per ognuno di essi si valuta il costo di accesso con l'impiego dell'indice. Viene così determinato il cammino di accesso di costo minimo. Il costo di accesso viene valutato come somma del costo di accesso all'indice e del costo di accesso ai dati. Il costo di accesso all'indice, se la selettività del predicato è  $F(P)$ , e trascurando il costo di accedere ai nodi intermedi dell'indice, è  $F(P) * L(I)$ . Il costo di accesso ai dati tramite l'indice, invece, varia sensibilmente a seconda che l'indice sia o meno clusterizzato. Se l'indice è clusterizzato, i RID delle entrate dell'indice puntano a collezioni contigue di record ed ogni blocco del file dei dati viene acceduto al più una volta. Si ha quindi un costo pari a  $F(P) * B(R)$ . Se l'indice non è clusterizzato, invece, ogni entrata potrebbe contenere RID che puntano a blocchi dati distinti, portando a tante operazioni di I/O quante sono le entrate a cui si accede. Il costo risulta quindi  $F(P) * V(A, R) * B_L$ , dove  $B_L$  è il numero di blocchi acceduti per un certo valore di chiave, che potrebbe quindi essere, come caso limite,  $T(R)/V(A, R)$ , cioè tanti quanti sono i record con tale valore di chiave, se questo non supera  $B(R)$ . Nella pratica esistono funzioni per stimare in modo più accurato tali accessi.

Il costo totale di accesso può comunque essere in prima approssimazione stimato come segue:

- se l'indice  $I$  è clusterizzato, il costo è  $F(P) * (L(I) + B(R))$ ;
- se l'indice non è clusterizzato, il costo è sicuramente inferiore al minimo tra  $F(P) * (L(I) + V(A, R) * B(R))$  e  $F(P) * (L(I) + T(R))$ .

Dobbiamo infine considerare se il cammino di accesso produce o meno tuple ordinate.

**Esempio 7.14** Consideriamo l’interrogazione dell’Esempio 7.9 e supponiamo di avere un indice ad albero su `regista` per la relazione `Film`. Poiché la selettività del predicato è molto alta ( $1/300$ ), l’accesso alla relazione attraverso tale indice, indipendentemente dal fatto che l’indice sia clusterizzato o meno, risulta probabilmente vantaggiosa rispetto alla scansione sequenziale. Poiché stimiamo ci siano meno di quattro film di Quentin Tarantino, recuperare i dati di tali film costa al più 1 accesso all’indice e 4 accessi ai dati (2 accessi in totale nel caso di indice clusterizzato). Se  $B(\text{Film}) \geq 5$  l’uso dell’indice risulta quindi vantaggioso.  $\square$

**Join.** Nel determinare i piani per l’esecuzione di un’interrogazione che coinvolge più relazioni collegate tramite join, un aspetto importante è l’ordine di esecuzione dei join. Tale ordine, insieme alla tecnica utilizzata per eseguire ogni join, determina il piano di esecuzione. Nella rappresentazione ad albero del piano, si utilizza la convenzione che il figlio sinistro corrisponda alla relazione outer ed il figlio destro alla relazione inner. Poiché considerare tutte le possibili alternative può essere costoso, una soluzione comune è concentrarsi su piani detti *left-deep*. In un albero left-deep il figlio destro di ogni nodo etichettato con un join è una relazione di base. La motivazione per considerare solo alberi left-deep è che tali alberi permettono di generare piani di esecuzione *fully-pipelined*, cioè in cui i risultati di tutte le relazioni intermedie sono passati in cascata agli operatori successivi, senza essere materializzati. Le relazioni inner devono comunque essere materializzate, quindi un piano in cui la relazione inner sia il risultato di un join forza a materializzare il risultato del join. Per restringere il numero di piani da considerare, inoltre, non vengono considerate permutazioni che implicano prodotti cartesiani, limitandosi alle sequenze che danno luogo a join tra relazioni che hanno attributi in comune.

**Esempio 7.15** Consideriamo l’interrogazione dell’Esempio 7.9. In base alle euristiche discusse, gli ordini di esecuzione del join che vengono considerati sono quelli corrispondenti alle seguenti espressioni algebriche:

- $(\text{Noleggio} \bowtie \text{Video}) \bowtie \text{Film}$ ;
- $(\text{Video} \bowtie \text{Noleggio}) \bowtie \text{Film}$ ;
- $(\text{Film} \bowtie \text{Video}) \bowtie \text{Noleggio}$ ;
- $(\text{Video} \bowtie \text{Film}) \bowtie \text{Noleggio}$ .

$\square$

L’approccio seguito consiste nel determinare una soluzione ottima per ogni sottoinsieme dei join dell’interrogazione e poi determinare il modo migliore per eseguire il join della relazione ottenuta con un’ulteriore relazione. Ad esempio, in riferimento all’interrogazione dell’Esempio 7.9, una volta determinata la soluzione per il join `Film`  $\bowtie$  `Video` si deve determinare il modo di effettuare il join di tale relazione con `Noleggio`.

Una soluzione è costituita da: una lista ordinata di relazioni su cui si esegue il join, il metodo di join usato (ad esempio, iterazione o merge join), un piano

indicante come ogni relazione deve essere acceduta (indice o scansione sequenziale), un'indicazione sulla necessità di ordinamento delle relazioni prima dell'esecuzione del join.

Un aspetto importante riguarda l'ordinamento delle tuple nelle relazioni. Definiamo *ordinamento interessante* un ordinamento su attributi che compaiono in clausole `GROUP BY` e `ORDER BY` e sugli attributi di join. Tale nozione è importante perché un piano parziale non ottimale, ma che ritrova le tuple ordinate rispetto ad un ordinamento interessante, potrebbe poi portare ad un piano globale ottimale.

Un algoritmo iterativo produce la soluzione tramite iterazione sul numero di relazioni. Al primo passo, si determina il modo più efficiente per accedere ad ogni singola relazione per ogni ordinamento interessante e per il caso non ordinato. Al secondo passo, si determina il modo migliore per eseguire il join di ogni singola relazione con ogni altra relazione, ma alcune combinazioni non sono valutate in base all'euristica di ritardare l'esecuzione del prodotto cartesiano. Il secondo passo produce quindi soluzioni per eseguire i join di coppie di relazioni. Per ogni coppia di relazioni vengono mantenute le soluzioni di costo minimo per ogni ordinamento interessante e per il caso non ordinato, mentre le altre soluzioni vengono scartate. Al terzo passo si determina il modo migliore per eseguire i join di tre relazioni, considerando ogni relazione ottenuta al passo precedente, e determinando il modo migliore per collegarla con ogni altra relazione (sempre limitandosi a piani left-deep ed utilizzando l'euristica del prodotto cartesiano). Si procede in questo modo fino a che tutte le relazioni sono state considerate.

**Esempio 7.16** Consideriamo nuovamente l'interrogazione dell'Esempio 7.9. Esaminiamo innanzitutto l'accesso alle singole relazioni. In particolare:

- Per `Film` vengono considerati i seguenti cammini di accesso: scansione sequenziale; uso di indice su `regista`, se presente; uso di indice su `(titolo, regista)`, se presente. L'ordinamento su `(titolo,regista)` è interessante.
- Per `Video` vengono considerati i seguenti cammini di accesso: scansione sequenziale; uso di indice su `colloc`, se presente; uso di indice su `(titolo, regista)`, se presente. Gli ordinamenti su `(titolo,regista)` e `colloc` sono interessanti.
- Per `Noleggio` vengono considerati i seguenti cammini di accesso: scansione sequenziale; uso di indice su `dataNol`, se presente e ad albero (notiamo che la condizione corrispondente è di intervallo e non di uguaglianza); uso di indice su `colloc`, se presente. L'ordinamento su `colloc` è interessante.

A questo punto vengono esaminati i join di `Film` con `Video` e di `Video` con `Noleggio` secondo le varie strategie viste e vengono determinate le soluzioni di costo minimo per il caso non ordinato e per i casi che corrispondono ad ordinamenti interessanti. Consideriamo infine il join con la relazione mancante ottenendo quindi i piani completi di esecuzione.  $\square$

### 7.5 Progettazione fisica

Nella progettazione fisica della base di dati, dallo schema logico della base di dati ne viene prodotto uno schema fisico, cioè un insieme di scelte sulle organizzazioni (primarie e secondarie) dei dati. Il primo passo per effettuare la fase di progettazione fisica è capire il *carico di lavoro*. Il carico di lavoro emerso dall'analisi dei requisiti e già considerato nella fase di progettazione logica (vedi Capitoli 5 e 6) viene ulteriormente dettagliato. In particolare, dobbiamo determinare quali sono le interrogazioni e gli aggiornamenti più importanti e con che frequenza vengono eseguiti. Vanno inoltre considerate le prestazioni desiderate o necessarie per tali interrogazioni ed aggiornamenti.

A questo punto, per ogni interrogazione nel carico di lavoro dobbiamo considerare a quali relazioni accede, quali attributi ritrova, quali attributi sono coinvolti in condizioni di selezione e join e quanto selettive sono queste condizioni. Analogamente, per ogni aggiornamento nel carico di lavoro dobbiamo considerare quali attributi sono coinvolti in condizioni di selezione e join, quanto selettive sono queste condizioni, il tipo di aggiornamento (inserimento, cancellazione o modifica) e gli attributi che saranno coinvolti nell'aggiornamento.

Le decisioni da prendere sono relative a quali indici creare, cioè su quali relazioni e su quali attributi. Inoltre, per ogni indice, bisogna stabilire di che tipo deve essere, in particolare se deve essere o meno clusterizzato e se deve essere ad albero piuttosto che hash. Per la scelta degli indici, un possibile approccio è considerare una alla volta le interrogazioni più importanti e considerare il piano di esecuzione migliore utilizzando gli indici attualmente disponibili. Se un indice addizionale permette di generare un piano di esecuzione migliore, tale indice viene aggiunto. Prima di creare un indice, però, bisogna anche considerare l'impatto degli aggiornamenti sul carico di lavoro. Ricordiamo infatti (vedi Paragrafo 7.3) che gli indici possono velocizzare le interrogazioni<sup>9</sup> ma rallentare gli aggiornamenti.

Gli attributi che appaiono in una clausola **WHERE** sono buoni candidati come chiavi per un indice. Se la condizione è di uguaglianza, può essere preferibile un indice hash. Se la condizione è di tipo intervallo, è preferibile un indice ad albero. Il clustering su tale attributo è particolarmente utile in caso di interrogazioni di tipo intervallo, ma può essere utile anche per l'uguaglianza se ci sono duplicati. In generale, dobbiamo cercare di scegliere indici che siano utilizzabili nel maggior numero possibile di interrogazioni. Inoltre dobbiamo considerare che un solo indice per relazione può essere clusterizzato; per sceglierlo vengono analizzate le interrogazioni importanti che trarrebbero maggior vantaggio dalla clusterizzazione. Notiamo inoltre che quasi tutti i DBMS allocano automaticamente un indice sugli attributi specificati come chiave primaria per la relazione, perché il DBMS stesso trae beneficio dalla presenza di tale indice nella verifica del vincolo di chiave e di eventuali chiavi esterne corrispondenti.

Se la clausola **WHERE** di un'interrogazione importante contiene diverse con-

---

<sup>9</sup>Un indice può velocizzare anche un aggiornamento se permette una valutazione efficiente della condizione nel corrispondente comando e quindi un accesso diretto alle tuple da aggiornare.

dizioni, si possono considerare anche indici con chiave multi-attributo. Se la selezione contiene una condizione di tipo intervallo, l'ordine degli attributi deve essere scelto con attenzione (vedi Paragrafo 7.3). Questo tipo di indici permette a volte di realizzare strategie basate solo sull'indice (index-only) per interrogazioni importanti. L'interrogazione può cioè essere eseguita accedendo solo alle entrate dell'indice e non alle entrate dei dati. Per le strategie basate solo sull'indice la clusterizzazione non è importante, perché non viene effettuato alcun accesso al file dei dati.

Quando si considera una condizione di join, un indice hash sulla relazione inner è molto utile per la strategia basata sull'utilizzo di indici. Tale indice dovrebbe essere clusterizzato se l'attributo di join non è una chiave per la relazione inner e le tuple di tale relazione devono essere effettivamente ritrovate (cioè non basta l'accesso all'indice per effettuare il join). Un indice clusterizzato di tipo B<sup>+</sup>-albero sull'attributo di join è invece utile per la strategia merge join.

**Esempio 7.17** Consideriamo lo schema logico ottimizzato per la base di dati della videoteca ottenuto nel Paragrafo 6.4.3 ed il carico di lavoro costituito dalle sei operazioni specificate nel Paragrafo 6.4. L'operazione 1 richiede la verifica dei vincoli di integrità associati alle relazioni **Video** e **Film**. Una verifica efficiente dei vincoli di chiave e chiave esterna beneficia dell'utilizzo di un indice sulla chiave primaria delle relazioni (**colloc** per **Video** e **codF** per **Film**). Un discorso analogo può essere effettuato per l'operazione 2.

L'operazione 3 richiede innanzitutto di verificare che il video non sia già in noleggio, tale controllo è facilitato da un indice su (**colloc**, **dataRest**) (non serve che sia clusterizzato perché ad ogni entrata dell'indice corrisponde una sola tupla). Per i controlli e gli aggiornamenti sulle relazioni **Standard** e **VIP**, è utile un indice su **codCli** (chiave primaria) su tali relazioni. Nel caso di azzeramento dei punti mancanti, l'operazione richiede di accedere contemporaneamente ai dati relativi allo stesso cliente nelle due relazioni, e quindi un join delle relazioni, per cui sarebbe utile ad esempio un co-clustering. Tale caso non è però molto frequente. L'operazione 4 è anch'essa agevolata da un indice su (**colloc**, **dataRest**), per ricercare il noleggio in corso, anche se richiede un aggiornamento di tale indice.

L'operazione 5 è agevolata da un indice ad albero su (**dataRest**, **dataNol**), clusterizzato. Tale operazione richiede inoltre di effettuare un join tra **Noleggio** e **Cliente**. Tale join può essere eseguito in maniera efficiente o mediante il co-clustering delle relazioni o mediante un indice (anche hash e non clusterizzato) su **codCli** in **Cliente**. L'operazione 6, infine, richiede di accedere a tutte le tuple della relazione **VIP**, quindi è agevolata dall'avere le tuple di tale relazione memorizzate fisicamente contigue.

Sulla base dell'analisi del carico di lavoro, e della frequenza delle operazioni, non viene effettuato il co-clustering di relazioni e vengono definiti indici (anche hash e non clusterizzati) sui seguenti attributi: **Cliente.codCli**, **VIP.codCli**, **Standard.codCli**, **Video.colloc**, **Film.codF**, **Noleggio.(colloc,dataRest)**. Un indice ad albero clusterizzato è definito su **Noleggio.(dataRest,dataNol)**. □

Si noti che la progettazione fisica, oltre agli aspetti discussi relativi all'organizzazione in cluster ed all'indicizzazione delle relazioni, comporta anche un'attività di *ottimizzazione dello schema logico*. In particolare, le esigenze di ottimizzazione e l'analisi del carico di lavoro possono portare a considerare schemi normalizzati alternativi (come visto nel Capitolo 6, nel decomporre una relazione ci sono varie alternative) o a “disfare” alcuni passi di decomposizione ed “accontentarsi” di uno schema non normalizzato, per migliorare le prestazioni di interrogazioni che coinvolgono attributi di diverse relazioni decomposte in precedenza (processo noto come *denormalizzazione*). La stessa motivazione, cioè il miglioramento di prestazioni di interrogazioni che richiedono l'accesso ad attributi di diverse relazioni, porta a riconsiderare accorpamenti di relazioni (analoghi agli accorpamenti di entità discussi nel Capitolo 6). In tale fase è possibile altresì riconsiderare partizionamenti, sia orizzontali sia verticali, di relazioni (analoghi ai partizionamenti di entità discussi nel Capitolo 6). Un partizionamento orizzontale corrisponde a partizionare una relazione in sotto-relazioni, ognuna contenente un sottoinsieme delle tuple della relazione originale. Ogni sotto-relazione può essere ottenuta mediante selezione dalla relazione originale, che, a sua volta, può essere ottenuta come unione delle sotto-relazioni. Un partizionamento verticale corrisponde a partizionare una relazione in sotto-relazioni, ognuna contenente un sottoinsieme degli attributi della relazione originale. Ogni sotto-relazione può essere ottenuta mediante proiezione dalla relazione originale, che, a sua volta, può essere ottenuta come join delle sotto-relazioni. Partizionamenti orizzontali e verticali possono essere utile per migliorare l'efficienza di interrogazioni che coinvolgono solo pochi attributi, o solo poche tuple, rispettivamente, della relazione.

**Esempio 7.18** In riferimento all'Esempio 7.17, dall'analisi del carico di lavoro appare abbastanza evidente l'utilità di un partizionamento orizzontale della relazione **Noleggio** sul valore dell'attributo **dataRest** per mantenere i noleggi in corso separati da quelli conclusi. Le due relazioni **NoleggioA(colloc,dataNol,codCli)** e **NoleggioC(colloc,dataNol,codCli,dataRest)** sostituiscono quindi **Noleggio**. L'operazione 3 richiede ora l'accesso a **NoleggioA** e sarebbe velocizzata da un indice su **colloc** (chiave primaria). Analogamente, l'operazione 4 sarebbe agevolata da un indice su **NoleggioA.colloc**, per ricercare il noleggio in corso, e richiede la cancellazione di una tupla da **NoleggioA** e l'inserimento della tupla corrispondente in **NoleggioC**. L'operazione 5 lavora anch'essa su **NoleggioA** e sarebbe agevolata da un indice ad albero su **dataNol**, clusterizzato (rimane invariata la necessità di effettuare il join con **Cliente**).

Il partizionamento orizzontale porta quindi miglioramenti significativi alle operazioni 3, 4 e 5. Oltre agli indici sulle altre relazioni determinati nell'Esercizio 7.17, vengono definiti indici su **NoleggioA.colloc** (anche hash e non clusterizzato), **NoleggioA.dataNol** ad albero clusterizzato. □

Un ultimo aspetto da considerare riguarda la riscrittura di interrogazioni eseguite frequentemente in modo che siano eseguite il più efficientemente possibile. Sottolineiamo, innanzitutto, che l'ottimizzazione è eseguita separatamente per ogni

interrogazione, pertanto è bene formulare interrogazioni complesse e non scomporle in tante interrogazioni più semplici perché ciò fa perdere i vantaggi dell'ottimizzazione. Inoltre, se un'interrogazione importante è più lenta di quel che ci aspettiamo, dobbiamo innanzitutto controllare che le statistiche siano aggiornate. A volte, poi, può succedere che il DBMS non stia eseguendo il piano che pensiamo; tipicamente questo succede, oltre che per una stima inesatta delle dimensioni, a causa di selezioni che coinvolgono valori nulli, espressioni aritmetiche o stringhe oppure che contengono condizioni in OR. Un'ulteriore causa può essere la mancanza di supporto nel sistema di caratteristiche quali strategie basate solo su indici o alcuni metodi di join. In tal caso, dobbiamo verificare quale piano viene effettivamente utilizzato (mediante gli opportuni comandi forniti dal DBMS o con l'utilizzo di strumenti ad hoc forniti dai vari sistemi) e rivedere la scelta degli indici o riscrivere l'interrogazione.

### Note conclusive

In questo capitolo abbiamo brevemente illustrato le principali problematiche collegate alle strutture di memorizzazione e di indicizzazione dei dati ed all'elaborazione di interrogazioni. Ci siamo limitati ad introdurre gli aspetti principali ed al solo modello relazionale. Una trattazione completa delle strutture di indicizzazione (ad albero ed hash) e delle relative operazioni, così come i dettagli degli algoritmi di realizzazione degli operatori relazionali e di ottimizzazione, esulano dagli obiettivi di questo libro.

Le strutture di memorizzazione dei dati e di indicizzazione nonché le tecniche di elaborazione delle interrogazioni discusse in questo capitolo costituiscono la base per la definizione di tecniche più sofisticate per dati più complessi. Tecniche di indicizzazione ed elaborazione ad hoc per basi di dati ad oggetti, per documenti XML, per immagini, per testi, per dati spaziali e temporali sono state proposte in letteratura [BOSD<sup>+</sup>97]. Allo stesso modo, le applicazioni di analisi dei dati (OLAP) hanno richiesto lo sviluppo di nuove tecniche di indicizzazione e di elaborazione. Infine, la necessità di gestire flussi continui (stream) di dati che devono essere processati in tempo reale, senza necessariamente essere memorizzati nella base di dati, modifica completamente lo scenario di elaborazione delle interrogazioni ed ha quindi richiesto lo sviluppo di nuove tecniche di esecuzione.

### Note bibliografiche

Data l'importanza delle strutture ausiliarie di accesso nelle basi di dati e dell'ottimizzazione di interrogazioni, a tali argomenti è dedicata una vasta letteratura scientifica ed esistono numerosi testi dedicati esclusivamente a questo argomento. Tra i libri di testo sulle basi di dati, particolarmente completi per quanto riguarda tali argomenti, sono il testo di Elmasri e Navathe [EN03], il testo di Garcia-Molina, Widom e Ullman [GMUW02] e quello di Ramakrishnan e Gehrke

[RG02]. Interamente dedicato agli aspetti architetturali dei DBMS è il testo di Albano [Alb01].

Un riferimento classico per gli alberi binari di ricerca e le organizzazioni hash, che dedica spazio anche alle tecniche per memoria secondaria ed alle tecniche di ordinamento esterno, è il testo di Knuth [Knu73]. Un ottimo riferimento per l'organizzazione di file, che tratta molte delle tecniche descritte in questo capitolo, è il testo di Salzberg [Sal88]. La gestione del buffer è trattata in modo approfondito nel testo di Gray e Reuter [GR93]. Una panoramica sull'esecuzione di interrogazioni, con ampia bibliografia sull'argomento, è fornita da Graefe in [Gra93]. Un ottimo riferimento per la progettazione fisica e il tuning è il testo di Bonnet e Shasha [BS02]. Per quanto riguarda infine i DBMS commerciali, per la sintassi di Oracle si può fare riferimento a [Ora05c].

## Esercizi

**7.1** Consideriamo lo schema relazionale dell'Esercizio 2.3:

```
Squadra(nomeS,città,sponsor,coloriSociali,allenatore)
Giocatore(nTessera,nomeSSquadra,numero, nome, cognome, annoN, ruolo)
Partita(idPartita,giornata,nomeSCasaSquadra,nomeSTrasfSquadra,goalCasa,goalTrasf)
Goal(idPartitaPartita,minuto,nTesseraGiocatore,autoGoal)
```

ed i seguenti profili delle relazioni:  $T(\text{Squadra}) = 12$ ,  $T(\text{Goal}) = 400$ ,  $T(\text{Partita}) = 150$ ,  $T(\text{Giocatore}) = 200$ ,  $V(\text{Giocatore}, \text{numero}) = 20$ ,  $V(\text{Giocatore}, \text{nome}) = 25$ ,  $V(\text{Giocatore}, \text{cognome}) = 190$ ,  $V(\text{Giocatore}, \text{annoN}) = 12$ ,  $V(\text{Giocatore}, \text{ruolo}) = 5$ ,  $V(\text{Goal}, \text{minuto}) = 80$ ,  $V(\text{Goal}, \text{nTessera}) = 120$ ,  $V(\text{Goal}, \text{idPartita}) = 120$ .

- Discutere la nozione di co-clustering di relazioni e fornire un esempio di interrogazione per cui il co-clustering sarebbe utile ed uno per cui sarebbe dannoso.
- Fissata la struttura di memorizzazione di cui al punto (a), mostrare (se ha senso) un esempio di indice per ognuna delle tipologie riassunte nella Tabella 7.1, evidenziando per ognuno il numero di entrate dell'indice ed il contenuto di una di tali entrate. Inoltre, per ognuno di tali indici, fornire un esempio di operazione velocizzata ed uno di operazione rallentata da tale indice.

**7.2** Consideriamo lo schema relazionale dell'Esercizio 2.1:

```
Ricetta(nomeR,genere,nPersone,tempoP,istruzioni)
Ingrediente(nomeI,nomeRRicetta,dose)
Menu(numM,occasione)
Contiene(numMMenu,nomeRRicetta,portata)
```

e la seguente interrogazione su tale schema:

```
SELECT portata
FROM Ricetta NATURAL JOIN Ingrediente NATURAL JOIN Contiene NATURAL JOIN Menu
WHERE occasione = 'natale' AND ingrediente = 'uova' AND tempoP < 30;
```

Consideriamo inoltre i seguenti profili delle relazioni:  $T(\text{Ricetta}) = 1'000$ ,  $B(\text{Ricetta}) = 50$ ,  $T(\text{Menu}) = 200$ ,  $B(\text{Menu}) = 10$ ,  $T(\text{Contiene}) = 800$ ,  $B(\text{Contiene}) = 40$ ,  $T(\text{Ingrediente}) = 12'000$ ,  $B(\text{Ingrediente}) = 600$ . Supponiamo infine che su tali relazioni siano definiti i seguenti indici:

- in **Ricetta**: su nomeR, tempoP e genere (clusterizzato);
- in **Menu**: su numM;

- in *Ingrediente*: su *nomeR* (clusterizzato) e su *nomeI*;
  - in *Contiene*, su *numM* (clusterizzato) e su *nomeR*.
- Identificare un piano logico di esecuzione che rifletta l'ordine delle operazioni che verrebbe prodotto dalla fase di ottimizzazione logica.
  - Discutere le strategie considerate e quelle scelte per l'accesso ad ogni singola relazione, supponendo che ci siano 10 valori per l'attributo *occasione*, che i valori dell'attributo *tempoP* siano nell'intervallo [10,600] e che ci siano 500 ingredienti diversi.
  - Stimare la cardinalità del risultato.
  - Elencare gli ordini di join (cioè l'ordine in cui viene effettuato il join di coppie di relazioni per produrre il risultato dell'interrogazione) considerati da un ottimizzatore.
  - L'aggiunta all'interrogazione delle clausole *DISTINCT* nel comando *SELECT, GROUP BY, ORDER BY* influenzerebbe i piani considerati?
  - Discutere se e come l'interrogazione potrebbe essere eseguita mediante la strategia di join basata sull'uso degli indici e mediante la strategia di merge join.
  - Discutere eventuali strategie di memorizzazione alternative delle relazioni che velocizzerebbero l'esecuzione dell'interrogazione.

**7.3** Consideriamo lo schema relazionale dell'Esercizio 2.2:

```

Cliente(numCC, nome, via, città, telefono, tipoCC, scadCC)
Albergo(telefonoA, nome, via, città, categoria)
Camera(telefonoAAlbergo, tipoC, numC, prezzo)
Prenotazione(numCCCliente, telefonoACamera, tipoCCamera, dataA, dataP, num)

```

ed i seguenti profili delle relazioni:  $T(\text{Cliente}) = 1'000$ ,  $B(\text{Cliente}) = 50$ ,  $T(\text{Albergo}) = 200$ ,  $B(\text{Albergo}) = 10$ ,  $T(\text{Camera}) = 500$ ,  $B(\text{Camera}) = 25$ ,  $T(\text{Prenotazione}) = 12'000$ ,  $B(\text{Prenotazione}) = 800$ . Supponiamo inoltre che su tali relazioni siano definiti i seguenti indici:

- in *Cliente*: su *numCC*;
  - in *Albergo*: su *telefonoA*, su *città* (clusterizzato) e su *categoria*;
  - in *Camera*: su *(telefonoA, tipoC)*;
  - in *Prenotazione*: su *numCC*, su *telefonoA* e su *dataA* (clusterizzato).
- Per ognuno degli indici precedenti, fornire un esempio di aggiornamento ed uno di interrogazione velocizzati e rallentati dalla presenza dell'indice. Fornire inoltre esempi di interrogazioni velocizzate dalla presenza dell'indice solo se l'indice è ad albero e solo se l'indice è clusterizzato.
  - Fornire un esempio di interrogazione velocizzata dal co-clustering delle relazioni.
  - Consideriamo ora la seguente interrogazione:

```

SELECT telA, numCC
FROM Albergo NATURAL JOIN Camere NATURAL JOIN Prenotazione
WHERE città = 'firenze' AND tipoC= 'doppia' AND 100 <= prezzo <= 200
      AND dataA = '15-Mar-2006';

```

- identificare un piano logico di esecuzione che rifletta l'ordine delle operazioni scelto da un ottimizzatore logico;
- supponiamo che le città degli alberghi siano 10, che i prezzi varino nell'intervallo [30,1'000], che le prenotazioni siano relative ad un arco temporale di un mese e che l'unico metodo di join a disposizione sia quello basato sull'uso degli indici:
  - per ogni relazione coinvolta stimare quante tuple sarebbero selezionate dalla relazione se tutti i predicati non di join venissero applicati prima di iniziare l'elaborazione del join;
  - sulla base della risposta alla domanda precedente, qual è l'ordine di join di minor costo stimato e qual è il suo costo?

## Capitolo 9

# Protezione dei dati

L'impiego sempre più diffuso delle basi di dati a supporto di funzioni di tipo gestionale ha rappresentato una condizione necessaria per la crescente automazione dei sistemi informativi, ma ha reso estremamente critico il problema della protezione dei dati da essi gestiti. In un ambiente di basi di dati, un danno al patrimonio informativo si ripercuote non solo sul singolo utente o sulla singola applicazione, ma investe l'intero sistema. In molte imprese ed organizzazioni, le informazioni sono così importanti che la loro, anche parziale, corruzione o distruzione può portare alla paralisi dell'intera struttura. La rapida diffusione di Internet ha ulteriormente acuito il problema della protezione, in quanto le informazioni residenti in una base di dati sono oggi potenzialmente accessibili da una vastissima comunità di utenti sparsa in tutto il mondo.

Solitamente, in una stessa base di dati sono memorizzate informazioni di differente importanza e sensibilità. Inoltre, i dati memorizzati sono condivisi tra più utenti con diverse responsabilità e privilegi. È indispensabile pertanto disporre di tecniche, strumenti e procedure di sicurezza volti sia a garantire l'affidabilità delle elaborazioni sia a proteggere le informazioni ed i programmi da intrusioni, modifiche, furti e divulgazioni non autorizzate. In effetti, un DBMS ed i suoi *meccanismi di sicurezza* devono rispondere ad una duplice esigenza: da un lato consentire la condivisione dei dati, dall'altro far sì che questa condivisione sia *selettiva* ed opportunamente regolamentata secondo le politiche dell'azienda od organizzazione.

Gli obiettivi della protezione dei dati sono principalmente tre: *riservatezza*, ovvero la protezione delle informazioni da letture non autorizzate (tale proprietà prende il nome di *privacy* quando si riferisce a dati di natura personale); *integrità*, ovvero la protezione dei dati da modifiche o cancellazioni non autorizzate; *disponibilità*, ovvero la garanzia che non si verifichino casi in cui ad utenti legittimi venga negato l'accesso ai dati (negazione di servizio). Tali obiettivi coesistono nelle esigenze di qualsiasi sistema. Tuttavia, l'importanza assegnata loro varia sensibilmente a seconda del sistema considerato. Ad esempio, l'aspetto della riservatezza è particolarmente rilevante in ambienti come quello militare, in cui sono gestiti dati molto delicati che, se scoperti, potrebbero danneggiare la sicurezza nazionale. In ambienti di tipo commerciale e aziendale risulta invece preponderante l'aspetto dell'integrità, poiché dalla correttezza dei dati manipolati dipende spesso tutta

l'attività dell'organizzazione che utilizza la base di dati stessa.

Esistono poi ulteriori proprietà di sicurezza, particolarmente rilevanti nel caso di accessi tramite Internet. Tra queste ricordiamo: l'*autenticità*, ovvero la garanzia da parte del ricevente dell'autenticità della fonte che ha generato i dati; la *completezza*, cioè la garanzia di aver ricevuto tutti i dati per cui si ha l'autorizzazione; l'*auto-protezione*, cioè il fatto che un utente possa specificare quale informazione non vuole ricevere.

Garantire la protezione delle informazioni contenute in una base di dati è un obiettivo che coinvolge non solo il DBMS ma anche il sistema operativo, l'hardware e le reti di comunicazione. In questo capitolo, concentreremo principalmente la nostra attenzione sui meccanismi offerti da un DBMS per garantire le varie proprietà di sicurezza.

In ambito basi di dati, la riservatezza delle informazioni è in primo luogo garantita dal *meccanismo di controllo dell'accesso*, un modulo del DBMS che intercetta ogni richiesta di lettura o scrittura e consente solo quelle per cui esistono le necessarie autorizzazioni. L'integrità è invece assicurata da un insieme di meccanismi. Il meccanismo di controllo dell'accesso, in analogia a quanto avviene per le operazioni di lettura, intercetta ogni richiesta di aggiornamento di dati consentendo solo quelle per cui esistono le necessarie autorizzazioni. Inoltre, una certa forma d'integrità è garantita anche dal meccanismo di verifica dei vincoli di integrità, che assicura che i dati inseriti siano corretti rispetto ai vincoli specificati, e dal gestore della concorrenza, che assicura la correttezza dei dati anche in presenza di accessi concorrenti da parte di più transazioni. Concorrono a garantire la riservatezza e l'integrità delle informazioni anche quei meccanismi, non di stretta competenza del DBMS, che assicurano tali proprietà durante la trasmissione in rete. In particolare, l'uso di tecniche di cifratura assicura che i dati trasmessi sulla rete di comunicazione possano essere decifrati solo da utenti autorizzati, mentre le tecniche di firma digitale sono usate per assicurare l'autenticità dei dati e la verifica della loro integrità. Infine, la disponibilità è in una certa misura garantita dal gestore del ripristino, che assicura la disponibilità dei dati in presenza di malfunzionamenti hardware e software. La disponibilità dei dati può essere ulteriormente migliorata dall'uso di tecniche per il rilevamento delle intrusioni in grado di determinare sequenze di accessi inusuali, e che quindi potrebbero essere indice di un attacco al sistema, quali ad esempio un numero di richieste eccessivo effettuato per sovraccaricare il sistema e degradarne le prestazioni.

Oltre ai meccanismi sopra descritti, concorre alla protezione dei dati anche il meccanismo di autenticazione di cui ogni DBMS è dotato. Tale meccanismo verifica l'identità dell'utente che si connette al sistema e la sua autorizzazione all'accesso. I meccanismi di autenticazione adottati comunemente sono basati sull'uso di login e password. Meccanismi più sofisticati, oggi utilizzati solo in contesti particolari, sono basati sulla rilevazione di dati biometrici, quali impronte digitali, riconoscimento della voce, scansione della retina, ecc.

In questo capitolo ci concentreremo principalmente sui meccanismi di sicurezza propri dei DBMS. In particolare, ci focalizzeremo sul controllo dell'accesso. Ut-

lizzeremo, ove possibile, l'esempio della videoteca per illustrare i vari concetti; utilizzeremo altri domini applicativi ove questo sarà necessario. Rimandiamo il lettore al Capitolo 8 per la gestione del ripristino ed il controllo della concorrenza, ed al Capitolo 3 per la specifica dei vincoli d'integrità.

### 9.1 Controllo dell'accesso: concetti fondamentali

Il controllo dell'accesso regola le operazioni che è possibile compiere sui dati e le risorse in una base di dati. Lo scopo è quello di limitare e controllare le operazioni che gli utenti, già riconosciuti dal meccanismo di autenticazione, effettuano, prevenendo azioni accidentali o deliberate che potrebbero compromettere la correttezza e la sicurezza dei dati. Il meccanismo di controllo dell'accesso riveste quindi un ruolo fondamentale per la protezione dei dati gestiti da un DBMS, basti ricordare che la maggior parte delle violazioni di sicurezza avvengono di solito ad opera di utenti legittimi.

Indipendentemente dal contesto in cui si applica, nel controllo dell'accesso distinguiamo tre entità fondamentali: gli *oggetti*, cioè le risorse a cui vogliamo garantire protezione; i *soggetti*, ovvero le entità “attive” che richiedono di poter accedere agli oggetti; i *privilegi*, che determinano le operazioni che i soggetti possono effettuare sugli oggetti. I soggetti possono essere ulteriormente classificati in: *utenti*, cioè singoli individui; *gruppi*, ovvero insiemi di utenti; *ruoli*, ovvero funzioni aziendali a cui assegnare un insieme di privilegi per lo svolgimento delle loro mansioni; *processi*, cioè programmi in esecuzione per conto di utenti. Naturalmente, la tipologia di soggetti, oggetti e privilegi dipende dal contesto da proteggere. In ambito basi di dati relazionali, esempi di oggetti sono tabelle, singole tuple o colonne di una tabella, viste, mentre esempi di privilegi sono quelli che denotano operazioni eseguibili tramite comandi SQL (ad esempio, SELECT, INSERT, UPDATE, ecc.).

Le componenti principali di un sistema per il controllo dell'accesso sono illustrate nella Figura 9.1. Innanzitutto, la concessione o meno di un determinato accesso deve rispecchiare le *politiche di sicurezza* dell'organizzazione, ovvero le regole ed i principi secondo cui l'organizzazione vuole che siano protette le proprie informazioni. Le politiche di sicurezza rappresentano quindi un insieme di direttive ad alto livello che esprimono le scelte compiute da un'organizzazione in merito alla protezione dei propri dati. Possono essere considerate come i requisiti per lo sviluppo di un sistema di controllo dell'accesso e dipendono da vari fattori, quali ad esempio l'ambito in cui l'organizzazione opera, la sua natura, la legislazione vigente, le esigenze degli utenti o i regolamenti interni. Esempi di politiche di sicurezza sono: “solo l'amministratore della base di dati può concedere o revocare privilegi di accesso” o “le valutazioni psicologiche di un impiegato possono essere viste solo dal suo responsabile”. Le politiche, per poter essere utilizzate ai fini del controllo dell'accesso, devono essere tradotte in un insieme di *autorizzazioni* che stabiliscono gli specifici diritti che i vari soggetti abilitati ad accedere al sistema possono esercitare sugli oggetti, in conformità con le politiche di sicurezza adottate. Le autorizzazioni, nel loro formato base, possono essere rappresentate

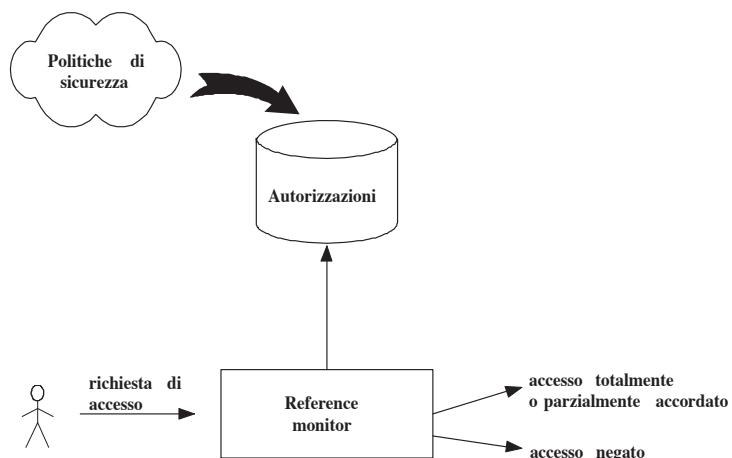


Figura 9.1: Controllo dell'accesso

mediante una tupla  $(s, o, p)$ , dove  $s$  è il soggetto a cui l'autorizzazione è concessa,  $o$  è l'oggetto sui cui è concessa l'autorizzazione e  $p$  è il privilegio che  $s$  può esercitare su  $o$ . Ad esempio, se consideriamo la politica illustrata precedentemente relativa alle valutazioni psicologiche e supponiamo che Mario Rossi sia il responsabile di Giovanni Bianchi, la tupla `(mario rossi, valutazionePsicologica(giovanni bianchi), lettura)` è un esempio di autorizzazione conforme alla politica, dove `valutazionePsicologica(giovanni bianchi)` indica il documento contenente la valutazione psicologica dell'impiegato Giovanni Bianchi. Le autorizzazioni devono essere memorizzate nel sistema per poter essere utilizzate ai fini del controllo dell'accesso. Solitamente, viene utilizzato per la loro rappresentazione lo stesso modello dei dati utilizzato per la rappresentazione degli oggetti. Ad esempio, in un DBMS relazionale le autorizzazioni sono memorizzate in cataloghi (vedi Capitolo 7).

Come illustrato nella Figura 9.1, il controllo dell'accesso è effettuato mediante il *meccanismo di controllo dell'accesso*, detto anche *reference monitor*, il cui compito è quello di intercettare ogni comando inviato al DBMS e stabilire, tramite l'analisi delle autorizzazioni, se il soggetto richiedente può essere autorizzato (totalmente o parzialmente) o meno a compiere l'accesso richiesto. Ogni meccanismo di controllo dell'accesso è basato su un *modello di controllo dell'accesso*, che definisce la natura delle componenti base di un'autorizzazione (soggetti, oggetti e privilegi), le loro inter-relazioni e tutte le funzionalità per la verifica e l'amministrazione delle autorizzazioni.

Nei paragrafi successivi illustreremo le più rilevanti politiche di sicurezza.

## 9.2 Politiche di sicurezza

Rispetto al controllo dell'accesso, le politiche di sicurezza possono essere suddivise in due classi fondamentali: *politiche per il controllo dell'accesso* e *politiche per l'amministrazione del controllo dell'accesso*, che verranno discusse nei paragrafi successivi.

### 9.2.1 *Politiche per il controllo dell'accesso*

Le politiche per il controllo dell'accesso definiscono i criteri secondo cui concedere o rifiutare l'accesso alle informazioni memorizzate in una base di dati. Nei paragrafi successivi illustreremo le dimensioni in base a cui possono essere classificate.

#### 9.2.1.1 *Limitazione degli accessi*

Un primo aspetto che le politiche di controllo dell'accesso devono regolamentare è relativo alla quantità di informazione a cui ciascun soggetto può accedere. Rispetto a questa dimensione, le principali opzioni sono:

- **Need to know – Principio del minimo privilegio.** È un principio molto restrittivo che permette ad ogni soggetto l'accesso solo a quei dati strettamente necessari per eseguire le proprie attività.
- **Maximized sharing – Principio della massima condivisione.** Lo scopo di questo principio è di consentire ai soggetti il massimo accesso alle informazioni nella base di dati, mantenendo comunque alcune informazioni riservate.

Entrambi i principi presentano vantaggi e svantaggi e la scelta di quale adottare dipende fortemente dal dominio considerato. Il principio del minimo privilegio offre ottime garanzie di protezione ed è pertanto adatto a contesti con forti esigenze di sicurezza. Il principale svantaggio è che può portare ad un sistema eccessivamente protetto, negando anche accessi che non comprometterebbero la sicurezza del sistema. Il principio della massima condivisione, invece, ha il vantaggio di soddisfare il massimo numero possibile di richieste di accesso; pertanto, viene solitamente utilizzato in ambienti, quali quelli accademici, in cui esiste una certa fiducia tra gli utenti che accedono alle informazioni ed in cui non è sentita una forte esigenza di protezione.

I sistemi per il controllo dell'accesso possono essere inoltre distinti in sistemi aperti e sistemi chiusi. In un sistema chiuso l'accesso è permesso *solo se* esplicitamente autorizzato, mentre in un sistema aperto l'accesso è permesso *a meno che* non sia esplicitamente negato. In un sistema chiuso, quindi, le autorizzazioni indicano per ogni soggetto i privilegi che egli *può* esercitare sugli oggetti del sistema. Tali privilegi sono i soli che verranno accordati dal meccanismo di controllo. Al contrario, in un sistema aperto le autorizzazioni stabiliscono, per ogni soggetto, i privilegi che egli *non può* esercitare sugli oggetti del sistema. Tali privilegi sono i

soli che gli saranno negati. Un sistema chiuso ben si adatta, quindi, a realizzare il principio del minimo privilegio, mentre un sistema aperto è adatto ad implementare il principio della massima condivisione. Un sistema chiuso offre maggiori garanzie di sicurezza rispetto ad un sistema aperto, in quanto un'autorizzazione inavvertitamente cancellata o non inserita restringe ulteriormente l'accesso, mentre in un sistema aperto la stessa operazione permette accessi non autorizzati. Per questa ragione la maggior parte dei DBMS commerciali si comporta come un sistema chiuso.

#### 9.2.1.2 Controllo dell'accesso

Un altro aspetto regolato dalle politiche per il controllo dell'accesso riguarda i criteri in base a cui i soggetti possono accedere agli oggetti nel sistema e se e come i diritti d'accesso possono venire trasmessi a terzi. Rispetto a queste dimensioni, le politiche possono essere classificate in tre categorie: *politiche discrezionali*, *politiche mandatorie* e *politiche basate su ruoli*.

**Politiche discrezionali.** Tali politiche attuano il controllo dell'accesso sulla base dell'identità del soggetto richiedente. In particolare, tali politiche sono tradotte in un insieme di autorizzazioni che stabiliscono esplicitamente, per ogni soggetto, i privilegi che questo può esercitare sugli oggetti del sistema. Il meccanismo di controllo esamina le richieste di accesso accordando solo quelle che sono concesse da un'autorizzazione. Queste politiche vengono dette discrezionali, in quanto permettono agli utenti di concedere o revocare dei diritti di accesso sugli oggetti, a loro discrezione.

Le politiche discrezionali hanno il vantaggio di essere estremamente flessibili e, quindi, adatte a numerosi contesti applicativi, in quanto possono modellare svariati requisiti di sicurezza, configurando opportunamente l'insieme di autorizzazioni. Il loro principale svantaggio è che non forniscono alcun meccanismo di controllo sul flusso di informazioni nel sistema, in quanto non impongono alcuna restrizione su come l'informazione possa essere trasmessa da un soggetto autorizzato ad un altro non autorizzato ad accedervi. Questo rende i meccanismi discrezionali vulnerabili rispetto ad attacchi, quali quelli perpetrati tramite *cavalli di Troia*. I cavalli di Troia sono programmi che apparentemente svolgono un compito utile (ad esempio stampare una lista di titoli di film) ma che al loro interno contengono delle istruzioni nascoste che utilizzano fraudolentemente le autorizzazioni degli utenti che li eseguono per trasferire illegalmente informazioni non violando, al contempo, le limitazioni imposte dalla politica discrezionale. Il seguente esempio chiarisce meglio il concetto.

**Esempio 9.1** Supponiamo che Anna e Paolo siano due dipendenti della videoteca e che Anna sia la responsabile di Paolo. Supponiamo che, in base alle politiche di sicurezza, Anna possa accedere sia in lettura sia in scrittura a tutte le relazioni della base di dati della videoteca, mentre a Paolo è proibito vedere i titoli dei film noleggiati dai clienti della videoteca. Supponiamo che Paolo regali ad Anna

un programma di utilità, ad esempio un programma per la gestione dell'agenda, in cui ha fraudolentemente inserito alcune istruzioni (il cavallo di Troia). Tali istruzioni: (i) creano una tabella `FilmClienti` che contiene, per ogni cliente, i titoli dei film che ha noleggiato; (ii) concedono a Paolo l'autorizzazione ad effettuare interrogazioni su di essa. Quando Anna esegue il programma, il meccanismo di controllo dell'accesso autorizzerà o meno le operazioni effettuate dal programma in base alle autorizzazioni possedute da Anna. In questo modo Paolo, senza violare i controlli della politica discrezionale, potrà accedere ad informazioni per cui non ha le necessarie autorizzazioni. □

Alcuni di questi limiti di sicurezza sono superati dalle politiche mandatorie.

