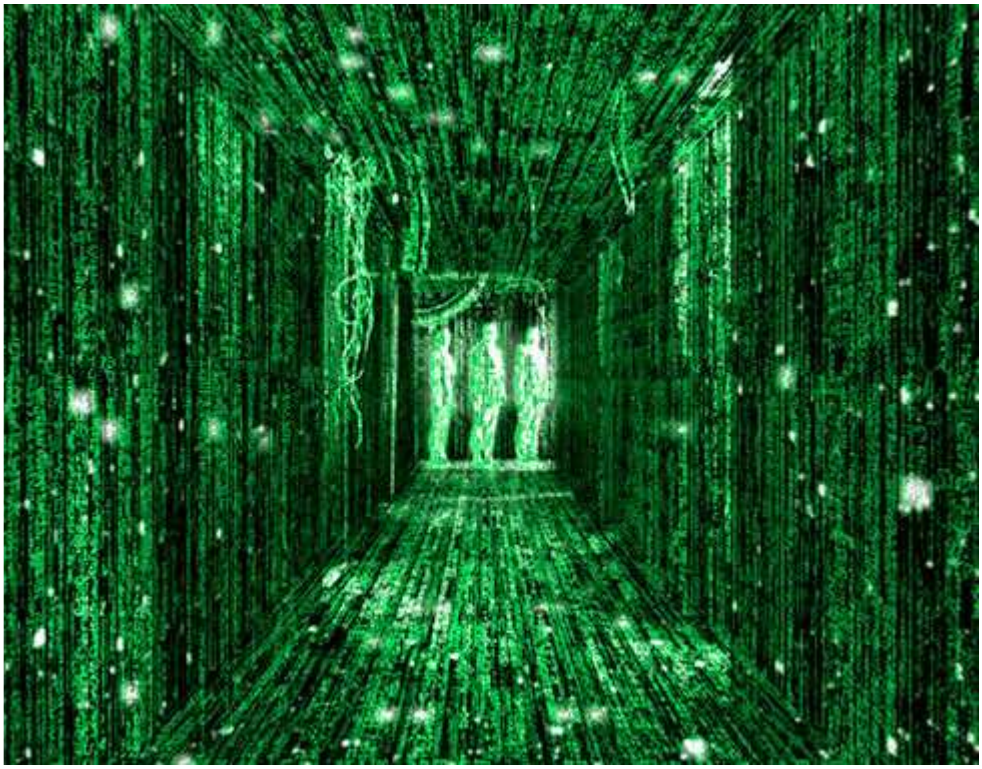
Basi di Dati
II Modulo

Mauro Mezzini

Gennaio 2010



Indice

Studia prima la scienza, e poi seguita la pratica, nata da essa scienza. Quelli che s'innamoran di pratica senza scienza son come 'l nocchier ch'entra in naviglio senza timone o bussola, che mai ha certezza dove si vada

LEONARDO

Capitolo 1

Basi di dati e sistemi informativi

L'informatica individuale ha trasformato il computer in un elettrodomestico di uso comune. Il Personal Computer (PC) è un oggetto familiare e universalmente conosciuto. Le applicazioni come *internet*, *posta elettronica*, *social networks* fanno parte dell'esperienza di ognuno.

Meno nota invece, ma di grande impatto ed importanza è un altro tipo di informatica. Quella che chiameremo *informatica industriale*, l'informatica delle grandi aziende come banche, industrie manifatturiere o di servizi o delle organizzazioni sociali, come scuole, ospedali, centri di ricerca o enti governativi.