**Politiche mandatorie.** Tali politiche regolano l'accesso ai dati mediante la definizione di *classi di sicurezza*, chiamate anche etichette, per i soggetti e gli oggetti del sistema. Le classi di sicurezza sono ordinate parzialmente (o totalmente) da una relazione d'ordine. La classe di sicurezza assegnata ad un oggetto è una misura della sensibilità dell'informazione che l'oggetto contiene: maggiore è la classe assegnata ad un oggetto più ingente è il danno derivante dal rilascio delle informazioni in esso contenute a soggetti non autorizzati. La classe di sicurezza assegnata ad un soggetto è, invece, una misura del grado di fiducia che si ha nel fatto che tale soggetto non commetta violazioni della politica ed, in particolare, non trasmetta ad altri informazioni riservate. Il controllo dell'accesso è regolato da una serie di *assiomi di sicurezza* che stabiliscono la relazione che deve intercorrere fra la classe di un soggetto e quella di un oggetto affinché al primo sia concesso di esercitare un privilegio sul secondo. La relazione che deve essere soddisfatta dipende dal tipo di privilegio considerato. In generale, gli assiomi sono volti ad evitare qualsiasi trasferimento d'informazione da un certo soggetto od oggetto verso soggetti od oggetti con classificazione minore o non comparabile.

Le politiche mandatorie sono applicate in ambienti, quali quello militare, dove ci sono forti esigenze di protezione ed è possibile classificare rigidamente soggetti ed oggetti del sistema. I sistemi che adottano politiche mandatorie sono spesso indicati come *sistemi multi-livello*.

La politiche mandatorie e discrezionali non sono però mutuamente esclusive, possono cioè essere applicate in combinazione. In questo caso, le politiche mandatorie non controllano più le richieste di accesso ma le autorizzazioni che vengono assegnate ad un soggetto, mentre alle politiche discrezionali è affidato il compito di controllare le richieste di accesso. Se una richiesta di accesso soddisfa il controllo discrezionale, cioè esiste per essa un'autorizzazione, allora soddisfa anche gli assiomi relativi al controllo mandatorio, per il controllo preventivo effettuato sulle autorizzazioni. Quando le due politiche sono utilizzate congiuntamente, le politiche discrezionali servono per restringere ulteriormente gli accessi consentiti dalle politiche mandatorie, per determinate classi di soggetti e/o oggetti.

**Politiche basate su ruoli.** In aggiunta alle politiche mandatorie e discrezionali, un terzo tipo di politiche, basate sul concetto di *ruolo*, è stato successivamente proposto. In base a tali politiche, i privilegi non sono direttamente assegnati agli

utenti ma sono mediati dal concetto di ruolo. Un ruolo rappresenta una funzione all'interno di un'azienda od organizzazione (sono esempi di ruoli per il dominio della videoteca *direttoreVideoteca, commesso e cliente*). Le autorizzazioni non sono concesse ai singoli utenti ma ai ruoli identificati. Ogni utente è poi abilitato a ricoprire uno o più ruoli ed in questo modo acquisisce le autorizzazioni ad essi associate. Il concetto di ruolo semplifica notevolmente la gestione delle autorizzazioni e rende più diretta la modellazione delle politiche aziendali. Infatti, in un'azienda/organizzazione spesso i privilegi esercitabili non sono legati all'identità dei singoli ma piuttosto al ruolo che essi ricoprono all'interno della stessa. La semplificazione amministrativa è dovuta principalmente a due fattori: il primo è che un ruolo di solito raggruppa un cospicuo numero di privilegi. Quando un utente deve ricoprire la mansione associata ad un certo ruolo, invece di specificare per lui tutte le autorizzazioni associate al ruolo, basta abilitarlo a ricoprire quel ruolo e questo comporta automaticamente l'acquisizione dei privilegi associati al ruolo stesso. La seconda semplificazione è dovuta al fatto che i ruoli rappresentano una componente del sistema più stabile rispetto agli utenti, che possono cambiare frequentemente e/o cambiare le loro mansioni. Ad esempio, un cambio di mansioni in un sistema che supporta i ruoli viene gestito molto facilmente: basta revocare all'utente interessato dal cambio di mansione l'autorizzazione a ricoprire i ruoli associati alla vecchia mansione ed abilitarlo a ricoprire i ruoli corrispondenti alla nuova.

Un'altra caratteristica importante delle politiche basate su ruoli è che, configurando opportunamente il sistema dei ruoli, possono essere usate per realizzare sia politiche discrezionali sia mandatorie, e questo accresce notevolmente il loro campo di impiego.

#### **9.2.2 Politiche per l'amministrazione del controllo dell'accesso**

Un altro importante aspetto regolato dalle politiche di sicurezza è chi può concedere e revocare privilegi. Questo aspetto è maggiormente rilevante per i meccanismi di controllo dell'accesso discrezionali e basati su ruoli, mentre nei sistemi mandatori di solito la politica di amministrazione è molto semplice in quanto non vi sono autorizzazioni esplicite e l'amministratore della sicurezza è di solito l'unico che può modificare le classi di sicurezza di soggetti ed oggetti. Le politiche discrezionali e basate su ruoli permettono, invece, di adottare soluzioni più articolate. In particolare, le principali politiche di amministrazione che possono essere adottate sono:

- **Centralizzata.** Un unico autorizzatore (o gruppo), detto amministratore della sicurezza, può specificare e revocare le autorizzazioni.
- **Basata su ownership.** L'utente che crea un oggetto, detto *owner*, gestisce anche la concessione e la revoca di autorizzazioni sull'oggetto.
- **Decentralizzata.** In base a questa politica, l'*owner* o l'amministratore della sicurezza possono delegare ad altri la facoltà di specificare e revocare autoriz-

zazioni per determinati oggetti e modi di accesso. Molti DBMS commerciali adottano la politica basata su ownership con delega dell'amministrazione.

- **Cooperativa.** Autorizzazioni su particolari oggetti non possono essere concesse da un singolo utente, ma richiedono il consenso di più utenti. Questa politica di amministrazione è particolarmente adatta ad ambienti cooperativi e distribuiti.

### 9.3 Modelli di controllo dell'accesso

I modelli di controllo dell'accesso per basi di dati sono stati sviluppati a partire dai modelli definiti per la protezione di risorse in un sistema operativo. Tuttavia, i modelli usati nell'ambito di un sistema operativo non sono completamente adatti ad essere utilizzati in una base di dati. Le differenze fondamentali tra i due contesti sono innanzitutto dovute al fatto che il controllo dell'accesso in una base di dati deve avvenire a diversi livelli di granularità (ad esempio, nel caso di basi di dati relazionali, a livello di relazione, vista, tupla o singolo attributo). Inoltre, un modello di controllo dell'accesso in un sistema operativo protegge risorse reali (ad esempio stampanti, supporti di memorizzazione, ecc.), mentre in una base di dati le risorse da proteggere possono essere strutture completamente logiche (ad esempio viste), alcune delle quali possono corrispondere allo stesso oggetto fisico.

Nei paragrafi successivi descriveremo i più significativi modelli di controllo dell'accesso.

#### 9.3.1 Modello a matrice di accesso

Tra i modelli sviluppati nell'ambito dei sistemi operativi, quello che ha maggiormente influenzato i modelli per il controllo dell'accesso per basi di dati è il modello di Lampson, successivamente esteso da Graham e Denning e formalizzato da Harrison, Ruzzo ed Ullman (quest'ultimo è conosciuto come *modello HRU* dalle iniziali dei cognomi dei suoi ideatori). Tale modello è il modello concettuale di riferimento per rappresentare autorizzazioni in sistemi che adottano politiche discrezionali. Il nome del modello deriva dal fatto che lo stato corrente del sistema rispetto alle autorizzazioni è descritto mediante una matrice. Più precisamente, nel modello a matrice di accesso lo stato del sistema è descritto dalla tripla  $(S, O, M)$ , dove:  $S$  è l'insieme dei soggetti, cioè delle entità che richiedono gli accessi,  $O$  è l'insieme degli oggetti, cioè delle entità su cui viene richiesto l'accesso, ed  $M$  è la *matrice di accesso*, in cui ogni riga corrisponde ad un soggetto, ogni colonna ad un oggetto e l'elemento  $M[i, j]$  contiene la lista dei privilegi esercitabili dal soggetto  $s_i$  sull'oggetto  $o_j$ , scelti tra l'insieme di privilegi che il modello mette a disposizione.

Il meccanismo di controllo dell'accesso autorizza solo gli accessi per cui esiste il relativo privilegio nella matrice di accesso.

**Esempio 9.2** La Figura 9.2 illustra un esempio di matrice di accesso, per la protezione di risorse in un sistema operativo. Ad esempio, in base alla Figura

	<code>marco.doc</code>	<code>edit.exe</code>	<code>giochi.dir</code>
<code>Marco</code>	{read,write}	{execute}	{execute}
<code>Anna</code>	-	{execute}	{execute,read,write}

Figura 9.2: Matrice di accesso

9.2 sia Marco sia Anna possono eseguire i programmi contenuti nella directory `giochi.dir`, ma solo Anna può leggere e modificare i file in essa contenuti. Viceversa, Marco è il solo che può leggere e modificare il file `marco.doc`.  $\square$

Notiamo come la matrice di accesso sia solo un modello concettuale, non adatto per un'effettiva implementazione. Infatti, in molte situazioni la matrice risulterebbe troppo sparsa e di dimensioni troppo elevate, in quanto ogni soggetto ha solitamente accesso solo ad una piccola porzione degli oggetti del sistema. Le soluzioni utilizzate nella pratica per implementare la matrice di accesso sono essenzialmente due:

- **Access Control List (ACL).** In questo caso la matrice viene implementata mediante un insieme di liste associate ad ogni oggetto del sistema. L'ACL associata ad un oggetto contiene un elemento per ogni soggetto che può esercitare un qualche privilegio sull'oggetto in questione. Tale elemento contiene la lista dei privilegi che il soggetto può esercitare sull'oggetto.
- **Capability List.** È un metodo complementare a quello delle ACL ed implementa la matrice di accesso mediante liste associate ai soggetti. La capability list associata ad un soggetto contiene un elemento per ogni oggetto su cui il soggetto può esercitare un qualche privilegio. Tale elemento contiene la lista dei privilegi che il soggetto ha su quel determinato oggetto.

L'approccio basato su ACL è quello utilizzato dai moderni sistemi operativi, mentre le capability list sono più adatte a sistemi distribuiti dove un soggetto potrebbe autenticarsi una sola volta presso un host, acquisire le sue capability ed utilizzarle per ottenere l'accesso presso tutti gli host che compongono il sistema distribuito. Naturalmente, in questo caso bisogna predisporre meccanismi che evitino che i soggetti possano coniare capability false o utilizzare capability non valide, quali ad esempio quelle relative ad un privilegio revocato.

### 9.3.2 Modello di Bell e LaPadula

Il modello sviluppato da Bell e LaPadula rappresenta il modello di riferimento per la maggior parte dei sistemi che adottano una politica mandatoria. Nel modello di Bell e LaPadula, i soggetti sono sia utenti sia processi. I privilegi previsti dal modello sono: `read`, che consente ad un soggetto di leggere ma non di modificare le informazioni contenute in un oggetto; `append`, che consente ad un soggetto di

modificare un oggetto ma non di leggere le informazioni in esso contenute; **write**, che consente sia la modifica sia la lettura di un oggetto; **execute**, che consente ad un soggetto di eseguire un oggetto (ad esempio un programma applicativo). Nel seguito considereremo solo i privilegi **read**, **write** ed **append** in quanto strettamente pertinenti alla gestione dati.

Essendo il modello di Bell e LaPadula un modello mandatorio, soggetti ed oggetti vengono classificati mediante l'assegnazione di una *classe di sicurezza*. Una classe di sicurezza è costituita da due componenti: un *livello di sicurezza* ed un *insieme di categorie*. Il livello di sicurezza è un elemento, scelto da un insieme totalmente ordinato. Esempi di livelli comunemente usati sono: **Top Secret** (TS), **Secret** (S), **Confidential** (C) e **Unclassified** (U), ordinati nel modo seguente:  $TS > S > C > U$ .

L'insieme di categorie è un insieme non ordinato di elementi, che può essere anche vuoto. Gli elementi di questo insieme dipendono dal tipo di ambiente considerato e denotano aree di competenza all'interno dell'organizzazione. Ad esempio, se consideriamo l'ambito militare, possibili categorie sono: **Army**, **Navy**, **Air Force** e **Nuclear**, mentre in ambito commerciale esempi di possibili categorie sono: **Finanza**, **Vendite** e **Ricerca e Sviluppo**. Un file contenente informazioni finanziarie riservate potrebbe, ad esempio, essere classificato (**Confidential**,  $\{\text{Finanza}\}$ ), mentre un generale di corpo d'armata potrebbe avere come classe di sicurezza (**Top Secret**,  $\{\text{Navy, Nuclear}\}$ ).

L'insieme delle classi di sicurezza è parzialmente ordinato da una relazione di dominanza formalmente definita come segue.

**Definizione 9.1 (Relazione di dominanza)** Una classe di sicurezza  $cs_1 = (L_1, Cat_1)$  domina una classe di sicurezza  $cs_2 = (L_2, Cat_2)$ , (denotato con  $c_1 \geq c_2$ ), se entrambe le seguenti condizioni sono verificate: (i) il livello di sicurezza di  $cs_1$  è maggiore o uguale al livello di sicurezza di  $cs_2$  (cioè  $L_1 \geq L_2$ ); (ii) l'insieme di categorie di  $cs_1$  include l'insieme di categorie di  $cs_2$  (cioè  $Cat_1 \supseteq Cat_2$ ).  $\diamond$

Se  $L_1 > L_2$  e  $Cat_1 \supset Cat_2$ , allora  $cs_1$  domina strettamente  $cs_2$  ( $cs_1 > cs_2$ ). Infine, due classi di sicurezza  $cs_1$  e  $cs_2$  sono incomparabili ( $cs_1 \sim cs_2$ ) se né  $cs_1 \geq cs_2$  né  $cs_2 \geq cs_1$  valgono.

**Esempio 9.3** Consideriamo le seguenti classi di sicurezza:

$$\begin{aligned} cs_1 &= (\text{TS}, \{\text{Nuclear, Army}\}) \\ cs_2 &= (\text{TS}, \{\text{Nuclear}\}) \\ cs_3 &= (\text{C}, \{\text{Army}\}) \end{aligned}$$

$cs_1 \geq cs_2$  in quanto  $cs_1$  ha lo stesso livello di sicurezza di  $cs_2$  ma un soprainsieme delle sue categorie;  $cs_1 > cs_3$ , in quanto il livello di sicurezza TS è strettamente maggiore di C e  $\{\text{Army}\}$  è un sottoinsieme proprio di  $\{\text{Nuclear, Army}\}$ . Infine,  $cs_2 \sim cs_3$ ; infatti,  $cs_2 \not\geq cs_3$ , in quanto  $\{\text{Nuclear}\} \not\supseteq \{\text{Army}\}$  e  $cs_3 \not\geq cs_2$ , in quanto  $TS > C$ .  $\square$

Lo stato del sistema è descritto dalla coppia  $(A, \mathcal{L})$ ,<sup>1</sup> dove:

- $A$  è l'insieme degli accessi correnti, ossia degli accessi correntemente in esecuzione;  $A$  è composto da triple della forma  $(s, o, p)$  che indicano che il soggetto  $s$  sta correntemente esercitando il privilegio  $p$  sull'oggetto  $o$ .
- $\mathcal{L}$  è la funzione di livello che associa ad ogni elemento del sistema la sua classe di sicurezza, formalmente:  $\mathcal{L} : O \cup S \rightarrow \mathcal{CS}$ , dove  $O$  ed  $S$  sono, rispettivamente, l'insieme degli oggetti e dei soggetti, e  $\mathcal{CS}$  è l'insieme delle classi di sicurezza.

Ogni modifica di stato è causata da una *richiesta*. Una richiesta può essere ad esempio una richiesta di accesso o una richiesta di modifica delle classi di sicurezza di soggetti ed oggetti. La risposta del sistema è chiamata *decisione*. Data una richiesta ed uno stato corrente, la decisione ed il nuovo stato sono determinati in base a degli *assiomi di sicurezza*. Tali assiomi stabiliscono le condizioni che devono essere soddisfatte nel sistema per effettuare una transizione di stato. Se le richieste sono soddisfatte solo se verificano gli assiomi, allora il sistema è *sicuro*.

Per quanto riguarda le richieste di accesso, il primo assioma del modello di Bell e LaPadula è il seguente:<sup>2</sup>

**Proprietà di sicurezza semplice.** Uno stato  $(A, \mathcal{L})$  soddisfa la proprietà di sicurezza semplice se per ogni elemento  $(s, o, p) \in A$  tale che  $p=\text{read}$  oppure  $p=\text{write}$ :  $\mathcal{L}(s) \geq \mathcal{L}(o)$ .

La proprietà di sicurezza semplice, conosciuta anche come *proprietà del no read-up*, assicura che i soggetti abbiano accesso diretto solo alle informazioni per cui hanno la necessaria classificazione, prevenendo letture di oggetti con classe di sicurezza maggiore o incomparabile.

**Esempio 9.4** Un soggetto con classe di sicurezza  $(C, \{\text{Army}\})$  non può leggere dati con classi di accesso  $(C, \{\text{Army}, \text{Air Force}\})$  o  $(U, \{\text{Air Force}\})$ , mentre può leggere oggetti con classe di sicurezza  $(U, \{\text{Army}\})$ .  $\square$

La proprietà di sicurezza semplice non evita però che una combinazione di accessi possa compromettere la sicurezza dei dati. Ad esempio un soggetto con classe di sicurezza  $(TS, \emptyset)$  potrebbe estrarre informazioni da un oggetto con classe di sicurezza  $(TS, \emptyset)$  ed inserirle in uno con classe di sicurezza  $(U, \emptyset)$ , rendendole quindi accessibili a soggetti con classe di sicurezza inferiore, senza violare la proprietà di sicurezza semplice, che è stata pensata principalmente per regolamentare le operazioni di lettura. Per evitare tali situazioni, è stato definito un ulteriore assioma, riferito specificatamente alle operazioni di scrittura.

---

<sup>1</sup>Nella formulazione originaria di Bell e LaPadula, lo stato del sistema è descritto da una quadrupla che contiene, in aggiunta alle due componenti descritte, anche la matrice di accesso e la gerarchia degli oggetti. Tali componenti vengono omesse in questa sede in quanto non rilevanti per la successiva trattazione.

<sup>2</sup>Nel seguito, i termini assioma e proprietà saranno usati come sinonimi.

**Proprietà \*.** Uno stato  $(A, \mathcal{L})$  soddisfa la proprietà \*, se per ogni elemento  $(s, o, p) \in A$  tale che  $p=\text{append}$  o  $p=\text{write}$ :  $\mathcal{L}(s) \leq \mathcal{L}(o)$ .

La proprietà \* (*\* property*), conosciuta anche come *proprietà del no write-down*, è definita per prevenire il flusso d'informazioni dovute a scritture verso classi di sicurezza minori o non comparabili.

**Esempio 9.5** Un soggetto con classe di sicurezza  $(C, \{\text{Army}, \text{Nuclear}\})$  non può scrivere informazioni in oggetti con classe di sicurezza  $(U, \{\text{Army}, \text{Nuclear}\})$  perché tali oggetti sono accessibili a soggetti che non possono leggere oggetti con classificazione  $(C, \{\text{Army}, \text{Nuclear}\})$ .  $\square$

Rispetto al controllo dell'accesso, un sistema si dice sicuro se ogni elemento aggiunto all'insieme degli accessi correnti verifica la proprietà di sicurezza semplice e la proprietà \*. Notiamo come questo implichi che, per il privilegio `write`, devono essere applicati entrambi gli assiomi. Un soggetto  $s$  può quindi esercitare il privilegio `write` sull'oggetto  $o$  solo se  $\mathcal{L}(s) = \mathcal{L}(o)$ .

L'applicazione delle due proprietà precedentemente enunciate può comportare un'eccessiva rigidità del sistema, come evidenziato dal seguente esempio.

**Esempio 9.6** Supponiamo che un generale abbia classe di sicurezza  $(TS, \{\text{Army}, \text{Nuclear}\})$ , mentre un suo colonnello abbia classe di sicurezza  $(C, \{\text{Army}\})$ . Il colonnello può potenzialmente comunicare con il generale, in quanto gli è concessa la scrittura in modalità append di oggetti con classe di sicurezza maggiore, mentre non è possibile per il generale comunicare con il suo colonnello in quanto, in base alla proprietà \*, il generale non può scrivere in oggetti con classe di sicurezza minore della sua e quindi accessibili al colonnello.  $\square$

Per ovviare agli inconvenienti discussi nell'Esempio 9.6, gli utenti possono connettersi al sistema con una qualsiasi classe di sicurezza dominata da quella a loro assegnata. Operativamente, quando un utente si connette al sistema con una certa classe di sicurezza, viene considerato dal sistema come un soggetto con classe di sicurezza pari a quella con cui si è connesso. Il generale dell'esempio precedente può connettersi al sistema con classe di sicurezza  $(C, \{\text{Army}\})$  e comunicare quindi con il suo colonnello.

In effetti, la precedente definizione di sistema sicuro non garantisce la totale sicurezza del sistema, come evidenziato da Mc Lean in [McL90]. Ad esempio, consideriamo un sistema in cui, quando un soggetto  $s$  richiede un qualsiasi tipo di accesso su un oggetto  $o$ , a tutti gli oggetti ed i soggetti del sistema è assegnata la classe di sicurezza più bassa e l'accesso è consentito. Tale sistema soddisfa la definizione di sistema sicuro enunciata in precedenza, in quanto non viola né la proprietà di sicurezza semplice né la proprietà \*, anche se presenta ovvi problemi di sicurezza, in quanto dopo la prima richiesta di accesso tutti possono accedere a tutto. In effetti, il modello di Bell e LaPadula garantisce buoni requisiti di sicurezza solo in presenza di una *classificazione statica* di soggetti ed oggetti in classi di sicurezza (tale principio è noto come *strong tranquillity principle*). Se tale principio non è

soddisfatto, la sicurezza o meno del sistema dipende dal modo con cui possono essere modificate le classi di sicurezza. Nella pratica, il principio di tranquillità sopra esposto è troppo restrittivo per essere applicato, in quanto ci possono essere situazioni in cui è necessaria una riclassificazione di soggetti ed oggetti rispetto alle classi di sicurezza. Per questo sono stati definiti principi meno restrittivi, che consentono la modifica delle classi di sicurezza sotto opportune condizioni. Viene inoltre introdotta la nozione di *soggetto fidato* (trusted), cioè un soggetto che può violare alcune delle restrizioni imposte dal modello. Ad esempio, Oracle Label Security, la componente del DBMS Oracle che consente di proteggere i dati secondo politiche mandatorie, prevede i seguenti privilegi speciali: **READ**, che consente di evitare i controlli in lettura degli assiomi mandatori; **FULL**, che consente di evitare sia i controlli in lettura sia in scrittura degli assiomi mandatori; **WRITEDOWN**, che consente di diminuire il livello di sicurezza di una classe di sicurezza; **WRITEUP**, che consente di aumentare il livello di sicurezza di una classe di sicurezza.

Notiamo, infine, come il modello di Bell e LaPadula permetta ad un soggetto di esercitare il privilegio **append** su oggetti con una classe di sicurezza superiore alla sua. Questo grado di libertà può essere utile in certi contesti e pericoloso in altri. Ad esempio, un documento a cui possono accedere in scrittura soggetti non autorizzati a leggerlo, può essere reso inconsistente da tali soggetti. Un approccio drastico al problema è quello di modificare il modello di Bell e LaPadula vietando le scritture verso classi di sicurezza maggiori o non comparabili, siano esse dovute all'esercizio del privilegio **append** o **write**.

### 9.3.3 Modello del System R

Uno dei modelli più rilevanti per sistemi che adottano politiche discrezionali è sicuramente il modello di controllo dell'accesso definito da Griffiths e Wade per il DBMS relazionale System R. Tale modello è stato uno dei primi che ha ricevuto un'effettiva implementazione, costituendo così la base per i meccanismi di controllo dell'accesso oggi presenti nei DBMS commerciali e per le funzionalità previste dallo standard SQL (vedi Paragrafo 9.4).

Essendo System R un DBMS relazionale, gli oggetti del modello sono relazioni di base e viste. Nel seguito utilizzeremo il termine relazione per far riferimento sia a relazioni di base sia a viste, quando una distinzione tra le due non sia necessaria. I privilegi previsti dal modello corrispondono alle operazioni effettuabili tramite comandi SQL (ad esempio, **SELECT**, **INSERT**, **DELETE**, **UPDATE**).

Il modello realizza una politica di tipo discrezionale adottando il paradigma di sistema chiuso: un accesso è concesso solo se esiste un'esplicita autorizzazione per esso. L'amministrazione dei privilegi è decentralizzata mediante ownership: l'utente che crea una relazione di base, riceve tutti i privilegi su di essa ed anche la possibilità di delegare ad altri tali privilegi. Per quanto riguarda le viste, invece, le regole sono leggermente diverse e verranno illustrate nel Paragrafo 9.3.3.6. La delega dei privilegi avviene mediante *grant option*. Se un privilegio è concesso con grant option l'utente che lo riceve può non solo esercitare il privilegio, ma

anche concederlo ad altri. Un utente può quindi concedere un privilegio su una determinata relazione solo se è il proprietario della relazione o ha ricevuto tale privilegio con grant option.

#### 9.3.3.1 Comando GRANT

Le autorizzazioni sono concesse tramite il comando **GRANT**, la cui sintassi è la seguente:

```
GRANT {<lista privilegi> | ALL [PRIVILEGES]}
ON <nome relazione>
TO {<lista utenti> | PUBLIC} [WITH GRANT OPTION];
```

dove:

- **<lista privilegi>** indica l'insieme dei privilegi concessi con il comando **GRANT**. Le parole chiave **ALL** o **ALL PRIVILEGES** (le due notazioni sono equivalenti) consentono di concedere con un solo comando tutti i privilegi previsti dal modello sulla relazione oggetto del comando.
- **<nome relazione>** indica il nome della relazione su cui sono concessi i privilegi.
- **<lista utenti>** indica l'insieme degli utenti a cui il comando si applica. La parola chiave **PUBLIC** consente di specificare le autorizzazioni implicate dal comando per tutti gli utenti del sistema.
- La clausola opzionale **WITH GRANT OPTION** consente la delega dell'amministrazione dei privilegi oggetto del comando, in aggiunta all'autorizzazione ad esercitarli. Se non è specificata, gli utenti che ricevono i privilegi non possono concederli ad altri utenti. Se invece è specificata, gli utenti che ricevono i privilegi possono sia esercitarli sia concederli a terzi, limitatamente alla relazione oggetto del comando.

I privilegi concessi con un comando **GRANT** si applicano ad intere relazioni, ad eccezione del privilegio **update** per cui è possibile specificare le colonne su cui si applica (racchiuse tra parentesi tonde e separate da virgole). È inoltre possibile concedere più privilegi su una stessa relazione con un unico comando (i privilegi devono essere separati tramite virgole). Analogamente, un unico comando **GRANT** può essere utilizzato per concedere privilegi sulla stessa relazione a più utenti.

**Esempio 9.7** Consideriamo la base di dati relativa alla gestione della videoteca illustrata nel Capitolo 2 e supponiamo che tutte le relazioni di questa base di dati siano state create dall'utente Luca. I seguenti sono esempi di comandi **GRANT**:<sup>3</sup>

---

<sup>3</sup>La notazione **u:** indica che **u** è l'utente che richiede l'esecuzione del comando, detto anche *grantor*.

---

```

luca: GRANT update(telefono) ON Clienti TO marco;
luca: GRANT select ON Film TO barbara, giovanna WITH GRANT OPTION;
giovanna: GRANT select ON Film TO matteo;
luca: GRANT ALL PRIVILEGES ON Video, Film TO elena WITH GRANT OPTION;
elena: GRANT insert, select ON Film TO barbara;

```

Il primo comando autorizza Marco a modificare l'attributo `telefono` delle tuple della relazione `Clienti`. Il secondo comando concede a Barbara e Giovanna la possibilità di effettuare interrogazioni sulla relazione `Film` e anche di concedere a terzi tale privilegio (che viene concesso da Giovanna a Matteo tramite il terzo comando). Il quarto comando concede ad Elena di esercitare tutti i privilegi previsti dal modello sulle relazioni `Video` e `Film`, unitamente alla possibilità di delegare ad altri tali privilegi, facendo diventare quindi Elena un ulteriore amministratore delle due relazioni, in aggiunta al loro proprietario Luca. Questo consente ad Elena di effettuare con successo l'ultimo comando `GRANT`, concedendo a Barbara il privilegio di inserire tuple ed effettuare interrogazioni sulla relazione `Film`. □

Dal momento che il modello del System R permette di concedere privilegi con grant option, è possibile che un utente riceva lo stesso privilegio sulla stessa relazione da utenti diversi. Ad esempio, con riferimento all'Esempio 9.7, Barbara riceve due volte il privilegio `select` sulla relazione `Film`, una volta da Luca con grant option e una volta da Elena senza grant option. Come vedremo nel Paragrafo 9.3.3.5, questa possibilità ha ripercussioni sull'operazione di revoca dei privilegi.

Inoltre, il fatto che i privilegi possano essere concessi con grant option fa sì che i privilegi che ogni utente possiede possano essere classificati in due categorie: *privilegi delegabili*, cioè quelli concessi all'utente con grant option, e *privilegi non delegabili*, cioè quelli che sono concessi senza grant option. Ai fini del controllo dell'accesso, entrambe le tipologie di privilegi sono rilevanti, mentre, per decidere l'esito di un comando `GRANT` dovremo prendere in considerazione solo i privilegi delegabili (vedi Paragrafo 9.3.3.4).

### 9.3.3.2 Comando REVOKE

I privilegi concessi possono essere successivamente revocati tramite il comando `REVOKE`, la cui sintassi è la seguente:

```

REVOKE {<lista privilegi> | ALL [PRIVILEGES]}
ON <nome relazione>
FROM {<lista utenti> | PUBLIC};

```

dove:

- `<lista privilegi>` indica l'insieme di privilegi oggetto del comando di revoca. Le parole chiave `ALL` (o `ALL PRIVILEGES`) indicano che tutti i privilegi

precedentemente concessi sulla relazione oggetto del comando dall'utente che esegue il comando sono revocati.

- <nome relazione> indica il nome della relazione su cui si vogliono revocare i privilegi.
- <lista utenti> indica l'insieme degli utenti a cui il comando si applica. La parola chiave PUBLIC consente di revocare i privilegi indicati nel comando a tutti gli utenti a cui erano stati precedentemente concessi sulla relazione oggetto del comando dall'utente che esegue il comando REVOKE.

Un utente può revocare solo i privilegi da lui stesso concessi tramite un precedente comando GRANT. Analogamente al comando GRANT, è possibile revocare più privilegi con un unico comando REVOKE, ed un unico comando REVOKE può essere utilizzato per revocare gli stessi privilegi sulla stessa relazione ad utenti diversi. Chiaramente la revoca di un privilegio implica la revoca anche della grant option.

**Esempio 9.8** Consideriamo i comandi GRANT dell'Esempio 9.7 ed i seguenti comandi REVOKE eseguiti da Luca:

```
REVOKE update, insert ON Video FROM elena;
REVOKE update ON Clienti FROM marco;
REVOKE select ON Film FROM giovanna;
```

Con il primo comando viene revocato ad Elena il privilegio di inserire e modificare tuple nella relazione **Video**. Il secondo comando revoca a Marco il diritto di effettuare modifiche sull'attributo **telefono** delle tuple della relazione **Clienti**. Infine, il terzo comando revoca a Giovanna la possibilità di effettuare interrogazioni sulla relazione **Film**. □

Il fatto che il modello di controllo dell'accesso del System R permetta di concedere privilegi con grant option implica che la revoca di un privilegio ad un utente non necessariamente comporta la perdita di quel privilegio da parte dell'utente in questione. Infatti, un utente può continuare ad esercitare un privilegio anche dopo un'operazione di revoca nel caso in cui lo abbia ricevuto da altre fonti *indipendenti* dal soggetto che effettua la revoca. L'esempio seguente chiarisce meglio il concetto.

**Esempio 9.9** Consideriamo la sequenza di comandi GRANT dell'Esempio 9.7 ed il seguente comando REVOKE:

```
luca: REVOKE select ON Film FROM barbara, giovanna;
```

Dopo l'esecuzione del comando di revoca da parte di Luca, Giovanna non può più effettuare interrogazioni sulla relazione **Film**, mentre Barbara mantiene ancora tale privilegio, grazie all'autorizzazione concessale da Elena. Barbara non potrà però più concedere a terzi il privilegio **select** sulla relazione **Film**. □

### 9.3.3.3 I cataloghi Sysauth e Syscolauth

Il DBMS System R, in analogia a quanto accade anche oggi nei DBMS commerciali, utilizza una rappresentazione uniforme per dati ed autorizzazioni. Le informazioni sull'insieme di autorizzazioni correntemente presenti nel sistema sono memorizzate in due cataloghi di sistema (vedi Capitolo 7) di nome **Sysauth** e **Syscolauth**.

Il catalogo **Sysauth** contiene informazioni sui privilegi che ogni utente ha sulle relazioni della base di dati, mentre il catalogo **Syscolauth** serve per la gestione del privilegio **update**. Lo schema di **Sysauth** è costituito dai seguenti attributi:

- **utente**, è l'identificatore dell'utente a cui sono concessi i privilegi.
- **nomeRel**, indica il nome della relazione su cui sono concessi i privilegi.
- **tipo**  $\in \{R,V\}$ , indica se l'oggetto dell'autorizzazione è una relazione di base (**tipo**=‘R’) o una vista (**tipo**=‘V’).
- **select**, indica se l'utente ha il privilegio di effettuare interrogazioni sulla relazione considerata. Tale colonna contiene un *timestamp*, che denota il tempo in cui il privilegio è stato concesso; il valore 0 indica che il relativo privilegio non è stato concesso. Esistono colonne analoghe per ogni privilegio previsto dal modello.
- **grantor**, è l'identificatore dell'utente che ha concesso l'autorizzazione.
- **grantopt**  $\in \{Y,N\}$ , indica se i privilegi sono delegabili (**grantopt** =‘Y’) o meno (**grantopt** =‘N’).

Le informazioni sul timestamp sono fondamentali per la corretta implementazione dell'operazione di revoca dei privilegi (vedi Paragrafo 9.3.3.5). Il timestamp può essere sia un semplice contatore sia indicare un tempo reale. L'importante è che soddisfi le seguenti proprietà: (i) sia monotonicamente crescente; (ii) non esistano due comandi GRANT con lo stesso timestamp. Privilegi concessi con lo stesso comando GRANT hanno lo stesso timestamp. Quando un utente crea una relazione, gli vengono concessi tutti i privilegi sulla relazione con grant option. Il loro timestamp sarà quello del comando CREATE TABLE.

**Esempio 9.10** La Figura 9.3 mostra un esempio di catalogo **Sysauth**, relativamente ai privilegi **select**, **insert** ed **update** ed alle relazioni **Clienti**, **Video** e **Film**, che riflette lo stato delle autorizzazioni dopo l'esecuzione dei comandi GRANT dell'Esempio 9.7. I timestamp sono stati assegnati arbitrariamente, tenendo conto delle considerazioni esposte in precedenza. Le prime tre entrate del catalogo sono automaticamente inserite dal sistema al momento della creazione delle relazioni **Clienti**, **Video** e **Film** da parte di Luca. Il timestamp coincide con quello del relativo comando CREATE TABLE.  $\square$

Come abbiamo visto, il catalogo **Sysauth** mantiene solo informazioni di tipo “qualitativo” sui privilegi **update** che ogni utente può esercitare, in quanto registra

utente	nomeRel	tipo	select	insert	update	grantor	grantopt
luca	Clienti	R	20	20	20		Y
luca	Video	R	22	22	22		Y
luca	Film	R	25	25	25		Y
marco	Clienti	R	0	0	30	luca	N
barbara	Film	R	32	0	0	luca	Y
giovanna	Film	R	32	0	0	luca	Y
matteo	Film	R	35	0	0	giovanna	N
elena	Video	R	40	40	40	luca	Y
elena	Film	R	40	40	40	luca	Y
barbara	Film	R	47	47	0	elena	N

Figura 9.3: Catalogo Sysauth

solo il fatto che un utente possa esercitare o meno il privilegio `update` su una certa relazione, ma non tiene traccia delle colonne specifiche su cui tale privilegio può essere esercitato. Tali informazioni sono mantenute nel catalogo `Syscolauth` che contiene una tupla: (`utente, nomeRel, colonna, grantor, grantopt`) per ogni colonna della relazione `nomeRel` su cui l'utente identificato da `utente` può esercitare il privilegio `update`.

#### 9.3.3.4 Concessione di privilegi

Quando un utente richiede l'esecuzione di un comando `GRANT`, il meccanismo di controllo dell'accesso legge i cataloghi `Sysauth` e `Syscolauth` per determinare se il richiedente possiede o meno il diritto di concedere i privilegi specificati nel comando. Per effettuare tale verifica, l'insieme dei privilegi delegabili che l'utente possiede è intersecato con l'insieme dei privilegi specificati nel comando `GRANT`. Sono possibili tre risultati. Se l'intersezione è vuota, il comando non viene eseguito; se l'intersezione coincide con i privilegi presenti nel comando, il comando viene eseguito totalmente e tutti i privilegi specificati vengono quindi concessi; altrimenti, il comando viene eseguito parzialmente, solo per i privilegi contenuti nell'intersezione. Un utente ha un privilegio delegabile su una relazione se ha creato la relazione oppure ha ottenuto il privilegio con grant option da un qualsiasi grantor.

**Esempio 9.11** Consideriamo il catalogo `Sysauth` illustrato nella Figura 9.3 ed i seguenti comandi `GRANT`:

```
marco: GRANT update(telefono) ON Clienti TO roberto;
luca: GRANT delete ON Clienti TO giovanni, anna;
barbara: GRANT select, insert ON Film TO alessandro;
```

Il primo comando non viene eseguito, in quanto Marco ha il privilegio `update` sull'attributo `telefono` ma senza grant option. Il secondo comando viene eseguito, in quanto Luca è il proprietario della relazione `Clienti`. Infine, il terzo comando

viene parzialmente eseguito; Barbara possiede il privilegio `select` sulla relazione `Film` con grant option, mentre può esercitare il privilegio `insert` ma senza concederlo a terzi. Il risultato dell'esecuzione del comando è quindi la concessione ad Alessandro del solo privilegio `select`.  $\square$

#### 9.3.3.5 Revoca ricorsiva

Un problema di notevole interesse è quello della semantica da attribuire all'operazione di revoca dei diritti delegabili, di quei diritti cioè concessi con grant option. Il modello di controllo dell'accesso del System R adotta la cosiddetta *revoca ricorsiva* (o *a cascata* o *basata su timestamp*), in base alla quale un'operazione di revoca del privilegio  $p$  concesso sulla relazione  $rel$  all'utente  $u_1$  da parte dell'utente  $u_2$  ha l'effetto non solo di far perdere ad  $u_1$  il privilegio  $p$  sulla relazione  $rel$ , se  $u_1$  non ha ottenuto tale privilegio da fonti indipendenti, ma anche di modificare l'insieme di autorizzazioni nel sistema portandolo in uno stato equivalente a quello in cui si sarebbe trovato se  $u_2$  non avesse mai concesso ad  $u_1$  il privilegio  $p$  sulla relazione  $rel$ . Pertanto, dopo un'operazione di revoca il sistema deve *ricorsivamente* revocare tutti i privilegi che non avrebbero potuto essere concessi se l'utente specificato nel comando di revoca non avesse mai ricevuto il privilegio revocato.

La semantica della revoca ricorsiva può essere formalmente definita come segue.

**Definizione 9.2 (Revoca ricorsiva)** Siano  $G_1, \dots, G_n$  una sequenza di comandi GRANT di un singolo privilegio sulla stessa relazione, tali che se  $i < j$ ,  $1 \leq i, j \leq n$ , allora il comando  $G_i$  è eseguito prima del comando  $G_j$ . Sia  $R_i$  il comando che revoca il privilegio concesso con  $G_i$ . La semantica della revoca ricorsiva impone che l'insieme di autorizzazioni nel sistema sistema dopo l'esecuzione della sequenza di comandi:

$G_1, \dots, G_n, R_i$

sia identico allo stato raggiunto dopo l'esecuzione della sequenza di comandi:

$G_1, \dots, G_{i-1}, G_{i+1}, \dots, G_n$ .  $\diamond$

Implementare l'operazione di revoca ricorsiva non è banale. Infatti, per decidere se un privilegio deve essere ricorsivamente revocato è necessario determinare se il privilegio non avrebbe potuto essere concesso qualora il privilegio revocato non fosse mai esistito. Per caratterizzare più precisamente il problema, è opportuno dare una rappresentazione a grafo dello stato delle autorizzazioni rispetto ad un dato privilegio  $p$  ed una data relazione  $rel$ . Il grafo, chiamato *grafo delle autorizzazioni*, contiene un nodo per ogni utente che possiede il privilegio  $p$  sulla relazione  $rel$ . Esiste un arco dal nodo  $u_1$  al nodo  $u_2$  se l'utente  $u_1$  ha concesso  $p$  ad  $u_2$  su  $rel$ . L'arco è etichettato con il timestamp del privilegio e con la lettera 'g' se il privilegio è stato concesso con grant option ed è quindi delegabile. Il grafo contiene sempre un nodo relativo al creatore della relazione, in quanto egli è l'unico che all'inizio può concedere privilegi sulla relazione stessa. La Figura 9.4(a) illustra il grafo delle autorizzazioni relativo al privilegio `select` e alla relazione `Film`, corrispondente al catalogo `Sysauth` della Figura 9.3.

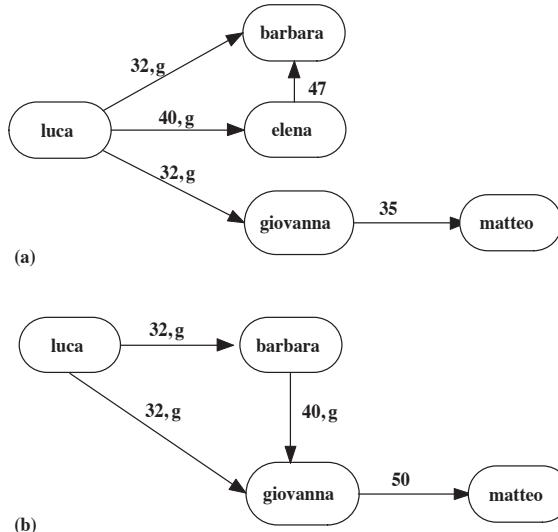


Figura 9.4: Grafi delle autorizzazioni

**Esempio 9.12** Con riferimento alla Figura 9.4(a), se Luca revocasse a Giovanna il privilegio `select` sulla relazione `Film` anche il privilegio concesso da Giovanna a Matteo sarebbe revocato in quanto non avrebbe mai potuto essere concesso se Luca non avesse concesso a Giovanna il privilegio per cui richiede la revoca.  $\square$

Le informazioni sul timestamp sono fondamentali per implementare correttamente la revoca ricorsiva, come dimostra l'esempio seguente.

**Esempio 9.13** Consideriamo il grafo delle autorizzazioni della Figura 9.4(b) relativo ad una sequenza di comandi `GRANT` per il privilegio `select` sulla relazione `Film` e supponiamo che Luca revochi a Giovanna il privilegio `select` sulla relazione `Film`. Questo non comporta la revoca in cascata del privilegio `select` che Giovanna ha concesso a Matteo, in quanto tale privilegio avrebbe potuto essere concesso da Giovanna al tempo 50 anche se Giovanna non lo avesse ricevuto da Luca al tempo 32, in virtù del comando `GRANT` effettuato da Barbara al tempo 40. Una situazione diversa si avrebbe se Barbara avesse concesso a Giovanna il privilegio `select` sulla relazione `Film` ad un tempo superiore al tempo 50. In questo caso, la revoca effettuata da Luca comporterebbe anche la revoca del privilegio concesso da Giovanna a Matteo in quanto al tempo 50 Giovanna non avrebbe potuto concedere il privilegio `select` a Matteo se non lo avesse ricevuto da Luca.  $\square$

La procedura che implementa la revoca ricorsiva è presentata nella Figura 9.5. La procedura riceve in ingresso una richiesta di revoca (*revoker,privilegio,relazione,revokee*), dove *revoker* è l'utente che richiede la revoca, mentre *revokee*

```

Procedura revoca_ricorsiva(revoker,privilegio,relazione,revokee)
Begin
    1. Seleziona da Sysauth le tuple con utente=revokee, nomeRel=relazione,
       grantor=revoker
    2. Poni a zero il valore dell'attributo privilegio nelle tuple selezionate, oppure cancella
       le tuple se tutte le entrate relative ai privilegi sono uguali a zero
    3. If privilegio=update Then
        Cancella le tuple con utente=revokee, nomeRel=relazione e grantor=revoker
        da Syscolauth
    EndIf
    4. min_tm := CURRENT_TIME
    5. For ogni tupla in Sysauth con utente=revokee, nomeRel=relazione e grantopt=Y Do
        If privilegio ≠ 0 e privilegio < min_tm Then min_tm := privilegio
    EndFor
    6. For ogni utente u tale che esiste in Sysauth una tupla con utente=u,
       nomeRel=relazione e grantor=revokee Do
        If privilegio < min_tm Then
            revoca_ricorsiva(revokee,privilegio,relazione,u)
        EndIf
    EndFor
End

```

Figura 9.5: Procedura per la revoca ricorsiva

è l'utente a cui il privilegio viene revocato. Tale procedura modifica i cataloghi **Sysauth** e **Syscolauth** per eliminare tutti i privilegi che, in base alla semantica della revoca ricorsiva, devono essere revocati in seguito all'esecuzione del comando di revoca. La procedura è una procedura ricorsiva di cui illustriamo il funzionamento tramite un esempio. Supponiamo che l'utente *A* revochi il privilegio *p* all'utente *B* sulla relazione *rel*. I passi (1) e (2) della procedura *revoca\_ricorsiva* aggiornano il catalogo **Sysauth** per eliminare il privilegio *p* concesso da *A* a *B* su *rel*. Se il privilegio revocato è **update** è inoltre necessario aggiornare il catalogo **Syscolauth** (passo (3)). Per determinare se le autorizzazioni per *p* concesse da *B* ad altri utenti debbano essere ricorsivamente revocate, la procedura determina, interrogando opportunamente il catalogo **Sysauth**, il tempo minimo, se diverso da zero, a cui *p* è stato concesso con grant option a *B* da utenti diversi da *A* (passi (4) e (5)). Un'autorizzazione per *p* concessa da *B* ad altri utenti è ricorsivamente revocata se il tempo in cui è stata specificata è maggiore del valore calcolato al passo precedente. Il procedimento è ripetuto per ogni utente i cui privilegi sono stati modificati durante l'esecuzione della procedura (passo (6)), fintantoché non sono necessarie ulteriori modifiche.

La procedura appena illustrata opera correttamente anche in presenza di cicli nel grafo delle autorizzazioni (vedi Esercizio 9.3 alla fine di questo capitolo). L'esempio seguente mostra il funzionamento della procedura di revoca.

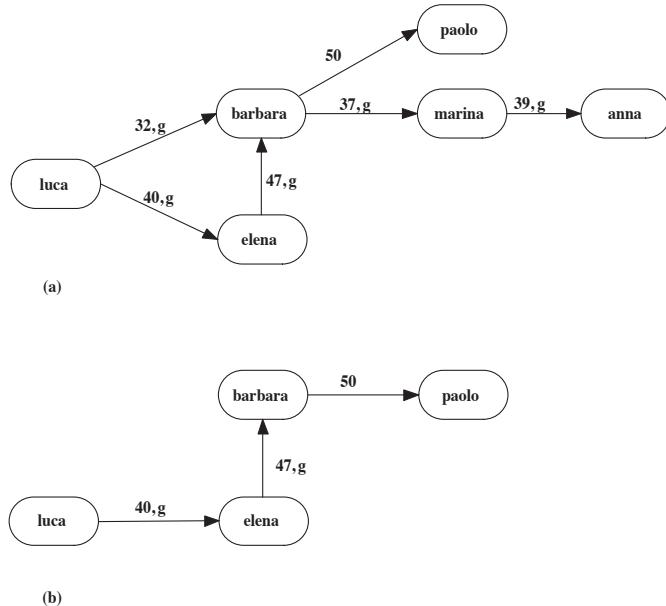


Figura 9.6: Autorizzazioni: (a) stato iniziale; (b) stato dopo la revoca di Luca a Barbara

**Esempio 9.14** Consideriamo il grafo delle autorizzazioni della Figura 9.6(a) relativo al privilegio `select` sulla relazione `Film` ed il relativo catalogo `Sysauth` riportato nella Figura 9.7. Supponiamo che Luca revochi a Barbara il privilegio `select`. La procedura della Figura 9.5 opera come segue:

1. Cancella dal catalogo `Sysauth` la tupla `(barbara,Film,R,32,luca,Y)`.
2. Calcola il minimo tra i timestamp dei privilegi `select` rimasti a Barbara con grant option sulla relazione `Film` dopo l'operazione di revoca:  $min\_tm = 47$ .
3. Considera i privilegi `select` sulla relazione `Film` concessi da Barbara ad altri utenti e confronta il loro timestamp con il minimo calcolato al passo precedente:
  - l'autorizzazione concessa da Barbara a Paolo ha timestamp  $50 > min\_tm$ : l'autorizzazione non viene revocata;
  - l'autorizzazione concessa da Barbara a Marina ha timestamp  $37 < min\_tm$ : la procedura viene ricorsivamente richiamata per eliminare tale autorizzazione.

I passi (1), (2) e (3) vengono eseguiti per ogni utente i cui privilegi sono stati modificati in seguito all'operazione di revoca. Come conseguenza, anche l'auto-

utente	nomeRel	tipo	select	grantor	grantopt
luca	Film	R	25		Y
barbara	Film	R	32	luca	Y
marina	Film	R	37	barbara	Y
anna	Film	R	39	marina	Y
elenia	Film	R	40	luca	Y
barbara	Film	R	47	elenia	Y
paolo	Film	R	50	barbara	N

Figura 9.7: Catalogo Sysauth per le autorizzazioni della Figura 9.6(a)

rizzazione concessa da Marina ad Anna viene ricorsivamente revocata. Il grafo risultante è mostrato nella Figura 9.6(b).  $\square$

La revoca ricorsiva, sebbene abbia una semantica ben definita e sia stata alla base di quasi tutti i modelli di controllo dell'accesso dei DBMS commerciali, è stata anche oggetto di critiche in quanto può portare alla cancellazione di un eccessivo numero di autorizzazioni. Infatti, spesso le autorizzazioni che un utente possiede dipendono dal ruolo che l'utente ricopre all'interno dell'organizzazione. Se un utente cambia ruolo, ad esempio in seguito ad una promozione, può essere utile revocare all'utente le autorizzazioni che possedeva, ma non quelle che egli aveva concesso ad altri utenti. Per tali motivi, sono state proposte dalla comunità scientifica semantiche alternative a quella della revoca ricorsiva. Ad esempio, è stato introdotto un nuovo tipo di revoca, chiamata *revoca senza cascata*, in base alla quale, quando un utente revoca un privilegio ad un altro utente, le autorizzazioni concesse da quest'ultimo in virtù del privilegio revocato non sono revocate ricorsivamente ma vengono rispecificate come se fossero state concesse dall'utente che ha richiesto l'operazione di revoca.

Anche lo standard SQL ha recepito questa linea di tendenza. Infatti, a partire dallo standard SQL:1999, la revoca può essere richiesta *con o senza cascata*. La revoca senza cascata prevede che un'operazione di revoca non venga eseguita se comporta la cancellazione di altre autorizzazioni, oltre a quella oggetto del comando. La revoca con cascata, invece, è una revoca ricorsiva, in cui però non sono considerati i timestamp delle autorizzazioni (vedi Paragrafo 9.4 per ulteriori dettagli).

#### 9.3.3.6 Autorizzazioni su viste

Le viste sono un importante meccanismo attraverso cui è possibile fornire forme più sofisticate di controllo dell'accesso, rispetto a quelle illustrate fino ad ora. Ad esempio, possono essere frequenti le situazioni in cui non tutti gli attributi di una relazione hanno gli stessi requisiti di protezione. La sintassi del comando GRANT discussa nel Paragrafo 9.3.3.1 non consente di concedere privilegi solo su alcuni attributi di una relazione, ad eccezione del privilegio update. Inoltre, per

la sintassi del comando **GRANT**, non è possibile autorizzare un utente a vedere solo alcune colonne di una relazione (ad esempio quelle relative al nome e cognome dei clienti). Questa politica per il controllo dell'accesso può essere implementata, nel modello del System R, mediante l'uso di viste. In questo caso, è sufficiente definire una vista come proiezione sugli attributi su cui vogliamo concedere i privilegi (**nome** e **cognome**) ed autorizzare l'accesso alla vista invece che alla relazione di base. Inoltre, le viste permettono di concedere privilegi *statistici*. Ad esempio, un utente potrebbe non essere autorizzato a vedere i titoli dei film noleggiati da un cliente, ma solo il numero di noleggi da lui effettuati. Tramite le viste, è possibile soddisfare questo requisito di sicurezza: basta definire una vista che computa il numero di noleggi effettuati da ogni cliente e concedere all'utente l'accesso alla vista invece che alle relazioni di base. Infine, le viste consentono di realizzare il cosiddetto *controllo dell'accesso in base al contenuto*, ovvero permettono di autorizzare l'accesso solo a specifiche tuple di una relazione, sulla base dei valori dei loro attributi. Ad esempio, se vogliamo autorizzare un utente a vedere solo le tuple della relazione **Film** relative a commedie, è sufficiente definire una vista che seleziona dalla relazione **Film** le tuple che soddisfano tale condizione e concedere all'utente il privilegio **select** sulla vista, invece che sulla relazione di base.

L'utilizzo delle viste comporta quindi notevoli vantaggi rispetto alla tipologia di politiche che possiamo implementare. È però opportuno chiarire alcuni aspetti connessi al loro corretto utilizzo.

In primo luogo, un utente può creare una vista solo se ha il privilegio **select** sulle relazioni/viste su cui è definita. Un'altra importante questione riguarda i privilegi che l'utente che crea una vista può esercitare sulla vista stessa. In precedenza, abbiamo visto come l'utente che crea una relazione di base possa esercitare su di essa tutti i privilegi previsti dal modello del System R con facoltà di delega. Per le viste, invece, i privilegi che l'utente che le definisce ha su di esse dipendono da due fattori: *(i)* le autorizzazioni che l'utente possiede sulle relazioni/viste su cui la vista è definita; *(ii)* la semantica della vista, ovvero la sua definizione in termini delle relazioni o viste componenti.

Rispetto al punto *(i)*, se una vista è definita su una singola relazione (o vista), i privilegi che l'utente che crea la vista ha su di essa sono gli stessi che ha sulla relazione o vista componente. I privilegi sulla vista saranno delegabili o meno, a seconda di come sono definiti per la relazione o vista componente. Nel caso in cui la vista sia definita su più relazioni (o viste), i privilegi che l'utente che crea la vista ha su di essa sono ottenuti intersecando i privilegi che l'utente ha su tutte le relazioni (o viste) componenti. Inoltre, un privilegio sulla vista è delegabile solo se il creatore della vista ha il diritto di delegare tale privilegio su tutte le relazioni o viste componenti. Per quanto concerne il punto *(ii)*, abbiamo visto nel Capitolo 3 come SQL ponga alcune restrizioni sulle operazioni che possono essere effettuate su una vista. Ad esempio, una vista che computa delle statistiche non può essere aggiornata. Queste restrizioni si riflettono anche sulle autorizzazioni che un utente che crea una vista ha sulla vista stessa. Le autorizzazioni che un utente ha sulle relazioni/viste su cui la vista è definita potranno essere esercitate anche sulla vista

solo se l'interrogazione di definizione della vista lo consente.

**Esempio 9.15** Con riferimento al catalogo **Sysauth** della Figura 9.3, supponiamo che Barbara esegua il comando:

```
CREATE VIEW Commedie AS
SELECT * FROM Film
WHERE genere = 'commedia';
```

L'esecuzione di tale comando viene autorizzata in quanto Barbara ha il privilegio **select** su **Film**. Questo è l'unico privilegio che Barbara può esercitare sulla vista appena creata, in quanto è l'unico che possiede sulla relazione **Film**. Inoltre, Barbara può concedere a terzi tale privilegio, avendolo ricevuto da Luca con grant option. Supponiamo ora che Elena crei la vista **NumNoleggi**, definita come segue:

```
CREATE VIEW NumNoleggi AS
SELECT codCli, COUNT(*) AS NumNol
FROM Noleggi
GROUP_BY codCli;
```

In base al catalogo **Sysauth** della Figura 9.3 Elena ha sulla relazione **Film** i privilegi **select**, **insert** ed **update**. Per le considerazioni viste in precedenza, questi privilegi dovrebbero essere esercitabili da Elena anche sulla vista a meno che l'interrogazione di definizione della vista sia tale che alcuni di essi non siano esercitabili. Le restrizioni discusse nel Capitolo 3, fanno sì che i privilegi **update** ed **insert** non siano esercitabili sulla vista. Elena può quindi esercitare sulla vista solo il privilegio **select**. Inoltre, siccome possedeva tale privilegio con grant option sulla relazione **Film** potrà concedere a terzi l'autorizzazione di selezionare tuple dalla vista **NumNoleggi**.  $\square$

Quando un utente crea una vista, il catalogo **Sysauth** viene aggiornato con due tuple a lui relative, corrispondenti ai privilegi, delegabili e non, che l'utente può esercitare. I privilegi sono determinati in base ai punti *(i)* e *(ii)* illustrati in precedenza. Il timestamp associato ai privilegi è quello del relativo comando **CREATE VIEW**.

La concessione di privilegi su una vista è molto simile a quella su relazioni di base: i privilegi che un utente può concedere ad altri su una vista sono quelli che possiede con grant option. Le operazioni di revoca sono, invece, più complicate, in quanto è necessario stabilire cosa succede ad una vista se un privilegio **select** su una delle relazioni su cui è definita viene revocato. L'approccio seguito dal System R, in accordo alla semantica della revoca ricorsiva, è quello di cancellare la vista se il privilegio revocato era l'unico utile per la sua definizione.

**Esempio 9.16** Consideriamo le viste dell'Esempio 9.15 e il catalogo **Sysauth** riportato nella Figura 9.3 e supponiamo che Luca revochi ad Elena il privilegio **select** sulla relazione **Noleggio**. In questo caso, la vista **NumNoleggi** viene a sua

volta cancellata in quanto il privilegio `select` che Luca aveva concesso ad Elena era per lei l'unico utile per creare la vista. Viceversa, se Luca revocasse a Barbara il privilegio `select` su `Film`, la decisione se cancellare o meno la vista `Commedie` creata da Barbara dipenderebbe dal timestamp del comando `CREATE VIEW`. Se il timestamp è maggiore di 47 la vista non è cancellata in quanto Barbara ha ricevuto da Elena il privilegio `select` su `Film` al tempo 47. Invece, nel caso in cui il comando `CREATE VIEW` sia stato eseguito da Barbara prima del tempo 47, la vista viene ricorsivamente revocata.  $\square$

#### 9.3.4 Modelli basati su ruoli

Una delle più rilevanti estensioni proposte ai modelli di controllo dell'accesso fino ad ora illustrati è l'introduzione del concetto di *ruolo*, brevemente descritto nel Paragrafo 9.2.1.2. Nel *controllo dell'accesso basato su ruoli (RBAC – Role Based Access Control)*, i ruoli rappresentano delle funzioni che gli utenti ricoprono all'interno dell'organizzazione o azienda in cui operano (sono esempi di ruolo per il dominio della videoteca `commesso`, `cliente` e `direttoreVideoteca`). I privilegi sono concessi ai ruoli invece che ai singoli utenti. Le autorizzazioni specificate per un ruolo sono quelle necessarie per esercitare le funzioni connesse al ruolo stesso. Gli utenti sono abilitati a ricoprire uno o più ruoli, in base alle mansioni che devono svolgere. L'abilitazione a ricoprire un ruolo implica l'acquisizione di tutte le autorizzazioni ad esso connesse.

Il controllo dell'accesso basato su ruoli ha suscitato subito un notevole interesse, dimostrato dai numerosi modelli di controllo dell'accesso proposti da gruppi di ricerca e dal fatto che quasi tutti i DBMS commerciali e lo standard SQL forniscono tale funzionalità, seppure con delle differenze. Per supplire a questa mancanza di standardizzazione e rendere più agevole lo sviluppo di meccanismi per il controllo dell'accesso basati su ruoli, è stato definito uno standard NIST (*National Institute of Standards and Technology*) che introduce un modello di riferimento. Lo standard è stato concepito in maniera modulare ed è costituito da tre componenti: *Modello base (Core RBAC)*, *Modello gerarchico (Hierarchical RBAC)*, *Modello con vincoli (Constrained RBAC)*. Il modello base definisce i requisiti minimi per realizzare il controllo dell'accesso basato su ruoli. Gli altri due modelli, non dipendenti l'uno dall'altro, aggiungono al modello base, rispettivamente, la strutturazione gerarchica dei ruoli e la possibilità di specificare vincoli sull'attivazione e sull'assegnazione dei ruoli. L'idea alla base di questa organizzazione modulare dello standard è che non esiste un unico modello adatto a tutti gli ambienti ed i domini applicativi. In questo modo, è possibile conformarsi allo standard realizzando una sola o più componenti, a seconda delle specifiche esigenze. Nel seguito, illustreremo brevemente ognuna delle componenti dello standard.

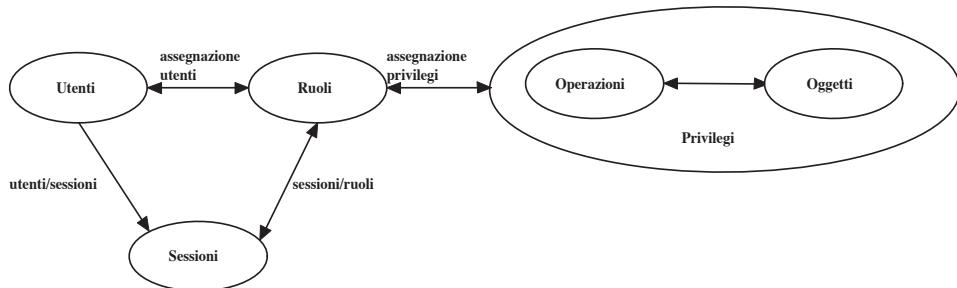


Figura 9.8: Modello RBAC base

#### 9.3.4.1 Modello base

Il modello base si compone di quattro componenti principali: *Utenti*, *Ruoli*, *Privilegi* e *Sessioni*. Le prime due componenti sono già state illustrate in precedenza. I *Privilegi* indicano i diritti d'accesso esercitabili sugli oggetti del sistema. Gli elementi dell'insieme *Privilegi* sono coppie  $(obj, op)$ , dove  $obj$  è un oggetto ed  $op$  è un'operazione. Infine, il concetto di sessione è stato introdotto perché un utente, quando effettua il log-in al sistema, può attivare un sottoinsieme dei ruoli che è abilitato a ricoprire. Una sessione stabilisce quindi la corrispondenza tra un utente ed i ruoli attivi durante la sua connessione al sistema. Il modello base stabilisce inoltre le relazioni che intercorrono tra le componenti appena introdotte, illustrate graficamente nella Figura 9.8, dove le doppie frecce indicano relazioni molti a molti.

Dall'analisi della Figura 9.8 è evidente come il modello base preveda che un utente possa essere assegnato a più ruoli, mentre un ruolo può essere ricoperto da più utenti (tutti quelli che ricoprono la funzione ad esso associata). Le autorizzazioni sono specificate per i ruoli e non per utenti singoli (infatti non esiste alcuna freccia che collega utenti a privilegi). Ruoli e privilegi sono anch'essi legati da una relazione molti a molti. Infine, ogni sessione mette in corrispondenza un utente con l'insieme dei ruoli attivi per quella sessione. Ogni sessione è associata ad un singolo utente, mentre un utente può attivare diverse sessioni (e diversi ruoli in ognuna di esse).

#### 9.3.4.2 Modello gerarchico

Il modello base non prevede alcuna strutturazione gerarchica dell'insieme dei ruoli (per questo è anche conosciuto come “*Modello flat*”). Il modello gerarchico aggiunge al modello base la possibilità di strutturare i ruoli in gerarchie. Questa funzionalità permette di modellare al meglio tutte quelle realtà dove le funzioni aziendali od organizzative sono strutturate in una gerarchia che riflette diversi livelli di autorità e responsabilità. Una gerarchia sui ruoli, di cui un esempio è illustrato nella Figura 9.9, definisce quindi un ordinamento parziale tra di essi, in

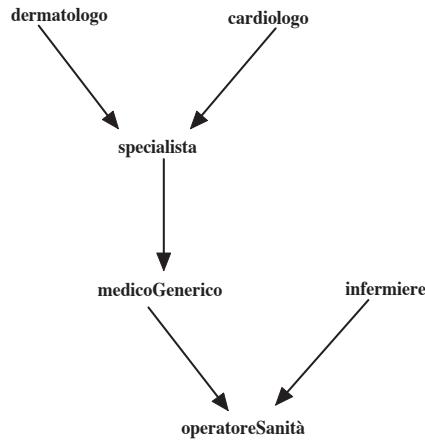


Figura 9.9: Gerarchia dei ruoli

quanto non tutti i ruoli sono necessariamente in relazione gerarchica. Tale gerarchia induce un'ereditarietà dei privilegi tra ruoli nella gerarchia e stabilisce una relazione tra gli utenti abilitati a ricoprire i vari ruoli.

Più precisamente, il modello gerarchico introduce una relazione d'ordine parziale sull'insieme dei ruoli, chiamata *relazione di ereditarietà* e denotata con  $\geq$ , tale per cui, dati due ruoli  $r_1$  ed  $r_2$ ,  $r_1 \geq r_2$ <sup>4</sup> implica che  $r_1$  eredita tutti i privilegi assegnati ad  $r_2$  e tutti gli utenti associati ad  $r_1$  sono anche utenti associati ad  $r_2$ . Questo principio di ereditarietà è motivato dal fatto che, in linea di principio, un ruolo dovrebbe poter effettuare sugli oggetti del sistema tutte le operazioni che possono essere effettuate da ruoli corrispondenti a funzioni più in basso nella gerarchia. Naturalmente, possono esistere casi in cui tale principio generale può non essere conveniente. Pensiamo ad esempio ad un sistema per la sorveglianza di un impianto critico, dove alcuni addetti molto specializzati sono capaci di compiere delle operazioni che i loro responsabili non sono in grado di effettuare. In queste situazioni, basta configurare la gerarchia dei ruoli in maniera tale da non mettere in relazione i due ruoli per cui il principio di ereditarietà non deve valere.

#### 9.3.4.3 Modello con vincoli

Il modello con vincoli aggiunge al modello base la possibilità di specificare vincoli. In realtà, lo standard attuale considera solo una tipologia di vincolo, nota come *separazione delle mansioni* (*SoD - Separation of Duties*), mentre altre categorie di vincoli, quali ad esempio vincoli sul tempo di attivazione dei ruoli, non sono al momento considerate nello standard. La separazione delle mansioni è un vincolo

<sup>4</sup> $r_1$  è detto anche *ruolo senior*, mentre  $r_2$  *ruolo junior*.

di sicurezza molto importante che impone delle restrizioni sull'insieme dei ruoli esercitabili o attivabili da un certo utente, per prevenire situazioni che potrebbero portare un utente ad avere troppo potere senza un adeguato controllo. I vincoli per la separazione delle mansioni possono essere classificati in due categorie: *vincoli statici* e *vincoli dinamici*. I vincoli statici stabiliscono delle relazioni di mutua esclusione tra ruoli assegnabili ad un certo utente. Ad esempio, la politica aziendale potrebbe impedire che un utente sia assegnato contemporaneamente al ruolo `segretarioAmministrativo` e `revisoreConti`, in quanto il secondo esercita una funzione di controllo sul primo. Nel modello con vincoli questa nozione di ruoli incompatibili è generalizzata a più di due ruoli. Ad esempio, per ragioni di garanzia, potremmo imporre che ogni utente non possa ricoprire più di due ruoli tra quelli che corrispondono a funzioni direttive all'interno di un'organizzazione. Più precisamente, un vincolo di separazione delle mansioni di tipo statico è formalizzato come una coppia  $(RS, n)$ , dove  $RS$  è un sottoinsieme dei ruoli previsti dal modello ed  $n$  è un numero naturale. Tale vincolo stabilisce che nessun utente può essere abilitato a ricoprire un numero di ruoli maggiore o uguale ad  $n$  tra quelli in  $RS$ . Se i ruoli sono strutturati in gerarchia, i vincoli di separazione delle mansioni vengono propagati lungo la gerarchia. Ad esempio, con riferimento alla Figura 9.9, se per il ruolo `specialista` è specificato un vincolo di separazione delle mansioni, questo vale anche per i ruoli `senior cardiologo` e `dermatologo`.

Abbiamo visto come i vincoli statici di separazione delle mansioni impongano delle restrizioni sull'insieme di ruoli che possono essere ricoperti da un utente. Ci sono però situazioni in cui un utente potrebbe essere autorizzato ad attivare due ruoli separatamente, perché questo non causa alcun problema di conflitto di interessi, mentre non dovrebbe essere autorizzato ad attivarli entrambi nella stessa sessione. Ad esempio, un utente potrebbe essere autorizzato ad attivare il ruolo `cassiere` e `supervisoreCassiere` in due momenti diversi, ma non contemporaneamente nella stessa sessione (in quanto il principio di separazione delle mansioni impone che le funzioni operative e di controllo siano mantenute separate). I vincoli dinamici di separazione delle mansioni consentono di imporre delle restrizioni sull'insieme dei ruoli che un utente può attivare in una sessione. Formalmente, sono modellati come i vincoli statici, cioè come una coppia  $(RS, n)$ , dove  $RS$  è un sottoinsieme dei ruoli previsti dal modello ed  $n$  è un numero naturale. La semantica è però diversa in quanto un vincolo dinamico impone che nessun utente possa attivare contemporaneamente un numero di ruoli maggiore o uguale ad  $n$  tra quelli in  $RS$ .

#### 9.4 Controllo dell'accesso in SQL

SQL è basato, per quanto riguarda il controllo dell'accesso, sul modello del System R illustrato nel Paragrafo 9.3.3, però con alcune importanti differenze ed estensioni. Tra le più rilevanti differenze vi è il supporto per i ruoli, una diversa semantica per l'operazione di revoca e l'aggiunta di nuovi privilegi ed oggetti di autorizzazione. Nel seguito, illustreremo solo le caratteristiche fondamentali del controllo

dell'accesso in SQL, focalizzandoci sulle differenze rispetto al modello di controllo dell'accesso del System R.

La prima differenza con il modello di controllo dell'accesso del System R è la possibilità di definire ruoli. I ruoli possono essere creati con il comando `CREATE ROLE <nome ruolo>` ed eliminati con il comando `DROP ROLE <nome ruolo>`, dove `<nome ruolo>` indica il nome del ruolo che vogliamo creare o cancellare, rispettivamente. Altre differenze riguardano i comandi `GRANT` e `REVOKE`, oggetto dei successivi paragrafi.

#### 9.4.1 Comando GRANT

Il comando `GRANT` del System R è stato esteso con la possibilità di concedere privilegi non solo ad utenti ma anche a ruoli. Inoltre, il comando è stato esteso con la possibilità di autorizzare un utente non solo all'esercizio di privilegi ma anche a ricoprire uno o più ruoli. Altre estensioni, come ad esempio l'insieme di privilegi ed oggetti previsti dal comando, riflettono le estensioni apportate al modello relazionale nel corso degli anni (discusse brevemente nel Capitolo 1). Alcune di tali estensioni saranno illustrate in dettaglio nei Capitoli 10 e 11 a cui rimandiamo il lettore per una piena comprensione degli argomenti nel seguito illustrati.

Più precisamente, il comando `GRANT` in SQL ha una duplice sintassi, a seconda che venga utilizzato per concedere privilegi (ad utenti o ruoli) o per abilitare utenti/ruoli a ricoprire ruoli. La prima forma ha la sintassi di seguito illustrata:

```
GRANT {<lista privilegi> | ALL PRIVILEGES}
    ON [<qualificatore oggetto>] <nome oggetto>
    TO {<lista utenti> | <lista ruoli> | PUBLIC}
        [{WITH GRANT OPTION | WITH HIERARCHY OPTION}];
```

dove:

- `<lista privilegi>` indica l'insieme dei privilegi concessi con il comando `GRANT`. La parola chiave `ALL PRIVILEGES` indica tutti i privilegi previsti dal modello.
- `<nome oggetto>` indica il nome dell'oggetto della base di dati su cui sono concessi i privilegi. In alcuni casi, è inoltre necessario specificare un qualificatore dell'oggetto (`<qualificatore oggetto>`) prima dell'oggetto stesso (ad esempio usiamo la parola chiave `TYPE` se l'oggetto dell'autorizzazione è un tipo definito dall'utente).<sup>5</sup>
- `<lista utenti>` indica l'insieme degli utenti a cui vengono concessi i privilegi.
- `<lista ruoli>` indica l'insieme dei ruoli a cui vengono concessi i privilegi. La parola chiave `PUBLIC` consente di specificare le autorizzazioni implicate dal comando per tutti gli utenti/ruoli del sistema.

---

<sup>5</sup>I tipi definiti dall'utente verranno illustrati nel Capitolo 10.

- La clausola opzionale **WITH GRANT OPTION** consente la delega dell'amministrazione dei privilegi oggetto del comando, in analogia al corrispondente comando nel modello del System R (vedi Paragrafo 9.3.3.1).
- La clausola opzionale **WITH HIERARCHY OPTION** può essere specificata solo per il privilegio **select** e consente, nel caso di relazioni legate da una gerarchia di ereditarietà (vedi Capitolo 10 per una trattazione di questi argomenti), di propagare il privilegio a tutte le sotto-tabelle della tabella riferita nel comando **GRANT**.

Vediamo ora quali sono le principali estensioni all'insieme dei privilegi ed oggetti di autorizzazione rispetto a quelli previsti dal modello del System R. Innanzitutto i privilegi **select** e **insert**, in aggiunta al privilegio **update**, possono essere concessi selettivamente solo su alcune colonne di una relazione (vale la stessa sintassi vista nel Paragrafo 9.3.3.1 per il privilegio **update**). Altri privilegi previsti dallo standard, dovuti alle estensioni apportate al modello relazionale nel corso degli anni, sono: **references**, che permette di utilizzare una colonna in un vincolo o asserzione, **trigger**, che permette di specificare trigger che operano su una certa relazione,<sup>6</sup> **under**, che permette la creazione di sotto-tipi o sotto-tabelle,<sup>7</sup> **usage**, che permette di utilizzare un oggetto dello schema (ad esempio un tipo) nella definizione di un altro oggetto, ed **execute**, che permette l'esecuzione di una procedura o funzione. SQL pone alcune restrizioni sui privilegi specificabili in relazione alla tipologia di oggetto. Ad esempio, i privilegi **usage** ed **execute** non possono essere specificati per relazioni, mentre **trigger** può essere specificato solo per relazioni di base.

Anche SQL, come il modello del System R, prevede l'amministrazione decentralizzata dei privilegi mediante ownership. Inoltre, il proprietario di un oggetto è l'unico abilitato ad esercitare su di esso i privilegi **drop** (cancellazione) e **alter** (modifica).

Come abbiamo detto in precedenza, il comando **GRANT** ha in SQL una duplice sintassi per consentire, oltre alla concessione di privilegi, anche di abilitare utenti/ruoli a ricoprire dei ruoli. In questa seconda accezione la sintassi è la seguente:

```
GRANT <lista ruoli concessi>
TO {<lista utenti> | <lista ruoli> | PUBLIC}
[WITH ADMIN OPTION];
```

dove:

- <lista ruoli concessi> indica l'insieme dei ruoli concessi con il comando **GRANT**.
- <lista utenti> indica l'insieme degli utenti per cui vengono abilitati i ruoli concessi con il comando.

---

<sup>6</sup>I trigger saranno oggetto del Capitolo 11.

<sup>7</sup>Vedi il Capitolo 10 per ulteriori dettagli.

- <lista ruoli> indica l'insieme dei ruoli per cui vengono abilitati i ruoli concessi con il comando. La parola chiave PUBLIC consente di abilitare i ruoli specificati nel comando per tutti gli utenti/ruoli del sistema.
- la clausola opzionale WITH ADMIN OPTION è l'analogo per i ruoli della grant option; quando si è abilitati a ricoprire un ruolo con admin option non solo vengono ereditate tutte le autorizzazioni specificate per quel ruolo, ma è anche possibile concedere a terzi la possibilità di ricoprire il ruolo.

Notiamo che la sintassi del comando GRANT permette di abilitare un ruolo a ricoprirne un altro (ereditando quindi le sue autorizzazioni). Questo è il modo con cui SQL fornisce implicitamente la possibilità di strutturare i ruoli in gerarchia (vedi Paragrafo 9.3.4.2).

**Esempio 9.17** I seguenti sono esempi di comandi GRANT:

```
GRANT usage ON TYPE indirizzo TO giovanna WITH GRANT OPTION;
GRANT execute ON aggiornaClienti TO elena;
GRANT select(nome, cognome), references(codCli) ON Clienti TO marco;
GRANT delete, update ON Clienti TO direttoreVideoteca;
GRANT direttoreVideoteca TO roberto WITH ADMIN OPTION;
```

Il primo comando consente a Giovanna di utilizzare il tipo **indirizzo** nella definizione di altri oggetti dello schema e di concedere a terzi tale privilegio. Il secondo comando consente ad Elena di eseguire la procedura **aggiornaClienti**, mentre il terzo comando concede a Marco la possibilità di formulare interrogazioni sugli attributi **nome** e **cognome** della relazione **Clienti** e di specificare vincoli che coinvolgono l'attributo **codCli** della stessa relazione. Gli ultimi due comandi, invece, coinvolgono ruoli. Il primo assegna al ruolo **direttoreVideoteca** il privilegio di cancellare e modificare tuple della relazione **Clienti**, mentre il secondo assegna a Roberto il ruolo **direttoreVideoteca** e la facoltà di abilitare terzi a ricoprire questo ruolo. □

#### 9.4.2 Comando REVOKE

Anche per quanto riguarda l'operazione di revoca lo standard SQL prevede alcune importanti novità rispetto al modello del System R. Come per il comando GRANT anche per il comando REVOKE vi è una duplice sintassi, a seconda che lo utilizziamo per la revoca di privilegi o di ruoli. Iniziamo, in primo luogo, a vedere la sintassi del comando per la revoca di privilegi:

```
REVOKE [{GRANT OPTION FOR | HIERARCHY OPTION FOR}]
ON [<qualificatore oggetto>] <nome oggetto>
FROM {<lista utenti> | <lista ruoli>}
{RESTRICT|CASCADE};
```

dove:

- le clausole opzionali GRANT OPTION FOR e HIERARCHY OPTION FOR servono per revocare la sola grant o hierarchy option, mantenendo il diritto ad esercitare i privilegi oggetto del comando di revoca.
- <lista privilegi> indica l'insieme di privilegi oggetto del comando di revoca.
- <nome oggetto> indica il nome dell'oggetto della base di dati su cui sono revocati i privilegi. In alcuni casi è inoltre necessario specificare un qualificatore dell'oggetto (<qualificatore oggetto>) prima dell'oggetto stesso, in analogia a quanto visto per il comando GRANT.
- <lista utenti> indica l'insieme degli utenti a cui sono revocati i privilegi.
- <lista ruoli> indica l'insieme dei ruoli a cui sono revocati i privilegi.
- Le clausole RESTRICT e CASCADE servono per gestire l'operazione di revoca ed il loro significato verrà precisato in seguito.

Il comando per revocare l'abilitazione a ricoprire ruoli ha la seguente sintassi:

```
REVOKE [ADMIN OPTION FOR] <lista ruoli revocati>
FROM {<lista utenti> | <lista ruoli>}
{RESTRICT | CASCADE};
```

dove:

- la clausola opzionale ADMIN OPTION FOR serve per revocare la sola admin option, mantenendo il diritto a ricoprire i ruoli oggetto del comando di revoca.
- <lista ruoli revocati> indica l'insieme di ruoli revocati con il comando.
- <lista utenti> indica l'insieme degli utenti a cui viene revocata l'abilitazione a ricoprire i ruoli oggetto della revoca.
- <lista ruoli> indica l'insieme dei ruoli a cui viene revocata l'abilitazione a ricoprire i ruoli oggetto della revoca.
- le clausole RESTRICT e CASCADE servono per gestire l'operazione di revoca ed il loro significato verrà precisato in seguito.

La prima differenza con l'analogico comando del System R è che il comando REVOKE in SQL può essere utilizzato per revocare sia privilegi sia autorizzazioni a ricoprire un ruolo. Inoltre, vi è la possibilità di revocare solo la grant o l'admin o la hierarchy option, senza revocare il corrispondente privilegio.

**Esempio 9.18** Consideriamo i comandi GRANT dell’Esempio 9.17 ed i seguenti comandi REVOKE:

```
REVOKE GRANT OPTION FOR usage ON TYPE indirizzo FROM giovanna;
REVOKE delete ON Clienti FROM direttoreVideoteca;
REVOKE direttoreVideoteca FROM roberto;
```

Il primo comando è un esempio di revoca della sola grant option. L’effetto è che Giovanna può ancora utilizzare il tipo `indirizzo` nella specifica di altri oggetti dello schema, ma non può più concedere a terzi questo privilegio. Con il secondo comando viene revocato al ruolo `direttoreVideoteca` (e quindi implicitamente anche a tutti gli utenti abilitati a ricoprirlo) il privilegio di cancellare tuple dalla relazione `Clienti`. Infine, con il terzo comando viene revocata a Roberto l’abilitazione a ricoprire il ruolo `direttoreVideoteca`. □

Una delle differenze più importanti dello standard SQL rispetto al modello di controllo dell’accesso del System R riguarda la semantica da attribuire all’operazione di revoca. Lo standard prevede due possibilità. Se la revoca di un privilegio è richiesta con l’opzione `RESTRICT`, allora l’esecuzione del comando non viene concessa se questo comporta la revoca di altri privilegi, oppure la cancellazione di oggetti dello schema (ad esempio nel caso di viste create grazie al privilegio revocato). Se la revoca invece è richiesta con l’opzione `CASCADE`, viene implementata una revoca, simile alla revoca ricorsiva del System R (vedi Paragrafo 9.3.3.5), ma senza considerare i timestamp. Le autorizzazioni che un utente a cui è stato revocato un privilegio ha specificato non sono ricorsivamente revocate se l’utente ha ricevuto da altre fonti indipendenti il privilegio specificato nel comando `REVOKE` con grant option, indipendentemente dal tempo in cui ha ricevuto il privilegio; sono ricorsivamente revocate, in caso contrario. Il seguente esempio chiarisce le differenze.

**Esempio 9.19** Consideriamo l’Esempio della Figura 9.4(b) e supponiamo che Barbara abbia concesso a Giovanna il privilegio `select` su `Film` al tempo 55 invece che 40. Supponiamo ora che Luca revochi a Giovanna il privilegio concesso al tempo 32. Questo non comporterebbe, come invece avviene nel modello del System R, la revoca del privilegio concesso da Giovanna a Matteo, in quanto Giovanna continua ad avere il privilegio `select` con grant option sulla relazione `Film` concesso da Barbara. □

Considerazioni analoghe valgono per l’operazione di revoca dei ruoli.

### Note conclusive

Proteggere le informazioni memorizzate in un DBMS è un compito complesso che coinvolge diversi attori: dai servizi del DBMS stesso, alle reti, ai sistemi operativi. In questo capitolo, ci siamo concentrati sul controllo dell’accesso in quanto

i meccanismi che lo realizzano sono per la maggior parte di stretta competenza del DBMS. Inoltre, ci siamo principalmente focalizzati sulla protezione dei dati memorizzati in un DBMS relazionale. Per ragioni di spazio, abbiamo tralasciato una serie di importanti problematiche che hanno dato origine ad una notevole attività di ricerca. Innanzitutto, una cospicua attività di ricerca è stata condotta per definire modelli di controllo dell'accesso per DBMS non relazionali, quali ad esempio DBMS ad oggetti, attivi [FT00] e, più recentemente, per sorgenti dati XML [BS05]. Inoltre, ci sono state numerose proposte volte ad estendere il potere espressivo dei modelli discrezionali più noti, quali ad esempio il modello del System R. Tra le principali proposte, ricordiamo il supporto per autorizzazioni positive e negative, autorizzazioni basate sul contenuto, autorizzazioni forti e deboli, autorizzazioni implicite definite opportune regole, ed autorizzazioni temporali [FT00]. Un'altra rilevante area di ricerca è stata quella connessa allo sviluppo di modelli di controllo dell'accesso di tipo mandatorio. Numerosi modelli sono stati proposti così come prototipi e prodotti commerciali [FT00]. In tale ambito, sono stati anche studiati problemi relativi all'inferenza e protocolli sicuri per il controllo della concorrenza e per le operazioni di recovery [AJG99]. Infine, il problema del controllo dell'accesso è stato investigato anche per nuove applicazioni dei sistemi di gestione dati, quali ad esempio i sistemi di data warehouse e data mining, i DBMS multimediali, i sistemi per la gestione di workflow [FT00], i sistemi informativi geografici e le applicazioni web [FT05].

### Note bibliografiche

Per una trattazione approfondita delle tematiche di sicurezza rimandiamo il lettore ai due testi di Bishop [Bis02] e [Bis05]. Il modello di Bell e LaPadula è illustrato in [BL75], mentre per il modello di controllo dell'accesso del System R gli articoli di riferimento sono [Fag76] e [GW76]. [FSG<sup>+</sup>01] è l'articolo di riferimento per quanto riguarda il controllo dell'accesso basato su ruoli. In tale articolo sono anche illustrate le funzionalità amministrative per la specifica ed il mantenimento delle componenti dei vari modelli previsti dallo standard e per effettuare il controllo dell'accesso, che non sono state trattate in questo capitolo. Infine, per SQL il riferimento è la documentazione relativa allo standard SQL:2003 [ISO03].

### Esercizi

**9.1** Consideriamo le seguenti classi di accesso:

$$\begin{aligned} c_1 &= (\text{TS}, \{\text{Navy}, \text{Nuclear}\}) \\ c_2 &= (\text{U}, \{\text{Navy}\}) \\ c_3 &= (\text{S}, \{\text{Navy}\}) \\ c_4 &= (\text{C}, \{\text{Air Force}, \text{Nato}\}) \\ c_5 &= (\text{TS}, \{\text{Navy}, \text{Army}\}) \end{aligned}$$

Determinare:

- a. Le relazioni che intercorrono tra tali classi.
- b. Gli accessi che soggetti con classe di accesso  $c_i$  possono esercitare su oggetti con classe di accesso  $c_j$ ,  $\forall i, j = 1, \dots, 5$ .

**9.2** Consideriamo il modello di controllo dell'accesso del System R e la relazione:

`AcquistoProdotti(numProdotto, prezzoUnitario, data, quantità, nome)`  
creata da Federica. Supponiamo che:

- al tempo 10 Federica crei una vista che restituisce per ogni prodotto la media del prezzo unitario;
  - al tempo 16 Federica conceda tutti i privilegi su tale vista a Matteo con grant option;
  - al tempo 20 Matteo conceda tutti i privilegi sulla vista a Michela con grant option;
  - al tempo 28 Federica conceda tutti i privilegi sulla vista a Michela;
  - al tempo 30 Federica revochi a Matteo i privilegi che gli aveva precedentemente concesso.
- a. Scrivere il comando per la definizione della vista e determinare le operazioni che Federica può eseguire su di essa.
  - b. Scrivere i comandi per concedere e revocare i privilegi sulla vista elencati in precedenza.
  - c. Determinare se dopo l'operazione di revoca eseguita da Federica, Michela può accedere alla vista e concedere ad altri utenti privilegi su tale vista.
  - d. Rispondere al quesito al punto (c) considerando lo standard SQL invece del modello di controllo dell'accesso del System R.

**9.3** Consideriamo il modello di controllo dell'accesso del System R ed i seguenti comandi:

```
luca,50:8 GRANT select on Film to marco WITH GRANT OPTION;
marco,52: GRANT select on Film to giovanna WITH GRANT OPTION;
giovanna,55: GRANT select on Film to matteo WITH GRANT OPTION;
matteo,57: GRANT select on Film to luca;
luca,60: REVOKE select on Film FROM marco;
```

- a. Costruire il grafo delle autorizzazioni relativo alla sequenza di comandi GRANT e le corrispondenti entrate nel catalogo Sysauth.
- b. Eseguire la procedura della Figura 9.5 e determinare lo stato delle autorizzazioni dopo la sua esecuzione.
- c. Descrivere il catalogo Sysauth dopo l'operazione di revoca al punto (b).
- d. Rispondere ai quesiti ai punti (b) e (c) considerando lo standard SQL invece del modello di controllo dell'accesso del System R.

**9.4** Consideriamo il modello basato su ruoli ed i ruoli `direttoreVideoteca`, `commesso`, `magazziniere` e `contabile`, tali che  $\text{direttoreVideoteca} \geq \text{commesso}$ ,  $\text{direttoreVideoteca} \geq \text{magazziniere}$  e  $\text{direttoreVideoteca} \geq \text{contabile}$ . Supponiamo che il ruolo `commesso` possa effettuare interrogazioni sulla relazione `Film`, mentre il ruolo `contabile` possa inserire tuple nella relazione `Cliente`.

- a. Costruire la gerarchia dei ruoli in base alle relazioni sopra descritte.
- b. Determinare le autorizzazioni per il ruolo `direttoreVideoteca`.
- c. Specificare un vincolo che vietи agli utenti di ricoprire contemporaneamente i ruoli `direttoreVideoteca` e `magazziniere`.

**9.5** Con riferimento alla base di dati della videoteca, scrivere i comandi SQL per:

- a. Creare il ruolo `regista`.
- b. Assegnare al ruolo `regista` il privilegio di eseguire interrogazioni sulle relazioni `Video` e `Film`.

---

<sup>8</sup>La notazione `u,t` indica il grantor ed il timestamp del comando GRANT.

- c. Abilitare Giovanni a ricoprire il ruolo **regista** e a concedere a terzi la possibilità di ricoprire tale ruolo.
- d. Creare un ruolo **aiutoRegista** sotto-ruolo del ruolo **regista**.
- e. Revocare al ruolo **regista** il privilegio di eseguire interrogazioni sulla relazione **Film**.
- f. Revocare a Giovanni la possibilità di concedere ad altri l'abilitazione a ricoprire il ruolo **regista**.

## Capitolo 4

# Sviluppo di applicazioni per basi di dati

Come abbiamo visto nel Capitolo 3, SQL è un linguaggio per definire e manipolare dati relazionali. Nel presentare il linguaggio SQL, ci siamo concentrati sui comandi principali per eseguire singole operazioni sulla base di dati. Tali comandi possono essere specificati direttamente ed eseguiti a partire da un’interfaccia interattiva, disponibile in ogni sistema di gestione dati. L’interfaccia interattiva è molto utile per eseguire operazioni semplici o poco frequenti, come la definizione dello schema della base di dati o dei vincoli di integrità, oppure interrogazioni formulate da utenti occasionali. Tuttavia, la maggioranza delle interazioni con la base di dati richiede l’esecuzione di operazioni più complesse ed eseguite ripetutamente dagli utenti, che non necessariamente possono essere rappresentate da una sequenza di comandi SQL come quelli introdotti nel Capitolo 3.

Lo sviluppo di una generica applicazione per basi di dati richiede infatti l’esecuzione dei comandi SQL introdotti nel Capitolo 3 insieme all’utilizzo di tecniche di programmazione più generali. Questa necessità nasce dal fatto che SQL non permette di sostituire completamente un linguaggio di programmazione generico nello sviluppo di un’applicazione. Al fine di permettere ai DBMS di ottimizzare efficacemente le interrogazioni specificate dagli utenti o contenute nei programmi applicativi (vedi Capitolo 7), il potere espressivo dei linguaggi di interrogazione in generale, e di SQL in particolare, è infatti limitato rispetto al potere espressivo di un generico linguaggio di programmazione. Per scelte progettuali, SQL non è né *computazionalmente completo* né *operazionalmente completo*. Con *completezza computazionale* intendiamo la capacità di esprimere nel linguaggio tutte le computazioni teoricamente possibili. SQL non mette a disposizione i costrutti tipici della programmazione imperativa (ad esempio, il costrutto di scelta ed il costrutto di iterazione) che permettono di garantire la completezza computazionale. Con *completezza operazionale* intendiamo, invece, la capacità di esprimere nel linguaggio operazioni che richiedono una comunicazione diretta con il sistema operativo e, attraverso tale sistema, con l’hardware del computer e le periferiche. SQL è operazionalmente incompleto in quanto, ad esempio, non comprende costrutti per scrivere in un file, stampare su una stampante o sviluppare un’interfaccia utente, come richiesto ormai dalla maggioranza delle applicazioni.

Per potere sviluppare un’applicazione per basi di dati arbitrariamente complessa nasce quindi la necessità di combinare l’utilizzo del linguaggio SQL con

quello di un generico linguaggio di programmazione. Nel linguaggio così ottenuto, SQL consentirà un accesso ottimizzato alla base di dati mentre il linguaggio di programmazione garantirà la completezza operazionale e computazionale.

In questo capitolo, per prima cosa illustreremo le soluzioni esistenti per integrare SQL in un linguaggio di programmazione. Identificheremo poi i passi principali di cui si compone un'arbitraria applicazione per basi di dati e descriveremo come tali passi vengono eseguiti nel contesto di ciascuna soluzione illustrata.

#### 4.1 Approcci all'integrazione

SQL può essere integrato in (accoppiato ad) un linguaggio di programmazione in diversi modi:

- **Accoppiamento interno.** SQL viene esteso con gli usuali costrutti di programmazione (ad esempio, costrutti per la definizione di procedure e funzioni, variabili e strutture di controllo). Il linguaggio esteso è chiamato *SQL procedurale* oppure *estensione procedurale di SQL*. Esso può essere utilizzato come un linguaggio di programmazione tradizionale all'interno del DBMS e le procedure e funzioni definite possono essere richiamate da qualunque applicazione utente.

L'accoppiamento interno permette di ottenere la completezza computazionale ma non sempre garantisce la completezza operazionale (ad esempio, le estensioni procedurali di SQL non sempre prevedono istruzioni per la lettura da input o la stampa). Inoltre, benché lo standard SQL proponga un'estensione procedurale di SQL, la sintassi varia sensibilmente da un DBMS all'altro, creando ovvi problemi di portabilità delle applicazioni.

- **Accoppiamento esterno.** Un linguaggio di programmazione esistente, come ad esempio il linguaggio Java od il linguaggio C, viene accoppiato con SQL. Il codice delle applicazioni sviluppate secondo questo approccio viene eseguito in un ambiente esterno al DBMS.

La completezza computazionale ed operazionale dei linguaggi di programmazione garantisce la completezza delle soluzioni ad accoppiamento esterno.

Dal punto di vista del programmatore, esistono due diversi approcci di accoppiamento esterno:

- **Libreria di funzioni.** Nel contesto di un linguaggio di programmazione, viene resa disponibile una libreria di funzioni, o più precisamente un'API (*Application Programming Interface*), che definisce l'interfaccia di comunicazione tra il linguaggio di programmazione ed il DBMS. Le funzioni permettono la connessione alla base di dati e l'esecuzione di comandi SQL nel contesto di programmi sviluppati utilizzando il linguaggio prescelto. Oltre alle librerie proprietarie dei vari DBMS commerciali, esistono anche librerie standard, che permettono

alle applicazioni di connettersi a molteplici DBMS relazionali (o relazionali ad oggetti, vedi Capitolo 10) utilizzando la stessa interfaccia di comunicazione, garantendo quindi un elevato livello di interoperabilità.