Queste organizzazioni hanno bisogno, per poter funzionare in modo corretto ed efficiente, di immagazzinare ed elaborare grandi quantità di informazioni. Un classico esempio è quello di una banca. In tale contesto occorre mantenere in modo affidabile e corretto le informazioni relative ai depositi dei clienti. Inoltre occorre mantenere aggiornate tali informazioni ogniqualvolta vengono effettuati movimenti, in entrata o in uscita, dai depositi della banca. Un altro esempio lo abbiamo nella gestione degli stipendi di una grande multinazionale. In questo caso, l'azienda deve mantenere le informazioni di migliaia o centinaia di migliaia di dipendenti e per poter elaborare ogni mese le buste paga, deve gestire una quantità molto grande di dati, che varia rapidamente nel tempo: occorre contabilizzare le ore effettivamente prestate, le ferie, le eventuali malattie e le ore di straordinario effettuato di ciascun dipendente; occorre tenere conto della fiscalità, delle detrazioni in funzione del reddito percepito e così via.

In generale le realtà industriali e le grandi organizzazioni, per poter svolgere correttamente le loro attività, dalla vendita di prodotti e servizi, al supporto dei processi interni, dall'amministrazione della contabilità all'elaborazione delle buste paga, hanno bisogno di manipolare e gestire grandi quantità di informazioni.

All'interno di queste organizzazioni sono quindi presenti sistemi e procedure che hanno come obiettivo centrale quello di gestire e conservare tali informazioni vitali per il funzionamento corretto ed efficiente dell'organizzazione stessa.

Con l'avvento delle tecnologie e delle scienze informatiche è stato possibile gestire, memorizzare e disseminare tali informazioni in modo sempre più effi-

ciente ed economico. Il complesso di sistemi hardware e software dedicati alla gestione ed alla memorizzazione dei dati viene detto *sistema informativo*.

Oggi non esiste azienda od organizzazione che non possegga al proprio interno uno o più (in alcuni casi centinaia o migliaia) sistemi informativi. Si può dire che se l'informatica individuale è costituita essenzialmente dal PC, allora l'informatica industriale è costituita dai sistemi informativi.

Sebbene i sistemi informativi siano composti da diversi apparati hardware e software come server, reti di interconnessione, sistemi di sicurezza e controllo, apparecchiature per la memorizzazione dei dati, sistemi operativi e procedure applicative, come già detto sopra il loro scopo ed obiettivo principale è quello di gestire ed immagazzinare informazioni.

La *base di dati*, ovvero l'insieme delle informazioni gestite ed immagazzinate rappresenta il cuore e l'elemento centrale del sistema informativo.

Nel corso di Basi di Dati - II Modulo vogliamo studiare i principi e le tecniche di progettazione e di implementazione di una base di dati. Non ci occuperemo di tutti gli altri elementi che compongono un sistema informativo industriale, dal server fisico, al sistema operativo, dai linguaggi di programmazione all'architettura di rete e via dicendo, visto che la base di dati è quell'elemento sempre presente e centrale all'interno di qualunque sistema informativo.

Introdurremo una metodologia di progettazione e modelli per le basi di dati, proposti, in ambito accademico e scientifico, agli inizi degli anni 70 che hanno dato prova, nella pratica, di robustezza, efficacia e facilità di uso. Tali metodologie e modelli sono ormai universalmente adottati, nella pratica industriale, per la progettazione ed implementazione delle basi di dati.

Il problema che si pone nella fase di progettazione ed implementazione di sistema informativo e di conseguenza di una base di dati, è quello di progettare il sistema in modo **corretto** ed **efficiente**.

Il requisito di correttezza implica che la base di dati contenga sempre solo e soltanto quelle le informazioni che effettivamente rappresentano la realtà di interesse. Ad esempio nel caso di una banca saremo interessati a fare sì che i dati relativi ai depositi siano corrispondenti all'effettivo deposito in denaro presente sui conti correnti e che le transazioni che vengono effettuate siano riportate correttamente.

L'efficienza implica che il sistema possa essere implementato in modo rapido ed economico e che i successivi costi di gestione e manutenzione siano minimizzati, soddisfacendo al contempo i requisiti di correttezza di cui sopra. In altri termini il committente di un sistema informativo ha interesse che il sistema sia realizzato nel più breve tempo possibile minimizzando i costi complessivi, sia quelli di sviluppo dei sistemi software che quelli dell'hardware necessario.