- **SQL ospitato.** In questo caso, i comandi SQL vengono ospitati direttamente in una versione estesa del linguaggio di programmazione. In genere, un pre-compilatore traduce il programma contenente i comandi SQL ospitati in un programma interamente scritto nel linguaggio ospite, sostituendo ogni comando SQL con chiamate alla base di dati tramite interfacce di connessione standard o proprietarie.

In pratica, la maggior parte delle applicazioni sono sviluppate secondo soluzioni ad accoppiamento esterno. È infatti molto frequente che un'applicazione esistente, scritta in un linguaggio di programmazione generico, debba essere estesa per interagire con una base di dati. L'accoppiamento interno viene invece utilizzato da nuove applicazioni che richiedono una forte interazione con la base di dati. L'approccio più diffuso combina entrambi i tipi di accoppiamento: utilizzando una soluzione ad accoppiamento interno, vengono inizialmente definite procedure e funzioni relative alle operazioni più frequentemente eseguite sulla base di dati; tali procedure e funzioni vengono quindi richiamate da applicazioni sviluppate secondo un approccio ad accoppiamento esterno.

## 4.2 Problemi dell'accoppiamento esterno

L'accoppiamento esterno, a differenza di quello interno, richiede l'integrazione tra un linguaggio di programmazione esistente ed SQL. A questo proposito, le differenze esistenti tra i modelli su cui i due linguaggi si basano generano una serie di problemi, noti come *confitto di impedenza (impedance mismatch)*. Ad esempio, i concetti su cui si basa il modello relazionale (attributo, tupla e relazione) non trovano un riscontro immediato in alcun linguaggio di programmazione. Andando ancora più nel dettaglio, i tipi di dato previsti da SQL non necessariamente coincidono con i tipi di dato del linguaggio di programmazione. Ad esempio, il tipo di dato VARCHAR non è un tipo di dato Java. Tuttavia, il tipo di dato String potrebbe essere utilizzato al posto di VARCHAR nel contesto del linguaggio Java. Nasce quindi l'esigenza di definire un collegamento (*binding*) tra i tipi di dato di SQL ed i tipi di dato del linguaggio di programmazione considerato.

Un secondo problema nasce dal fatto che, mentre il risultato di un'interrogazione è sempre un insieme di tuple (cioè, SQL è un linguaggio *set-oriented*), i linguaggi di programmazione possono generalmente elaborare un insieme accedendo ad una tupla alla volta (sono quindi *tuple-oriented*). Per risolvere questo problema, è necessario tradurre il risultato di un'interrogazione, quindi una relazione, in una struttura dati prevista dal linguaggio di programmazione. Se il risultato dell'interrogazione contiene una singola tupla, la stessa può essere rappresentata utilizzando un insieme di variabili, una per ogni valore di attributo. Al contrario, se la dimensione del risultato non è nota o è troppo grande per essere

caricata in memoria centrale, è necessario utilizzare un meccanismo per accedere alle singole tuple appartenenti al risultato ed estrarne i valori, mentre il risultato complessivo viene mantenuto nella base di dati. Questo è possibile utilizzando un *cursor*. Intuitivamente, un cursore è un puntatore ad una tupla contenuta nel risultato di un'interrogazione, memorizzata nella base di dati. Un cursore è quindi associato alla valutazione di un'interrogazione. In ogni momento, la tupla puntata dal cursore, ed i valori degli attributi di tale tupla, possono essere letti ed inseriti in una struttura dati del linguaggio ospite (ad esempio, una variable scalare od un vettore). Ogni cursore è corredata da alcune funzioni che abilitano il programmatore al controllo del movimento del cursore sull'insieme risultato, permettendo il posizionamento del cursore sulla tupla successiva o precedente rispetto a quella correntemente puntata, o sulla prima ovvero sull'ultima tupla del risultato. I cursori sono in genere aperti e chiusi esplicitamente mediante opportune primitive del linguaggio di programmazione. L'apertura di un cursore comporta l'esecuzione dell'interrogazione a cui il cursore è associato.

### 4.3 Flusso di esecuzione

Qualunque sia l'approccio utilizzato per lo sviluppo, un'applicazione per basi di dati è caratterizzata da una sequenza comune di passi, illustrata nel seguito:

- **Connessione alla base di dati.** Affinché un'applicazione possa utilizzare una base di dati, l'applicazione deve aprire una connessione verso il DBMS su cui la base di dati risiede. Nelle soluzioni ad accoppiamento interno, tale connessione è già implicitamente aperta in quanto l'applicazione viene eseguita direttamente dal DBMS. Nelle soluzioni ad accoppiamento esterno è invece necessario specificare il server con cui vogliamo aprire la connessione, precisandone l'indirizzo Internet (URL), nonché opportune credenziali di accesso (ovvero un nome utente ed una password).
- **Esecuzione dei comandi SQL.** Una volta aperta una connessione, è possibile interagire con la base di dati tramite l'esecuzione di comandi SQL.
- **Chiusura della connessione.** Quando l'interazione con la base di dati è terminata, la connessione con il DBMS dovrebbe essere chiusa. I DBMS ammettono infatti un numero massimo di connessioni contemporaneamente attive. Chiudere una connessione significa quindi dare la possibilità a nuove applicazioni di accedere al DBMS. In genere, la terminazione di un'applicazione comporta la chiusura automatica di tutte le connessioni aperte.

Per ogni comando SQL che deve essere eseguito, l'interazione tra il linguaggio di programmazione ed il DBMS coinvolge i seguenti tre passi principali:

- **Preparazione del comando.** La preparazione del comando consiste nella generazione delle strutture dati necessarie per la comunicazione con il sistema di gestione dati e nell'eventuale compilazione ed ottimizzazione del comando da parte del DBMS (vedi Capitolo 7).

- **Esecuzione del comando.** Un comando SQL viene eseguito utilizzando le strutture dati e le informazioni generate durante la fase di preparazione. L'esecuzione di comandi `INSERT`, `DELETE` e `UPDATE`, analogamente all'esecuzione di comandi del DDL, restituisce un valore numerico che rappresenta il numero delle tuple inserite/cancellate/modificate oppure i dettagli relativi al completamento, in modo corretto od errato, dell'operazione. Il risultato della valutazione di un comando `SELECT` è, invece, sempre un insieme di tuple.
- **Manipolazione del risultato.** Il risultato di un comando SQL, una volta tradotto nelle strutture del linguaggio di programmazione, può essere manipolato secondo la logica richiesta dall'applicazione. Come già discusso, il risultato di un comando `SELECT` può richiedere l'utilizzo di un cursore, per accedere alle varie tuple contenute nel risultato.

Spesso la fase di preparazione e la fase di esecuzione di un comando SQL sono disaccoppiate. Un comando SQL può infatti essere preparato, in modo separato dalla sua esecuzione, durante la compilazione o l'esecuzione dell'applicazione che lo contiene (vedi Paragrafo 4.5.2). Il disaccoppiamento è molto utile nel caso in cui lo stesso comando SQL debba essere eseguito più volte, magari cambiando soltanto i valori di alcuni parametri. Per esempio, un'applicazione potrebbe voler determinare più volte il numero medio dei noleggi per un particolare genere di film, specificando ogni volta un genere diverso. In questo caso, poiché il comando SQL da eseguire è sempre lo stesso (ciò che cambia è solo il valore di una costante), la fase di preparazione permette di determinare il piano più efficiente per l'esecuzione dell'operazione, indipendentemente dal valore del parametro (vedi Capitolo 7). Tale piano di esecuzione può poi essere utilizzato in tutte le elaborazioni successive del comando nel contesto della stessa esecuzione dell'applicazione, con un notevole risparmio di tempo.

#### 4.4 SQL dinamico

SQL dinamico fornisce un insieme di funzionalità che hanno lo scopo di facilitare lo sviluppo delle cosiddette *applicazioni on-line*. Un'applicazione on-line è un programma sviluppato per permettere l'accesso alla base di dati da parte di un utente connesso da un terminale. Una classica applicazione “on-line” può essere schematizzata come segue:

1. accetta una richiesta da terminale;
2. analizza la richiesta;
3. invia l'appropriato comando SQL al DBMS;
4. restituisce a terminale un messaggio e/o i risultati del comando eseguito.

Per comprendere meglio l'organizzazione di un'applicazione on-line, consideriamo il seguente esempio, relativo al dominio applicativo della videoteca.

**Esempio 4.1** Supponiamo di voler realizzare un'applicazione che esegua i seguenti passi:

1. prende in input il titolo ed il regista di un film;
2. se la valutazione del film, corrispondente al titolo ed al regista forniti in input, è minore od uguale a 1, allora si ritirano dal noleggio (e quindi si cancellano dalla base di dati) tutti i video relativi al film, in quanto si ritiene che il film non sarà noleggiato di frequente;
3. se la valutazione del film è maggiore od uguale a 4, i video di tipo vhs vengono rimpiazzati da video di tipo dvd, in quanto si ritiene che il film sarà noleggiato spesso dai clienti e si vuole quindi migliorare la qualità dei supporti;
4. in tutti gli altri casi, seleziona il titolo, il regista e la valutazione di tutti i film.

Osserviamo che l'applicazione deve eseguire comandi SQL diversi in relazione alla valutazione assegnata al film identificato dai dati forniti in input.  $\square$

Se le richieste che possono essere immesse da terminale sono poche e note a priori, come nell'esempio precedente, è possibile "cablare" nel programma applicativo, utilizzando una logica a casi, tutte le possibili richieste in ingresso; ad ogni richiesta corrisponderanno uno o più comandi SQL "potenzialmente" eseguibili. Il comando SQL che verrà "effettivamente" eseguito diventerà noto solo al momento dell'esecuzione, quando l'utente preciserà la sua richiesta (nel caso dell'esempio, quando l'utente fornirà il titolo ed il regista di un film). Questo approccio non è ovviamente adattabile in situazioni in cui non sia possibile prevedere tutte le possibili richieste in ingresso, ad esempio perché lo stesso comando SQL, o parte di esso, viene fornito direttamente in input o perché l'insieme delle richieste potenziali è molto grande.

Per agevolare lo sviluppo di applicazioni "on-line", indipendentemente dalla tipologia delle richieste considerate, le soluzioni ad accoppiamento interno ed esterno permettono l'esecuzione di comandi SQL non necessariamente noti al momento della compilazione del programma. Per mettere in evidenza che tali comandi vengono creati dinamicamente all'atto dell'esecuzione del programma, essi vengono chiamati *comandi dinamici* ed il codice SQL corrispondente viene chiamato *SQL dinamico*. I comandi SQL dinamici sono contrapposti ai comandi *SQL statici*, direttamente codificati nel programma sorgente. I comandi statici, a differenza di quelli dinamici, sono completamente noti nel momento in cui l'applicazione viene compilata. Al contrario, un comando SQL dinamico viene parzialmente o completamente costruito durante l'esecuzione del programma, utilizzando per esempio dati forniti in input dall'utente.

I comandi SQL dinamici comportano maggiori problemi in termini di ottimizzazione, compilazione ed esecuzione sia per il DBMS sia per il linguaggio ospite, ed è per questo motivo che alcuni approcci di accoppiamento non ne permettono l'esecuzione. Infatti, un comando SQL statico può essere preparato durante la compilazione del programma ed il risultato della preparazione può essere utilizzato un numero arbitrario di volte, nel contesto di esecuzioni distinte. Al contrario, un comando SQL dinamico può essere preparato soltanto durante l'esecuzione del programma, con un conseguente aumento del tempo di risposta. Ovviamente, il risultato della preparazione può poi essere utilizzato un numero arbitrario di volte, nel contesto della stessa esecuzione.

Nel seguito verranno descritte con maggior dettaglio le tipologie di accoppiamento presentate nel Paragrafo 4.1. Gli esempi che verranno riportati si riferiscono allo schema della base di dati relativa alla videoteca, introdotto nel Capitolo 2.

#### 4.5 Estensioni procedurali di SQL

Nelle soluzioni ad accoppiamento interno, SQL viene esteso con tipici costrutti della programmazione imperativa. In genere, i programmi risultanti sono organizzati in moduli, cioè in procedure e funzioni. L'uso dei moduli permette di modellare e memorizzare nella base di dati operazioni significative in un certo dominio applicativo. Tali funzioni e procedure possono essere eseguite direttamente dall'interfaccia resa disponibile dai DBMS ovvero utilizzate nel contesto di applicazioni ad accoppiamento esterno, che richiedono l'esecuzione di tali funzionalità.

Lo standard SQL propone un'estensione procedurale di SQL nella parte *SQL/PSM – SQL/Persistent Stored Module* ed illustra come i nuovi comandi possano essere utilizzati per definire funzioni e procedure.<sup>1</sup> È importante osservare che, benché la maggior parte dei DBMS commerciali proponga un'estensione procedurale di SQL, la sintassi di tali estensioni spesso non è conforme allo standard, creando ovvi problemi di portabilità delle applicazioni.

Nel seguito, presenteremo inizialmente SQL/PSM, descrivendone le caratteristiche generali. Introdurremo quindi i comandi per l'esecuzione di codice SQL dinamico. Infine, illustreremo come sia possibile definire procedure e funzioni in SQL/PSM. Quando non altrimenti specificato, SQL indicherà il linguaggio SQL Core esteso con i comandi di SQL/PSM.

##### 4.5.1 *SQL/PSM*

SQL/PSM è la parte dello standard SQL che descrive un'estensione procedurale di SQL Core. Ogni programma scritto in SQL/PSM è composto da due parti: una parte di *dichiarazione* ed una parte di *esecuzione*. Notiamo che, poiché l'applicazione risultante viene eseguita dal DBMS, non è necessario alcun comando esplicito di connessione alla base di dati che si intende utilizzare.

---

<sup>1</sup>Ricordiamo che i comandi per la definizione di funzioni e procedure sono parte di SQL Core, così come i comandi per l'uso dei cursori e di SQL dinamico.

La parte di dichiarazione contiene la dichiarazione dei tipi delle variabili utilizzate dal programma. A questo proposito, possono essere utilizzati tutti i tipi SQL introdotti nel Capitolo 3. Un tipo può essere associato ad una o più variabili nel contesto della stessa istruzione. La dichiarazione di un insieme di variabili aventi lo stesso tipo deve essere preceduta dalla parola chiave **DECLARE**. Una variabile può inoltre essere associata ad un valore di default, in modo del tutto simile a quanto avviene per una colonna di una relazione.

**Esempio 4.2** Il seguente è un esempio di dichiarazione di variabili:

```
DECLARE valutazMedia NUMERIC(3,2);
DECLARE valutaz NUMERIC(3,2) DEFAULT 2.50;
DECLARE nome, cognome VARCHAR(20);
```

□

La parte di esecuzione di un programma SQL/PSM è identificata da un costrutto di blocco, identificato dalle parole chiave **BEGIN** ed **END**. Il blocco può contenere tipici costrutti procedurali e comandi SQL. I costrutti procedurali ed i comandi SQL comunicano mediante le variabili dichiarate nella parte di dichiarazione. I valori possono essere assegnati alle variabili tramite il comando **SET**, usando la stessa sintassi vista per la clausola **SET** del comando **UPDATE** (vedi Capitolo 3). Le variabili possono poi essere utilizzate in ogni punto di un comando SQL dove può essere inserita un'espressione avente lo stesso tipo della variabile: nella clausola **WHERE** di un comando **SELECT**, **DELETE** ed **UPDATE**, nella clausola **SET** di un comando **UPDATE** e nella clausola **VALUES** di un comando **INSERT**.

**Esempio 4.3** Nel seguente codice, la variabile **ilGenere** di tipo **CHAR(15)** contiene il genere da utilizzare per selezionare i film di interesse:

```
DECLARE ilGenere CHAR(15);
BEGIN
    SET ilGenere = 'comico';
    SELECT titolo
    FROM Film
    WHERE genere = ilGenere;
END;
```

□

SQL/PSM permette l'uso di strutture di controllo e costrutti tipici della programmazione imperativa, come il costrutto di scelta (**IF\_THEN\_ELSE\_END IF**) e quelli di iterazione (**LOOP-END LOOP**, **WHILE\_DO-END WHILE**, **REPEAT\_UNTIL-END REPEAT**). La semantica di questi costrutti è analoga a quella dei corrispondenti costrutti nei linguaggi di programmazione.

Benché una descrizione dettagliata dei costrutti previsti da SQL/PSM non rientri nello scopo di questo libro, a titolo esemplificativo presentiamo la sintassi del costrutto di scelta:

```

IF <condizione> THEN <elenco istruzioni>
...
[ELSEIF <condizione> THEN <elenco istruzioni>]
[ELSE <elenco istruzioni>]
END IF;

```

dove <condizione> rappresenta un'espressione SQL di tipo booleano mentre <elenco istruzioni> rappresenta un singolo comando oppure un blocco di istruzioni SQL.

**Esempio 4.4** Il seguente codice inserisce un nuovo film nella base di dati, supponendo che la valutazione del film dipenda da una certa analisi effettuata sui noleggi di film dello stesso genere (indicata con [...] nel codice seguente):

```

DECLARE infoNoleggi INTEGER;
DECLARE laValutaz  NUMERIC(3,2);
BEGIN
[...]
IF infoNoleggi > 5000 THEN SET laValutaz = 3.00;
ELSEIF infoNoleggi > 3000 THEN SET laValutaz = 2.00;
ELSE SET laValutaz = 1;
END IF;
INSERT INTO Film VALUES ('kill bill I',
                         'quentin tarantino',2003,
                         'thriller',laValutaz);
END;

```

□

Quando il risultato di un comando **SELECT** deve essere manipolato, è necessario distinguere due casi. Il primo caso si riferisce a comandi **SELECT** scalari, cioè a comandi che restituiscono una singola tupla, il secondo a comandi non scalari.

Nel caso di comandi **SELECT** scalari, i valori degli attributi della singola tupla ottenuta come risultato possono essere inseriti in variabili opportune mediante il comando **SELECT INTO**, come illustrato dall'esempio seguente.<sup>2</sup>

**Esempio 4.5** Il seguente codice assegna la valutazione del film “Nirvana”, ottenuta come risultato dell'esecuzione di un comando **SELECT** scalare, alla variabile **laValutaz**:

```

DECLARE laValutaz NUMERIC(3,2);
BEGIN
    SELECT valutaz INTO laValutaz
    FROM Film WHERE titolo = 'nirvana';
END;

```

□

---

<sup>2</sup>Il comando **SELECT INTO** è un comando previsto da SQL Core.

Quando il comando **SELECT** non è scalare, la sua esecuzione restituisce un insieme di tuple ed è necessario utilizzare un cursore per elaborare ciascuna tupla singolarmente. Tutti i comandi relativi alla gestione di un cursore sono previsti da SQL Core. Un cursore è una struttura che deve essere inizialmente dichiarata, specificandone il nome e l'interrogazione associata. La sintassi è la seguente:

```
DECLARE <nome cursore> CURSOR [SCROLL] FOR
<interrogazione>;
```

dove **<nome cursore>** è il nome del cursore che viene definito ed **<interrogazione>** è l'interrogazione ad esso associata. Se la clausola opzionale **SCROLL** viene specificata, il cursore potrà essere spostato liberamente sul risultato dell'interrogazione. Altrimenti, come vedremo meglio nel seguito, i movimenti del cursore saranno limitati.<sup>3</sup>

**Esempio 4.6** Il seguente codice dichiara un cursore di nome **valElevata** associato ad un'interrogazione per determinare i film aventi una valutazione superiore a 3:

```
DECLARE valElevata CURSOR FOR
    SELECT titolo, regista
    FROM Film
    WHERE valutaz > 3.00;
```

□

La dichiarazione di un cursore associa semplicemente un cursore ad un'interrogazione SQL. Questa interrogazione viene però eseguita solo quando il cursore viene esplicitamente aperto, utilizzando il comando **OPEN <nome cursore>**.

**Esempio 4.7** Il comando **OPEN valElevata** apre il cursore **valElevata**, introdotto nell'Esempio 4.6. □

Quando un cursore viene aperto, esso inizialmente “punta” alla posizione precedente a quella della prima tupla della relazione ottenuta come risultato dall'esecuzione dell'interrogazione. Ciascuna tupla del risultato può essere prelevata tramite il comando **FETCH\_FROM\_INTO**. Questo comando ha un duplice effetto: (*i*) sposta il cursore su una certa tupla; (*ii*) carica i valori degli attributi della tupla in variabili. La sintassi del comando **FETCH\_FROM\_INTO** è la seguente:

```
FETCH [<orientamento>] FROM <nome cursore>
INTO <lista variabili>;
```

dove:

- **<nome cursore>** è il nome del cursore da spostare.

---

<sup>3</sup>Il comando di dichiarazione di un cursore permette anche di indicare se il cursore verrà utilizzato per modificare o cancellare tuple all'interno di una relazione. Questo aspetto non verrà considerato nella trattazione.

- <orientamento> è una clausola opzionale che indica come spostare il cursore nell’insieme delle tuple ottenute come risultato, prima che i valori degli attributi della tupla correntemente puntata vengano assegnati alle variabili elencate in <lista variabili>. I valori possibili per <orientamento> sono: NEXT, per muovere il cursore sulla tupla successiva a quella correntemente puntata (default); PRIOR, per muovere il cursore sulla tupla precedente a quella correntemente puntata; FIRST, per muovere il cursore sulla prima tupla del risultato; LAST, per muovere il cursore sull’ultima tupla del risultato; ABSOLUTE *n*, per muovere il cursore sull’*n*-esima tupla del risultato, a partire dalla prima; RELATIVE *n* per muovere il cursore sull’*n*-esima tupla del risultato, a partire dalla posizione corrente. L’unico orientamento possibile per un cursore per il quale non sia stata specificata la clausola SCROLL è NEXT.
- <lista variabili> è la lista delle variabili a cui vengono assegnati i valori degli attributi della tupla correntemente puntata dal cursore. I tipi delle variabili devono essere compatibili con i tipi di tali attributi; inoltre, le variabili devono essere elencate nello stesso ordine con cui gli attributi compaiono nel risultato dell’interrogazione associata al cursore.

**Esempio 4.8** Il seguente comando posiziona il cursore `valElevata` definito nell’Esempio 4.6 sulla tupla successiva a quella correntemente puntata ed inserisce i valori degli attributi di tale tupla in due variabili, `ilTitolo` ed `ilRegista`, che assumiamo precedentemente dichiarate:

```
FETCH NEXT FROM valElevata
  INTO ilTitolo, ilRegista;
```

□

Quando un cursore ha raggiunto la prima o l’ultima tupla di un risultato e viene richiesta l’esecuzione di un comando `FETCH PRIOR` o `FETCH NEXT`, rispettivamente, SQL prevede che il DBMS rilevi una condizione di errore chiamata `NOT FOUND`. Vedremo nel seguito il meccanismo proposto da SQL per la gestione delle condizioni di errore (e quindi per il controllo delle eccezioni). Per il momento, supponiamo che esista una variabile `fine` che assume il valore vero quando la condizione di errore `NOT FOUND` diventa vera. Utilizzando la variabile `fine` insieme ad un costrutto di iterazione, è quindi possibile analizzare tutte le tuple ottenute come risultato di un’interrogazione associata al cursore.

Quando l’elaborazione di tutte le tuple è terminata, il cursore può essere chiuso mediante il comando `CLOSE <nome cursore>`. Dopo aver chiuso un cursore non è più possibile accedere alle tuple del risultato. Per accedere nuovamente alle tuple è quindi necessario riaprire il cursore.

**Esempio 4.9** Il codice mostrato nella Figura 4.1 illustra la tipica elaborazione applicata al risultato di un’interrogazione mediante un cursore. In particolare, viene utilizzato un cursore per determinare il titolo ed il regista dei film con valutazione superiore a 3. Notiamo che, benché SQL/PSM non preveda operazioni di lettura e scrittura a video, l’estensione procedurale di SQL proposta dai DBMS

```

DECLARE ilTitolo  VARCHAR(30);
DECLARE ilRegista VARCHAR(20);
DECLARE valElevata CURSOR FOR
    SELECT titolo, regista
    FROM Film
    WHERE valutaz > 3.00;
BEGIN
    [...]
    OPEN valElevata;
    FETCH NEXT FROM valElevata INTO ilTitolo, ilRegista;
    WHILE NOT fine DO
        BEGIN
            PRINT('Titolo: ' || ilTitolo);
            PRINT('Regista: ' || ilRegista);
            FETCH NEXT FROM valElevata INTO ilTitolo, ilRegista;
        END;
    END WHILE;
    CLOSE valElevata;
END;

```

Figura 4.1: Utilizzo di un cursore

commerciali ammette generalmente almeno il comando di scrittura (ad esempio, il comando di scrittura a video in Microsoft SQL Server è `PRINT` mentre in Oracle è `dbmsoutput.put_line`). A titolo esemplificativo, nel codice di Figura 4.1 utilizziamo il comando `PRINT` come comando per la scrittura a video. In tale codice, [...] indica le operazioni da eseguire per inizializzare la variabile `fine`. Tali operazioni verranno discusse nel Paragrafo 4.5.3. □

#### 4.5.2 SQL dinamico

Tutti gli esempi riportati nel Paragrafo 4.5.1 si riferiscono ad SQL statico. Tuttavia, SQL permette di eseguire anche comandi dinamici. Poiché un comando SQL dinamico viene creato durante l'esecuzione dell'applicazione, l'idea è quella di memorizzare il comando SQL in una variabile di tipo stringa, passata poi come parametro ad un comando opportuno per la preparazione e/o l'esecuzione. I comandi dinamici possono essere eseguiti in un unico passo (in questo caso, il comando verrà preparato ed immediatamente eseguito) od in due passi, corrispondenti ad una fase di preparazione ed una fase di esecuzione.

Per eseguire un comando dinamico in un unico passo, la sintassi è la seguente:

```
EXECUTE IMMEDIATE <variabile comando>;
```

dove `<variabile comando>` rappresenta una variabile di tipo stringa che contiene il codice SQL da eseguire. La stringa contenuta nella variabile può anche essere costruita a tempo di esecuzione.

```

DECLARE ilTitolo VARCHAR(30);
DECLARE ilRegista VARCHAR(20);
DECLARE ilComando VARCHAR(100) := 'SELECT titolo, regista, valutaz
                                    FROM Film';
BEGIN
    [...]
    IF (SELECT valutaz FROM Film
        WHERE titolo = ilTitolo AND regista = ilRegista) <= 1.00 THEN
        SET ilComando = 'DELETE FROM Video
                          WHERE titolo = ilTitolo AND regista = ilRegista';
    ELSEIF
        (SELECT valutaz FROM Film
        WHERE titolo = ilTitolo AND regista = ilRegista) >= 4.00 THEN
        SET ilComando = 'UPDATE Video
                          SET tipo = ''d''
                          WHERE titolo = ilTitolo AND regista = ilRegista AND
                                tipo = ''v'' ';
    END IF;
    EXECUTE IMMEDIATE ilComando;
END;

```

Figura 4.2: Utilizzo di comandi SQL dinamici

**Esempio 4.10** Il programma mostrato nella Figura 4.2 realizza il comportamento descritto nell’Esempio 4.1, utilizzando SQL dinamico. Nel codice, [...] indica una qualche computazione che permette di inizializzare i valori delle variabili *ilTitolo* ed *ilRegista*. Notiamo che la variabile *ilComando* conterrà stringhe diverse in relazione alla valutazione assegnata al film considerato. È interessante osservare che lo stesso comportamento potrebbe essere ottenuto inserendo la condizione di scelta all’interno del comando *DELETE* e del comando *UPDATE*. Tuttavia, in questo caso, verrebbero eseguiti due comandi SQL, uno dei quali non comporta alcuna modifica alla base di dati, anziché uno soltanto. □

Il comando per la preparazione di un comando dinamico è il seguente:

```

PREPARE <nome comando>
FROM <variabile comando>;

```

dove *<variabile comando>* è la variabile di tipo stringa che contiene il comando SQL da eseguire e *<nome comando>* è un identificatore associato a tale comando, da utilizzare per l’esecuzione successiva.

Durante la fase di preparazione di un comando, il valore di alcune costanti potrebbe non essere noto e determinato solo successivamente, prima di ciascuna esecuzione del comando stesso. Il comando può quindi essere parametrico. Ogni parametro viene indicato con il simbolo ‘?’ nella stringa contenuta in *<variabile comando>*. Un comando preparato può poi essere eseguito mediante il comando *EXECUTE* avente la seguente sintassi:

---

```
EXECUTE <nome comando>
[INTO <lista variabili>]
[USING <lista valori parametri>];
```

dove:

- <nome comando> è l'identificatore assegnato al comando SQL da un precedente comando di preparazione.
- La clausola USING permette di associare un valore a ciascun parametro eventualmente specificato al momento della preparazione. Ogni valore deve essere il risultato di un'espressione specificata in <lista valori parametri>, di tipo opportuno. Le espressioni devono essere fornite nell'ordine con cui sono stati specificati i parametri nel comando dinamico.
- Se il comando da eseguire è un comando SELECT scalare oppure una chiamata di funzione o procedura (vedi Paragrafo 4.5.4), la clausola INTO permette di specificare in <lista variabili> una lista di variabili nelle quali memorizzare il risultato ottenuto dall'esecuzione, analogamente alla clausola INTO del comando SELECT discusso in precedenza.

**Esempio 4.11** Supponiamo di voler aumentare la valutazione dei film di un certo genere dello 0.5%. Se il genere non è noto a priori, questo comportamento può essere ottenuto preparando un comando SQL dinamico ed assegnando al parametro una stringa che rappresenta il genere di interesse al momento dell'esecuzione. Notiamo che, una volta preparato, il comando può essere eseguito varie volte, cambiando solo il valore assegnato ai parametri, senza ripetere la preparazione. Il seguente codice realizza questo comportamento, aumentando la valutazione dei film comici e dei film drammatici:

```
DECLARE ilComando VARCHAR(100);
SET ilComando = 'UPDATE Film
                  SET valutaz = valutaz * 1.05
                  WHERE genere = ?';

PREPARE comandoP FROM ilComando;
EXECUTE comandoP USING 'comico';
EXECUTE comandoP USING 'drammatico';
```

□

#### 4.5.3 Gestione degli errori

Per gestire in modo adeguato il flusso applicativo, è necessario rilevare gli errori generati durante l'esecuzione di un programma e gestirli in modo opportuno. A questo scopo, SQL prevede una variabile di sistema SQLSTATE, che rappresenta una stringa di cinque caratteri. Il valore 00000 indica che non si è verificato alcun errore. Un valore diverso da 00000 indica che si è verificato un errore oppure una situazione che richiede attenzione. Ad esempio, la stringa 02000 indica "fine

dati”. Tale errore viene, per esempio, riportato quando la posizione corrente di un cursore non corrisponde ad alcuna tupla. I codici di errore restituiti in `SQLSTATE` sono standardizzati, quindi sono comuni a tutti i DBMS e le piattaforme conformi allo standard SQL.

Gli errori possono essere gestiti a livello applicativo in SQL tramite il meccanismo delle eccezioni. Esistono due tipi di eccezioni: *eccezioni di sistema*, definite e sollevate direttamente dal sistema al verificarsi di un errore durante l’interazione del programma applicativo con la base di dati, ed *eccezioni utente*, definite e sollevate dal programma applicativo. Nel seguito illustreremo solo le eccezioni di sistema.

Ogni eccezione può essere associata ad un’azione da eseguire quando l’eccezione viene sollevata tramite la dichiarazione di un *handler*. La sintassi per definire un handler è la seguente:

```
DECLARE <tipo handler> HANDLER FOR <lista condizioni> <azione>;
```

dove:

- **<tipo handler>** indica quale comportamento adottare dopo l’esecuzione dell’azione. I comportamenti possibili sono: `CONTINUE`, per continuare l’esecuzione del programma dal punto in cui è stato interrotto; `EXIT`, per terminare l’esecuzione del programma; `UNDO`, per disfare le operazioni che hanno portato al sollevamento dell’eccezione.
- **<lista condizioni>** è una lista di identificatori di errori. Nella lista, gli errori possono essere identificati dai valori della variabile `SQLSTATE` o dalle parole chiave `SQLEXCEPTION`, `SQLWARNING` e `NOT FOUND` per indicare, rispettivamente, un errore arbitrario, il verificarsi di una condizione che richiede attenzione benché non corrisponda ad un errore o l’esecuzione di un’operazione che non ha restituito alcun dato.
- **<azione>** rappresenta un qualunque comando SQL da eseguire al verificarsi di una delle condizioni indicate in `<lista condizioni>`.

**Esempio 4.12** Il programma riportato nella Figura 4.1 può essere riscritto, utilizzando il meccanismo delle eccezioni, come mostrato nella Figura 4.3. In tale programma, al verificarsi della condizione `NOT FOUND`, la variabile `fine` viene posta a `true` e, come conseguenza, il ciclo termina. □

#### 4.5.4 Funzioni e procedure

SQL permette la definizione di moduli, cioè di procedure e funzioni, memorizzate direttamente nel DBMS. Tali moduli sono noti come *stored procedure* e *stored function* e vengono anche chiamati *SQL invoked routine* (moduli SQL invocati) nello standard SQL.

```

DECLARE ilTitolo  VARCHAR(30);
DECLARE ilRegista VARCHAR(20);
DECLARE valElevata CURSOR FOR
    SELECT titolo, regista
    FROM Film
    WHERE valutaz > 3.00;
DECLARE fine BOOLEAN DEFAULT FALSE;
DECLARE CONTINUE HANDLER FOR NOT FOUND
    SET fine = TRUE;
BEGIN
    OPEN valElevata;
    FETCH NEXT FROM valElevata INTO ilTitolo, ilRegista;
    WHILE NOT fine DO
        BEGIN
            PRINT('Titolo: ' || ilTitolo);
            PRINT('Regista: ' || ilRegista);
            FETCH NEXT FROM valElevata INTO ilTitolo, ilRegista;
        END WHILE;
        CLOSE valElevata;
    END;

```

Figura 4.3: Utilizzo del meccanismo delle eccezioni

In modo analogo alle procedure ed alle funzioni definite in un qualunque linguaggio di programmazione, i moduli SQL facilitano lo sviluppo delle applicazioni in quanto aiutano il programmatore a fattorizzare il codice. Evitano inoltre che la conoscenza sul dominio applicativo venga dispersa nelle applicazioni. I moduli sono infatti memorizzati e gestiti dal DBMS e possono essere condivisi e riutilizzati da diverse applicazioni, mantenendo al contempo controllato l'accesso al codice sorgente.

In SQL, un modulo è composto da tre componenti di base: un nome, una lista di dichiarazioni di parametro ed un corpo. Il nome e la lista di dichiarazioni per i parametri rappresentano la *segnatura* del modulo. Il corpo del modulo può essere scritto in vari linguaggi di programmazione, incluso SQL.

Una procedura può essere creata usando il comando `CREATE PROCEDURE`, avente la seguente sintassi:

```

CREATE PROCEDURE <nome procedura> (<lista parametri>)
<corpo procedura>

```

dove:

- `<nome procedura>` è il nome della procedura che viene creata.
- `<lista parametri>` è una lista di dichiarazioni di variabili SQL, separate da virgola. Ogni parametro può essere preceduto dalla parola chiave `IN`, se il parametro è di sola lettura (`default`), `OUT`, se il parametro è di sola

scrittura, oppure **IN OUT**, se il parametro può essere sia letto sia scritto dalla procedura.

- <corpo procedura> corrisponde al codice da eseguire.

**Esempio 4.13** Il seguente comando permette di creare una procedura per aumentare la valutazione di un certo film, il cui titolo e regista sono specificati in input, insieme all'incremento richiesto:

```
CREATE PROCEDURE IncrVal(IN ilTitolo VARCHAR(30),
                         IN ilRegista VARCHAR(20),
                         IN incr NUMERIC(3,2))

UPDATE Film
SET valutaz = valutaz + incr
WHERE titolo = ilTitolo AND regista = ilRegista;
```

Il corpo della procedura in questo caso corrisponde all'esecuzione di un singolo comando SQL. Notiamo che tutti i parametri della procedura sono parametri di input. □

Una procedura può essere invocata usando la seguente sintassi:

```
CALL <nome procedura> (<lista argomenti>);
```

dove <nome procedura> è il nome della procedura da eseguire e <lista argomenti> è una lista di argomenti ciascuno dei quali corrisponde ad: un'espressione, per ogni parametro di input alla procedura; una variabile, per ogni parametro di output o di input output (in questo caso la variabile deve essere inizializzata). I tipi degli argomenti devono essere conformi con la dichiarazione dei parametri corrispondenti, effettuata durante la creazione della procedura.

**Esempio 4.14** Il seguente comando richiama la procedura **IncrVal**, definita nell'Esempio 4.13, ed incrementa la valutazione del film “Pulp fiction” di Quentin Tarantino di 0.50:

```
CALL IncrVal('pulp fiction', 'quentin tarantino', 0.50);
```

Una funzione può essere creata usando il comando **CREATE FUNCTION**, avente la seguente sintassi:

```
CREATE FUNCTION <nome funzione> (<lista parametri>
RETURNS <tipo risultato>
<corpo funzione>)
```

dove <nome funzione> è il nome della funzione che viene creata, <tipo risultato> è il tipo del valore restituito dalla funzione, mentre <lista parametri> e <corpo funzione> hanno lo stesso significato di <lista parametri> e <corpo procedura> nel comando **CREATE PROCEDURE** con le seguenti differenze: i parametri sono solo di input ed il corpo della funzione deve contenere un comando **RETURN** per restituire un valore di tipo <tipo risultato> all'ambiente chiamante.

**Esempio 4.15** Il seguente comando permette la creazione di una funzione per determinare la valutazione media dei film aventi un certo genere, specificato in input alla funzione:

```
CREATE FUNCTION ValMedia(ilGenere CHAR(15))
RETURNS NUMERIC(3,2)
RETURN (SELECT AVG(valutaz)
        FROM Film
        WHERE genere = ilGenere);
```

Come possiamo osservare, il valore di ritorno coincide con il valore restituito da un'interrogazione di aggregazione scalare.  $\square$

Una chiamata di funzione può apparire ovunque possa apparire un valore del tipo restituito della funzione. Una funzione può essere invocata usando la stessa sintassi vista per le procedure o tralasciando la parola chiave **CALL**, usando quindi la seguente sintassi:

```
<nome funzione> (<lista argomenti>);
```

Poiché i parametri di una funzione sono sempre di sola lettura, ogni argomento in **<lista argomenti>** corrisponde in questo caso ad un'espressione, di tipo conforme con la dichiarazione del parametro corrispondente, effettuata durante la creazione della funzione.

**Esempio 4.16** Il seguente comando richiama la funzione **ValMedia**, definita nell'Esempio 4.15, per calcolare la valutazione media per i film di genere comico:

```
ValMedia('comico');
```

Il seguente esempio presenta una procedura che utilizza, nel corpo, molti dei comandi SQL precedentemente introdotti.

**Esempio 4.17** Il codice della Figura 4.4 è un esempio completo di procedura. La procedura prende in input un genere di film e calcola la valutazione media di tutti i film del genere specificato. Se la valutazione media è inferiore a 4, aumenta la valutazione di tutti i film presenti nella videoteca del 5%. Altrimenti, riduce la stessa del 5%. Vengono, quindi, stampati a video il titolo, il regista e la valutazione dei film il cui genere è fornito come valore di input alla procedura.  $\square$

#### 4.6 Approcci basati su librerie di funzioni

L'approccio di accoppiamento esterno basato su librerie di funzioni (vedi Paragrafo 4.1) prevede che l'accesso al DBMS venga effettuato utilizzando un'opportuna libreria di funzioni, implementata nel linguaggio di programmazione con cui intendiamo sviluppare l'applicazione. Tale libreria corrisponde in realtà ad una particolare API per la comunicazione con il DBMS.

```

CREATE PROCEDURE AggiornaVal(ilGenere CHAR(15))
DECLARE laVal      NUMERIC(3,2);
DECLARE ilTitolo   VARCHAR(30);
DECLARE ilRegista  VARCHAR(20);
DECLARE valCr      CURSOR FOR
    SELECT titolo, regista, valutaz
    FROM Film
    WHERE genere = ilGenere;
DECLARE fine BOOLEAN DEFAULT FALSE;
DECLARE CONTINUE HANDLER FOR NOT FOUND
    SET fine = TRUE;
BEGIN
    SELECT AVG(valutaz) INTO laVal FROM Film WHERE genere = ilGenere;
    IF (laVal < 4) THEN
        UPDATE Film SET valutaz = valutaz * 1.05;
    ELSE
        UPDATE Film SET valutaz = valutaz * 0.95;
    END IF;
    OPEN valCr;
    FETCH NEXT FROM valCr INTO ilTitolo, ilRegista, laVal;
    WHILE NOT fine DO
        BEGIN
            PRINT('Titolo: ' || ilTitolo);
            PRINT('Regista: ' || ilRegista);
            PRINT('Valutazione: ' || laVal);
            FETCH NEXT FROM valCr INTO ilTitolo, ilRegista, laVal;
        END;
    END WHILE;
    CLOSE valCr;
END;

```

Figura 4.4: Stored procedure

Nel passato, la maggior parte dei DBMS commerciali ha proposto API specifiche per implementare soluzioni di accoppiamento esterno. Attualmente, esistono molteplici tentativi per standardizzare questi approcci. A questo proposito, lo standard SQL ha proposto *SQL/CLI – SQL Call Level Interface* (interfaccia a livello di chiamata), una specifica per la definizione di tali librerie di funzioni. Contemporaneamente, alcuni approcci di precedente definizione sono diventati ormai uno standard di riferimento per lo sviluppo di applicazioni. Tra le interfacce più comunemente utilizzate, ricordiamo *ODBC – Open Database Connectivity*, proposta originariamente da Microsoft Corp., e *JDBC – Java Database Connectivity*, per interfacciare applicazioni Java e sistemi di gestione dati. Per quanto riguarda ODBC, la libreria è attualmente disponibile per molti sistemi operativi, inclusi Microsoft Windows, Unix, Linux, OS/2 e Mac OS X. JDBC è invece una componente della Java Standard Edition, a partire dalla versione JDK 1.1. Entrambe le interfacce propongono le stesse funzionalità anche se la sintassi di JDBC può risultare più semplice a coloro che si avvicinano a questo tipo di programmazione.

La maggior parte dei DBMS commerciali forniscono supporto per entrambe le librerie.

È importante osservare che l'utilizzo di tali librerie nello sviluppo delle applicazioni garantisce un elevato livello di portabilità tra DBMS diversi. Infatti, tali librerie sono state progettate per permettere ad un'applicazione di connettersi a DBMS differenti utilizzando la stessa interfaccia di comunicazione, garantendo quindi un elevato livello di interoperabilità.

Nel seguito, per prima cosa introdurremo l'architettura di riferimento per gli approcci basati su librerie di funzioni. Quindi, vista la crescente diffusione dell'utilizzo di Java per lo sviluppo di applicazioni, illustreremo le caratteristiche principali di JDBC.

#### **4.6.1 Architettura di riferimento**

L'architettura di riferimento per l'utilizzo di una libreria di funzioni è illustrata nella Figura 4.5. Tale architettura è valida sia per ODBC sia per JDBC. Durante l'esecuzione del programma applicativo, una chiamata ad una funzione di libreria viene catturata da una componente software chiamata *driver manager* (gestore dei driver). Tale componente risiede sulla macchina su cui il programma è in esecuzione. Il driver manager indirizza la chiamata ad una componente software, chiamata *driver*, direttamente collegata con il DBMS a cui la chiamata era inizialmente indirizzata. Il compito del driver è quello di fornire un'implementazione per le funzioni specificate dall'interfaccia nel contesto di uno specifico DBMS. I driver sono quindi forniti dai produttori dei DBMS relazionali. La lista dei driver disponibili nel contesto di un'applicazione, e quindi dei DBMS utilizzabili, deve essere resa nota dall'applicazione al driver manager affinché le chiamate di funzioni eseguite dall'applicazione possano essere indirizzate al driver opportuno.

Una volta ricevuta una richiesta da parte del driver manager, il driver la traduce in una richiesta per il DBMS. A questo punto, il DBMS riceve la richiesta da parte del driver, esegue i comandi SQL richiesti e restituisce il risultato al driver. Tale risultato viene quindi collezionato dal driver, tradotto nel formato richiesto dall'interfaccia e trasmesso al driver manager e quindi all'applicazione che aveva effettuato la richiesta.

Notiamo che la presenza delle componenti driver manager e driver nasconde all'applicazione l'eterogeneità dei linguaggi e dei protocolli di comunicazione dei DBMS utilizzati. L'applicazione comunica, infatti, soltanto con il driver manager, secondo modalità che non dipendono dallo specifico sistema di gestione dati che intendiamo utilizzare.

L'accesso ad un DBMS mediante applicazioni Java può avvenire tramite l'utilizzo di JDBC, la cui definizione risale al 1996. Da un punto di vista applicativo, JDBC corrisponde ad un'API Java. In tale API, il concetto di driver manager corrisponde ad una classe mentre i driver sono modellati come interfacce. Caricare un driver nel contesto di un'applicazione JDBC corrisponde a caricare

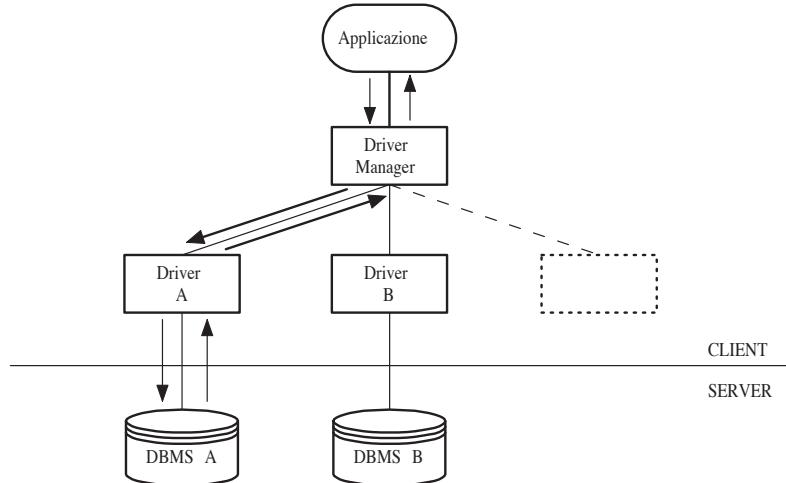


Figura 4.5: Architettura di riferimento per soluzioni ad accoppiamento esterno basato su librerie di funzioni

le classi che implementano un driver JDBC e quindi utilizzarne una particolare implementazione.

JDBC prevede quattro diverse tipologie di driver, che si contraddistinguono per la modalità con cui comunicano con la base di dati. I driver di tipo 1, chiamati anche *ponti JDBC-ODBC (JDBC-ODBC bridge)*, traducono le chiamate JDBC in chiamate ad un driver ODBC per il DBMS considerato. Questa soluzione richiede quindi che la macchina che esegue il codice Java possieda un'installazione di ODBC ed un driver ODBC specifico per il DBMS a cui vogliamo accedere. I driver di tipo 2, chiamati anche *driver nativi*, traducono le chiamate JDBC in chiamate ad un driver specifico del DBMS a cui vogliamo accedere. I driver di tipo 3, chiamati anche *driver middleware*, sono supportati da un software intermedio che traduce le chiamate JDBC in chiamate direttamente eseguibili dal DBMS che intendiamo utilizzare. Tale software è in grado di effettuare la traduzione per i DBMS relazionali più diffusi sul mercato. Infine, i driver di tipo 4, chiamati genericamente *driver JDBC*, sono specifici del DBMS a cui intendiamo accedere e traducono le chiamate JDBC in chiamate direttamente eseguibili dal DBMS, senza l'utilizzo di un software intermedio. Questi driver vengono offerti dagli stessi produttori di RDBMS.

Indipendentemente dal tipo di driver utilizzato, un'applicazione JDBC è caratterizzata dai passi elencati nel Paragrafo 4.3 e descritti con maggior dettaglio nel seguito. Le classi ed i relativi metodi che illustreremo nel seguito sono riassunti nella Tabella 4.1.

Classe	Metodo
DriverManager	getConnection(String url)
Connection	createStatement() prepareStatement(String sql) prepareCall(String sql) close()
Statement	executeQuery(String sql) executeUpdate(String sql) execute(String sql) close();
PreparedStatement sotto-classe di Statement	setXXX(int indiceColonna, XXX valore) con XXX tipo Java executeQuery() executeUpdate() execute()
CallableStatement sotto-classe di PreparedStatement	registerOutParameter(int indiceColonna, int tipoSQL)
ResultSet	next() getXXX(String nomeColonna) getXXX(int indiceColonna) con XXX tipo Java close()
SQLException	getSQLState() getErrorCode() getMessage()

Tabella 4.1: Principali classi e metodi JDBC

#### 4.6.2 Connessione

Come illustrato nel Paragrafo 4.3, il primo passo di un'applicazione basata sull'accoppiamento esterno consiste nello stabilire una connessione con il DBMS che intendiamo utilizzare. È in questa fase che è necessario indicare il driver da utilizzare, che verrà quindi caricato dall'applicazione. Ad ogni driver corrisponde infatti una classe Java. Nel caso in cui venga utilizzato un ponte JDBC-ODBC, la classe è disponibile nel contesto della Java Virtual Machine e può essere caricata con la seguente istruzione:

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

Dopo avere caricato il driver, è possibile stabilire una connessione con il DBMS. Il sistema di gestione dati deve essere univocamente identificato da una stringa di locazione avente la seguente forma:

```
jdbc : <subprotocol> : <subname>
```

Nella stringa precedente, **<subprotocol>** identifica il driver da utilizzare mentre **<subname>** identifica la specifica base di dati a cui ci vogliamo connettere. Se il driver è di tipo 1, la stringa di locazione ha la seguente forma:

---

```
jdbc : odbc : <subname>
```

La connessione viene stabilita utilizzando il metodo `getConnection` della classe `DriverManager`. Il metodo restituisce un oggetto di classe `Connection`. Oltre alla stringa di locazione, il metodo richiede la specifica di un nome utente e di una password.

**Esempio 4.18** La seguente istruzione Java crea una connessione alla base di dati `ilMioDB` utilizzando le credenziali di accesso `myLogin` e `myPassword`:

```
Connection con =
    DriverManager.getConnection("jdbc:odbc:ilMioDB",
                               "myLogin", "myPassword");
```

La connessione viene stabilita se uno dei driver caricati, e quindi noti al driver manager, riconosce la stringa di locazione. Nell'esempio, la connessione viene stabilita se la classe `JdbcOdbcDriver` è stata caricata (quindi è possibile utilizzare il driver di tipo 1) ed `ilMioDB` è un identificatore ODBC valido per il DBMS considerato. □

Una connessione può essere chiusa al termine dell'elaborazione tramite il metodo `close` della classe `Connection`.

#### 4.6.3 Elaborazione

Dopo avere stabilito una connessione, JDBC può essere utilizzato per eseguire comandi SQL sulla base di dati a cui è stato ottenuto l'accesso. In JDBC, i comandi SQL sono sempre valori di tipo stringa, che possono essere noti a tempo di compilazione (SQL statico) o costruiti durante l'esecuzione del programma (SQL dinamico). Il compilatore Java non può tuttavia trarre vantaggio dalla natura statica del comando in quanto l'applicazione comunicherà con il DBMS solo a tempo di esecuzione. È per questo motivo che in JDBC la distinzione tra comandi statici e dinamici è solo formale. Entrambi i tipi di comando vengono, infatti, gestiti in modo omogeneo come comandi dinamici.

In analogia a quanto visto per SQL, anche in JDBC esistono funzioni per gestire il risultato di un'interrogazione tramite cursori. Inoltre, anche in JDBC è possibile separare la fase di preparazione da quella di esecuzione, in modo del tutto analogo a quanto visto per SQL.

I comandi preparati ed eseguiti in un unico passo (chiamati *comandi non preparati* in JDBC) sono oggetti, istanze della classe `Statement`, e possono essere creati a partire da un oggetto di classe `Connection`. Il codice seguente illustra la creazione di un oggetto di classe `Statement`:

```
Connection con;
Statement stmt = con.createStatement();
```

Notiamo che, al momento della creazione dell'oggetto, la stringa corrispondente al codice SQL da eseguire non viene fornita poiché il piano di esecuzione verrà determinato contestualmente all'esecuzione. Tale codice viene quindi fornito solo al momento in cui l'oggetto di classe **Statement** viene eseguito. Questo è possibile invocando un opportuno metodo di esecuzione della classe **Statement**, in relazione al tipo di comando da eseguire. Se il comando SQL da eseguire è un comando **SELECT**, il metodo da eseguire è **executeQuery**. Per comandi di definizione dei dati (ad esempio, **CREATE TABLE**) e per comandi di manipolazione dei dati (**INSERT**, **DELETE** ed **UPDATE**), il metodo da utilizzare è **executeUpdate**. Nel caso in cui la tipologia del comando non sia nota, è possibile utilizzare il metodo **execute**. L'esempio seguente illustra l'utilizzo di questi metodi.

**Esempio 4.19** I seguenti sono alcuni esempi di esecuzione di comandi SQL non preparati:

```
stmt.executeUpdate("CREATE TABLE Film
    (titolo  VARCHAR(30),
     regista VARCHAR(20),
     anno    DECIMAL(4) NOT NULL,
     genere  CHAR(15) NOT NULL,
     valutaz NUMERIC(3,2),
     PRIMARY KEY (titolo,regista))");
stmt.executeUpdate("INSERT INTO Film
    VALUES ('salvate il soldato ryan',
            'steven spielberg',1998,
            'drammatico',3.00)");
stmt.executeQuery("SELECT * FROM Film"); □
```

I comandi preparati ed eseguiti in due passi distinti (chiamati *comandi preparati* in JDBC) sono oggetti, istanze della classe **PreparedStatement** (sotto-classe della classe **Statement**). Poiché la preparazione di un comando avviene prima della sua esecuzione, la stringa corrispondente al comando SQL da eseguire deve essere specificata, tramite il metodo **prepareStatement**, al momento della creazione dell'oggetto. Per eseguire un comando preparato, è possibile utilizzare gli stessi metodi visti per i comandi non preparati. Tuttavia, in questo caso, al momento dell'esecuzione, non deve essere fornito alcun comando SQL.

**Esempio 4.20** Il codice seguente illustra la creazione di alcuni comandi SQL preparati:

```
PreparedStatement queryPstmt =
    con.prepareStatement("SELECT * FROM Film");
PreparedStatement updatePstmt =
    con.prepareStatement("INSERT INTO Film
        VALUES ('salvate il soldato ryan',
                'steven spielberg',1998,
                'drammatico',3.00)");
```

Tali comandi possono poi essere eseguiti come segue:

```
queryPstmt.executeQuery();
updatePstmt.executeUpdate();
```

□

Anche in JDBC, i comandi preparati possono essere parametrici. In questo caso, i parametri sono indicati dal simbolo ‘?’ all’interno del codice SQL. Prima dell’esecuzione del comando, i valori possono essere assegnati ai parametri usando opportuni metodi **setXXX**, dove **XXX** indica un tipo Java. L’input dei metodi **setXXX** è rappresentato dall’identificatore di un parametro di tipo **XXX** e da un’espressione avente tipo **XXX**. Il parametro viene identificato utilizzando una notazione posizionale, da sinistra a destra, secondo l’ordine con cui appare nella stringa che corrisponde al comando che intendiamo eseguire. Il tipo di ogni espressione deve essere compatibile con il tipo del parametro corrispondente. La conversione da tipi Java a tipi SQL viene eseguita implicitamente. Per esempio, il tipo Java **Integer** viene convertito nel tipo SQL **INTEGER**, il tipo Java **Double** viene convertito in **NUMERIC**, mentre il tipo Java **String** viene convertito in **VARCHAR**.

**Esempio 4.21** Il seguente codice illustra la creazione di un comando preparato con un parametro. Tale comando permette di determinare i film aventi un certo genere, fornito come parametro:

```
PreparedStatement queryParPstmt =
    con.prepareStatement("SELECT *
                        FROM Film
                        WHERE genere = ?");
```

Il valore per il parametro verrà fornito prima dell’esecuzione mediante la seguente istruzione:

```
queryParPstmt.setString(1,"thriller");
```

A questo punto, il comando può essere eseguito come segue:

```
queryParPstmt.executeQuery();
```

□

L’esecuzione di un comando **SELECT** restituisce un oggetto istanza della classe **ResultSet**, corrispondente all’insieme delle tuple ottenute come risultato dall’esecuzione del comando stesso. È possibile accedere a ciascuna tupla utilizzando un cursore, implementato mediante metodi della classe **ResultSet**.

Analogamente a quanto visto per SQL, il cursore inizialmente punta alla posizione precedente alla prima tupla del risultato. Il cursore può quindi essere spostato utilizzando metodi opportuni. Il metodo **next** permette di spostare il cursore alla tupla successiva e restituisce **true** se la tupla correntemente puntata esiste, **false** altrimenti.<sup>4</sup> I valori per gli attributi della tupla puntata possono

---

<sup>4</sup> Altri orientamenti sono possibili se lo statement è stato creato come “scrollable”. Questo aspetto non verrà considerato nella trattazione.

quindi essere recuperati utilizzando opportuni metodi `getXXX`, dove `XXX` è il tipo Java in cui il valore recuperato deve essere convertito. L'input dei metodi `getXXX` è rappresentato da un nome di attributo o dalla posizione dell'attributo nello schema del risultato. Oggetti istanze della classe `ResultSet`, cioè risultati di interrogazioni e cursori ad esse associati, vengono implicitamente chiusi al momento della chiusura della connessione. Essi possono anche essere chiusi in modo esplicito utilizzando il metodo `close` della classe `ResultSet`.

**Esempio 4.22** Il seguente codice illustra come sia possibile stampare a video il titolo ed il regista dei film di genere thriller:

```
Statement selStmt = con.createStatement();
String stmt = "SELECT titolo, regista
               FROM Film
              WHERE genere = 'thriller'";
ResultSet rsStmt = selStmt.executeQuery(stmt);
while (rsStmt.next())
{
    System.out.println(rsStmt.getString("titolo"));
    System.out.println(rsStmt.getString("regista"));
}
```

Le chiamate al metodo `getString` possono anche essere effettuate specificando la posizione degli attributi come valore di input, anziché il loro nome. I comandi diventerebbero rispettivamente i seguenti:

```
rsStmt.getString(1);
rsStmt.getString(2);
```

□

JDBC permette di eseguire chiamate a procedure o funzioni SQL, precedentemente create utilizzando i comandi di definizione introdotti nel Paragrafo 4.5. Il metodo da utilizzare è `prepareCall`, definito per la classe `Connection`. Il parametro di questo metodo è una stringa che rappresenta la chiamata della procedura o funzione che vogliamo eseguire. Il risultato è un oggetto istanza della classe `CallableStatement` (sotto-classe di `PreparedStatement`). Tale oggetto può quindi essere eseguito come un comando preparato usando il metodo `executeQuery`, se il corpo del modulo (procedura o funzione) contiene un singolo comando `SELECT`, usando il metodo `executeUpdate`, se contiene comandi di definizione o di manipolazione, ed usando il metodo `execute`, in tutti gli altri casi.

**Esempio 4.23** Il codice seguente illustra come la procedura `AggiornaVal`, presentata nell'Esempio 4.17, può essere chiamata da un programma Java mediante l'uso di JDBC:

```
CallableStatement cs=
    con.prepareCall("call AggiornaVal('comico')");
cs.execute();
```

I parametri di input alla procedura possono non essere noti al momento della preparazione della stessa; in tal caso, vengono lasciati parametrici ed istanziati solo prima dell'invocazione della procedura, come mostra il codice seguente:

```
CallableStatement cs=con.prepareCall("call AggiornaVal(?)");
cs.setString(1,"comico");
cs.execute();
```

□

Il tipo del valore di ritorno di una funzione e dei valori di output di una procedura vengono specificati utilizzando il metodo `registerOutParameter` definito per la classe `CallableStatement`. Tale metodo richiede come input la posizione di un parametro presente nella chiamata di funzione o procedura precedentemente preparata e l'indicazione del tipo corrispondente. Il metodo deve essere eseguito una volta per ogni parametro di output o valore di ritorno. Il seguente esempio ne illustra l'utilizzo.

**Esempio 4.24** Consideriamo la funzione `ValMedia`, definita nell'Esempio 4.15. Il codice da utilizzare per calcolare la valutazione media dei film comici, utilizzando JDBC, è il seguente:

```
CallableStatement cs = con.prepareCall("? = call ValMedia(?)");
cs.registerOutParameter(1,Types.NUMERIC);
cs.setString(2,"comico");
cs.executeQuery();
Double n = cs.getDouble(1);
```

Al termine del codice precedente, la variabile Java `n` conterrà il valore restituito dalla chiamata della funzione `ValMedia('comico')`, opportunamente convertito nel tipo Java `Double`.

□

Le eccezioni sollevate dal DBMS possono essere catturate e gestite da un'applicazione Java utilizzando i metodi della classe `java.sql.SQLException`, che estende la classe `java.lang.Exception`. Ad esempio, il metodo `getErrorCode` restituisce il codice dell'errore generato dal DBMS, mentre il metodo `getMessage` restituisce una descrizione testuale dell'errore.

**Esempio 4.25** La Figura 4.6 mostra un esempio completo di programma Java che utilizza JDBC per comunicare con un DBMS. Il programma esegue le stesse operazioni che definiscono la procedura `AggiornaVal` presentata nell'Esempio 4.17. A titolo di esempio, nel programma il genere del film sul quale viene basata l'elaborazione corrisponde al contenuto della variabile `ilGenere`, di tipo stringa (in un'applicazione reale, tale valore potrebbe essere fornito in input all'applicazione). Questa variabile viene utilizzata per costruire dinamicamente la stringa corrispondente alla selezione della valutazione media dei film aventi il genere specificato, gestita come comando non preparato. In questo caso, il contenuto della variabile, racchiuso tra apici, viene concatenato alla stringa che corrisponde al comando di

```

import java.sql.*;
import java.io.*;
class exampleJDBC
{ public static void main (String args [])
{
    Connection con = null;
    try {
        String ilGenere = "comico";
        Class.forName ("sun.jdbc.odbc.JdbcOdbcDriver");
        con = DriverManager.getConnection("jdbc:odbc:ilMioDB",
                                         "laMiaLogin",
                                         "laMiaPassword");
        Statement st = con.createStatement();
        ResultSet rs = st.executeQuery("SELECT AVG(valutaz)
                                       FROM Film
                                       WHERE genere = '" + ilGenere + "',");
        rs.next();
        if (rs.getDouble(1) < 4)
            st.executeUpdate("UPDATE Film
                             SET valutaz = valutaz * 1.05");
        else
            st.executeUpdate("UPDATE Film
                             SET valutaz = valutaz * 0.95");
        PreparedStatement pst =
            con.prepareStatement("SELECT titolo, regista, valutaz
                               FROM Film
                               WHERE genere = ?");
        pst.setString(1,ilGenere);
        rs = pst.executeQuery();
        while (rs.next())
            System.out.println("Titolo: " + rs.getString(1) +
                               " Regista: " + rs.getString(2) +
                               " Valutazione: " + rs.getDouble(3));
        con.close();
    }
    catch(java.lang.ClassNotFoundException e) {
        System.err.print("ClassNotFoundException: ");
        System.err.println(e.getMessage());
    }
    catch (SQLException e) {
        while( e!=null){
            System.out.println("SQLState: " + e.getSQLState());
            System.out.println("Code: " + e.getErrorCode());
            System.out.println("Message: " + e.getMessage());
            e = e.getNextException();
        }}}}}

```

Figura 4.6: Programma Java che utilizza JDBC per interagire con la base di dati

selezione da eseguire. La variabile `ilGenere` viene inoltre utilizzata per assegnare un valore al parametro del comando preparato che seleziona tutti i film di un certo genere.  $\square$

## 4.7 SQL ospitato

Un secondo approccio all'accoppiamento esterno permette di utilizzare SQL come linguaggio "ospitato" da un linguaggio di programmazione generico (detto *linguaggio ospite*). Utilizzando questo approccio, i comandi SQL possono essere direttamente specificati nel programma ospite, senza la necessità di utilizzare una particolare libreria di funzioni.

La variante di SQL per l'utilizzo da linguaggio di programmazione è comunemente chiamata *SQL ospitato* (*embedded SQL*). Il linguaggio risultante mette a disposizione tutte le funzionalità del linguaggio ospite e permette di utilizzare direttamente SQL per l'accesso e la manipolazione dei dati.

Lo standard SQL indica chiaramente come SQL possa essere ospitato in un certo insieme di linguaggi di programmazione, tra cui C, COBOL, Pascal e Fortran. Benché Java non sia incluso in questa lista, successivamente allo sviluppo di JDBC, è stata proposta una specifica ANSI/ISO per incapsulare comandi SQL in Java. Tale standard, noto come SQLJ (SQL Java), è stato sviluppato dall'SQLJ Group, un consorzio che comprende produttori di DBMS e Sun Microsystems. Vista la crescente diffusione dell'utilizzo di Java per lo sviluppo di applicazioni per basi di dati, in questo paragrafo, dopo avere introdotto l'architettura di riferimento per SQL ospitato, a titolo di esempio illustreremo le caratteristiche principali di SQLJ.

### 4.7.1 Architettura di riferimento

Un programma sviluppato secondo l'approccio di SQL ospitato contiene sia comandi espresi secondo la sintassi del linguaggio ospite sia comandi SQL. Un comando SQL si distingue dalle istruzioni del linguaggio ospite per la presenza di un prefisso ed un terminatore. Lo standard SQL indica come prefisso, per i linguaggi considerati, la parola chiave `EXEC SQL`. In SQLJ, il prefisso è `#sql`. In entrambi i casi, il punto e virgola viene usato come terminatore del comando.

La compilazione di un programma sviluppato secondo questo approccio avviene in due passi distinti:

1. **Pre-compilazione.** Il programma viene passato ad un pre-compilatore specifico del linguaggio ospite. Il pre-compilatore elimina i comandi SQL sostituendoli con chiamate al DBMS, espresse nella sintassi del linguaggio ospite. Il risultato è un programma scritto completamente nel linguaggio ospite.
2. **Compilazione.** Il programma risultante dalla fase precedente viene compilato utilizzando un compilatore per il linguaggio ospite.

Nel caso di SQLJ, il pre-compilatore viene chiamato *traduttore* (*translator*). Il traduttore legge un file SQLJ sorgente, con estensione `.sqlj`, e lo traduce in un file sorgente Java, con estensione `.java`. Questo programma contiene chiamate alle librerie messe a disposizione dalle implementazioni di SQLJ per il DBMS utilizzato. Spesso, i sistemi di gestione dati permettono di utilizzare a questo scopo direttamente JDBC. Per ogni connessione richiesta dall'applicazione, il traduttore genera anche un profilo, contenente informazioni relative ai comandi SQL da eseguire nel contesto di ciascuna connessione. L'ultima operazione eseguita dal traduttore consiste nella compilazione del file Java sorgente in codice eseguibile.

#### 4.7.2 Connessione

La connessione ad un DBMS viene stabilita chiamando il metodo `connect`, definito in una particolare classe resa disponibile dal DBMS considerato. Analogamente al metodo JDBC `getConnection`, il metodo `connect` accetta tre parametri: una stringa di locazione per la base di dati che vogliamo utilizzare, un nome utente ed una password. Ad esempio, la connessione ad una base di dati Oracle in SQLJ, supponendo di utilizzare un driver JDBC di tipo 1, è la seguente:

```
oracle.connect("jdbc:odbc:ilMioDB",
               "lamiaLogin",
               "lamiaPassword");
```

Al termine dell'elaborazione, una connessione aperta può essere chiusa mediante il metodo `close`.

#### 4.7.3 Elaborazione

SQLJ permette solo l'esecuzione di comandi SQL statici.<sup>5</sup> I comandi dinamici possono comunque essere eseguiti tramite JDBC. I comandi SQL sono preceduti dalla parola chiave `#sql`, terminati da un punto e virgola e posizionati tra parentesi graffe, secondo la seguente sintassi:

```
#sql {<comando SQL>};
```

**Esempio 4.26** La seguente istruzione SQLJ inserisce un nuovo film nella base di dati:

```
#sql {INSERT INTO Film VALUES ('salvate il soldato ryan',
                               'steven spielberg', 1998,
                               'drammatico', 3.00)}; □
```

Le variabili Java possono essere utilizzate nei comandi SQL, precedute dal simbolo `:` per distinguerle dai nomi degli attributi delle relazioni. Ogni volta che una variabile viene utilizzata nel contesto di un comando SQL, viene applicata una

---

<sup>5</sup>Ricordiamo che tale limitazione non vale per altri linguaggi ospite.

conversione implicita dal tipo Java al tipo SQL corrispondente. Le regole applicate sono simili a quelle introdotte per JDBC. Analogamente, il comando SQL può includere riferimenti ad altri elementi Java, come elementi di array, attributi di oggetti e chiamate di funzioni.

Il risultato di un comando **SELECT** scalare può essere collezionato in un insieme di variabili Java, utilizzando il comando **SELECT INTO** introdotto nel Paragrafo 4.5.1, come illustrato dal seguente esempio.

**Esempio 4.27** Il seguente codice SQLJ seleziona il genere del film avente come titolo e regista quelli contenuti rispettivamente nelle variabili **ilTitolo** ed **ilRegista**:

```
String ilTitolo = "big fish";
String ilRegista = "tim burton";
String ilGenere;
#sql{SELECT genere INTO :ilGenere
      FROM Film
      WHERE titolo = :ilTitolo AND regista = :ilRegista}; □
```

Se l'interrogazione non è scalare, è necessario utilizzare un cursore per elaborare il risultato. In SQLJ, un cursore è un oggetto istanza di una classe **iterator**. Una classe iterator deve essere definita dall'applicazione per ogni tipo di risultato da elaborare; essa contiene un attributo per ogni attributo delle tuple del risultato. Una classe iterator è definita con la seguente sintassi:

```
#sql iterator <nome classe> (<lista attributi>);
```

dove **<nome classe>** è il nome della classe iterator che viene definita e **<lista attributi>** è una lista di dichiarazioni di attributi, separate da virgola.

**Esempio 4.28** Il seguente comando definisce una classe iterator per tuple aventi due attributi, **titolo** e **regista**:

```
#sql iterator FilmIter (String titolo, String regista); □
```

Il risultato della valutazione di un comando **SELECT** viene assegnato ad un'istanza di una classe iterator, di tipo compatibile con il risultato dell'interrogazione. Analogamente a quanto avviene in JDBC, il metodo **next** permette di muovere il cursore sulla tupla successiva del risultato. Se tale tupla esiste, il metodo restituisce **true**, altrimenti restituisce **false**. I metodi di accesso, associati a ciascun attributo della classe iterator, possono quindi essere utilizzati per recuperare i valori corrispondenti alla tupla correntemente puntata. Un oggetto istanza di una classe iterator può infine essere chiuso tramite il metodo **close**.

**Esempio 4.29** L'esempio seguente illustra un possibile utilizzo della classe iterator **FilmIter**, dichiarata nell'Esempio 4.28:

```

FilmIter ilFilmIter = null;
#sql ilFilm_iter = {SELECT titolo, regista FROM Film};
while (ilFilmIter.next())
{
    System.out.println("Titolo: " + ilFilmIter.titolo());
    System.out.println("Regista: " + ilFilmIter.regista());
}
ilFilmIter.close();
```

□

Anche in SQLJ è possibile chiamare procedure precedentemente create, utilizzando la seguente sintassi:

```
#sql {CALL <nome procedura> (<lista argomenti>)};
```

dove **<nome procedura>** è il nome della procedura da eseguire e **<lista argomenti>** è una lista di argomenti ciascuno dei quali corrisponde ad: un'espressione, preceduta dalla parola chiave **IN**, per ogni parametro di input alla procedura; una variabile, preceduta dalla parola chiave **OUT**, per ogni parametro di output; una variabile, preceduta dalla parola chiave **IN OUT**, per ogni parametro di input output (in questo caso la variabile deve essere inizializzata). I tipi degli argomenti devono essere conformi con la dichiarazione dei parametri corrispondenti, effettuata durante la creazione della procedura.

Una funzione può essere richiamata in modo simile. In questo caso però, il valore di ritorno può essere immediatamente memorizzato in una variabile di programma, utilizzando la seguente sintassi:

```
#sql <nome variabile> = {VALUES <nome funzione>
                           (<lista valori parametri>)};
```

La gestione delle eccezioni è, infine, del tutto simile a quella introdotta nel Paragrafo 4.6.3 per JDBC.

**Esempio 4.30** La Figura 4.7 riporta un esempio completo di programma SQLJ. Il programma esegue le stesse operazioni eseguite dalla procedura **AggiornaVal** presentata nell'Esempio 4.17. A titolo di esempio, nel programma il genere del film sul quale viene basata l'elaborazione corrisponde al contenuto della variabile **ilGenere**, di tipo stringa (in un'applicazione reale, tale valore potrebbe essere fornito in input all'applicazione). Come possiamo osservare, la possibilità di specificare direttamente i comandi SQL, senza l'uso di funzioni di libreria, rende il codice più leggibile rispetto al codice presentato nella Figura 4.6. □

### Note conclusive

Il linguaggio SQL, benché ricco di funzionalità, non fornisce i costrutti tipici di un linguaggio di programmazione e deve quindi essere esteso od integrato con un

```
import java.sql.*;
import java.io.*;
import java.math.*;
import sqlj.runtime.*;
import sqlj.runtime.ref.*;
import oracle.sqlj.runtime.*;
class exampleSQLj
{
    #sql iterator FilmIter(String titolo, String regista);
    public static void main (String args [])
    {
        try{
            Double valMedia;
            String ilGenere = "comico";
            oracle.connect("jdbc:odbc:ilMioDB",
                           "laMiaLogin",
                           "laMiaPassword");
            #sql{SELECT AVG(valutaz) INTO :valMedia
                  FROM Film
                  WHERE genere = :ilGenere};
            if (valMedia > 4)
                #sql{UPDATE Film SET valutaz = valutaz * 1.05};
            else
                #sql{UPDATE Video SET valutaz = valutaz * 0.95};
            FilmIter ilFilmIter = null;
            #sql ilFilmIter ={SELECT titolo, regista, valutaz
                               FROM Film
                               WHERE genere = :ilGenere};
            while (ilFilmIter.next())
                System.out.println("Titolo: " + ilFilmIter.titolo() +
                                   " Regista: " + ilFilmIter.regista() +
                                   " Valutazione: " + ilFilmIter.valutaz());
        }
        catch(java.lang.ClassNotFoundException e) {
            System.err.print("ClassNotFoundException: ");
            System.err.println(e.getMessage());
        }
        catch (SQLException e) {
            while(e!=null){
                System.out.println("SQLState: " + e.getSQLState());
                System.out.println("Code: " + e.getErrorCode());
                System.out.println("Message: " + e.getMessage());
                e = e.getNextException();
            }
        }
    }
}
```

Figura 4.7: Programma Java che utilizza SQLJ per interagire con la base di dati

## Capitolo 8

# Gestione delle transazioni

Tra i compiti principali di un DBMS vi è quello di fornire un ambiente efficiente per accedere e memorizzare i dati, garantendo nel contempo la correttezza dei dati stessi. Uno dei principali fattori da considerare al fine di garantire tale proprietà è che l'accesso concorrente ai dati da parte di più applicazioni può, se non opportunamente regolamentato, causare inconsistenze ai dati. Allo stesso tempo, è di fondamentale importanza aumentare quanto più possibile il grado di concorrenza (in termini di processi in esecuzione) in quanto questo comporta un miglior utilizzo delle risorse ed una conseguente diminuzione dei tempi di risposta. Occorre pertanto che il DBMS sia dotato di opportuni *meccanismi di controllo della concorrenza* che permettano l'esecuzione concorrente di più processi preservando nel contempo la consistenza dei dati memorizzati nella base di dati. Inoltre, durante l'esecuzione di un processo è possibile che si verifichino malfunzionamenti di varia natura, ad esempio causati da errori nei programmi applicativi o nel DBMS stesso, o da guasti di componenti hardware, quali ad esempio la rottura di un disco. Se non è previsto alcun meccanismo in grado di affrontare tali anomalie, allora il malfunzionamento può generare dati scorretti e può causare serie perdite di dati. Ad esempio, consideriamo un programma che effettua un'operazione di giroconto, trasferendo un certo importo da un conto corrente ad un altro. Supponiamo che il programma, dopo aver effettuato il prelevamento dal primo conto e prima di aver effettuato il versamento sul secondo, termini erroneamente a causa di un malfunzionamento software o hardware. Se il DBMS non effettua opportune operazioni di *ripristino (recovery)*, la base di dati rimane in uno stato scorretto, in quanto la somma di denaro prelevata dal primo conto non ha più alcun riscontro nella base di dati.

Data l'estrema importanza di agire sempre su dati consistenti, grossi sforzi di ricerca e sviluppo sono stati fatti, volti alla definizione di tecniche sia per la gestione di errori e malfunzionamenti sia per il controllo della concorrenza. La nozione chiave che sta alla base di tutte queste tecniche è il concetto di *transazione*. Una transazione è un insieme di operazioni di lettura e scrittura sulla base di dati a cui il DBMS assicura particolari proprietà (dette *proprietà ACIDe*, descritte nel Paragrafo 8.1.1) che garantiscono la consistenza dei dati in presenza di transazioni concorrenti e di guasti/malfunzionamenti di varia natura. In questo modo, il programmatore non deve preoccuparsi di questi aspetti. Dovrà solo, tra-

```

BeginWork
  Read lr
  If lr < 150 Then Abort
  Else
    lr := lr - 150
    Write lr
    Read cc
    cc := cc + 150
    Write cc
  EndIf
  Commit
EndWork

```

Figura 8.1: Transazione relativa ad un dominio bancario

mite opportune primitive, identificare le porzioni di codice a cui vuole assicurare un comportamento transazionale; in questo modo, la gestione delle problematiche relative a concorrenza e fallimenti sarà demandata al DBMS.

Lo scopo di questo capitolo è presentare una trattazione formale del concetto di transazione e del controllo della concorrenza. Illustreremo inoltre le principali tecniche adottate dai DBMS per il controllo della concorrenza e per il ripristino della base di dati a seguito di malfunzionamenti hardware/software. Infine, descriveremo il supporto alle transazioni offerto da SQL e dalle soluzioni per la programmazione di applicazioni descritte nel Capitolo 4. Per illustrare in maniera ottimale i concetti relativi al controllo della concorrenza ed alla gestione del ripristino utilizzeremo in questo capitolo un esempio relativo ad un dominio bancario, invece del solito esempio della videoteca.

### 8.1 Concetti di base

Come accennato nell'introduzione, una transazione può essere informalmente definita come un insieme di operazioni di lettura e scrittura su una base di dati, svolte da un'applicazione a cui vogliamo garantire particolari proprietà di correttezza rispetto alle operazioni effettuate sulla base di dati. Linguaggi diversi mettono a disposizione primitive diverse per indicare l'inizio e la fine di una transazione (vedi Paragrafo 8.7 per ulteriori dettagli). Nel seguito, astraendo dai dettagli sintattici dei vari linguaggi, utilizzeremo le istruzioni `BeginWork` ed `EndWork` per indicare l'inizio e la fine di una transazione. Una transazione può o terminare con successo la sua esecuzione (cioè effettuare il *commit*), oppure non essere completata (cioè effettuare l'*abort*). L'abort di una transazione può essere dovuto a varie anomalie che accadono durante la sua esecuzione, quali errori hardware o software. Può anche essere richiesto da programma applicativo per gestire condizioni anomale. Per questo motivo, all'interno di una transazione possono comparire altre due istruzioni: `Abort` e `Commit`. Se non esplicitamente richiesto, assumiamo

che il commit coincida con l'istruzione di fine transazione. Un semplice esempio di transazione che effettua un trasferimento di 150 euro da un libretto di risparmio (*lr*) ad un conto corrente (*cc*) è mostrato nella Figura 8.1. Il codice presentato in tale figura rappresenta un'astrazione dell'effettivo codice della transazione, che dovrà contenere opportune istruzioni SQL per registrare l'effetto delle operazioni di trasferimento all'interno della base di dati.

### 8.1.1 Proprietà ACIDe

A tutte le istruzioni contenute tra le primitive `BeginWork` ed `EndWork` il DBMS assicura quattro proprietà, conosciute come proprietà *ACIDe*: *Atomicità*, *Consistenza*, *Isolamento* e *Persistenza*. Il termine *ACIDe* deriva dall'acronimo dei termini inglesei per denotare tali proprietà: *Atomicity*, *Consistency*, *Isolation*, *Durability*. Vediamo ora in dettaglio le quattro proprietà:

- **Atomicità** (detta anche *proprietà tutto-o-niente*). Tutte le operazioni di una transazione sono considerate dal DBMS come una singola unità; quindi, o vengono eseguite tutte e la transazione effettua il commit, oppure non ne viene eseguita alcuna. In questo secondo caso, il DBMS annulla gli effetti dell'esecuzione parziale della transazione (*rollback*), riportando la base di dati allo stato in cui si sarebbe trovata se la transazione che effettua l'abort non fosse mai stata eseguita.
- **Consistenza**. La proprietà di consistenza assicura che l'esecuzione di una transazione non violi i vincoli d'integrità specificati per la base di dati (vedi Capitolo 3 per la specifica di tali vincoli). Ricordiamo che la valutazione di un vincolo può essere immediata, cioè dopo ogni operazione all'interno di una transazione, o differita al termine dell'esecuzione della transazione. Nel primo caso, una violazione del vincolo comporta l'annullamento degli effetti dell'operazione che ha violato il vincolo, ma non causa l'abort della transazione. Nel secondo caso, invece, la violazione del vincolo comporta l'abort della transazione. La proprietà di consistenza si riferisce esclusivamente al soddisfacimento dei vincoli d'integrità specificati nella base di dati e non ad eventuali inconsistenze dovute alla logica dell'applicazione. Ad esempio, in riferimento alla transazione nella Figura 8.1, è compito del programmatore far sì che l'importo prelevato dal libretto di risparmio sia equivalente all'importo accreditato sul conto corrente.
- **Isolamento**. Questa proprietà assicura che l'esito di una transazione non sia influenzato dall'esecuzione concorrente di altre transazioni e che la transazione operi, quindi, come se disponesse di un sistema dedicato in cui nessuna altra transazione è in esecuzione. Uno degli effetti di questa proprietà è che una transazione non può leggere risultati intermedi di altre transazioni.
- **Persistenza**. I risultati di una transazione terminata con successo devono essere resi permanenti nella base di dati nonostante possibili malfunzionamenti del sistema.

Le proprietà ACIDe vengono garantite da diversi moduli del DBMS. In particolare, il *gestore del ripristino* applica le strategie necessarie ad assicurare l'atomicità delle transazioni ed a gestire correttamente il rollback. Questo modulo assicura inoltre la persistenza dei risultati di una transazione che ha effettuato il commit. La proprietà di isolamento è assicurata, invece, dal *gestore della concorrenza*, che isola gli effetti di una transazione fino alla terminazione della stessa. In particolare, tale modulo sincronizza l'esecuzione di transazioni in modo da garantire esecuzioni concorrenti libere da interferenze. Infine, la consistenza è assicurata sia dal gestore della concorrenza sia dal meccanismo per la specifica e la verifica dei vincoli d'integrità messo a disposizione dal DBMS.

### 8.1.2 Un semplice modello di transazioni

In letteratura sono stati proposti diversi modelli di transazioni, alcuni dei quali saranno discussi nel Paragrafo 8.6. La maggior parte di questi modelli propone estensioni al modello base di transazioni oggetto di questo paragrafo. Le transazioni definite in accordo a questo modello sono dette *transazioni flat* (o transazioni semplici). Le transazioni flat rappresentano il tipo di transazione più semplice; sono utilizzate in tutti i DBMS disponibili in commercio, pur con qualche estensione, e verranno utilizzate nel prosieguo della trattazione per illustrare le tecniche che i DBMS utilizzano al fine di garantire le proprietà ACIDe.

In una transazione flat esiste un solo livello di controllo, a cui appartengono tutte le istruzioni contenute tra le primitive `BeginWork` ed `EndWork`. Una transazione flat non presenta quindi alcuna struttura gerarchica. La definizione seguente introduce formalmente tale concetto.

**Definizione 8.1 (Transazione flat)** Una transazione flat  $T_i$  è un insieme di operazioni sui dati su cui è definito un ordinamento parziale  $<_i$ , che specifica l'ordine in cui le operazioni vengono eseguite. Sia  $O = \{\text{Read}_i[x] \mid x \text{ è un dato}\} \cup \{\text{Write}_i[x] \mid x \text{ è un dato}\}$  e sia  $\mathcal{I} = 2^O$  l'insieme delle parti di  $O$ ; allora:

- $T_i \subseteq \mathcal{I} \cup \{a_i, c_i\}$ , dove:
  1.  $a_i$  indica che la transazione non è terminata con successo (cioè ha effettuato l'operazione di *abort*);
  2.  $c_i$  indica che la transazione è terminata correttamente (cioè ha effettuato l'operazione di *commit*);
- $a_i \in T_i$  se e solo se  $c_i \notin T_i$ ;
- se  $op \in \{a_i, c_i\}$ , allora per ogni operazione  $\overline{op}$  tale che  $\overline{op} \in T_i$  e  $\overline{op} \neq op$ , abbiamo  $\overline{op} <_i op$ ;
- per ogni coppia di operazioni  $op_h, op_k \in T_i$  tali che  $op_h$  e  $op_k$  accedono allo stesso dato e una delle due è un'operazione di scrittura, allora o vale  $op_h <_i op_k$  oppure vale  $op_k <_i op_h$ .  $\diamond$

La definizione di transazione flat stabilisce che una transazione possa, in modo esclusivo, effettuare abort oppure commit, e che l'istruzione di abort o di commit debba essere l'ultima istruzione effettuata dalla transazione. Viene inoltre imposto un ordinamento sulle operazioni di lettura/scrittura sullo stesso dato. Infatti, la terza condizione della Definizione 8.1 stabilisce che due operazioni sullo stesso dato  $x$ , di cui almeno una delle due sia una scrittura, siano ordinabili tramite la relazione di ordinamento  $<_i$ . Questo vincolo è motivato dal fatto che, se non venisse posta alcuna restrizione sull'ordinamento delle operazioni di lettura e scrittura sullo stesso dato, non potrebbe essere determinato con certezza l'effetto della loro esecuzione sul dato in questione. Per semplicità, nel seguito considereremo solo transazioni la cui sequenza di operazioni è ordinata secondo un ordinamento totale. Iniziamo la nostra trattazione affrontando il problema del controllo della concorrenza.

## 8.2 Controllo della concorrenza

Scopo del controllo della concorrenza è garantire che l'esecuzione concorrente di più transazioni non violi le proprietà di isolamento e consistenza, portando le transazioni ad agire su dati non corretti o a far sì che dati non corretti siano memorizzati nella base di dati. A tali situazioni viene dato il nome di *anomalie*.<sup>1</sup> Il paragrafo successivo illustra gli esempi più rilevanti di tali anomalie.

### 8.2.1 Anomalie nell'esecuzione concorrente di transazioni

Le anomalie causate dall'esecuzione concorrente di transazioni sono dovute al fatto che, se le loro operazioni non sono opportunamente regolamentate, alcune transazioni leggono risultati intermedi di transazioni non ancora completate. Una delle più importanti anomalie è la cosiddetta *perdita di aggiornamento* (*lost update*) che si verifica quando l'aggiornamento effettuato da una transazione viene perso perché sovrascritto in modo non corretto da un'altra transazione. L'esempio seguente illustra tale anomalia.

**Esempio 8.1** Consideriamo ancora il dominio delle transazioni bancarie e supponiamo di avere due transazioni  $T_1$  e  $T_2$ . La prima trasferisce 150 euro da un libretto di risparmio ad un conto corrente, mentre la seconda accredita 5.000 euro sullo stesso conto corrente. Qui e nel seguito utilizziamo una notazione che evidenzia la progressione temporale delle due transazioni e la loro esecuzione concorrente ed omettiamo per semplicità le primitive `BeginWork` ed `EndWork`. Supponiamo dunque che l'esecuzione concorrente di  $T_1$  e  $T_2$  sia la seguente:

T <sub>1</sub>	T <sub>2</sub>
Read lr	
lr := lr - 150	

---

<sup>1</sup>Notiamo che le anomalie discusse in questo capitolo sono del tutto differenti da quelle discusse nel Capitolo 6.

```

    Read cc
Write lr
    cc := cc + 5'000
Read cc
    Write cc
    Commit
    cc := cc + 150
Write cc
Commit

```

Questa esecuzione, a causa di un'interferenza tra  $T_1$  e  $T_2$ , non produce l'effetto desiderato sulla base di dati, cioè un incremento del saldo del conto corrente di 5'150 euro, nonostante entrambe le transazioni terminino la propria esecuzione con successo. Il problema è che la somma depositata da  $T_2$  su  $cc$  viene persa perché sovrascritta dalla scrittura effettuata successivamente da  $T_1$ . L'incremento del saldo alla fine delle due transazioni è quindi pari a 150 euro.  $\square$

Un'ulteriore fonte di anomalie è determinata dalle cosiddette *lettture sporche* (*dirty read*), che possono verificarsi quando una transazione legge un risultato intermedio di un'altra transazione non ancora completata.

**Esempio 8.2** Consideriamo le transazioni  $T_1$  e  $T_2$  dell'Esempio 8.1 e supponiamo che siano eseguite concorrentemente nel modo seguente:

$T_1$	$T_2$
Read $lr$	
$lr := lr - 150$	
Write $lr$	
Read $cc$	
$cc := cc + 150$	
Write $cc$	
	Read $cc$
	$cc := cc + 5'000$
	Write $cc$
Abort	
	Commit

In questo caso, per la proprietà di atomicità, gli effetti di  $T_1$  devono essere annullati e, quindi, alla fine dell'esecuzione delle due transazioni il conto corrente dovrebbe essere incrementato solo di 5'000 euro (quelli depositati dalla transazione  $T_2$  che termina con successo). Invece, il conto corrente alla fine dell'esecuzione è incrementato di 5'150. Tale anomalia è dovuta al fatto che la transazione  $T_2$  legge la modifica effettuata sul conto corrente da  $T_1$  prima che questa sia terminata con successo, operando quindi su un dato non corretto a fronte di un abort di  $T_1$ .  $\square$

Un'altra situazione anomala si ha quando una transazione legge due volte lo stesso dato ma tra la prima lettura e la seconda il dato viene modificato da un'altra transazione. Questo fenomeno, chiamato delle *lettture non ripetibili* (*non repeatable read*) comporta la violazione della proprietà di isolamento.

**Esempio 8.3** Consideriamo la seguente esecuzione concorrente di due transazioni  $T_1$  e  $T_2$ :

$T_1$	$T_2$
Read $lr$	
	Read $lr$
	$lr := lr - 150$
	Write $lr$
	Commit
Read $lr$	
Commit	

In questo caso la transazione  $T_1$  legge due valori di  $lr$  che differiscono per 150 euro, a causa della modifica effettuata da  $T_2$ .  $\square$

Infine, un'ulteriore anomalia, causata sempre da interferenze tra transazioni, è quella dovuta ai cosiddetti *fantasmi* (*phantom*). Questo fenomeno accade ad esempio quando una transazione effettua nel corso della sua esecuzione due volte la stessa interrogazione, ottenendo come risposta alla seconda esecuzione un numero maggiore di tuple. Queste “tuple fantasma” sono causate da modifiche effettuate nel frattempo da un'altra transazione. L'esempio seguente chiarisce meglio il concetto.

**Esempio 8.4** Supponiamo che la base di dati del dominio bancario considerato in questo capitolo contenga una relazione **Conto** che memorizzi informazioni sui conti correnti aperti presso la banca (nome, cognome, codice fiscale del titolare, saldo, ecc.) Consideriamo la seguente esecuzione concorrente di due transazioni  $T_1$  e  $T_2$  che operano sulla relazione **Conto**:

$T_1$	$T_2$
SELECT * FROM Conto;	
	INSERT INTO Conto
	VALUES('mario','rossi',...);
...	...
SELECT * FROM Conto;	
Commit	
	Commit

In questo caso la seconda esecuzione della stessa interrogazione da parte di  $T_1$  restituisce una tupla in più, il “fantasma” appunto, quella relativa al conto di Mario Rossi inserita nel frattempo da  $T_2$ , anche se  $T_1$  non ha modificato la relazione `Conto`.  $\square$

### 8.2.2 Serializzabilità

I meccanismi di controllo della concorrenza adottati dai DBMS evitano le anomalie illustrate nel paragrafo precedente. L’idea alla base di questi meccanismi è di regolamentare l’esecuzione di transazioni concorrenti, prevenendo quelle combinazioni di esecuzioni concorrenti che non garantiscono l’isolamento e la consistenza. Questo significa consentire solo quelle esecuzioni concorrenti che siano *equivalenti* ad un’esecuzione in serie (cioè una dopo l’altra) delle transazioni coinvolte. Prima di precisare formalmente il concetto di equivalenza, introduciamo alcuni concetti e notazioni, utili nel prosieguo della discussione.

L’esecuzione concorrente di due o più transazioni genera un’alternanza di computazioni da parte delle varie transazioni, nota con il nome di *interleaving*. La corrispondente sequenza di operazioni viene definita *schedule*. In particolare, uno schedule è una lista di operazioni di lettura, scrittura, abort e commit, eventualmente appartenente a più transazioni, che preserva l’ordine (vedi Definizione 8.1) con cui le operazioni vengono eseguite nella corrispondente transazione; cioè se due operazioni di una transazione  $T$  appaiono nello schedule, il loro ordine nello schedule deve essere uguale a quello in  $T$ . Nel seguito, ai fini della trattazione teorica, consideriamo schedule che non contengono operazioni di abort. Questo equivale ad ipotizzare un ambiente ideale, dove non si verificano malfunzionamenti e tutte le transazioni terminano con successo. Naturalmente, questa ipotesi è valida solo per una trattazione teorica del problema del controllo della concorrenza. Vedremo nel paragrafo successivo come i meccanismi utilizzati dai DBMS non effettuino questa assunzione.

**Esempio 8.5** Consideriamo l’esecuzione concorrente dell’Esempio 8.1. Il corrispondente schedule è: `[Read1[lr],2 Read2[cc], Write1[lr], Read1[cc], Write2[cc], Commit2, Write1[cc], Commit1]`.  $\square$

Come discusso nel Paragrafo 8.2.1, l’interleaving tra due transazioni  $T_1$  e  $T_2$  può causare delle anomalie e portare la base di dati in uno stato non corretto. Tuttavia, qualsiasi transazione produrrebbe uno stato corretto se fosse eseguita da sola. In altri termini, non si sarebbero verificati problemi se le transazioni  $T_1$  e  $T_2$  fossero state eseguite l’una dopo l’altra, consecutivamente. Uno schedule con questa proprietà viene detto *schedule seriale*. Più formalmente, uno schedule è seriale se per ogni coppia di transazioni  $T_i$ ,  $T_j$ ,  $i \neq j$ , le cui operazioni appartengono allo

---

<sup>2</sup>Qui e nel seguito denotiamo con  $op_i$  le operazioni nello schedule che appartengono alla transazione  $T_i$ .

schedule, tutte le operazioni di  $T_i$  appaiono nello schedule prima delle operazioni di  $T_j$ , o viceversa.

Intuitivamente, gli schedule seriali realizzano la proprietà di isolamento. Chiaramente, nella pratica, non è opportuno restringere l'esecuzione ai soli schedule seriali, in quanto questo comporterebbe un notevole spreco di risorse con conseguente degrado delle prestazioni. Un criterio di correttezza può però essere formulato in termini di schedule seriali. In generale, non produrranno inconsistenze, e quindi potranno essere ammessi, tutti gli schedule *equivalenti* ad un qualche schedule seriale. Tali schedule vengono detti *schedule serializzabili* e saranno i soli ammessi dal gestore della concorrenza. È però necessario chiarire cosa intendiamo per equivalenza tra schedule. In letteratura esistono varie nozioni di equivalenza, caratterizzate da complessità diverse per la loro verifica. La prima nozione è introdotta formalmente nel seguito.

**Definizione 8.2 (Equivalenza rispetto alle viste)** Due schedule  $S_1$  ed  $S_2$  sono equivalenti rispetto alle viste, denotato con  $S_1 \equiv_V S_2$ , se:

1. l'insieme delle operazioni in  $S_1$  coincide con l'insieme delle operazioni in  $S_2$ ;
2. per ogni dato  $d$  e per ogni coppia di transazioni  $T_i, T_j$  appartenenti agli schedule, se in  $S_1$  la transazione  $T_i$  esegue `Read[d]` e  $T_j$  è l'ultima transazione che modifica il valore di  $d$  prima della lettura effettuata da  $T_i$ , lo stesso avviene in  $S_2$ ;
3. per ogni dato  $d$ , se in  $S_1$  la transazione  $T_i$  è l'ultima transazione ad eseguire `Write[d]`, lo stesso avviene in  $S_2$ .  $\diamond$

La condizione (1) assicura che lo stesso insieme di transazioni partecipi ad entrambi gli schedule. La condizione (2) assicura che ogni transazione riceva gli stessi valori in ingresso in entrambi gli schedule (e quindi esegua la stessa computazione), mentre la condizione (3), insieme alla condizione (2), assicura che entrambi gli schedule producano lo stesso stato finale della base di dati.

**Esempio 8.6** Consideriamo lo schedule dell'Esempio 8.5:

$$S_1 = [\text{Read}_1[lr], \text{Read}_2[cc], \text{Write}_1[lr], \text{Read}_1[cc], \\ \text{Write}_2[cc], \text{Commit}_2, \text{Write}_1[cc], \text{Commit}_1]$$

e lo schedule seriale:

$$S_2 = [\text{Read}_2[cc], \text{Write}_2[cc], \text{Commit}_2, \text{Read}_1[lr], \\ \text{Write}_1[lr], \text{Read}_1[cc], \text{Write}_1[cc], \text{Commit}_1]$$

corrispondente all'esecuzione sequenziale di  $T_2$  e  $T_1$ . I due schedule non sono equivalenti rispetto alle viste in quanto in  $S_2$  la transazione  $T_1$  legge il valore di  $cc$  modificato da  $T_2$  mentre questo non è vero in  $S_1$ . Consideriamo ora lo schedule:

$$S_3 = [\text{Read}_1[lr], \text{Read}_2[cc], \text{Write}_1[lr], \text{Write}_2[cc], \\ \text{Commit}_2, \text{Read}_1[cc], \text{Write}_1[cc], \text{Commit}_1]$$

Questo schedule è equivalente rispetto alle viste allo schedule seriale  $S_2$  in quanto: (1) gli schedule  $S_2$  ed  $S_3$  contengono le stesse operazioni; (2) sia in  $S_2$  che in  $S_3$  la transazione  $T_1$  legge il valore di  $cc$  modificato da  $T_2$ ; (3) sia in  $S_2$  che in  $S_3$  l'ultima transazione che modifica  $lr$  e  $cc$  è  $T_1$ . Notiamo che l'esecuzione concorrente di  $T_1$  e  $T_2$  secondo lo schedule  $S_3$  non avrebbe generato le anomalie dello schedule  $S_1$ .  $\square$

Utilizzando la nozione di equivalenza rispetto alle viste, possiamo definire il concetto di *schedule serializzabile rispetto alle viste*.

**Definizione 8.3 (Serializzabilità rispetto alle viste)** *Uno schedule  $S$  è serializzabile rispetto alle viste se e solo se esiste uno schedule seriale  $S'$  tale che  $S \equiv_V S'$ .*  $\diamond$

Con riferimento all'Esempio 8.6, lo schedule  $S_3$  è serializzabile rispetto alle viste.

È facile dimostrare come, accettando solo schedule serializzabili rispetto alle viste, è possibile prevenire le anomalie dovute a perdita di aggiornamento, letture non ripetibili e fantasmi, illustrate nel Paragrafo 8.2.1.

Iniziamo a considerare il problema della perdita di aggiornamento. Schedule che presentano tale anomalia (vedi Esempio 8.1) sono della forma:<sup>3</sup>

$$S = [\text{Read}_1[x], \text{Read}_2[x], \text{Write}_1[x], \text{Write}_2[x], \text{Commit}_1, \text{Commit}_2]$$

dove  $x$  indica un generico dato. I due schedule seriali corrispondenti sono:

$$S_1 = [\text{Read}_1[x], \text{Write}_1[x], \text{Commit}_1, \text{Read}_2[x], \text{Write}_2[x], \text{Commit}_2]$$

che corrisponde allo schedule in cui  $T_1$  è eseguita prima di  $T_2$  e:

$$S_2 = [\text{Read}_2[x], \text{Write}_2[x], \text{Commit}_2, \text{Read}_1[x], \text{Write}_1[x], \text{Commit}_1]$$

che corrisponde allo schedule in cui  $T_2$  è eseguita prima di  $T_1$ .

$S$  non è serializzabile rispetto alle viste in quanto non è equivalente rispetto alle viste né ad  $S_1$  né ad  $S_2$ .  $S$  non è equivalente rispetto alle viste ad  $S_1$  in quanto in  $S_1$  la transazione  $T_2$  legge il dato  $x$  modificato da  $T_1$ , mentre questo non accade in  $S$ . Anche  $S$  ed  $S_2$  non sono equivalenti rispetto alle viste, in quanto non verificano la terza condizione della Definizione 8.2 (l'ultima transazione che modifica  $x$  è  $T_2$  in  $S$  e  $T_1$  in  $S_2$ ).

Analogamente, uno schedule che presenta l'anomalia dovuta a letture non ripetibili è della forma:

---

<sup>3</sup>Senza perdita di generalità consideriamo schedule corrispondenti a due sole transazioni.

$$S = [\text{Read}_1[x], \text{Read}_2[x], \text{Write}_2[x], \text{Commit}_2, \text{Read}_1[x], \text{Commit}_1]$$

i corrispondenti schedule seriali sono:

$$S_1 = [\text{Read}_1[x], \text{Read}_1[x], \text{Commit}_1, \text{Read}_2[x], \text{Write}_2[x], \text{Commit}_2]$$

ed:

$$S_2 = [\text{Read}_2[x], \text{Write}_2[x], \text{Commit}_2, \text{Read}_1[x], \text{Read}_1[x], \text{Commit}_1]$$

È facile verificare, con considerazioni analoghe alle precedenti, che entrambi non sono equivalenti rispetto alle viste ad  $S$ . Un discorso analogo vale per gli schedule che presentano il problema dei fantasmi. Non consideriamo in questa sede l'anomalia dovuta a letture sporche in quanto causata dall'abort di transazioni.

Il problema che rimane da affrontare è quanto sia oneroso verificare la serializzabilità rispetto alle viste di uno schedule. Verificare se due schedule sono equivalenti rispetto alle viste ha una complessità polinomiale nel numero di operazioni negli schedule; possono quindi essere definiti algoritmi efficienti per effettuare tale test. Viceversa, verificare se uno schedule è serializzabile rispetto alle viste è un problema NP-Completo. La nozione di equivalenza rispetto alle viste è quindi di scarsa utilità pratica perché è impossibile implementare un meccanismo efficiente che determini se uno schedule è o meno serializzabile rispetto alle viste e quindi possa essere eseguito. Per questo motivo, nella pratica, viene adottata una nozione più restrittiva di equivalenza, denominata *equivalenza rispetto ai conflitti*, che ha il vantaggio di essere caratterizzata da una complessità inferiore. Come dice il nome, l'equivalenza rispetto ai conflitti si basa sulla nozione di *conflitto*. Due operazioni appartenenti a transazioni diverse sono in conflitto se operano sullo stesso dato ed almeno una delle due è un'operazione di scrittura. Dato uno schedule  $S$ , denotiamo con  $\text{conf}(S)$  l'insieme di coppie  $(op_1, op_2)$  tali che  $op_1, op_2 \in S$ ,  $op_1$  ed  $op_2$  sono in conflitto e  $op_1$  precede  $op_2$  nello schedule. La nozione di equivalenza rispetto ai conflitti è formalizzata dalla seguente definizione.

**Definizione 8.4 (Equivalenza rispetto ai conflitti)** Due schedule  $S_1$  ed  $S_2$  sono equivalenti rispetto ai conflitti, denotato con  $S_1 \equiv_C S_2$ , se:

1. l'insieme delle operazioni in  $S_1$  coincide con l'insieme delle operazioni in  $S_2$ ;
2.  $\text{conf}(S_1) = \text{conf}(S_2)$ .

In base alla definizione precedente, due schedule sono equivalenti rispetto ai conflitti se tutte le operazioni in conflitto compaiono nello stesso ordine nei due schedule. La definizione seguente formalizza il concetto di serializzabilità rispetto ai conflitti.

**Definizione 8.5 (Serializzabilità rispetto ai conflitti)** Uno schedule  $S$  è serializzabile rispetto ai conflitti se e solo se esiste uno schedule seriale  $S'$  tale che  $S \equiv_C S'$ .  $\diamond$

Il vantaggio di utilizzare questa nozione di serializzabilità è che, a differenza della serializzabilità rispetto alle viste, verificare se uno schedule è serializzabile rispetto ai conflitti ha una complessità polinomiale nel numero di transazioni che appartengono allo schedule. Allo stesso tempo, è stato dimostrato formalmente che la classe degli schedule serializzabili rispetto ai conflitti, denotata con *SRC*, è strettamente inclusa nella classe degli schedule serializzabili rispetto alle viste (denotata con *SRV*); quindi, utilizzando la nozione di serializzabilità rispetto ai conflitti, vengono scartate delle possibili esecuzioni che sono in effetti corrette, ottenendo però una complessità accettabile. Per questo motivo, il concetto di serializzabilità rispetto ai conflitti è alla base dei più utilizzati protocolli per il controllo della concorrenza.

### 8.3 Protocolli per il controllo della concorrenza

Nel paragrafo precedente abbiamo affrontato sul piano teorico il problema della correttezza degli schedule definendo la nozione di equivalenza e serializzabilità. Vediamo ora come tali concetti vengono utilizzati nella pratica. Per assicurare la serializzabilità durante l'esecuzione di più transazioni, i DBMS utilizzano un protocollo per il controllo della concorrenza, che viene eseguito dallo *scheduler*. In letteratura sono stati proposti molti protocolli, tra cui i più noti sono i *protocolli basati su lock* ed i *protocolli basati su timestamp*. Nel seguito, dopo aver introdotto il concetto di lock, illustreremo entrambe le tipologie di protocolli.

#### 8.3.1 Il concetto di lock

Il lock è un concetto molto semplice e comunemente usato per il controllo della concorrenza. L'idea è quella di ritardare l'esecuzione di operazioni che possono causare la violazione delle proprietà ACID imponendo che le transazioni pongano dei blocchi sui dati (i *lock* appunto) prima di poter effettuare tali operazioni. Più specificatamente, una transazione può accedere ad un dato solo se può porre un lock su quel dato. La concessione o meno di un lock dipende dagli altri lock presenti sul dato. Se il lock non può essere concesso, la transazione viene messa in attesa.

Due sono i principali tipi di lock, che corrispondono alle operazioni di lettura e scrittura che una transazione può compiere sulla base di dati:

- **Lock condiviso.** Se una transazione vuole leggere un dato deve prima ottenere un lock condiviso (*shared*) su quel dato. Più transazioni possono contemporaneamente avere un lock condiviso sullo stesso dato. Per le sue caratteristiche, un lock condiviso è anche chiamato *lock in lettura* (*read lock*).
- **Lock esclusivo.** Se una transazione  $T$  vuole scrivere un dato deve prima ottenere un lock esclusivo (*exclusive*) su quel dato. Naturalmente un lock esclusivo consente anche la lettura del dato. Su un dato su cui vi è un lock esclusivo non può essere concesso alcun altro lock, finché il lock esclusivo

non viene rilasciato. Per le sue caratteristiche, un lock esclusivo è anche chiamato *lock in scrittura (write lock)*.

Nei protocolli basati su lock, ogni richiesta di lettura o scrittura deve essere preceduta da una richiesta di lock in modalità opportuna. Se la richiesta può essere accordata, in base alle regole descritte in precedenza, allora il lock viene concesso ed il dato può essere acceduto nella corrispondente modalità. Se il lock non può essere concesso, la transazione viene messa in attesa fintantoché i lock sul dato che non consentivano di soddisfare la richiesta vengono rilasciati. Assumiamo, inoltre, che le transazioni rilascino i lock entro il termine della loro esecuzione. I lock possono essere rilasciati anche in un ordine diverso rispetto a quello con cui sono stati acquisiti. Per la corretta implementazione di un protocollo basato su lock sono quindi necessarie tre primitive:  $Lc(x)$ , per richiedere un lock condiviso sul dato  $x$ ;  $Le(x)$ , per richiedere un lock esclusivo sul dato  $x$ ;  $Un(x)$ , per rilasciare un lock sul dato  $x$  (unlock). Le richieste di lock ed unlock sono inserite automaticamente dallo scheduler in modo trasparente ad utenti ed applicazioni ed il loro ordine dipende dallo specifico protocollo adottato.

Il semplice meccanismo di lock illustrato fino ad ora assicura la mutua esclusione degli accessi in lettura e scrittura allo stesso dato, ma non garantisce la serializzabilità degli schedule. Per ottenere questa garanzia, occorre porre delle ulteriori restrizioni sulle operazioni di acquisizione e rilascio dei lock, che prendono il nome di *locking a due fasi (two-phase locking)*, abbreviato in *2PL*.

### 8.3.2 Protocollo di locking a due fasi

Il protocollo di locking a due fasi richiede che ogni transazione effettui richieste di lock ed unlock in due fasi distinte: *fase di acquisizione*, in cui una transazione può ottenere dei lock, ma non può rilasciarne; *fase di rilascio*, in cui una transazione può rilasciare dei lock, ma non ottenerne dei nuovi. Inizialmente, una transazione si trova nella fase di acquisizione in cui acquisisce i lock necessari. Non appena la transazione rilascia un lock, subentra la fase di rilascio durante la quale non può essere soddisfatta alcuna richiesta di ulteriore acquisizione di lock. Durante la fase di acquisizione è anche possibile effettuare una conversione di un lock condiviso in esclusivo (ad esempio nel caso di una transazione che deve prima leggere un dato e successivamente modificarlo).

L'esempio seguente illustra come il 2PL prevenga le anomalie dovute a perdita di aggiornamento discusse nel Paragrafo 8.2.1, non consentendo l'esecuzione di schedule che le generano.

**Esempio 8.7** Consideriamo l'esecuzione concorrente di transazioni dell'Esempio 8.1 e vediamo perché lo schedule corrispondente non è consentito dal 2PL. Per semplicità assumiamo che quando una transazione deve sia leggere sia scrivere un dato richieda subito un lock esclusivo su di esso:

$T_1$	$T_2$
<code>Le(<math>lr</math>) <math>\Rightarrow</math> concesso</code>	
<code>Read <math>lr</math></code>	
$lr := lr - 150$	
	<code>Le(<math>cc</math>) <math>\Rightarrow</math> concesso</code>
	<code>Read <math>cc</math></code>
<code>Write <math>lr</math></code>	
	$cc := cc + 5'000$
<code>Le(<math>cc</math>) <math>\Rightarrow</math> attesa</code>	
	<code>Write <math>cc</math></code>
	<code>Un(<math>cc</math>)</code>
	<code>Commit</code>
<code>Le(<math>cc</math>) <math>\Rightarrow</math> concesso</code>	
<code>Read <math>cc</math></code>	
$cc := cc + 150$	
<code>Write <math>cc</math></code>	
<code>Commit</code>	

Nello schedule, ogni operazione di lettura/scrittura è preceduta dalla corrispondente richiesta di lock. La richiesta di lock esclusivo su  $cc$  da parte di  $T_1$  non viene concessa perché  $T_2$  detiene già un lock sul conto corrente. L'operazione di lettura di  $T_1$ , così come tutte le operazioni successive, sono ritardate fino a che  $T_2$  rilascia il lock, e questo può avvenire solo dopo l'operazione di scrittura sul conto corrente. In questo modo il problema della perdita di aggiornamento non si verifica, in quanto  $T_1$  legge il saldo del conto corrente già modificato da  $T_2$ .  $\square$

È stato formalmente dimostrato che il protocollo di locking a due fasi assicura la serializzabilità rispetto ai conflitti degli schedule generati. In particolare, la classe degli schedule ammissibili dal 2PL corrisponde ad un sottoinsieme delle classi degli schedule serializzabili rispetto ai conflitti. Il 2PL previene quindi anche le anomalie dovute a fantasmi e letture non ripetibili. Tra le anomalie illustrate nel Paragrafo 8.2.1 rimane ancora il problema dell'anomalia dovuta a letture sporche, causata dalla lettura di risultati intermedi di una transazione che successivamente effettua l'abort.

**Esempio 8.8** Riprendiamo lo schedule dell'Esempio 8.2 che presentava l'anomalia dovuta a letture sporche ed eseguiamolo secondo il 2PL:

$T_1$	$T_2$
<code>Le(<math>lr</math>) <math>\Rightarrow</math> concesso</code>	
<code>Read <math>lr</math></code>	
$lr := lr - 150$	
<code>Write <math>lr</math></code>	

```

Le( $cc$ ) $\Rightarrow$  concessa
Read  $cc$ 
 $cc := cc + 150$ 
Write  $cc$ 
Un( $cc$ )
Le( $cc$ ) $\Rightarrow$  concessa
Read  $cc$ 
 $cc := cc + 5\cdot000$ 
Write  $cc$ 
Abort
Un( $cc$ )
Commit

```

Tale schedule non viola le regole del 2PL, anche se comporta la memorizzazione di dati non corretti nella base di dati.  $\square$

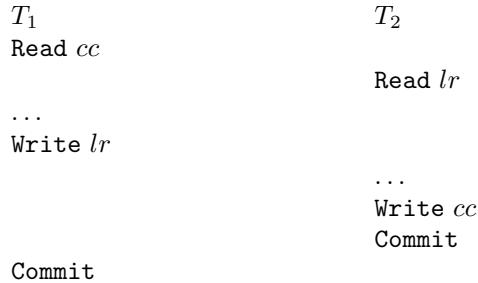
Per evitare anche l'anomalia da letture sporche, viene introdotta una variante del 2PL, chiamato *locking a due fasi stretto (strict 2PL)*, che è la versione utilizzata dalla maggioranza dei DBMS commerciali. Tale variante aggiunge al 2PL il seguente vincolo:

*Tutti i lock esclusivi di una transazione sono rilasciati solo al termine (abort o commit) della transazione.*

Un'ulteriore variante, chiamata *locking a due fasi forte (strong 2PL)*, richiede che la condizione precedente valga anche per i lock condivisi. Se torniamo allo schedule con anomalia da letture sporche dell'Esempio 8.8, vediamo che tale schedule non è permesso dal locking a due fasi stretto, in quanto l'unlock del conto corrente da parte di  $T_1$  avrebbe potuto essere eseguito solo al termine della sua esecuzione. In questo modo, la richiesta di lock sul conto corrente da parte di  $T_2$  avrebbe causato un'attesa di  $T_2$  fino all'abort di  $T_1$  ed il conseguente ripristino del valore del conto corrente al valore che aveva prima che la transazione  $T_1$  fosse eseguita.

Il 2PL stretto evita quindi tutti i tipi di anomalie discussi nel Paragrafo 8.2.1. Il problema di tale protocollo, così come del 2PL, è che può portare a situazioni di *deadlock*. Un sistema che adotta il 2PL è in deadlock (cioè in una situazione di stallo) quando esistono un insieme di transazioni che sono reciprocamente bloccate a causa dei lock che detengono e che devono acquisire. L'esempio seguente chiarisce il problema.

**Esempio 8.9** Siano  $T_1$  e  $T_2$  due transazioni che agiscono su  $cc$  e  $lr$  come segue:



Se utilizziamo il protocollo di locking a due fasi,  $T_1$  pone un lock condiviso su  $cc$ , mentre  $T_2$  pone un lock condiviso su  $lr$ . Dopo aver effettuato l'operazione di lettura,  $T_1$  deve acquisire un lock esclusivo su  $lr$  prima di effettuare la scrittura, ma non lo può fare a causa del lock che detiene  $T_2$ . Per le regole del 2PL,  $T_2$  non può rilasciare il lock su  $lr$  in quanto deve ancora scrivere  $cc$  ed acquisire il relativo lock, che non può acquisire finantoché  $T_1$  non rilascia il lock che ha su  $cc$ . Vi è quindi una situazione di stallo in cui nessuna delle due transazioni può procedere nella computazione.  $\square$

Vedremo nel Paragrafo 8.4 quali tecniche i DBMS adottano per gestire le situazioni di deadlock.

Infine, un ulteriore aspetto riguarda la *granularità* dei lock. Infatti, finora abbiamo ragionato in termini di un modello astratto, in cui i lock vengono posti su dati generici. Gli “oggetti” di una base di dati sono relazioni e tuple, ed è su questi che debbono essere posti i lock. Chiaramente, la soluzione più semplice è porre un lock su un’intera relazione quando una transazione richiede di accedere ad alcune delle sue tuple. Lo svantaggio di questa soluzione è che limita la concorrenza creando facilmente situazioni di conflitto. Se una transazione effettua operazioni confinate ad un insieme ben preciso di tuple, potrebbe essere opportuno effettuare il lock ad una granularità più fine, cioè a livello di tupla. Il potenziale svantaggio di questa soluzione è che può portare a richiedere un eccessivo numero di lock, degradando le prestazioni del gestore della concorrenza. SQL consente di effettuare scelte sulla granularità dei lock, così come altre scelte simili, sulla base delle caratteristiche delle transazioni (vedi Paragrafo 8.7.1).

### 8.3.3 Protocolli basati su timestamp

I protocolli basati su timestamp costituiscono un’alternativa ai protocolli basati su lock per il controllo della concorrenza. Il loro utilizzo in DBMS commerciali è limitato ma possono essere utili in contesti distribuiti. Tali protocolli assegnano un timestamp univoco ad ogni transazione  $T_i$ , denotato con  $TS(T_i)$ . Il timestamp è assegnato dal sistema, prima che la transazione  $T_i$  inizi la propria esecuzione. L’assegnazione del timestamp è monotona, cioè se ad una transazione  $T_i$  è stata assegnato un timestamp  $TS(T_i)$  e viene attivata una nuova transazione  $T_j$ , allora  $TS(T_i) < TS(T_j)$ . Il valore del timestamp può essere o il valore dell’orologio

di sistema nel momento in cui la transazione viene sottoposta al sistema, oppure può essere generato da un contatore logico che viene incrementato a seguito dell'assegnazione di ogni nuovo timestamp.

L'ordine tra operazioni in conflitto viene effettuato in base ai timestamp delle corrispondenti transazioni; tale schema è chiamato *schema basato sull'ordinamento dei timestamp – timestamp ordering*. In particolare, date due operazioni in conflitto  $op_1$  e  $op_2$  sul dato  $x$ ,  $op_1$  può essere eseguita prima di  $op_2$  solo se la transazione che contiene  $op_1$  ha timestamp minore di quella che contiene  $op_2$ . Per soddisfare tale vincolo, vengono associati due timestamp ad ogni dato  $x$ :

- **W-timestamp( $x$ )**. Denota il timestamp più alto fra quelli di tutte le transazioni che hanno eseguito l'operazione **Write**[ $x$ ].
- **R-timestamp( $x$ )**. Denota il timestamp più alto fra quelli di tutte le transazioni che hanno eseguito l'operazione **Read**[ $x$ ].

Lo scheduler opera come segue:

- Supponiamo che una transazione  $T_i$  richieda di eseguire l'operazione **Read**[ $x$ ]:
  - Se  $TS(T_i) < W\text{-timestamp}(x)$ , l'operazione di lettura è rifiutata e viene effettuato il rollback di  $T_i$ .
  - In caso contrario, la lettura viene eseguita e **R-timestamp( $x$ )** assume come valore il massimo tra **R-timestamp( $x$ )** e  $TS(T_i)$ .
- Supponiamo che la transazione  $T_i$  richieda di eseguire l'operazione **Write**[ $x$ ]:
  - Se  $TS(T_i) < R\text{-timestamp}(x)$  o  $TS(T_i) < W\text{-timestamp}(x)$ , l'operazione di scrittura viene rifiutata e viene eseguito il rollback di  $T_i$ .
  - Altrimenti, l'operazione di scrittura viene eseguita e **W-timestamp( $x$ )** è posto uguale a  $TS(T_i)$ .

Queste due regole evitano che una transazione possa leggere o scrivere un dato modificato da una transazione con timestamp maggiore o modificare un dato già letto da una transazione con timestamp superiore. Quando una transazione viene abortita come conseguenza del protocollo di controllo della concorrenza, le viene assegnato un nuovo timestamp e riavviata.

**Esempio 8.10** Siano  $T_1$  e  $T_2$  due transazioni.  $T_1$  stampa a video la somma dei saldi del libretto di risparmio e del conto corrente, mentre  $T_2$  trasferisce 50 euro dal libretto di risparmio al conto corrente. Consideriamo il seguente schedule:

$T_1$	$T_2$
Read $lr$	
	Read $lr$
	$lr := lr - 50$

```

Write lr
Read cc
cc := cc + 50
Write cc
Commit

Read cc
Display cc + lr
Commit

```

e supponiamo di eseguirlo con il protocollo basato sull'ordinamento dei timestamp. Supponiamo che:  $TS(T_1) = 100$ ,  $TS(T_2) = 102$ ,  $R\text{-timestamp}(cc) = 80$ ,  $W\text{-timestamp}(cc) = 80$ ,  $R\text{-timestamp}(lr) = 90$ ,  $W\text{-timestamp}(lr) = 90$ . La prima operazione dello schedule è  $\text{Read}_1[lr]$ :

- viene eseguito il test:  $TS(T_1) < W\text{-timestamp}(lr)$ . Poiché il test dà esito negativo, l'operazione viene eseguita e  $R\text{-timestamp}(lr)$  viene posto uguale a 100 (cioè al valore di  $TS(T_1)$ ).

In base allo schedule, devono adesso essere eseguite tutte le operazioni della transazione  $T_2$ . Per verificare se queste operazioni possono essere eseguite, vengono effettuati i seguenti test:

- $\text{Read}_2[lr]$ :  $TS(T_2) < W\text{-timestamp}(lr) = 90$ . Il test dà esito negativo, l'operazione  $\text{Read}_2[lr]$  viene eseguita e  $R\text{-timestamp}(lr)$  viene posto uguale a 102 (cioè al valore di  $TS(T_2)$ ).
- $\text{Write}_2[lr]$ :  $TS(T_2) < R\text{-timestamp}(lr) = 102$  e  $TS(T_2) < W\text{-timestamp}(lr) = 90$ . Il test dà esito negativo, l'operazione  $\text{Write}_2[lr]$  viene eseguita e  $W\text{-timestamp}(lr)$  viene posto uguale a 102.
- $\text{Read}_2[cc]$ :  $TS(T_2) < W\text{-timestamp}(cc) = 80$ . Il test dà esito negativo, l'operazione  $\text{Read}_2[cc]$  viene eseguita e  $R\text{-timestamp}(cc)$  viene posto uguale a 102.
- $\text{Write}_2[cc]$ :  $TS(T_2) < R\text{-timestamp}(cc) = 102$ .  $TS(T_2) < W\text{-timestamp}(cc) = 80$ . Il test dà esito negativo, l'operazione  $\text{Write}_2[cc]$  viene eseguita e  $W\text{-timestamp}(cc)$  viene posto uguale a 102.

Al termine dell'esecuzione di  $T_2$ , il valore dei timestamp associati sia a  $cc$  sia a  $lr$  è uguale a 102.

A questo punto, viene schedulata l'operazione  $\text{Read}_1[cc]$ . Eseguiamo il test:  $TS(T_1) < W\text{-timestamp}(cc) = 102$ . Poiché la risposta è positiva, l'operazione  $\text{Read}_1[cc]$  viene rifiutata e viene eseguito il rollback della transazione  $T_1$ . Lo schedule presentato non è uno schedule ammissibile per il protocollo basato sull'ordinamento dei timestamp. Notiamo infatti, come lo schedule porterebbe  $T_1$  a stampare a video dati non corretti in quanto  $T_1$  legge il valore di  $lr$  prima della modifica di  $T_2$  ed il valore di  $cc$  dopo. Il risultato è che la somma dei saldi stampata da  $T_1$  differirebbe di 50 euro dal valore corretto.  $\square$

Il protocollo basato sull'ordinamento dei timestamp assicura la serializzabilità rispetto ai conflitti. Questo dipende dal fatto che le operazioni in conflitto sono eseguite in ordine di timestamp. Il protocollo assicura inoltre l'assenza di deadlock, dato che nessuna transazione viene posta in attesa di un lock. Uno degli svantaggi è che possono essere prodotti molti rollback a cascata (questo avviene quando il rollback di una transazione provoca il rollback di un'altra transazione e così via). Per evitare questo problema, il protocollo deve essere modificato per assicurare che una transazione possa leggere solo valori modificati da transazioni che hanno terminato la loro esecuzione con successo. Ciò può essere ottenuto associando un bit di commit ad ogni transazione  $T_i$  ed un riferimento a quel bit di commit da ogni dato scritto da  $T_i$ .

Un'altra variante del protocollo basato sull'ordinamento dei timestamp, volta a minimizzare l'abort di transazioni, è basata su un principio conosciuto come *Thomas' Write Rule (TWR)*. In base a questo principio, il comportamento dello scheduler viene modificato come segue. Supponiamo che una transazione  $T_i$  richieda di eseguire l'operazione  $\text{Write}_i[x]$ . Se  $TS(T_i) < \text{W-timestamp}(x)$ , l'operazione di scrittura è semplicemente ignorata senza abortire la transazione. In tutti gli altri casi, lo scheduler agisce secondo le regole illustrate in precedenza. L'intuizione alla base della TWR è che l'operazione di scrittura può essere semplicemente ignorata senza portare a problemi di consistenza in quanto è una scrittura “senza effetti”. Infatti, per il principio di ordinamento dei timestamp,  $x$  dovrebbe essere scritto dalla transazione con timestamp  $\text{W-timestamp}(x)$  dopo la scrittura effettuata da  $T_i$ , i cui effetti sono quindi sovrascritti. Inoltre, la modifica effettuata da  $T_i$  anche se fosse stata effettuata nel giusto ordine rispetto ai timestamp, non sarebbe stata letta da alcun'altra transazione (altrimenti la condizione  $TS(T_i) < \text{R-timestamp}(x)$  avrebbe causato l'abort della transazione). Quindi, può essere semplicemente ignorata non causando effetti collaterali.

#### 8.4 Gestione dei deadlock

Come già accennato nel Paragrafo 8.3.2, il 2PL può portare un insieme di transazioni in uno stato di deadlock. Più precisamente, un deadlock occorre quando esiste un insieme di transazioni  $\{T_0, T_1, \dots, T_n\}$  tale che  $T_0$  attende di ottenere un lock su un dato che è bloccato da  $T_1$ ;  $T_1$  attende di ottenere un lock su un dato che è bloccato da  $T_2$ ;  $\dots$ ;  $T_{n-1}$  attende di ottenere un lock su un dato che è bloccato da  $T_n$  e  $T_n$  attende di ottenere un lock su un dato che è bloccato da  $T_0$ . In ogni DBMS che implementa il 2PL, serve quindi un meccanismo per gestire i possibili deadlock. Vi sono tre metodi principali per risolvere il deadlock. Il metodo più semplice, utilizzato dalla maggioranza dei DBMS commerciali, è quello del *timeout*, in base a cui viene posto un limite al tempo massimo di attesa di un lock da parte di una transazione. Trascorso tale tempo, la transazione viene abortita e rieseguita, indipendentemente dal fatto che si sia verificato o meno un deadlock. Chiaramente, lo svantaggio di questo metodo è che una attesa oltre il timeout non sempre è dovuta ad un deadlock e quindi possono essere abortite delle transazioni

che avrebbero potuto terminare con successo. Il punto più critico di tale metodo è quale valore assegnare al timeout. Valori troppo piccoli possono portare ad abortire inutilmente molte transazioni; viceversa, valori troppo grandi portano a ritardi nella risoluzione dei deadlock con conseguente spreco nell'allocazione delle risorse e penalizzazioni rispetto alle prestazioni.

L'altra classe di metodi per la gestione dei deadlock è quella dei protocolli per la *prevenzione* dei deadlock, che assicurano che il sistema non entri mai in uno stato di deadlock. Alternativamente, è possibile lasciare che il sistema entri in uno stato di deadlock e poi eliminare la situazione di stallo tramite un meccanismo di *rilevazione e risoluzione*. Tale soluzione è utilizzata da alcuni DBMS commerciali. Nel seguito, illustreremo brevemente entrambi gli approcci.

#### 8.4.1 Prevenzione dei deadlock

Vi sono diversi metodi che possono essere usati per prevenire il deadlock. L'approccio più semplice richiede che ogni transazione ponga un lock su tutti i dati, prima di iniziare l'esecuzione (questa variante del 2PL è chiamata *2PL conservativo*). Questo schema presenta però due svantaggi. Innanzitutto, l'utilizzazione dei dati può essere molto bassa; i dati possono essere infatti bloccati e non utilizzati per un lungo periodo di tempo. Inoltre, è possibile che accada una situazione detta di *starvation*; questo significa che una transazione che necessita di accedere a dati condivisi può dover attendere indefinitamente che tutti i dati di cui ha bisogno siano rilasciati da altre transazioni.

Un ulteriore approccio per prevenire il deadlock consiste nell'usare un meccanismo basato sui timestamp assegnati alle transazioni. I timestamp sono usati per decidere se una transazione che richiede un lock su un dato su cui vi è già un lock incompatibile debba essere posta in attesa o abortita. A tal fine, due sono gli schemi principali adottabili:

- **Schema wait-die.** Quando una transazione  $T_i$  richiede un lock su un dato, correntemente bloccato da  $T_j$ ,  $T_i$  è autorizzata ad aspettare (wait) solo se ha un timestamp minore di quello di  $T_j$  (cioè  $T_i$  è più anziana di  $T_j$ ). Altrimenti,  $T_i$  viene abortita (die).
- **Schema wound-wait.** Quando una transazione  $T_i$  richiede un lock su un dato, correntemente bloccato da  $T_j$ ,  $T_j$  viene abortita se il timestamp di  $T_i$  è minore del suo ( $T_i$  “wounds”  $T_j$ ), altrimenti  $T_i$  è autorizzata ad aspettare (wait).

Entrambi gli schemi potrebbero portare alla starvation. Un metodo per evitare questo è mantenere il vecchio timestamp quando una transazione viene rieseguita.

Schemi alternativi possono essere quello di abortire sempre una transazione se la sua richiesta di lock non può essere esaudita, indipendentemente dal timestamp della transazione che detiene il lock, oppure abortire la transazione che detiene il lock, se quest'ultima sta a sua volta aspettando che una terza transazione rilasci un altro lock per continuare la sua esecuzione.

**Esempio 8.11** Consideriamo tre transazioni  $T_1$ ,  $T_2$  e  $T_3$ , aventi rispettivamente timestamp 7, 12 e 15. In base allo schema wait-die, se  $T_1$  richiede un dato bloccato da  $T_2$ ,  $T_1$  è messa in attesa. Se  $T_3$  richiede un dato bloccato da  $T_2$ , viene eseguito il rollback di  $T_3$ . In base allo schema wound-wait, se  $T_1$  richiede un dato bloccato da  $T_2$ , viene eseguito il rollback di  $T_2$ . Se  $T_3$  richiede un dato bloccato da  $T_2$ ,  $T_3$  è messa in attesa.  $\square$

Il problema maggiore che sorge con entrambi gli schemi è la generazione di rollback non necessari. Per questo motivo, sebbene i protocolli per la prevenzione dei deadlock siano concettualmente molto semplici, essi non vengono effettivamente usati nei DBMS commerciali.

#### 8.4.2 Rilevazione dei deadlock

Se il sistema non utilizza alcun protocollo che assicura l'assenza dei deadlock, allora deve essere usato un meccanismo di rilevazione e risoluzione del deadlock che periodicamente esamina lo stato delle transazioni in esecuzione per determinare se vi è un deadlock. La presenza di un deadlock può essere rilevata mediante un grafo, detto *grafo delle attese* (*wait-for graph*), in cui esiste un nodo per ogni transazione attiva nel sistema, ed esiste un arco dal nodo rappresentante la transazione  $T_i$  al nodo rappresentante la transazione  $T_j$ , se  $T_i$  è in attesa che  $T_j$  rilasci un lock di cui necessita per proseguire la propria esecuzione. Rilevare un deadlock equivale quindi a determinare un ciclo nel grafo delle attese.

**Esempio 8.12** Consideriamo quattro transazioni  $T_1$ ,  $T_2$ ,  $T_3$  e  $T_4$ . Supponiamo che  $T_1$  aspetti il rilascio di un lock da parte di  $T_2$  e  $T_3$ ,  $T_3$  aspetti il rilascio di un lock da parte di  $T_2$ , mentre  $T_2$  aspetti il rilascio di un lock da parte di  $T_4$ . Il corrispondente grafo delle attese è rappresentato nella Figura 8.2(a). Queste attese non sono dovute ad un deadlock in quanto il grafo non contiene cicli. Viceversa, supponiamo adesso che la transazione  $T_4$  richieda un lock su dati bloccati da  $T_3$ . Il grafo risultante è presentato in Figura 8.2(b). Poiché il grafo presenta un ciclo che coinvolge le transazioni  $T_2$ ,  $T_3$  e  $T_4$ , rileviamo una situazione di deadlock.  $\square$

Se l'algoritmo di rilevazione determina la presenza di un deadlock, il sistema deve risolvere tale situazione abortendo una o più transazioni per spezzare il ciclo ed eliminare il deadlock. Uno dei fattori più importanti è quale criterio adottare per la *selezione della vittima*, per decidere cioè quali transazioni conviene abortire. Possibili criteri sono: la selezione casuale, la selezione della transazione bloccata più di recente oppure di quella con timestamp maggiore (cioè la più giovane), la selezione della transazione il cui abort elimina il maggior numero di cicli o il maggior numero di archi nel grafo delle attese, oppure di quella che ha utilizzato il minor numero di risorse (dati, tempo di CPU, ecc.). Chiaramente, il pericolo, qualsiasi sia la politica adottata, è che la stessa transazione sia selezionata sempre come vittima, non riuscendo mai a completare la sua esecuzione. Deve quindi essere assicurato che una transazione possa essere selezionata come vittima solo

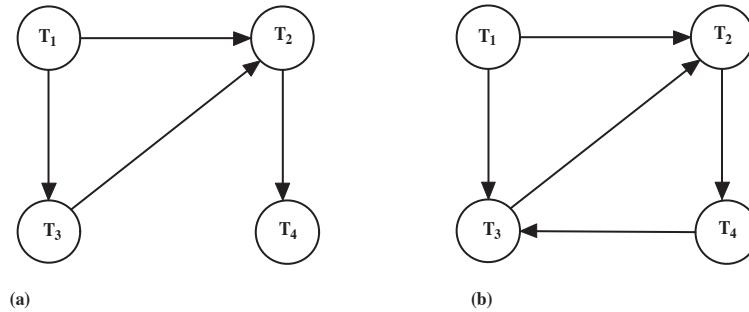


Figura 8.2: Grafi delle attese: (a) grafo aciclico; (b) grafo ciclico

un numero limitato di volte. Una soluzione è quella di tenere in considerazione, nella selezione della vittima, anche il numero di rollback che la transazione ha già effettuato.

### 8.5 Tecniche di rispristino

Una delle componenti fondamentali di un DBMS è il *gestore del ripristino*, che è responsabile della gestione di malfunzionamenti hardware e software e del ripristino della base di dati allo stato (consistente) che esisteva prima che si verificasse il malfunzionamento. Dal punto di vista della gestione delle transazioni, il gestore del ripristino assicura le proprietà di atomicità e persistenza.

Iniziamo a precisare quali sono i principali tipi di malfunzionamento che possono accadere in un DBMS:

- **Malfunzionamenti del disco.** Le informazioni residenti su disco vengono perse. I tipi più comuni di tali malfunzionamenti sono la rottura della testina o gli errori che accadono durante le operazioni di trasferimento dei dati.
- **Malfunzionamenti di alimentazione.** Le informazioni memorizzate in memoria centrale e nei registri vengono perse.
- **Errori software.** I tipi più comuni di tali malfunzionamenti sono errori logici del programma in esecuzione (il programma non può continuare la propria esecuzione a causa del verificarsi di certe condizioni interne, come ad esempio dati di ingresso scorretti, dati sconosciuti, overflow, ecc.) ed errori di sistema (il sistema è entrato in uno stato indesiderabile ed il programma non può continuare la propria normale esecuzione).

È inoltre utile classificare i vari supporti di memorizzazione rispetto a come le informazioni in essi memorizzate sono influenzate dai vari tipi di malfunzionamento:

- **Memoria volatile.** Le informazioni residenti nella memoria volatile non vengono mantenute in caso di cadute (crash) del sistema. La memoria principale e la memoria cache sono esempi di memoria volatile.
- **Memoria non volatile.** Le informazioni residenti nella memoria non volatile solitamente sopravvivono a cadute di sistema. Esempi di tali tipi di memorie sono dischi e nastri magnetici. Entrambi sono però soggetti a malfunzionamenti (ad esempio la rottura della testina), che possono generare perdita di informazioni.
- **Memoria stabile.** Le informazioni residenti nella memoria stabile non possono essere perse. Chiaramente, la memoria stabile esiste solo a livello teorico. Per poter operare in modo corretto, il gestore del ripristino deve però poter disporre di memoria stabile. Per realizzare un'approssimazione di tale tipo di supporto, vengono replicati i dati in diversi supporti di memorizzazione non volatili con probabilità di fallimento indipendenti. Questo serve anche a garantire la proprietà di persistenza del risultato delle transazioni.

### 8.5.1 Il problema del ripristino

Il seguente esempio introduce alcune problematiche legate alle procedure di ripristino.

**Esempio 8.13** Consideriamo una transazione  $T$  che trasferisce 10'000 euro dal conto  $A$  al conto  $B$ , i cui saldi ammontano, rispettivamente, a 50'000 e 150'000 euro. Supponiamo che, durante l'esecuzione di  $T$ , dopo la modifica di  $A$  e prima della modifica di  $B$ , si verifichi una caduta del sistema. I contenuti della memoria volatile vengono persi, quindi non è possibile determinare lo stato corrente della transazione. A questo punto:

- Se rieseguiamo  $T$  arriviamo ad uno stato inconsistente in cui il saldo di  $A$  assume il valore 30'000 e quello di  $B$  il valore 160'000.
- Se non rieseguiamo  $T$ , arriviamo sempre ad uno stato inconsistente in cui il saldo del conto  $A$  è 40'000 e quello di  $B$  150'000.  $\square$

Il problema illustrato nell'esempio precedente è causato dal fatto di avere modificato la base di dati prima di avere la certezza che la transazione terminasse con successo. Uno dei meccanismi più utilizzati per superare questi problemi è il cosiddetto *ripristino con log*. Durante l'esecuzione di una transazione tutte le operazioni di scrittura sono registrate in un particolare file, detto *file di log*, che risiede in memoria stabile. Le informazioni nel file di log vengono utilizzate dal gestore del ripristino nel caso si verifichino malfunzionamenti prima che la transazione abbia completato la propria esecuzione. Concettualmente, il file di log può essere pensato come un file sequenziale. Nell'implementazione effettiva, possono essere usati molteplici file fisici per facilitare il lavoro di archiviazione e ricerca dei record. Ad ogni record viene attribuito un identificatore unico.

Un’importante regola per la correttezza delle operazioni di ripristino è che i record di log relativi ad operazioni di modifica della base di dati vengano scritti su memoria stabile prima che i dati siano effettivamente modificati in memoria non volatile. Questa regola è nota come regola *WAL*, dall’inglese *Write Ahead Log*.

Inoltre, al fine di garantire la persistenza dei risultati di una transazione, viene eseguito, ad intervalli regolari, un *checkpoint*. Durante un checkpoint, il contenuto della memoria di massa viene allineato al contenuto del buffer. Dopo aver determinato e registrato nel file di log l’insieme delle transazioni correntemente attive (che non hanno cioè ancora effettuato l’abort o il commit), vengono trasferite su memoria di massa tutte le pagine del buffer relative a transazioni che hanno già effettuato l’abort o il commit. Inoltre, al fine di aumentare la robustezza a guasti e malfunzionamenti, il DBMS effettua periodicamente, ma meno frequentemente delle operazioni di checkpoint (ad esempio, ogni notte oppure ogni domenica), un’operazione di *dump*, che consiste in un backup completo della base di dati su supporto di memorizzazione di massa.

Per implementare correttamente le procedure di ripristino, sono necessarie due primitive: la primitiva **redo**, che esegue nuovamente le azioni effettuate da una transazione, e la primitiva **undo**, che annulla gli effetti delle azioni compiute da una transazione. Tali primitive devono essere *idempotenti*, cioè, più esecuzioni in sequenza di tali primitive devono essere equivalenti ad un’esecuzione singola. Questo assicura un comportamento corretto anche in presenza di malfunzionamenti durante il ripristino.

Possono essere definiti diversi meccanismi di ripristino, che si differenziano rispetto al modo in cui sincronizzano le operazioni di scrittura nel file di log e nella base di dati, e nell’utilizzo di entrambe le primitive **undo** e **redo** o solo di una. Nel seguito illustreremo i principali.

### 8.5.2 *Protocolli con modifiche differite*

Durante l’esecuzione di una transazione, tutte le operazioni di scrittura vengono differite fino a che la transazione non entra nello stato *partially committed*, cioè nello stato raggiunto dopo l’esecuzione con successo della sua ultima istruzione. Tutte le modifiche sono però registrate nel file di log. L’esecuzione di una transazione  $T_i$  procede come segue. Prima che  $T_i$  cominci la propria esecuzione viene scritto nel file di log il record  $\langle T_i, \text{start} \rangle$ , che indica l’inizio della transazione. Durante l’esecuzione, qualsiasi operazione di scrittura produce un nuovo record nel file di log, contenente l’identificatore della transazione ed il nuovo valore del dato. Quando  $T_i$  entra nello stato partially committed, viene scritto nel file di log il record  $\langle T_i, \text{commit} \rangle$  e vengono effettuate le scritture differite. Dato che un malfunzionamento può verificarsi mentre le modifiche vengono eseguite, tutti i record del file di log sono scritti su memoria stabile prima di effettuare le modifiche.

A seguito di un malfunzionamento, il gestore del ripristino consulta il file di log per determinare quali transazioni debbano essere rieseguite. Una transazione  $T_i$  deve essere rieseguita se il file di log contiene entrambi i record  $\langle T_i, \text{start} \rangle$

e  $< T_i, \text{commit} >$ . Questo è effettuato tramite la primitiva **redo** che assegna a tutti i dati aggiornati dalla transazione i nuovi valori, contenuti nel file di log. Non è invece necessario l'utilizzo della primitiva **undo** in quanto le modifiche sono differite fino al commit della transazione.

**Esempio 8.14** Consideriamo le seguenti transazioni:

$T_1$	$T_2$
<b>Read</b> $lr$	<b>Read</b> $cc2$
$lr := lr - 1'500$	$cc2 := cc2 - 20'000$
<b>Write</b> $lr$	<b>Write</b> $cc2$
<b>Read</b> $cc1$	<b>Commit</b>
$cc1 := cc1 + 1'500$	
<b>Write</b> $cc1$	
<b>Commit</b>	

Supponiamo che venga eseguita prima la transazione  $T_1$  e poi la transazione  $T_2$ . Supponiamo che prima dell'esecuzione delle transazioni si abbia:

$$lr = 50'000 \quad cc1 = 60'000 \quad cc2 = 80'000.$$

Il contenuto del file di log al termine dell'esecuzione delle due transazioni è il seguente:

```

<  $T_1, \text{start}$  >
<  $T_1, lr, 48'500$  >
<  $T_1, cc1, 61'500$  >
<  $T_1, \text{commit}$  >
<  $T_2, \text{start}$  >
<  $T_2, cc2, 60'000$  >
<  $T_2, \text{commit}$  >

```

Supponiamo adesso che si verifichi un crash del sistema subito dopo che sia stato scritto in memoria stabile il record di log relativo all'operazione **Write**<sub>1</sub>[ $cc1$ ]. Lo stato del file di log al momento del crash è il seguente:

```

<  $T_1, \text{start}$  >
<  $T_1, lr, 48'500$  >
<  $T_1, cc1, 61'500$  >

```

Quando il sistema inizia il ripristino, esamina il file di log e poiché non esiste alcun record di commit per  $T_1$ , non esegue alcuna azione e lo stato dei dati non viene quindi modificato. Supponiamo adesso che si abbia un crash subito dopo aver scritto in memoria stabile il record di log relativo al commit di  $T_2$ . Il file di log in questo caso è quello presentato all'inizio dell'esempio. Quando il sistema inizia il ripristino, esamina il file di log e determina che esistono due transazioni per cui sono presenti sia il record di start sia quello di commit. Esegue quindi le operazioni **redo**( $T_1$ ) e **redo**( $T_2$ ). Lo stato della base di dati dopo il ripristino diventa il seguente:

$$lr = 48\cdot500 \quad cc1 = 61\cdot500 \quad cc2 = 60\cdot000.$$

□

Lo svantaggio dei protocolli con modifiche differite, che non prevedono operazioni di **undo**, è che la loro implementazione causa un notevole sovraccarico del gestore del ripristino. Per differire le modifiche, infatti, bisogna impedire al gestore del buffer di scaricare su memoria di massa i blocchi del buffer contenenti i dati modificati da transazioni che non hanno ancora effettuato il **commit**. Per tali motivi, i DBMS commerciali adottano un approccio alternativo, descritto nel paragrafo successivo.

### 8.5.3 Protocolli con modifiche non differite

Una tecnica alternativa, che consente una gestione più efficiente delle risorse, prevede che gli aggiornamenti possano essere applicati alla base di dati anche immediatamente, una volta registrati nel file di log, ed in generale in qualunque momento rispetto al **commit** della transazione. Questo consente al gestore del ripristino di ottimizzare le operazioni di scrittura, poiché viene rimosso il vincolo sui trasferimenti di blocchi dal buffer a memoria di massa. Tutte le operazioni di scrittura su un certo dato  $x$  devono essere precedute dall'inserimento di un nuovo record nel file di log che, a differenza dello schema precedente, contiene oltre all'identificatore della transazione, sia il vecchio sia il nuovo valore del dato. Come nel caso precedente, poiché le informazioni nel file di log sono usate per ricostruire uno stato consistente della base di dati, non è consentito l'aggiornamento effettivo della base di dati prima che il corrispondente record di log sia scritto in memoria stabile. Il meccanismo di ripristino utilizza sia la primitiva **redo** sia la primitiva **undo**.

A seguito di un malfunzionamento, il gestore del ripristino consulta il file di log per determinare quali transazioni debbano essere rieseguite e di quali debbano essere annullati gli effetti. La procedura è la seguente:

- Gli effetti della transazione  $T_i$  sono annullati se il file di log contiene il record  $\langle T_i, \text{start} \rangle$  ma non contiene il record  $\langle T_i, \text{commit} \rangle$ .
- La transazione  $T_i$  è rieseguita se il file di log contiene sia il record  $\langle T_i, \text{start} \rangle$  sia il record  $\langle T_i, \text{commit} \rangle$ .

**Esempio 8.15** Consideriamo le transazioni introdotte nell'Esempio 8.14 e supponiamo che venga eseguita prima  $T_1$  e poi  $T_2$ . Supponiamo che prima dell'esecuzione delle transazioni si abbia:

$$lr = 50\cdot000 \quad cc1 = 60\cdot000 \quad cc2 = 80\cdot000.$$

Il contenuto del file di log al termine dell'esecuzione delle due transazioni è il seguente:

$\langle T_1, \text{start} \rangle$   
 $\langle T_1, lr, 50\cdot000, 48\cdot500 \rangle$

---

```
< T1,cc1,60'000,61'500>
< T1,commit>
< T2,start>
< T2,cc2,80'000,60'000>
< T2,commit>
```

Supponiamo che si verifichi un crash subito dopo aver scritto in memoria stabile il record di log per l'operazione `Write1[cc1]`. Lo stato del file di log al momento del crash è il seguente:

```
< T1,start>
< T1, lr,50'000,48'500>
< T1,cc1,60'000,61'500>
```

Quando il sistema inizia il ripristino, esamina il file di log e, poiché non esiste alcun record di commit per  $T_1$ , esegue  $\text{undo}(T_1)$ . Il nuovo stato dei dati dopo l'esecuzione dell'operazione è il seguente:

$$lr = 50'000 \quad cc1 = 60'000 \quad cc2 = 80'000.$$

Supponiamo adesso che si abbia un crash subito dopo aver scritto in memoria stabile il record di log per l'operazione `Write2[cc2]`. Lo stato del file di log al momento del crash è il seguente:

```
< T1,start>
< T1, lr,50'000,48'500>
< T1,cc1,60'000,61'500>
< T1,commit>
< T2,start>
< T2,cc2,80'000,60'000>
```

Quando il sistema inizia il ripristino, esamina il file di log e determina che per  $T_1$  sono presenti sia il record di start sia quello di commit. Esegue quindi l'operazione `redo( $T_1$ )`. Determina inoltre che per  $T_2$  è presente solo il record di start. Esegue quindi l'operazione `undo( $T_2$ )`. Lo stato della base di dati dopo il ripristino diventa il seguente:

$$lr = 48'500 \quad cc1 = 61'500 \quad cc2 = 80'000.$$

Supponiamo adesso che si abbia un crash subito dopo aver scritto su memoria stabile il record di log relativo al commit di  $T_2$ . Quando il sistema inizia il ripristino, esamina il file di log e determina che sia per  $T_1$  sia per  $T_2$  sono presenti il record di start e quello di commit. Esegue quindi le operazioni `redo( $T_1$ )` e `redo( $T_2$ )`. Lo stato della base di dati diventa il seguente:

$$lr = 48'500 \quad cc1 = 61'500 \quad cc2 = 60'000.$$

□

## 8.6 Limitazioni delle transazioni flat

Il modello flat di transazione considerato nel Paragrafo 8.1.2 è un modello molto semplice ma presenta alcuni limiti, primo fra tutti quello di non permettere di effettuare il commit oppure l'abort selettivo solo di alcune parti della transazione. Per gestire attività semplici, quali ad esempio operazioni di addebito e accredito su un conto corrente bancario, è certamente sufficiente un'unica unità atomica di esecuzione. Tuttavia, per operazioni più complesse, occorre fornire un maggior potere espressivo, mantenendo, per quanto possibile, la semplicità del modello flat. Gli esempi seguenti presentano alcune applicazioni per cui il modello flat non è completamente appropriato.

### 8.6.1 Esempi

Introduciamo ora un primo esempio per chiarire meglio le limitazioni delle transazioni flat.

**Esempio 8.16** Supponiamo che il Sig. Rossi voglia effettuare un viaggio da Singapore a Milano, e che non esistano voli diretti tra le due località ma esista un volo diretto da Singapore a Francoforte. Occorre quindi prenotare un certo numero di voli intermedi. Supponiamo che la rappresentazione ad alto livello della corrispondente transazione sia la seguente:

```
BeginWork
    prenota il volo da Singapore a Francoforte
    prenota il volo da Francoforte a Milano nello stesso giorno
EndWork
```

Supponiamo che nella tratta Francoforte-Milano siano rimasti solo posti di business e che il cliente giudichi questa tariffa troppo costosa. In questo caso potremmo considerare delle soluzioni alternative. Ad esempio, prendere un volo da Francoforte a Bergamo e poi raggiungere Milano in pullman, oppure cercare un volo da Francoforte a Torino e proseguire per Milano in treno. In generale, utilizzando transazioni flat, sono possibili due sole scelte: *(i)* abortire l'intera transazione, con l'effetto però di annullare l'intero lavoro fatto, in particolare la prenotazione del volo da Singapore a Francoforte; *(ii)* terminare in ogni caso la transazione e cancellare poi le prenotazioni che non sono più utili, con conseguente aumento del lavoro di gestione. L'ideale sarebbe avere a disposizione un *rollback selettivo*; invece di disfare l'intera transazione, potremmo effettuare un rollback dell'ultima prenotazione, mantenendo la prenotazione del volo da Singapore a Francoforte. □

Il seguente è un altro esempio di applicazione che mostra i limiti delle transazioni flat.

**Esempio 8.17** Supponiamo che, in un sistema bancario, alla fine di ogni mese sia necessario modificare un milione di conti correnti accreditando i relativi interessi. Supponiamo inoltre che l'intero lavoro sia effettuato da una sola transazione flat, il cui codice sia approssimativamente il seguente:

```

BeginWork
  For ogni conto  $C$  Do
    aggiorna  $C$ 
  EndFor
  Commit
EndWork

```

Se tutto va a buon fine, allora tutti gli aggiornamenti sono resi permanenti quando la transazione esegue il commit. Al contrario, se, ad esempio, sono stati modificati 999'999 conti e subito dopo l'ultimo aggiornamento si verifica una caduta di sistema, è necessario disfare 999'999 aggiornamenti, in base alla proprietà di atomicità. Naturalmente questo comporta la perdita di lavoro utile, in quanto gli aggiornamenti eseguiti non sono scorretti e, inoltre, disfare 999'999 aggiornamenti può richiedere tanto tempo quanto quello impiegato per eseguirli.  $\square$

Per superare le limitazioni delle transazioni flat, sono state proposte molte estensioni. Nel seguito ne illustreremo due, che sono quelle recepite dallo standard SQL.

### 8.6.2 Transazioni con savepoint

La caratteristica “tutto-o-niente” delle transazioni flat fornisce la semantica di fallimento più semplice. La gestione delle situazioni di errore in transazioni organizzate secondo tale modello è tuttavia molto primitiva. Ci sono però situazioni, come quelle illustrate negli esempi precedenti, in cui viene eseguito molto lavoro da parte della transazione e non tutto è necessariamente reso invalido dal verificarsi di un singolo errore nell'esecuzione della transazione stessa. Un approccio alternativo è permettere alla transazione di poter tornare ad un qualche stato, interno alla transazione, precedente lo stato in cui si è verificato l'errore. Tutte le modifiche eseguite dalla transazione tra i due stati sono annullate, ma non le modifiche precedenti allo stato a cui si ritorna in caso di errore. Tale approccio richiede di poter informare il sistema circa uno stato, detto *savepoint*, del programma applicativo che deve essere opportunamente salvato in modo da potervi ritornare in caso di malfunzionamenti, mediante un rollback selettivo.

Un savepoint è stabilito tramite l'invocazione di un'opportuna primitiva (denotata nel seguito con **SaveWork**). Tale primitiva richiede al sistema di registrare lo stato corrente dell'esecuzione e restituisce un'etichetta che verrà successivamente utilizzata per tornare a quel savepoint. Per ripristinare lo stato ad un certo savepoint viene invocata la primitiva **RollBackWork** passandogli l'identificatore del savepoint a cui vogliamo tornare. L'approccio è illustrato graficamente nella Figura 8.3. Nella figura, la parte tratteggiata evidenzia le operazioni il cui effetto viene

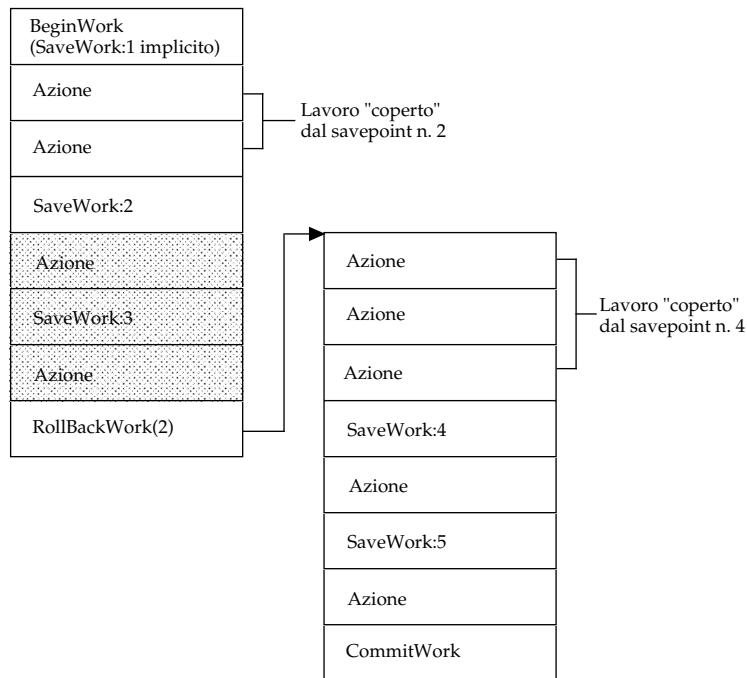


Figura 8.3: Uso di savepoint all'interno di una transazione

annullato in seguito all'invocazione di `RollBackWork(2)`; la freccia conduce alle istruzioni eseguite successivamente al rollback. Sono inoltre indicati due esempi di porzioni di applicazione “coperte” dai savepoint. L'esecuzione dell'istruzione `BeginWork` stabilisce implicitamente il primo savepoint della transazione.

**Esempio 8.18** Se consideriamo la transazione relativa alla pianificazione di un viaggio illustrata nell'Esempio 8.16, potremmo definire un savepoint dopo la prenotazione del volo aereo da Singapore a Francoforte in modo da poter mantenere tale prenotazione anche se la prenotazione della seconda tratta del volo dovesse essere annullata.  $\square$

È interessante notare la differenza tra il rollback completo della transazione ed il rollback al primo savepoint. Nel primo caso, la transazione scompare e perde tutte le risorse possedute. Se invece viene effettuato un rollback al primo savepoint, la transazione resta attiva, mantiene le risorse a lei allocate e semplicemente ritorna ad uno stato in cui non è stata eseguita alcuna operazione.

#### 8.6.3 Transazioni concatenate

La concatenazione è una variante del concetto di savepoint. L'idea delle transazioni concatenate è di permettere l'esecuzione di un'operazione di commit del lavoro

fatto fino ad un dato momento, non rilasciando il contesto della transazione. In particolare, non vengono rilasciati gli oggetti della base di dati acquisiti fino a quel momento, quali, ad esempio, eventuali lock in possesso della transazione. La corrispondente primitiva viene denotata con **ChainWork** ed è una combinazione delle primitive **CommitWork** e **BeginWork**. Notiamo però che l'esecuzione di una primitiva **ChainWork** non è esattamente equivalente all'esecuzione di una primitiva **CommitWork** ed al successivo avvio di una nuova transazione. L'esecuzione di **ChainWork** ha l'effetto di mantenere il contesto della transazione, mentre il commit normale rilascia il contesto.

Se confrontiamo il modello di transazioni concatenate con quello di transazioni con savepoint, possiamo effettuare le seguenti considerazioni:

- **Strutturazione del flusso di lavoro.** Sia le transazioni concatenate sia quelle con savepoint permettono di articolare in modo più strutturato il programma applicativo, rispetto alle transazioni flat.
- **Commit selettivi.** Dato che la primitiva **ChainWork** esegue un commit irrevocabile, nelle transazioni concatenate il rollback è limitato alla transazione correntemente attiva. Ciò corrisponde al ripristino dello stato all'ultimo savepoint, invece di selezionare un savepoint arbitrario.
- **Rollback.** Se si verifica un malfunzionamento durante l'esecuzione di una transazione con savepoint, l'effetto dell'intera transazione viene annullato indipendentemente da qualsiasi savepoint eseguito fino a quel momento. In una transazione concatenata, un rollback annulla invece solo gli effetti delle operazioni compiute sulla base di dati dall'ultima porzione di transazione.

## 8.7 Gestione delle transazioni in SQL, JDBC ed SQLJ

Nel seguito illustreremo il supporto alle transazioni offerto da SQL (vedi Capitolo 3) e da JDBC ed SQLJ, discussi nel Capitolo 4. SQL consente la specifica sia di transazioni concatenate sia di transazioni con savepoint, discusse in precedenza. Permette inoltre di specificare il grado di isolamento durante l'esecuzione di una transazione. JDBC ed SQLJ mettono a disposizione opportune primitive per iniziare una transazione ed effettuarne il commit o l'abort.

### 8.7.1 Gestione delle transazioni in SQL

Iniziamo a vedere come in SQL sia possibile iniziare e terminare una transazione.

#### 8.7.1.1 Creazione e terminazione di transazioni

In SQL una transazione inizia implicitamente all'esecuzione di un comando quale **SELECT**, **INSERT**, **UPDATE**, **DELETE** e **CREATE**, se non è già attiva un'altra transazione, e viene terminata dai comandi **COMMIT** o **ROLLBACK**. Una nuova transazione può

anche essere esplicitamente iniziata con il comando **START TRANSACTION**, oggetto del paragrafo successivo. SQL consente di specificare sia transazioni concatenate sia transazioni con savepoint. Una transazione concatenata può essere specificata mediante l'opzione **END CHAIN**, utilizzabile nel comando **COMMIT**, con la seguente sintassi:

```
COMMIT [WORK] [AND [NO] CHAIN];
```

Notiamo come le formulazioni: **COMMIT**, **COMMIT WORK**, **COMMIT WORK AND NO CHAIN** e **COMMIT AND NO CHAIN** siano del tutto equivalenti e servano per la specifica di una transazione flat, mentre i comandi **COMMIT AND CHAIN** e **COMMIT WORK AND CHAIN** servono per specificare una transazione concatenata (identiche opzioni sono fornite per il comando **ROLLBACK**).

Un savepoint può essere specificato all'interno di una transazione mediante il comando:

```
SAVEPOINT <nome savepoint>;
```

tramite cui viene associata un'etichetta (**nome savepoint**) al savepoint, per futuri riferimenti. Il numero massimo di savepoint definibili in una transazione dipende dallo specifico DBMS. Un savepoint può essere esplicitamente eliminato mediante il comando **RELEASE SAVEPOINT**, specificando il nome del savepoint che vogliamo eliminare. Questo comando ha l'effetto di eliminare il savepoint specificato e tutti i savepoint successivi a questo nella transazione.

Infine, la sintassi del comando **ROLLBACK** è stata estesa per permettere di effettuare il rollback ad uno specifico savepoint:

```
ROLLBACK [WORK] [AND [NO] CHAIN] [TO SAVEPOINT <nome savepoint>];
```

Le clausole **AND CHAIN** e **TO SAVEPOINT** sono mutuamente esclusive.

#### *8.7.1.2 Opzioni nell'esecuzione di una transazione*

Abbiamo visto in precedenza come la proprietà fondamentale assicurata dal gestore della concorrenza sia l'isolamento dell'esecuzione di una transazione dagli effetti di altre transazioni eseguite concorrentemente. A tal fine, SQL offre all'utente la possibilità di specificare il *livello di isolamento* voluto per una transazione, scegliendolo da un insieme pre-definito. Queste ed altre caratteristiche della transazione possono essere specificate mediante opportune opzioni del comando **START TRANSACTION**, la cui sintassi, semplificata, è la seguente:

```
START TRANSACTION [<modo di accesso>
                  <livello di isolamento>
                  <dimensione diagnostica>];
```

dove:

- <modo di accesso> definisce la tipologia di operazioni che la transazione eseguirà sulla base di dati. Sono possibili due opzioni: se la transazione è definita **READ ONLY**, non può modificare la base di dati, sia a livello di schema sia a livello di istanza (ad esempio eseguire inserimenti, cancellazioni, ecc.), mentre queste restrizioni non si applicano ad una transazione **READ WRITE**, che costituisce il valore di default. La specifica del modo di accesso consente al DBMS di gestire in modo più efficiente il controllo della concorrenza. Ad esempio, un insieme di transazioni **READ ONLY** necessita solo di lock condivisi, con conseguente aumento del livello di concorrenza.
- <livello di isolamento> consente di specificare il grado di protezione che vogliamo ottenere rispetto all'esecuzione di transazioni concorrenti e quindi quali anomalie possono essere tollerate. La sintassi è **ISOLATION LEVEL <livello>**, dove <livello> può assumere uno tra un insieme pre-definito di valori, illustrati nel seguito.
- <dimensione diagnostica> indica il numero massimo di condizioni di errore che possono essere gestite. La sua sintassi è la seguente: **DIAGNOSTICS SIZE <numero condizioni>**. Non approfondiremo oltre questo aspetto in quanto non strettamente legato al controllo della concorrenza.

In alternativa è possibile utilizzare il comando **SET TRANSACTION**, con una sintassi simile. L'unica differenza è che il comando **START TRANSACTION**, oltre a definire le caratteristiche della transazione, la inizia.

Come detto in precedenza la clausola <livello di isolamento> permette di stabilire il grado di protezione che vogliamo ottenere rispetto all'esecuzione concorrente di altre transazioni, scegliendolo tra quattro opzioni pre-definite. Naturalmente, più elevato è il grado di isolamento richiesto, maggiori saranno le garanzie rispetto all'isolamento, ma più onerosa sarà la gestione dell'esecuzione della transazione da parte del DBMS. In particolare, per livelli di isolamento bassi possono essere utilizzate alternative o varianti dei protocolli di locking, meno onerose del 2PL e granularità fini per i lock. I DBMS possono gestire tutte le quattro opzioni previste dal standard, oppure un sottoinsieme; in questo caso, il livello di isolamento effettivamente garantito non può essere inferiore a quello specificato.

Il livello più basso di isolamento è dato dall'opzione **READ UNCOMMITTED**. Una transazione con tale livello di isolamento può leggere le modifiche effettuate da una transazione che non ha ancora effettuato il commit ed è quindi soggetta alle anomalie dovute a letture sporche, fantasmi e letture non ripetibili esposte nel Paragrafo 8.2.1. Per evitare però che transazioni **READ UNCOMMITTED** provochino la memorizzazione di dati scorretti nella base di dati, viene posta la restrizione che tali transazioni possano essere solo di tipo **READ ONLY**, evitando quindi le anomalie da perdita di aggiornamento. Nonostante sia abbastanza probabile che transazioni **READ UNCOMMITTED** siano esposte alle anomalie discusse nel Paragrafo 8.2.1, ci possono essere casi in cui tali anomalie possono essere tollerabili. È il caso ad

```

import java.sql.*;
import java.io.*;
class exampleJDBC
{public static void main (String args [])
{
    Connection con = null;
    try {

        String ilGenere = "comico";
        Class.forName ("sun.jdbc.odbc.JdbcOdbcDriver");
        con = DriverManager.getConnection("jdbc:odbc:ilMioDB",
                                         "laMiaLogin",
                                         "laMiaPassword");

        con.setAutoCommit(false);
        Statement st = con.createStatement();
        ResultSet rs = st.executeQuery("SELECT AVG(valutaz)
                                      FROM Film
                                      WHERE genere = '" + ilGenere + "'");

        rs.next();
        if (rs.getDouble(1) < 4)
            st.executeUpdate("UPDATE Film
                             SET valutaz= valutaz*1.05");
        else
            st.executeUpdate("UPDATE Film
                             SET valutaz = valutaz*0.95");
        PreparedStatement pst =
            con.prepareStatement("SELECT titolo, regista, valutaz
                                FROM Film
                                WHERE genere = ?");
        pst.setString(1,ilGenere);
        rs = pst.executeQuery();
        ...
        con.commit();
        con.close();
    }
    catch(java.lang.ClassNotFoundException e) {
        System.err.print("ClassNotFoundException:  ");
        System.err.println(e.getMessage());
    }
    catch (SQLException e1) {try{if (con != null) con.rollback(); };}
    catch (SQLException e) {
        while( e!=null){
            System.out.println("SQLState:  "+ e.getSQLState());
            System.out.println("Code:  "+ e.getErrorCode());
            System.out.println("Message:  "+ e.getMessage());
            e = e.getNextException();
        }}}
```

Figura 8.4: Transazione in JDBC

esempio di una transazione che effettua operazioni statistiche su una grossa mole di dati, in cui un'inconsistenza su uno o pochi dati non inficia la valutazione statistica. Il livello di isolamento **READ COMMITTED** evita l'anomalia dovuta a letture sporche in quanto garantisce che una transazione legga solo modifiche di transazioni che hanno effettuato il commit. In questo caso, come per i successivi livelli di isolamento, la transazione può anche essere di tipo **READ WRITE**. Il successivo livello di isolamento, **REPEATABLE READ** aggiunge al livello precedente la garanzia che non si verifichi l'anomalia dovuta a letture non ripetibili, evitando quindi che una transazione possa modificare un dato letto precedentemente dalla transazione in oggetto, fino a quando la transazione termina. È però ancora possibile il fenomeno dei fantasmi. Il livello di isolamento maggiore è rappresentato dalla opzione **SERIALIZABLE**, che rappresenta l'opzione di default e garantisce l'esecuzione della transazioni solo nel contesto di schedule serializzabili. Sono quindi evitate tutte le anomalie discusse nel Paragrafo 8.2.1.

### 8.7.2 Gestione delle transazioni in JDBC ed SQLJ

Se non altrimenti specificato, JDBC gestisce l'esecuzione di ogni singolo comando SQL come una transazione. Questo comportamento è chiamato *auto-commit* e può essere disabilitato utilizzando il metodo `setAutoCommit` della classe `Connection` (vedi Capitolo 4), a cui deve essere passato un valore booleano (`false` nel caso in cui volessimo disabilitare l'auto-commit). Se l'auto-commit è disabilitato, una transazione viene creata all'inizio dell'esecuzione dell'applicazione. Tale transazione può essere terminata esplicitamente, invocando il metodo `commit` o `rollback` della classe `Connection`, o implicitamente, al termine dell'esecuzione.

**Esempio 8.19** La Figura 8.4 illustra una versione semplificata del programma JDBC dell'Esempio 4.25 del Capitolo 4 a cui sono stati aggiunti i comandi per la gestione delle transazioni. Tutti i comandi SQL compresi tra le istruzioni `con.setAutoCommit(false)` e `con.close()` costituiscono un'unica transazione e questo implica che o vengono eseguiti tutti oppure non ne viene eseguito alcuno. Viceversa, se non fossero state inserite queste due istruzioni, per default ogni singolo comando SQL sarebbe stato considerato una transazione a se stante. Inoltre, il meccanismo delle eccezioni consente di richiedere il rollback della transazione quando viene sollevata un'eccezione a causa dell'esecuzione di un comando SQL. □

SQLJ, invece, per default disabilita l'auto-commit. Quindi, una transazione viene creata all'inizio dell'esecuzione dell'applicazione. Tale transazione può essere esplicitamente terminata usando i comandi **COMMIT** e **ROLLBACK**, con la seguente sintassi:

```
#sql {COMMIT};  
#sql {ROLLBACK};
```

Dopo l'esecuzione di uno di questi due comandi, viene automaticamente creata una nuova transazione. L'auto-commit può comunque essere richiesto al momento

della connessione con il DBMS. Ad esempio, in Oracle, il metodo `connect` (vedi Capitolo 4) prevede un ulteriore parametro booleano. Se tale parametro viene posto a `true`, l'auto-commit viene abilitato. La sintassi per abilitare l'auto-commit è quindi la seguente:

```
oracle.connect("jdbc:odbc:il_mio_DB","la_mia_login",
               "la_mia_password",true);
```

### Note conclusive

Modelli, architetture e protocolli per la gestione delle transazioni sono sempre stati oggetto di una notevole attività di ricerca a causa della loro estrema rilevanza per la tecnologia delle basi di dati. Alcune proposte scaturite da questa attività sono volte ad eliminare parte dei problemi dei meccanismi di controllo della concorrenza classici illustrati in questo capitolo. Ad esempio, sono state proposte varianti del protocollo 2PL per limitare il problema del deadlock, che prevedono, ad esempio, l'introduzione di nuovi tipi di lock, in aggiunta ai lock esclusivi e condivisi, quali i lock di tipo *update* [GMUW02]. Ulteriori estensioni ai meccanismi classici possono essere suddivise in tre categorie: estensioni derivanti dalla definizione di nuovi modelli dei dati, estensioni derivanti dagli sviluppi di nuove architetture di sistema ed estensioni derivanti da nuove applicazioni.

Rispetto alla prima categoria, il settore delle basi di dati è caratterizzato da un'intensa ricerca relativa alla definizione di modelli semanticamente più ricchi del modello relazionale. Pensiamo ad esempio al modello orientato ad oggetti, al modello attivo (vedi Capitolo 11) ed ai più recenti modelli relazionali ad oggetti (vedi Capitolo 10). L'espressività semantica di tali modelli ha richiesto l'estensione dei modelli di transazione sotto diversi aspetti. Ad esempio, nozioni come le gerarchie di ereditarietà, oggetti composti e versioni, tipiche del modello orientato ad oggetti, hanno richiesto l'introduzione di nuovi meccanismi di lock. Le regole attive hanno richiesto l'analisi di differenti modalità di accoppiamento, per eseguire aggiornamenti generati dall'esecuzione dei trigger.

Tra gli sviluppi nel settore delle architetture di sistema che hanno un impatto sulle tecniche di esecuzione delle transazioni ricordiamo i sistemi main-memory, i sistemi paralleli ed i più recenti *sistemi per la gestione di stream di dati* (*DSMS – Data Stream Management System*). Nei sistemi main-memory, l'intera base di dati risiede in memoria centrale, mentre copie di backup risiedono in memoria secondaria. Tale architettura ha richiesto la rivisitazione delle tecniche di ripristino per assicurare che l'informazione necessaria al ripristino della base di dati possa essere salvata in memoria secondaria senza ridurre le prestazioni delle normali operazioni in main-memory. Nelle architetture parallele, gli insiemi di dati sono spesso partizionati tra più nodi. Questo approccio permette di eseguire la stessa operazione in parallelo su tutti i dati che appartengono allo stesso insieme. Questo richiede di mantenere la consistenza tra le varie partizioni e tra tutte le relative strutture di accesso. Infine, i sistemi per la gestione di stream di dati rappresentano attualmente una delle più rilevanti tendenze in ambito basi di dati, dal momento

che molte applicazioni hanno oggi la necessità di gestire flussi continui (stream) di dati che devono essere processati in tempo reale senza necessariamente essere memorizzati nella base di dati. Pensiamo, ad esempio, alle informazioni generate da una rete di sensori o alle informazioni riguardanti il traffico di rete o quello telefonico. Per questi sistemi le tradizionali nozioni di equivalenza ed i corrispondenti meccanismi di locking non sono più adeguati e nuove strategie devono quindi essere definite [GBO06].

Infine, i DBMS vengono ormai utilizzati in svariati domini applicativi, molti dei quali richiedono una sostanziale estensione al modello di transazione flat e alle architetture che lo supportano (alcune di queste estensioni sono state trattate nel Paragrafo 8.6). Tra i nuovi requisiti, ricordiamo la necessità di gestire transazioni a lunga durata, distribuite, generate da utenti mobili, oppure da sistemi real-time o con particolari requisiti di sicurezza.

### Note bibliografiche

Le problematiche riguardanti i modelli teorici e gli aspetti architetturali relativi alle transazioni sono state ampiamente investigate ed esiste una vasta letteratura in merito. In particolare, le transazioni sono trattate nei libri di testo relativi a sistemi di basi di dati ed in testi specializzati. Tra questi ultimi ricordiamo il testo di Gray e Reuter [GR93], quello di Lynch et al. [LMWF94] ed il testo di Weikum e Vossen [WV02]. Il testo di Gray e Reuter tratta molto in dettaglio da un punto di vista sistemistico la gestione delle transazioni. Il testo riporta inoltre numerosi dettagli relativi all'implementazione, spesso includendo la definizione delle strutture dati in linguaggio C. Il testo di Lynch et al. ha invece un approccio nettamente teorico alla trattazione dei concetti di base relativi alle transazioni; il testo tra l'altro discute in modo molto approfondito tutte le problematiche relative alle transazioni distribuite che non sono state affrontate in questo capitolo. Il testo di Weikum e Vossen fornisce una trattazione completa sia dal punto di vista teorico sia applicativo delle principali problematiche connesse alla gestione delle transazioni. Inoltre, illustra le principali tendenze di ricerca in tale ambito. Infine, il testo di Papadimitriou [Pap86] tratta in modo completo la teoria della concorrenza.

### Esercizi

**8.1** Dare un esempio di schedule serializzabile rispetto alle viste ma non rispetto ai conflitti.

**8.2** Consideriamo tre transazioni  $T_1$ ,  $T_2$  e  $T_3$  ed i dati  $a$ ,  $b$  e  $c$ :

$T_1$	$T_2$	$T_3$
Read $a$	Read $a$	Read $a$
$a := a + 10$	Read $b$	Read $c$
Write $a$	$b := b + a$	$c := c + a$
Commit	Write $b$	Write $c$
	Commit	Commit

- a. Determinare uno schedule serializzabile e non banale (cioè diverso da un'esecuzione in sequenza) per le transazioni  $T_1$ ,  $T_2$  e  $T_3$ .

- b. Dimostrare formalmente la serializzabilità dello schedule determinato al punto (a).

**8.3** Dimostrare che lo schedule presentato nell'Esempio 8.10 non è serializzabile rispetto ai conflitti.

**8.4** Consideriamo tre transazioni  $T_1, T_2, T_3$  aventi rispettivamente timestamp 10, 14 e 18. Descrivere qual è il comportamento del sistema in base agli schemi wait-die e wound-wait se:

- a.  $T_1$  richiede un dato bloccato da  $T_2$ .
- b.  $T_3$  richiede un dato bloccato da  $T_1$ .

**8.5** Presentare un esempio di schedule di transazioni in deadlock. Costruire il grafo delle attese corrispondente e specificare alcune modalità per risolvere il deadlock.

**8.6** Consideriamo lo schedule presentato nell'Esempio 8.7. Specificare le operazioni eseguite in base alle tecniche di ripristino con modifiche differite e non differite, se:

- a. Si verifica un crash dopo l'esecuzione dell'operazione  $\text{Write}_1[lr]$ .
- b. Si verifica un crash dopo l'esecuzione del commit da parte di  $T_1$ .
- c. Si verifica un crash dopo l'esecuzione dell'operazione  $\text{Write}_2[cc]$ .
- d. Si verifica un crash dopo l'esecuzione del commit da parte di  $T_2$ .

Assumiamo che lo stato della base di dati prima dell'esecuzione sia:

$$lr = 50'000 \quad cc = 60'000.$$

**8.7** Data la transazione di Figura 8.1, sviluppare una transazione equivalente in SQL, JDBC ed SQLJ.

## Capitolo 11

# Basi di dati attive

Molte applicazioni recenti, quali il controllo dei processi, la gestione di reti di generazione e distribuzione di potenza, la gestione automatizzata del lavoro d'ufficio o di magazzino ed i sistemi di controllo in ambito medico, richiedono risposte tempestive a situazioni critiche. Tale comportamento non è gestito in maniera soddisfacente dai DBMS tradizionali, in quanto essi eseguono interrogazioni e transazioni solo a fronte di esplicite richieste da parte di un utente o di un programma applicativo, sono quindi *passivi*. Per le applicazioni sopra citate, è necessario che determinate situazioni siano monitorate e, nel momento in cui si verificano, che determinate azioni siano eseguite, possibilmente entro prestabiliti vincoli temporali. Un possibile esempio nell'ambito della gestione automatizzata di un magazzino è il riordino automatico di un certo prodotto quando la sua disponibilità scende sotto una certa soglia. Altri esempi di problemi che richiedono un comportamento reattivo sono il controllo ed il mantenimento dei vincoli di integrità, il calcolo di dati derivati e la gestione delle eccezioni.

La necessità di reagire automaticamente a situazioni critiche ha portato alla definizione ed allo sviluppo delle *basi di dati attive*, come integrazione della tecnologia delle basi di dati con le regole di produzione sviluppate inizialmente nel contesto dell'intelligenza artificiale. Tali regole, chiamate anche *regole attive* o *trigger* nei sistemi di gestione dati, rappresentano uno strumento aggiuntivo nel contesto dello sviluppo di applicazioni. Tramite la loro definizione, infatti, una parte della logica applicativa può essere memorizzata direttamente nella base di dati ed utilizzata automaticamente da tutte le applicazioni che generano determinate situazioni, con conseguente vantaggio in termini di flessibilità di manutenzione, sia della base di dati sia delle applicazioni, e di tempestività nell'esecuzione.

In questo capitolo, discuteremo le caratteristiche generali dei linguaggi per la specifica di trigger nei DBMS relazionali ed i corrispondenti modelli di esecuzione. Presenteremo quindi le caratteristiche di SQL:2003 per la definizione dei trigger e ne illustreremo vari esempi di utilizzo.

### 11.1 Approcci all'esecuzione automatica di azioni

Nei DBMS tradizionali (passivi), i requisiti delle applicazioni che richiedono l'esecuzione automatica di (re-)azioni possono essere soddisfatti secondo due diversi

approcci, illustrati nelle Figure 11.1(a) e (b), rispettivamente. Il primo consiste nello sviluppare un programma applicativo che interroga periodicamente la base di dati per determinare se la situazione monitorata si è verificata (*polling*) ed in questo caso eseguire opportune operazioni. Il secondo consiste nel modificare ogni programma che aggiorna la base di dati incorporandovi dei controlli per determinare se la situazione monitorata si è verificata. La maggiore difficoltà del primo approccio consiste nel determinare la frequenza ottima di polling: la strategia può essere inefficiente se il polling è troppo frequente o se il costo di esecuzione dell'interrogazione è alto, mentre, se il periodo tra due controlli successivi è troppo lungo, alcune situazioni che richiederebbero l'esecuzione della (re)-azione possono non essere determinate in maniera tempestiva. Il secondo approccio, invece, compromette la modularità e la riusabilità del software. In particolare, il cambiamento delle (re)-azioni richiede la modifica del codice in ogni applicazione; inoltre, la correttezza del comportamento del sistema è garantita solo se l'implementazione dei programmi applicativi è corretta.

Le basi di dati attive superano le limitazioni delle soluzioni sopra descritte, integrando il monitoraggio delle situazioni in maniera omogenea con le altre componenti del DBMS. Un *DBMS attivo* monitora in maniera continua lo stato della base di dati e reagisce spontaneamente all'occorrere di determinate situazioni predefinite. Ad esempio, il sistema può reagire a modifiche della base di dati, all'esecuzione di particolari operazioni di sistema oppure al verificarsi di eventi esterni alla base di dati (come un fallimento hardware rilevato da un programma diagnostico). Funzionalmente, un sistema di gestione dati attivo controlla il verificarsi di determinate situazioni chiamate *eventi* e, al verificarsi di tali situazioni, esegue automaticamente un'*azione*. L'approccio integrato dei DBMS attivi è illustrato nella Figura 11.1(c).

Una *regola attiva*, o *trigger*, è lo strumento sintattico per definire la (re-)azione del sistema. L'introduzione dei trigger in un sistema di gestione dati permette di sottrarre la conoscenza di tipo reattivo ai programmi applicativi e rappresentare tale conoscenza sotto forma di regole, che diventano un nuovo elemento dello schema logico. Il maggiore vantaggio di questo approccio risiede nel fatto che tale conoscenza viene definita una volta e condivisa da tutte le applicazioni. Una modifica al comportamento reattivo comporterà quindi solo una modifica ai trigger che lo implementano, mentre le applicazioni verranno mantenute inalterate.

I sistemi di gestione dati attivi sono stati studiati in maniera intensiva a partire dagli anni '90, sia da un punto di vista teorico sia architettonale. Questa ricerca ha portato alla realizzazione di molteplici prototipi. Contemporaneamente, tutti i maggiori DBMS commerciali, tra cui Oracle, IBM DB2, Microsoft SQL Server e PostgreSQL, sono stati estesi con la possibilità di definire trigger. A partire da SQL:1999, anche lo standard SQL propone un approccio alla definizione e gestione dei trigger. Tale specifica è stata consolidata in SQL:2003, tuttavia molti DBMS commerciali non sono ancora allineati con le nuove specifiche. Per questo motivo, e per chiarire meglio le potenzialità dei trigger, nel seguito descriveremo in generale la sintassi e la modalità di esecuzione dei trigger discutendo varie alternative, non

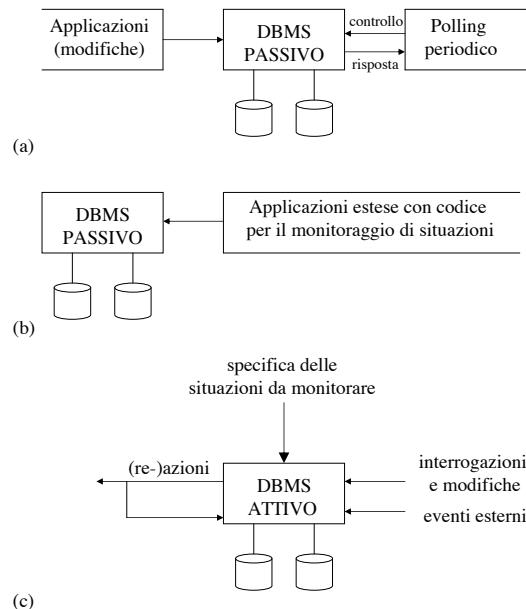


Figura 11.1: Approcci al supporto di un comportamento reattivo: (a) polling; (b) estensione dei programmi applicativi; (c) approccio integrato dei DBMS attivi

tutte previste da SQL. Nel Paragrafo 11.4 introdurremo quindi la sintassi e la modalità di esecuzione dei trigger in SQL.

## 11.2 Linguaggi per la specifica di trigger

Nei sistemi di basi di dati attivi, il sistema deve essere in grado di riconoscere determinate “situazioni”, interne od esterne alla base di dati, descritte in precedenza ed osservabili, e di attivare “(re-)azioni”, quali operazioni sulla base di dati o programmi arbitrari, all’occorrere di tali situazioni. Deve quindi essere possibile definire situazioni e (re-)azioni mediante regole attive. Le situazioni sono abitualmente specificate in termini di *eventi* (che accadono) e *condizioni* (che devono essere verificate).

Le regole attive sono usualmente specificate attraverso il paradigma *evento-condizione-azione* (*ECA* – *Event-Condition-Action*). Una regola ECA ha il seguente formato:

ON *evento* IF *condizione* THEN *azione*

il cui significato è: se si verifica l'evento, la condizione è valutata e, se la condizione è soddisfatta, allora l'azione viene eseguita.

Poiché una regola attiva rappresenta un nuovo oggetto (il trigger appunto) di una base di dati, i DBMS rendono disponibili comandi specifici per la creazione, la cancellazione e la modifica di tali regole. Alcuni sistemi prevedono, inoltre, comandi per abilitare e disabilitare i trigger al fine di escluderne temporaneamente alcuni.

I comandi di creazione dei trigger permettono di specificare le tre componenti principali: evento, condizione ed azione, secondo le seguenti regole generali.

**Evento.** Un evento è qualcosa che accade, che è di interesse (per la definizione delle regole) e che può essere mappato, almeno dal punto di vista del sistema, in un istante temporale. Nella maggioranza dei casi vogliamo descrivere una classe di eventi, di cui possono verificarsi nel tempo occorrenze multiple. Esempi tipici di eventi sono: modifiche dei dati (inserimenti, cancellazioni, modifiche); accesso ai dati (operazioni di selezione su di una particolare tabella); operazioni di sistema (login di utenti, operazioni legate alla gestione delle transazioni od alla gestione delle autorizzazioni); eventi temporali espressi in tempo assoluto (es. 1 gennaio 2006 alle 12:00), in intervalli di tempo (es. ogni 10 minuti) o periodicamente (es. ogni giorno alle ore 12:00); eventi esterni, cioè definiti da una condizione applicativa (es. data-troppo-grande).

Spesso gli eventi possono essere qualificati specificando, mediante i qualificatori BEFORE e AFTER, se la regola debba essere considerata prima o dopo l'esecuzione dell'operazione corrispondente all'evento. Un esempio in cui è significativo considerare una regola prima dell'esecuzione dell'operazione che l'attiva è quello in cui vogliamo verificare una pre-condizione che, nel caso in cui non sia verificata, impedisca (per esempio, abortendola) l'esecuzione dell'evento stesso. Questo comportamento evita l'esecuzione di operazioni non desiderate, con un conseguente miglioramento delle prestazioni del sistema.

Gli eventi di base possono anche essere combinati per costruire eventi composti, tramite l'utilizzo di operatori logici o sequenze.

Tutti i DBMS commerciali che supportano i trigger permettono di specificare operazioni di aggiornamento ai dati (inserimento, cancellazione, modifica) come eventi. Alcuni, come Oracle, sono più flessibili ed ammettono come eventi anche alcune operazioni di sistema (inclusa la rilevazione di errori). Solo alcuni (tra cui Oracle e PostgreSQL) permettono di specificare eventi composti. Per motivi di efficienza, gli eventi temporali non sono previsti né da SQL né dai DBMS commerciali.

**Condizione.** La condizione specifica un ulteriore controllo che viene eseguito quando la regola è selezionata e prima che l'azione venga eseguita. Le condizioni possono essere espresse come predicati, che devono essere soddisfatti dalla base di dati, oppure come interrogazioni sulla base di dati. In questo ultimo caso, la condizione si assume vera se e solo se l'interrogazione restituisce un insieme non vuoto. In entrambi i casi, è spesso possibile utilizzare solo una parte limitata del

linguaggio di interrogazione, al fine di valutare le condizioni in modo più efficiente e quindi migliorare le prestazioni del sistema. La condizione può inoltre essere specificata come chiamata ad una funzione che può o meno accedere alla base di dati.

In genere, la condizione può utilizzare variabili di sistema (ad esempio, l'identificatore dell'utente) e stati passati (oltre a quello corrente) della base di dati, ad esempio per confrontare il valore aggiornato di un attributo con il suo valore prima della modifica. In particolare, l'insieme delle tuple inserite, cancellate o modificate da un evento viene chiamato *tabella di transizione*. La possibilità di utilizzare la tabella di transizione nella definizione della condizione permette di fare riferimento a due stati consecutivi della base di dati e rappresenta il meccanismo principale per l'implementazione dei vincoli di transizione che, come abbiamo osservato nel Capitolo 3, non possono essere rappresentati in SQL utilizzando il meccanismo dei vincoli di integrità o le asserzioni. Numerosi DBMS permettono, inoltre, l'omissione della condizione: in questi casi la condizione si assume sempre vera.

Per motivi di efficienza, non sempre la condizione è prevista nella definizione dei trigger nei DBMS commerciali (ad esempio, Microsoft SQL Server e PostgreSQL non ne permettono la specifica).

**Azione.** L'azione è eseguita quando la regola è selezionata e la sua condizione è vera. Possibili azioni sono: operazioni di aggiornamento od accesso ai dati; chiamate di procedura; operazioni di definizione dei dati; altre operazioni per il controllo delle transazioni (quali rollback e commit, vedi Capitolo 8) o per concedere o revocare privilegi (vedi Capitolo 9); sequenze delle operazioni precedenti. Anche l'azione può utilizzare la tabella di transizione per specializzare le operazioni da eseguire. Il tipo di azione permesso dai DBMS commerciali e da SQL dipende in genere dalla modalità con cui è stato specificato l'evento (**BEFORE** oppure **AFTER**).

**Esempio 11.1** In riferimento alla base di dati della videoteca, supponiamo di voler assegnare un valore di default all'attributo **valutaz** della relazione **Film**, se non viene specificato alcun valore al momento dell'inserimento di un nuovo film. Supponiamo però che questo valore non sia fisso ma venga calcolato come media dei valori di valutazione di tutti i film presenti nella base di dati al momento dell'inserimento del nuovo film.

Per implementare questo comportamento, possiamo definire un trigger caratterizzato dalle seguenti componenti:

- L'evento in questo caso è rappresentato da un'operazione di inserimento sulla relazione **Film**.
- La condizione determina se il valore per l'attributo **valutaz** di almeno una tupla inserita è **NULL**.<sup>1</sup> Solo in questo caso verrà eseguita l'azione.

---

<sup>1</sup>Ricordiamo che un singolo comando **INSERT** può inserire anche più di una tupla (vedi Capitolo 3).

```

While ci sono regole da considerare Do
    (1) seleziona una regola R
    (2) valuta la condizione di R
    (3) If la condizione di R è vera Then
        esegui l'azione di R
    EndIf
EndWhile

```

Figura 11.2: Algoritmo corrispondente al processo reattivo

- L'azione corrisponde al calcolo del valore medio per l'attributo **valutaz** dei film presenti nella base di dati ed all'assegnazione di tale valore all'attributo **valutaz** delle tuple inserite.  $\square$

### 11.3 Modello di esecuzione

Rispetto ad un DBMS passivo tradizionale, in un DBMS attivo sono presenti due attività aggiuntive: l'attività di rilevazione degli eventi e di attivazione delle regole corrispondenti ed il *processo reattivo* vero e proprio, cioè la selezione e l'esecuzione delle regole. Entrambe queste attività sono in genere invocate solo in determinati momenti di tempo per la scelta dei quali sono possibili diverse alternative.

L'algoritmo per l'esecuzione del processo reattivo, nella sua forma più semplice, è rappresentato nella Figura 11.2. Il processo reattivo può essere visto come l'esecuzione iterativa di tre passi principali, noti come selezione, valutazione ed esecuzione della regola:

1. **Selezione.** Consiste nello scegliere una delle regole attivate dall'evento.
2. **Valutazione.** Consiste nella valutazione della condizione della regola selezionata; a questo punto la regola selezionata non è più attivata.
3. **Esecuzione.** Questa fase viene eseguita solo se la condizione per la regola selezionata è soddisfatta e consiste nell'esecuzione delle operazioni corrispondenti all'azione di tale regola.

Notiamo che le regole attivate ma non selezionate al passo (1) rimangono comunque attivate e quindi possono essere selezionate in successive iterazioni dell'algoritmo, mentre la regola selezionata viene disattivata durante l'esecuzione del passo (2). Inoltre, l'esecuzione dell'azione di una regola può provocare nuovi eventi che possono a loro volta attivare nuove regole. Tali regole possono essere aggiunte all'insieme delle regole da considerare, ed essere quindi selezionate al termine dell'esecuzione della regola correntemente attivata, in successive iterazioni dell'algoritmo (modalità *iterativa* di gestione delle attivazioni), oppure, relativamente a tali nuove regole, può iniziare una nuova esecuzione dell'algoritmo durante l'esecuzione

dell'azione della regola correntemente attivata (modalità *ricorsiva* di gestione delle attivazioni).

Il processo reattivo è influenzato da molteplici parametri. Tra questi ricordiamo la granularità di attivazione del processo, la modalità di esecuzione delle regole, le modalità di accoppiamento tra le varie fasi del processo reattivo e la terminazione del processo reattivo. Nel seguito, illustreremo con maggior dettaglio ciascuno di questi aspetti.

#### 11.3.1 *Granularità del processo reattivo*

Con *granularità del processo reattivo* intendiamo la frequenza di attivazione dell'algoritmo presentato nella Figura 11.2. La granularità più fine è “sempre”, cioè il processo reattivo viene attivato non appena un evento si verifica. La frequenza con cui eseguire l'attivazione dipende in questo caso dal tipo di evento considerato. Nel caso di eventi temporali, la frequenza di attivazione dipende dalla frequenza con cui l'evento temporale si verifica; nel caso di eventi che corrispondono ad operazioni sulla base di dati (ad esempio, inserimento, cancellazione o modifica di una singola tupla in una base di dati relazionale), il processo può essere attivato più volte nel contesto dell'esecuzione di uno stesso comando SQL. In caso di eventi che corrispondono all'esecuzione di comandi SQL (situazione tipica nei DBMS commerciali), la frequenza di attivazione è invece minore.

Esistono poi diversi livelli di granularità meno fini. Un esempio è dato dall'attivazione del processo reattivo dopo l'esecuzione di ogni comando SQL nel caso in cui gli eventi corrispondano a singole operazioni sulla base di dati. Un altro esempio significativo si ottiene attivando il processo al termine di ogni transazione (vedi Capitolo 8), nel contesto della quale si possono verificare molteplici eventi.

Notiamo che più grossolana è la granularità prescelta, maggiore sarà il numero di regole attivate e quindi potenzialmente selezionabili dall'algoritmo nella Figura 11.2. La selezione avviene ordinando le regole sulla base della *priorità* e scegliendo una tra le regole a priorità maggiore. Le priorità possono essere assegnate alle regole o dall'utente all'atto della creazione delle regole in modo esplicito o dal sistema in modo implicito. Le priorità possono essere relative, espresse cioè come relazioni di precedenza tra coppie di regole, od assolute, tramite priorità numeriche. Esempi tipici di priorità assolute implicite sono costituite da proprietà statiche di ogni singola regola, come il momento della creazione, o proprietà dinamiche, come l'istante di ultimo utilizzo. Per motivi di efficienza, i DBMS commerciali spesso prevedono priorità assolute basate su proprietà statiche delle regole.

**Esempio 11.2** Consideriamo una transazione per la base di dati della videoteca, che esegue le seguenti operazioni:

1. inserisce un insieme di nuovi film, mediante un singolo comando **INSERT** (cioè è possibile supponendo che i dati dei film da inserire siano contenuti in una qualche relazione di appoggio);

2. inserisce un insieme di nuovi video, mediante un singolo comando `INSERT`, ma solo se la valutazione dei film in essi contenuti è maggiore di 2; altrimenti, si ritiene che il film non riscuota interesse e quindi i nuovi video non vengono inseriti.

Supponiamo innanzitutto che la granularità del processo reattivo sia “sempre”. Bisogna allora distinguere il caso in cui gli eventi corrispondono all’esecuzione di comandi SQL ed il caso in cui gli eventi corrispondono all’esecuzione di operazioni sulla base di dati che coinvolgono una singola tupla. Nel primo caso, il processo reattivo viene attivato una volta dopo l’esecuzione del comando (1) ed una volta dopo l’esecuzione del comando (2). Nel secondo caso, invece, il processo reattivo viene attivato una volta per ogni film e video inserito.

Consideriamo ora la granularità a livello di comando (cioè, “dopo l’esecuzione di ogni comando SQL”). Indipendentemente da cosa denotino gli eventi, il processo viene comunque attivato solo due volte, una dopo l’esecuzione di ciascun comando SQL.

Con la granularità a livello transazionale, il processo reattivo viene infine attivato una sola volta, al commit di  $T$ .  $\square$

### 11.3.2 Esecuzione dei trigger

Vi sono due modalità per l’esecuzione di un trigger: *orientata all’istanza* (instance-oriented) e *orientata all’insieme* (set-oriented). Nel primo caso, la regola viene eseguita una volta per ogni tupla della base di dati coinvolta nell’evento che attiva la regola e soddisfa la condizione; nel secondo caso, la regola è eseguita una sola volta per l’insieme di tuple coinvolte nell’evento. Il seguente esempio illustra la differenza, sottile, tra le due modalità.

**Esempio 11.3** Consideriamo la regola progettata nell’Esempio 11.1. Con l’esecuzione orientata all’insieme, la regola viene eseguita una sola volta dopo l’inserimento dei nuovi film. In questo caso, se la condizione della regola è soddisfatta da almeno una tupla inserita, durante l’esecuzione dell’operazione di modifica viene calcolato un unico valore medio che viene assegnato all’attributo `valutaz` delle tuple appena inserite. Con l’esecuzione orientata all’istanza, invece, l’azione della regola viene eseguita per ogni tupla appena inserita che soddisfa la condizione; quindi, il valore medio viene calcolato una volta dopo ogni inserimento, considerando un insieme potenzialmente differente di valori ad ogni computazione. Come conseguenza, ad ogni film inserito può essere assegnato un valore di valutazione differente.  $\square$

Notiamo che la differenza tra le due modalità di esecuzione si può perdere variando la granularità del processo reattivo. Se ad esempio il processo reattivo viene attivato dopo ogni singola operazione sulla singola istanza, allora l’azione di un trigger viene sempre eseguita per al massimo un’istanza; di conseguenza, le due modalità di esecuzione generano lo stesso risultato. Questa situazione non si

verifica nei DBMS commerciali in cui, come già osservato, gli eventi generalmente corrispondono all'esecuzione di comandi SQL o ad operazioni di sistema. Entrambe le modalità di esecuzione sono quindi in generale disponibili.

L'algoritmo corrispondente al processo reattivo, illustrato nella Figura 11.2, può essere specializzato per ciascuna modalità di esecuzione come segue:

- **Orientata all'insieme.** I passi (2) e (3) sono eseguiti una sola volta per l'insieme di istanze coinvolte nell'evento che ha attivato la regola selezionata nel passo (1). L'insieme delle tuple aggiornate dall'evento viene chiamato *tabella di transizione*.
- **Orientata all'istanza.** I passi (2) e (3) sono eseguiti una volta per ogni istanza coinvolta nell'evento che ha attivato la regola. Ad ogni esecuzione, l'istanza coinvolta nell'evento viene chiamata *variabile* o *tupla di transizione*.

#### 11.3.3 Modalità di accoppiamento

La *modalità di accoppiamento* (*coupling mode*) viene definita tra coppie di fasi adiacenti nel processo reattivo e permette di stabilire quando una fase deve essere eseguita, rispetto all'esecuzione della fase precedente. La modalità di accoppiamento permette quindi di indicare quando deve essere controllata la condizione rispetto al verificarsi dell'evento (ad esempio, subito, quando si verifica l'evento, o ad un certo punto nel tempo, come alla fine della transazione che ha generato l'evento) e quando deve essere eseguita l'azione rispetto alla verifica della condizione (ad esempio, appena la condizione è soddisfatta, come estensione della transazione corrente, o alla fine della transazione corrente, come sua estensione o in una nuova transazione).

Le varie modalità di accoppiamento sono illustrate graficamente nella Figura 11.3. La figura si riferisce alla modalità di accoppiamento tra evento e condizione, assumendo che l'azione sia eseguita immediatamente dopo la valutazione della condizione. Il verificarsi dell'evento che attiva la regola è indicato con un pallino nero, mentre la linea orizzontale indicata come "processo reattivo" indica l'esecuzione della condizione e dell'azione. L'inizio e la fine delle transazioni, infine, sono indicate come segmenti verticali. Notiamo che, se l'esecuzione dei trigger è orientata all'istanza, la transazione corrispondente al processo reattivo coinvolge la verifica della condizione e l'esecuzione dell'azione per ogni tupla coinvolta nell'evento. Sono possibili le seguenti modalità di accoppiamento:

- **Modalità immediata**, cioè nel contesto della stessa transazione (vedi Figura 11.3(a)). Per esempio, la modalità immediata per la condizione rispetto all'evento significa che la condizione viene valutata non appena l'evento si verifica.
- **Modalità differita**, cioè al momento del commit della transazione corrente (vedi Figura 11.3(b)). Per esempio, la modalità differita per la condizione

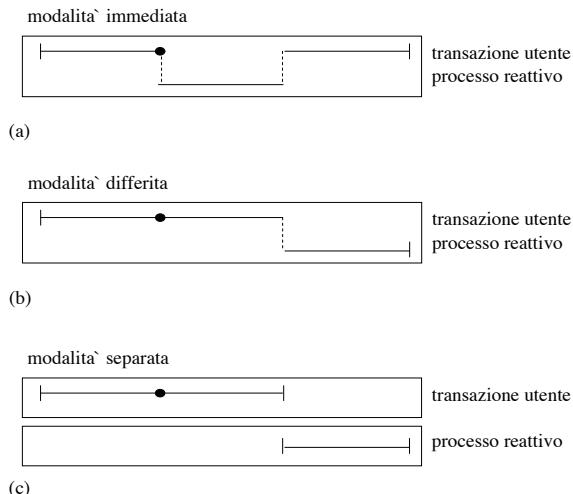


Figura 11.3: Modalità di accoppiamento

rispetto all'evento significa che se l'evento si verifica durante l'esecuzione della transazione  $T$ , allora la condizione è valutata al termine della transazione  $T$ , come sua estensione.

La modalità differita può essere utile, ad esempio, per i trigger che impongono vincoli d'integrità. Infatti, una transazione può eseguire numerose operazioni che violano un vincolo (così da attivare i trigger corrispondenti), ma la stessa transazione potrebbe riportare la base di dati in uno stato consistente rispetto ai vincoli prima di raggiungere il momento del commit. Verificare la condizione ed eseguire l'azione solo al termine della transazione permette di rilevare i vincoli di integrità effettivamente violati.

- **Modalità separata**, cioè in una transazione distinta (vedi Figura 11.3(c)). Per esempio, la modalità separata per la condizione rispetto all'evento significa che la verifica della condizione avviene in una transazione separata da quella durante l'esecuzione della quale si è verificato l'evento.

La modalità separata può essere utile in presenza di un numero elevato di regole da attivare, al fine di scomporre la conseguente ampia transazione in un insieme di transazioni più piccole.

È da notare che la decisione riguardo alla modalità di accoppiamento non è completamente indipendente dalle altre dimensioni del processo reattivo. Ad esempio, se la granularità del processo reattivo è a livello di transazione, allora la modalità immediata tra evento e condizione non è possibile in quanto la selezione,

e di conseguenza la valutazione e l'esecuzione della regola, verranno effettuate solo al termine dell'esecuzione della transazione.

In genere, i DBMS commerciali adottano la modalità di accoppiamento immediata, poiché permette una maggiore semplicità di implementazione del processo reattivo. Alcuni prototipi di sistemi di gestione dati attivi permettono, tuttavia, la specifica di una particolare modalità di accoppiamento nella definizione di una regola.

**Esempio 11.4** Consideriamo la regola progettata nell'Esempio 11.1 e la transazione presentata nell'Esempio 11.2. Supponiamo che la granularità del processo reattivo sia a livello di singolo comando SQL. Supponiamo inoltre che la transazione al passo (1) inserisca il film "Kill Bill I" di Quentin Tarantino senza specificare alcun valore per la valutazione. Poiché l'inserimento di un nuovo film rappresenta l'evento della regola introdotta nell'Esempio 11.1, la regola viene attivata dopo l'esecuzione del passo (1). Supponendo che la modalità di accoppiamento tra condizione ed azione sia immediata, per la modalità di accoppiamento tra evento e condizione sono possibili le seguenti alternative:

- Se la modalità di accoppiamento è immediata, il trigger viene eseguito prima del passo (2); poiché non è stato specificato alcun valore per l'attributo **valutaz**, l'azione della regola calcola il valore medio delle valutazioni dei film esistenti e lo associa al nuovo film. Supponiamo che tale valore sia 2.5. Dopo l'esecuzione della regola, viene eseguito il passo (2). Poiché la valutazione per il film inserito è superiore a 2, eventuali video corrispondenti a questo film vengono inseriti.
- Se la modalità di accoppiamento è differita, il passo (2) viene eseguito prima dell'esecuzione della regola. Poiché nessun valore è stato specificato per il nuovo film durante l'inserimento, il valore della valutazione è NULL; quindi eventuali video relativi al nuovo film non verranno inseriti al passo (2). La valutazione del film inserito verrà poi modificata dall'azione della regola, selezionata ed eseguita al termine della transazione. Notiamo che, in base alla modalità di accoppiamento differita, se durante l'esecuzione di questa modifica, si verifica un errore che genera un rollback, verranno annullate non solo le modifiche eseguite dall'azione della regola ma anche tutti gli aggiornamenti eseguiti dalla transazione.
- Se la modalità di accoppiamento è separata, il comportamento è analogo a quello discusso per la modalità differita. Tuttavia, se durante l'esecuzione della modifica che rappresenta l'azione del trigger si verifica un errore, i film ed i video inseriti dalla transazione vengono mantenuti, in quanto rappresentano una transazione separata. Al contrario, la transazione corrispondente all'esecuzione dell'azione del trigger viene abortita e quindi il valore dell'attributo **valutaz** per il nuovo film inserito non viene modificato. □

#### 11.3.4 Terminazione

Se l'esecuzione di un'azione può produrre eventi che causano l'attivazione di altre regole, o della stessa regola, allora è possibile attivare regole all'infinito.

Al fine di evitare questa situazione, la soluzione più semplice è accettare la non terminazione come possibilità, lasciando al progettista delle regole il compito di assicurare che non si verifichi, analogamente a come la non terminazione viene trattata nei linguaggi di programmazione. Alternativamente, è possibile fissare un limite superiore, dipendente dai parametri del sistema, al numero di regole che possono essere eseguite, ricorsivamente o iterativamente, durante l'elaborazione. Se questo limite è superato, l'elaborazione termina in modo anormale. Questa soluzione è attualmente adottata dalla maggioranza dei DBMS commerciali. Un'altra soluzione può essere quella di imporre restrizioni sintattiche sull'insieme di regole per assicurare che l'elaborazione delle regole termini sempre. Al livello più semplice, le restrizioni possono assicurare che le regole non si attivino a vicenda (tale restrizione è però molto severa). Ad un livello più sofisticato, le limitazioni possono assicurare che le regole si attivino a vicenda ma senza formare un ciclo, richiedendo quindi che una regola non possa direttamente o indirettamente attivare se stessa. Soluzioni più complesse possono infine permettere le attivazioni ricorsive purché sia possibile garantire che, lungo ogni ciclo, la condizione di qualche regola diventerà falsa e quindi il ciclo verrà interrotto.

### 11.4 Trigger in SQL

A partire da SQL:1999, SQL permette di creare e manipolare trigger. Nel seguito, illustreremo la sintassi per la definizione dei trigger in SQL e ne descriveremo la modalità di esecuzione.

#### 11.4.1 Sintassi

In SQL, ogni trigger reagisce ad una specifica operazione di aggiornamento su una specifica tabella, chiamata *tabella soggetto*. Il comando per la creazione di un trigger in SQL ha il seguente formato:

```
CREATE TRIGGER <nome trigger>
  {BEFORE | AFTER}  <evento>
  ON <tabella soggetto>
  [REFERENCING {OLD [ROW] AS <variabile> |
                NEW [ROW] AS <variabile> |
                OLD TABLE AS <variabile> |
                NEW TABLE AS <variabile>}]
  [FOR EACH {ROW | STATEMENT}]
  [WHEN <condizione>]
  {<comando SQL> |
  BEGIN ATOMIC <sequenza di comandi SQL> END};
```

dove:

- <nome trigger> è il nome del trigger che viene definito.
- <tabella soggetto> è il nome della tabella sulla quale intendiamo monitorare l'esecuzione degli eventi.
- BEFORE o AFTER indica se la regola deve essere attivata prima o dopo l'esecuzione dell'evento.
- <evento> è un'operazione di aggiornamento per la tabella soggetto. L'evento può essere INSERT, DELETE o UPDATE [OF <lista attributi>]. Nel caso di evento UPDATE OF <lista attributi>, la regola viene attivata se viene eseguito un comando UPDATE sulla tabella soggetto, che modifica i valori degli attributi indicati in <lista attributi>. Gli eventi non possono essere combinati, quindi ogni regola può essere attivata solo da un singolo evento.
- La clausola REFERENCING permette di specificare opportuni alias per la tabella o la tupla di transizione.<sup>2</sup> Le parole chiave OLD e NEW permettono di specificare un alias per la tabella o la tupla di transizione esistente prima (OLD) o dopo (NEW) l'esecuzione dell'evento, secondo le regole che discuteremo nel seguito.
- Un trigger può essere eseguito FOR EACH ROW, cioè con modalità orientata alla tupla, o FOR EACH STATEMENT, cioè con modalità orientata all'insieme. In questo secondo caso, il trigger viene eseguito anche se nessuna tupla è stata effettivamente aggiornata dall'esecuzione del comando. Il default per la modalità di esecuzione di un trigger è FOR EACH STATEMENT.
- <condizione> è un'espressione booleana SQL arbitraria.
- <comando SQL> è un singolo comando SQL mentre <sequenza di comandi SQL> identifica una sequenza di comandi SQL separati da ';' (vedi Capitolo 4). Tali comandi non possono contenere parametri di connessione.

Gli stati (OLD e NEW) delle tabelle e delle tuple di transizione ai quali può essere assegnato un alias nella clausola REFERENCING dipendono dal tipo di trigger (BEFORE o AFTER), dalla modalità di esecuzione (FOR EACH ROW o FOR EACH STATEMENT) e dal tipo di evento (INSERT, DELETE, UPDATE). Ogni combinazione di questi elementi, oltre a determinare quando e come il trigger debba essere eseguito, influenza infatti quali tuple sono visibili nella tabella soggetto e nelle tabelle/tuple di transizione, durante la valutazione della condizione e l'esecuzione dell'azione. In particolare, le tuple di transizione sono visibili solo per esecuzioni orientate all'istanza e, se il trigger è di tipo BEFORE, la tabella di transizione non è mai visibile (ma, in caso di esecuzione orientata all'istanza, la tupla di transizione è visibile). Inoltre, valgono le seguenti regole di visibilità:

---

<sup>2</sup>Ricordiamo che, nello standard SQL, viene utilizzato il termine "variabile" anziché "tupla" di transizione.

- nei trigger attivati dal comando **INSERT**:
  - se i trigger sono di tipo **BEFORE**, le tuple inserite non sono visibili come parte della tabella soggetto ma possono essere accedute usando la clausola **REFERENCING NEW**;
  - se i trigger sono di tipo **AFTER**, le tuple inserite sono visibili sia nella tabella soggetto sia tramite la clausola **REFERENCING NEW**;
- nei trigger attivati dal comando **DELETE**:
  - se i trigger sono di tipo **BEFORE**, le tuple cancellate sono visibili come parte della tabella soggetto e possono anche essere accedute usando la clausola **REFERENCING OLD**;
  - se i trigger sono di tipo **AFTER**, le tuple cancellate non sono visibili come parte della tabella soggetto e possono essere accedute solo usando la clausola **REFERENCING OLD**;
- nei trigger attivati dal comando **UPDATE**:
  - per tutti i tipi di trigger, i valori precedenti e correnti delle tuple possono essere acceduti usando la clausola **REFERENCING (OLD e NEW, rispettivamente)**;
  - se i trigger sono di tipo **BEFORE**, l'effetto della modifica non è visibile nello stato corrente della tabella soggetto, mentre se i trigger sono di tipo **AFTER** tale effetto è visibile.

Notiamo che trigger attivati dal comando **INSERT** non possono contenere una clausola **REFERENCING OLD**, in quanto le tuple coinvolte nell'evento non esistevano prima dell'esecuzione dell'evento stesso. Analogamente, i trigger attivati dal comando **DELETE** non possono contenere una clausola **REFERENCING NEW**, in quanto le tuple coinvolte nell'evento non esistono al termine dell'esecuzione dell'evento stesso.

Nel caso di trigger di tipo **BEFORE**, SQL sconsiglia l'esecuzione di comandi di aggiornamento dei dati nel contesto dell'azione, ma non la vieta. Questo suggerimento è dettato dal fatto che un trigger di tipo **BEFORE** potrebbe aggiornare alcuni dati prima dell'esecuzione dell'evento che ha attivato la regola, generando comportamenti anomali.

La Tabella 11.1 riassume le regole di visibilità sopra descritte.

**Esempio 11.5** Il trigger introdotto informalmente nell'Esempio 11.1 può essere rappresentato in SQL, considerando una modalità di esecuzione orientata all'insieme, come segue:

```
CREATE TRIGGER ModificaValNull
AFTER INSERT ON Film
REFERENCING NEW TABLE AS NT
```

Tipo di trigger e modalità di esecuzione	Evento	Tabelle/tuple di transizione
BEFORE ROW*	INSERT	NEW
	DELETE	OLD
	UPDATE	NEW, OLD
AFTER ROW	INSERT	NEW, NEW TABLE
	DELETE	OLD, OLD TABLE
	UPDATE	tutte
BEFORE STATEMENT*	INSERT	-
	DELETE	-
	UPDATE	-
AFTER STATEMENT	INSERT	NEW TABLE
	DELETE	OLD TABLE
	UPDATE	NEW TABLE, OLD TABLE

\* I comandi di aggiornamento nell'azione del trigger sono sconsigliati

Tabella 11.1: Regole di visibilità

```

FOR EACH STATEMENT
WHEN EXISTS(SELECT *
            FROM NT
            WHERE valutaz IS NULL)
UPDATE Film
SET valutaz = (SELECT AVG(valutaz) FROM Film)
WHERE (titolo,regista) IN (SELECT titolo,regista FROM NT);

```

Scegliendo una modalità di esecuzione orientata all'istanza, il trigger deve essere modificato come segue:

```

CREATE TRIGGER ModificaValNull
AFTER INSERT ON Film
REFERENCING NEW ROW AS NR
FOR EACH ROW
WHEN (NR.valutaz IS NULL)
UPDATE Film
SET valutaz = (SELECT AVG(valutaz) FROM Film)
WHERE titolo = NR.titolo AND regista = NR.regista;

```

Come discusso nell'Esempio 11.3, il risultato ottenuto dall'esecuzione delle azioni dei due trigger è diverso. □

SQL prevede un comando `DROP TRIGGER <nome trigger>` per la cancellazione del trigger con nome `<nome trigger>`.

#### **11.4.2 Modello di esecuzione**

SQL prevede una granularità di esecuzione del processo reattivo a livello di singolo comando SQL. Quindi, dopo l'esecuzione di un comando, verrà determinato l'insieme dei trigger attivati dall'evento e si procederà con i passi illustrati nel Paragrafo 11.3, utilizzando una modalità di accoppiamento immediata.

La selezione del trigger da eseguire viene effettuata sulla base del tipo dei trigger (BEFORE o AFTER), della modalità di esecuzione specificata (FOR EACH ROW o FOR EACH STATEMENT) e di priorità implicite assolute, basate sul tempo di creazione dei trigger. Un trigger “vecchio” viene infatti eseguito prima di un trigger “giovane”. Poiché possono esistere vincoli di integrità definiti per la tabella soggetto e l'esecuzione di eventi può comportare violazioni a tali vincoli di integrità, nel definire l'ordine di esecuzione è necessario considerare anche l'attività di verifica dei vincoli, ottenendo il seguente ordine di selezione:

1. vengono inizialmente selezionati i trigger di tipo BEFORE, con modalità FOR EACH STATEMENT;
2. per ogni tupla oggetto del comando che rappresenta l'evento, vengono selezionate le regole di tipo BEFORE, con modalità FOR EACH ROW;
3. viene eseguito l'evento per la singola tupla e verificati i vincoli di integrità su tale tupla, con valutazione immediata (NOT DEFERRABLE oppure DEFERRABLE ma con modalità di esecuzione IMMEDIATE, vedi Capitolo 3);
4. per ogni tupla oggetto del comando che rappresenta l'evento, vengono selezionate le regole di tipo AFTER, con modalità FOR EACH ROW;
5. a questo punto, vengono verificati i vincoli di integrità immediati sulla tabella, con valutazione immediata (NOT DEFERRABLE oppure DEFERRABLE ma con modalità di esecuzione IMMEDIATE, vedi Capitolo 3);
6. vengono infine selezionate le regole di tipo AFTER, con modalità FOR EACH STATEMENT.

Ad ogni passo, se esiste più di un trigger del tipo considerato, la selezione viene effettuata in base all'ordine di creazione delle regole.

Per quanto riguarda la terminazione, SQL non fornisce indicazioni e non vengono fornite condizioni sintattiche per evitare situazioni di non terminazione. Come già osservato nel Paragrafo 11.3.4, in genere i DBMS commerciali fissano un limite superiore, dipendente da parametri di sistema, al numero di regole che possono essere attivate ricorsivamente.

### **11.5 Applicazioni dei trigger**

I trigger sono oggi sempre più utilizzati per svariate tipologie di applicazioni. Tra gli utilizzi più comuni ricordiamo: la verifica di vincoli di integrità, il calcolo di

dati derivati, il mantenimento di repliche dei dati e la specifica di regole operative, dipendenti dal dominio. Nel seguito, presenteremo alcuni esempi di trigger in SQL per lo schema relazionale della videoteca presentato nella Figura 6.14.

#### **11.5.1 Specifica di vincoli di integrità**

Come abbiamo visto nel Capitolo 3, SQL permette la specifica di diversi tipi di vincoli di integrità statici, alcuni dei quali rappresentabili come asserzioni (vedi Capitolo 3). Tuttavia, per motivi di efficienza, il meccanismo delle asserzioni è raramente presente nei DBMS commerciali. Inoltre, i vincoli di transizione, che mettono in relazione stati diversi della base di dati, non possono essere rappresentati in alcun modo in SQL.

Per imporre vincoli complessi o vincoli di transizione è quindi possibile utilizzare un insieme di trigger in cui la tabella soggetto è una tabella coinvolta nella definizione del vincolo, l'evento rappresenta un'operazione che può comportare una violazione del vincolo e la condizione indica quando il vincolo di integrità viene violato.

Per quanto riguarda l'azione, sono possibili diversi comportamenti. Se intendiamo simulare il comportamento adottato da SQL per la verifica dei vincoli di integrità, allora nel caso in cui la condizione sia vera, e quindi il vincolo sia violato, è necessario abortire la transazione (vedi Capitolo 8).<sup>3</sup> Il seguente esempio illustra questo tipo di applicazione dei trigger.

**Esempio 11.6** Consideriamo la base di dati della videoteca ed il seguente vincolo di integrità:

**V<sub>1</sub>:** *Ogni cliente non può noleggiare più di 3 video contemporaneamente.*

Questo vincolo può essere rappresentato dall'asserzione presentata nella Figura 11.4(a). Le operazioni che possono violare l'asserzione sono l'inserimento di nuove tuple nella tabella **Noleggio** e la modifica del valore dell'attributo **dataRest** in **NULL**. Quando una di queste operazioni viene eseguita, il vincolo viene verificato e, se la condizione non è soddisfatta, la transazione nel contesto della quale è stata eseguita l'operazione viene abortita.

Tale comportamento può anche essere specificato utilizzando due trigger, uno per ogni operazione che può violare il vincolo di integrità.<sup>4</sup> Il trigger attivato dall'inserimento di un nuovo noleggio è presentato nella Figura 11.4(b). Per tale trigger, è prevista un'esecuzione orientata all'istanza; quindi, per ogni tupla inserita nella tabella **Noleggio**, riferita con l'alias **NR**, si verifica se il numero di noleggi corrispondente al cliente che ha effettuato il nuovo noleggio è maggiore di 3. Se questa condizione è soddisfatta, l'azione esegue il rollback della transazione

<sup>3</sup>A questo proposito ricordiamo che, spesso, i DBMS permettono di sollevare eccezioni ma non permettono di abortire una transazione direttamente nell'azione di una regola.

<sup>4</sup>Ricordiamo che in SQL un trigger viene attivato da un singolo evento.

```

CREATE ASSERTION VerificaNoleggi
CHECK (NOT EXISTS
       (SELECT * FROM Noleggio
        WHERE dataRest IS NULL
        GROUP BY codCli
        HAVING COUNT(*) > 3));

```

(a)

```

CREATE TRIGGER VerificaNoleggi
AFTER INSERT ON Noleggio
REFERENCING NEW ROW AS NR
FOR EACH ROW
WHEN (SELECT COUNT(*)
      FROM Noleggio
      WHERE dataRest IS NULL AND
            codCli = NR.codCli) > 3
ROLLBACK;

```

(b)

Figura 11.4: Implementazione di un vincolo di integrità: (a) asserzione; (b) trigger

e quindi, in base alla modalità di accoppiamento immediata, l'inserimento viene annullato.

Una regola analoga dovrebbe essere specificata per la modifica dell'attributo `dataRest`. Notiamo tuttavia che la modifica in `NULL` del valore dell'attributo `dataRest` per le tuple di `Noleggio` non è un'operazione significativa per il dominio applicativo considerato, in quanto una volta che un video è stato restituito (quindi il valore di `dataRest` è diverso da `NULL`), il suo valore non dovrebbe essere modificato dalle applicazioni.  $\square$

Mentre un DBMS esegue sempre il rollback delle operazioni che hanno determinato la violazione di un vincolo di integrità, l'uso dei trigger rende più flessibile il mantenimento dell'integrità, permettendo la specifica di operazioni di ripristino alternative, come illustrato dal seguente esempio.

**Esempio 11.7** Relativamente al vincolo  $V_1$  presentato nell'Esempio 11.6, supponiamo di non volere abortire una transazione che esegua l'inserimento di un nuovo noleggio, se questo comporta una violazione del vincolo di integrità, ma di volere semplicemente annullare l'inserimento. In questo modo, le eventuali altre operazioni eseguite dalla transazione non verranno annullate. La Figura 11.5(a) presenta il trigger che implementa questo comportamento. Poiché la modalità di esecuzione è orientata all'istanza, la condizione e l'azione verranno eseguite una volta per ogni tupla inserita.

Il comportamento precedente può anche essere descritto mediante il trigger presentato nella Figura 11.5(b), con esecuzione orientata all'insieme. In questo caso, l'alias `NT` rappresenta l'intera tabella di transizione. Questa tabella può essere utilizzata come ogni altra tabella nella definizione della condizione e dell'azione. A differenza dell'esempio precedente, in questo caso la condizione e l'azione vengono eseguite una sola volta.  $\square$

```
CREATE TRIGGER VerificaNoleggi
AFTER INSERT ON Noleggio
REFERENCING NEW ROW AS NR
FOR EACH ROW
WHEN (SELECT COUNT(*)
      FROM Noleggio
      WHERE dataRest IS NULL AND
            codCli = NR.codCli) > 3
DELETE FROM Noleggio
WHERE colloc = NR.colloc AND
      dataNol = NR.dataNol;
```

(a)

```
CREATE TRIGGER VerificaNoleggi
AFTER INSERT ON Noleggio
REFERENCING NEW TABLE AS NT
FOR EACH STATEMENT
WHEN EXISTS
  (SELECT *
   FROM Noleggio
   WHERE Noleggio.dataRest IS NULL AND
         Noleggio.codCli IN (SELECT codCli FROM NT)
   GROUP BY codCli
   HAVING COUNT(*) > 3)
DELETE FROM Noleggio
WHERE (colloc,dataNol) IN (SELECT colloc, dataNol FROM NT) AND
      codCli IN (SELECT codCli
                  FROM Noleggio
                  WHERE Noleggio.dataRest IS NULL AND
                        Noleggio.codCli IN (SELECT codCli FROM NT)
                  GROUP BY codCli
                  HAVING COUNT(*) > 3);
```

(b)

Figura 11.5: Trigger per la verifica di vincoli di integrità: (a) esecuzione orientata all'istanza; (b) esecuzione orientata all'insieme

```

CREATE TRIGGER CalcolaPtiMancanti
AFTER INSERT ON Noleggio
REFERENCING NEW ROW AS NR
FOR EACH ROW
BEGIN ATOMIC
    UPDATE Standard
    SET ptiMancanti = ptiMancanti - 1
    WHERE codCli = NR.codCLI AND
          NR.colloc IN (SELECT colloc FROM Video
                         WHERE tipo = 'v');

    UPDATE Standard
    SET ptiMancanti = ptiMancanti - 2
    WHERE codCli = NR.codCLI AND
          NR.colloc IN (SELECT colloc FROM Video
                         WHERE tipo = 'd');
END;

```

Figura 11.6: Trigger per il calcolo di un dato derivato

### 11.5.2 Calcolo di dati derivati

Un altro utilizzo frequente dei trigger consiste nel mantenere aggiornato il valore di una colonna calcolata. Benché le colonne calcolate rappresentino informazione ridondante, la loro presenza può essere utile nel caso in cui tale informazione debba essere acceduta frequentemente (vedi Capitolo 6). Nel Capitolo 3 abbiamo descritto come SQL possa essere utilizzato per definire colonne calcolate. Tuttavia, tale definizione può utilizzare solo colonne della tabella a cui appartiene la colonna calcolata. Nel caso in cui sia invece necessario basare la definizione su colonne contenute in tabelle diverse, è possibile utilizzare un trigger, come illustrato dal seguente esempio.

**Esempio 11.8** In riferimento allo schema presentato nella Figura 6.14, supponiamo di volere mantenere aggiornato in modo automatico l'attributo `ptiMancanti` della tabella `Standard`. Supponiamo inoltre che il noleggio di ogni video di tipo vhs permetta di accumulare 1 punto mentre il noleggio di ogni video di tipo dvd permetta di accumulare 2 punti. Il trigger presentato nella Figura 11.6 implementa questo comportamento. □

### 11.5.3 Regole operative

I trigger possono essere utilizzati anche per imporre determinati comportamenti dipendenti dal dominio applicativo considerato. Tali comportamenti, noti anche come *business rule* o *regole operative*, possono essere implementati utilizzando un approccio passivo, mediante la definizione di opportune procedure e funzioni, memorizzate nel DBMS ed utilizzate dalle applicazioni, secondo l'approccio illustrato

```

CREATE PROCEDURE OrganizzaClienti()
BEGIN
    INSERT INTO VIP
    SELECT codCli, 5.00
    FROM Standard
    WHERE ptiMancanti <= 0;
    DELETE FROM Standard
    WHERE ptiMancanti <=0;
END;

CREATE TRIGGER OrganizzaClienti
AFTER UPDATE OF ptiMancanti ON Standard
REFERENCING NEW ROW AS NR
FOR EACH ROW
WHEN NR.ptiMancanti <= 0
BEGIN ATOMIC
    INSERT INTO VIP
    VALUES (NR.codCli,5.00);
    DELETE FROM Standard
    WHERE codCli = NR.codCli;
END;

```

(a) (b)

Figura 11.7: Implementazione di regole operative: (a) stored procedure; (b) trigger

nella Figura 11.1(b). Come discusso in precedenza, il principale svantaggio di questo approccio consiste nel fatto di dovere modificare tutte le applicazioni, al fine di richiamare opportunamente le funzioni o le procedure che garantiscono il corretto comportamento operazionale. La definizione di opportuni trigger costituisce un'alternativa a tale approccio, come illustrato dal seguente esempio.

**Esempio 11.9** In riferimento all'esempio della videoteca, supponiamo di voler implementare la seguente regola operativa:

*Quando il valore dell'attributo ptiMancanti per un cliente standard diventa 0, tale cliente viene rimosso dalla tabella Standard e viene inserito nella tabella VIP, con bonus pari a 5 euro.*

Questo comportamento può essere implementato tramite la procedura presentata nella Figura 11.7(a), che dovrà poi essere richiamata da ogni applicazione che modifica l'attributo ptiMancanti.<sup>5</sup> Una soluzione alternativa consiste nel definire il trigger presentato nella Figura 11.7(b), attivato automaticamente ad ogni modifica dell'attributo ptiMancanti. Notiamo che questa regola può essere attivata dall'esecuzione dell'azione della regola descritta nell'Esempio 11.8. □

### Note conclusive

I trigger sono sempre più utilizzati nello sviluppo di applicazioni per basi di dati. Come abbiamo già osservato, la maggior parte dei DBMS commerciali supporta la definizione dei trigger, benché spesso con lievi differenze sintattiche rispetto allo standard SQL e spesso con qualche funzionalità aggiuntiva. In particolare, i

<sup>5</sup>Notiamo che, in presenza del vincolo V<sub>5</sub> presentato nella Tabella 6.2 (ogni cliente è alternativamente un cliente standard od un cliente VIP), il vincolo non viene violato dall'esecuzione della procedura se la sua valutazione è differita.

DBMS commerciali, inclusi Oracle e Microsoft SQL Server, permettono di definire trigger non solo su tabelle ma anche su viste e, oltre ai trigger di tipo **BEFORE** e di tipo **AFTER**, propongono anche trigger di tipo **INSTEAD OF**. Tali trigger rimpiazzano l'esecuzione dell'evento con l'esecuzione dell'azione. Sono quindi molto utili per implementare operazioni di aggiornamento su viste che non sarebbero altrimenti concesse dal sistema. Inoltre, spesso, i sistemi impediscono l'uso della tabella soggetto nell'azione ed impongono restrizioni sulla sintassi relativa a trigger di tipo **BEFORE** (ad esempio, impediscono l'esecuzione di comandi DML nell'azione), al fine di evitare comportamenti anomali. Ricordiamo infine che il concetto di trigger è un concetto ortogonale rispetto al modello dei dati. Essi possono quindi essere utilizzati anche nel contesto del modello relazionale ad oggetti.

### Note bibliografiche

Alle basi di dati attive è dedicato il libro di Ceri e Widom [CW96] che, oltre ad illustrare in dettaglio i vari aspetti dei linguaggi e dei modelli di esecuzione delle regole attive, discute molteplici aspetti architettonici ed implementativi, che non sono stati trattati in questo capitolo, e presenta svariate applicazioni delle regole attive. Il libro presenta inoltre alcuni sistemi di basi di dati attivi (Starburst, Ariel, RDL, HiPAC, Ode e Chimera), alcuni dei quali sono stati sviluppati a livello prototipale per analizzare alternative sintattiche, semantiche ed architettoniche. Dedica inoltre un capitolo alla descrizione di SQL:1999 e dei trigger nei DBMS relazionali commerciali. Aspetti avanzati relativi alle basi di dati attive ed alle loro applicazioni sono trattati in [Pat99]. Per la documentazione sulle tipologie di trigger supportate dai DBMS commerciali, rimandiamo ai manuali dei vari sistemi ([Ora05a] per Oracle, [IBM04] per IBM DB2, [SQL05] per Microsoft SQL Server e [Pos06] per PostgreSQL).

### Esercizi

#### 11.1 Consideriamo il seguente schema relazionale:

```
Impiegato(imp#, nome, mansione, dataA, stipendio, premioP, dip#Dipartimento)
Dipartimento(dip#, nome, divisione, sede, dirigenteImpiegato).
```

Definire in SQL i seguenti trigger, utilizzando sia la modalità di esecuzione orientata all'istanza sia quella orientata all'insieme:

- Un trigger che, ogni volta che un dipartimento è cancellato dalla base di dati, ponga a **NULL** il valore dell'attributo **dip#** delle tuple della relazione **Impiegato** che avevano come precedente valore di **dip#** il numero del dipartimento cancellato.
- Un trigger che, ogni volta che un dipartimento è cancellato dalla base di dati, cancelli tutti gli impiegati che lavoravano nel dipartimento cancellato.
- Un trigger che impedisca la cancellazione di un dipartimento se esiste almeno un impiegato che vi lavora.
- Un trigger che, ogni volta che lo stipendio di un impiegato supera lo stipendio del direttore del suo dipartimento, ponga lo stipendio dell'impiegato uguale al valore dello stipendio del direttore del dipartimento.

- e. Un trigger che, ogni volta che gli stipendi degli impiegati sono modificati, se il totale degli stipendi aggiornati supera il loro totale prima della modifica, riduce del 5% gli stipendi di tutti gli impiegati del dipartimento 2.

**11.2** In riferimento allo schema relazionale dell'Esercizio 11.1, definire in SQL un trigger  $R_1$  che, ogni volta che qualche direttore di dipartimento è cancellato, cancelli anche tutti gli impiegati del dipartimento diretto dall'impiegato cancellato, ed il dipartimento stesso. Definire inoltre un trigger  $R_2$  che, ogni volta che gli stipendi vengono modificati, controlli la media degli stipendi aggiornati e, se tale media supera 2'500, cancelli tutti gli impiegati il cui stipendio è stato modificato e supera nello stato corrente 3'000.

Consideriamo adesso una base di dati contenente sei impiegati, Gianna, Chiara, Luigi, Giorgio, Luca e Serena, con le seguenti relazioni di dipendenza:

- Gianna è direttore di Chiara e Giorgio;
- Chiara è direttore di Luigi;
- Giorgio è direttore di Luca e Serena.

ed una transazione che cancella l'impiegato Gianna ed aggiorna gli stipendi in modo tale che la media dei valori aggiornati degli stipendi superi 2'500 ed il valore aggiornato dello stipendio di Chiara superi 3'000. Descrivere lo stato della base di dati al termine dell'esecuzione di tale transazione.

**11.3** Consideriamo il seguente schema relazionale:

```
Conto(num,nomeCli,saldo,tasso)
ContoBasso(numConto,data)
Movimento(num,numConto,importo).
```

Definire in SQL i trigger per implementare le seguenti regole operative, scegliendo per ciascuna regola la modalità di esecuzione ritenuta più opportuna:

- a. Un trigger che, se un conto ha un saldo inferiore a 500 ed un tasso di interesse superiore allo 0%, ponga il tasso di interesse del conto allo 0%.
- b. Un trigger che, se un conto ha un tasso di interesse superiore all'1%, ma inferiore al 2%, ponga il tasso di interesse del conto al 2%.
- c. Un trigger che, se un conto ha un saldo inferiore a 500 e non è stato ancora inserito nella relazione ContoBasso, inserisca il conto nella relazione ContoBasso, associandogli la data corrente.
- d. Un trigger che, se un conto nella relazione ContoBasso ha un saldo superiore a 500 nella relazione Conto, lo cancelli dalla relazione ContoBasso.
- e. Un trigger che, per ogni nuovo movimento effettuato, aggiorni il saldo del conto corrispondente nella relazione Conto.

**11.4** Consideriamo il seguente schema relazionale:

```
StudenteDottorato(e-mail,nome,area,supervisoreProf)
Prof(e-mail,nome,area)
Corso(titolo,profProf)
Segue(stuaStudenteDottorato,titoloCorso).
```

Specificare in SQL i trigger necessari per mantenere i seguenti vincoli di integrità, evitando di eseguire il rollback delle operazioni che comportano una violazione di tali vincoli:

- a. Ogni studente di dottorato deve lavorare nella stessa area del suo supervisore.
- b. Ogni studente di dottorato deve seguire almeno un corso.
- c. Ogni studente di dottorato deve seguire tutti i corsi tenuti dal suo supervisore.

**11.5** Consideriamo il seguente schema relazionale:

```
Fornisce(nomeF,codProdProdotto,prezzo)
Prodotto(codProd,descr,prezzo)
InfoOrdine(nOrdineOrdine,codProdProdotto,quantita)
Cliente(codCli,nome,indirizzo)
Ordine(nOrdine,data,codCliCliente,importoTot).
```

Definire in SQL i trigger per implementare le seguenti regole operative, scegliendo per ciascuna regola la modalità di esecuzione ritenuta più opportuna:

- a. Al momento dell'inserimento di un ordine, controllare che la data non sia successiva alla data corrente; se lo è inserire comunque l'ordine con come data la data corrente.
- b. Per assicurare che tutte le tuple della relazione **InfoOrdine** corrispondano ad ordini e prodotti esistenti, si propaghi la cancellazione di un ordine alle informazioni sul suo contenuto, e si impedisca la cancellazione di prodotti che compaiono in un ordine (evitando di effettuare il rollback della transazione).
- c. Reagire all'inserimento di un ordine calcolando l'importo totale dell'ordine a partire dai dati contenuti nelle relazioni **InfoOrdine** e **Prodotto** ed assegnare tale valore all'attributo **importoTot**, indipendentemente dal valore eventualmente specificato per tale campo nel comando di inserimento.