Vedremo che la metodologia di progettazione e realizzazione delle basi di dati descritta nei successivi capitoli ha dato prova di soddisfare al meglio i requisiti di correttezza ed efficienza suddetti.

1.1 Ciclo di vita dei sistemi informativi

Per meglio introdurre la metodologia di progettazione di una base di dati è utile vedere quali sono le fasi di vita di un sistema informativo industriale, riportate

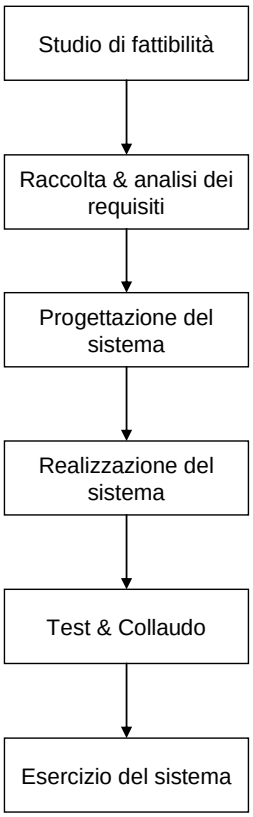


Figura 1.1: Il ciclo di vita dei sistemi informativi

in Fig. 1.1.

Studio di fattibilità. In questa fase si analizzano per grandi linee i requisiti del sistema informativo e si valutano le possibili scelte in termini tecnologici ed economici per la realizzazione del sistema. Ad esempio si decidono il tipo e le dimensioni di massima dei sistemi hardware; si definiscono i costi ed i tempi complessivi di sviluppo del software e di approvvigionamento dell'hardware.

Raccolta ed analisi dei requisiti. In questa fase vengono raccolte ed analizzate nel dettaglio tutte le richieste del committente ed i requisiti che il sistema informativo deve soddisfare. Si definiscono quali dati devono essere memorizzati e quali operazioni e applicazioni occorre sviluppare per l'accesso e la gestione dei dati.

Progettazione del sistema. Sulla base dei requisiti espressi nella fase precedente in questa fase avviene la progettazione del sistema in tutte le sue componenti, dalla base di dati ai programmi applicativi. Inoltre si definiscono nel dettaglio le esigenze di acquisto sia del software di base (sistemi operativi, protocolli di rete, compilatori, ecc.) che dell'hardware necessario.

Realizzazione del sistema. In questa fase, sulla base delle specifiche progettuali, viene creata e popolata la base di dati e vengono implementati i programmi applicativi. Inoltre viene installato il complesso di sistemi hardware e software del sistema informativo.

Test e collaudo. In questa fase il sistema nel suo complesso viene sottoposto a test di funzionalità e prestazionali. Particolare attenzione viene dedicata ai test del software applicativo.

Esercizio del sistema. In questa fase il sistema entra in produzione ed avvia i suoi servizi.

Tutte queste fasi non vengono nella pratica eseguite in stretta cascata ma possono esserci dei ricircoli tra una fase e la precedente. Ad esempio in caso di test o collaudo negativi occorre ritornare nella fase implementativa e modificare il software non funzionante. Se la semplice modifica del software non è in grado di risolvere il problema occorre ritornare alla fase progettuale e così via.

Come già detto nel precedente paragrafo, andremo ad analizzare solo le fasi che riguardano la progettazione ed implementazione della base di dati del sistema informativo. Non ci occuperemo di descrivere le attività svolte durante la raccolta ed analisi dei requisiti utente, poiché essendoci nella pratica innumerevoli situazioni, molto differenti tra di loro, è difficile standardizzare le attività che vengono svolte durante questa fase.

1.2 Progettazione ed implementazione di una base di dati

L'obiettivo della progettazione di una base di dati è quello di rappresentare la struttura dei dati di un frammento del mondo reale. Tale frammento del mondo reale è un insieme di fatti, cose, persone, elementi, processi o altro specificato dal committente del sistema informativo.

Il frammento del mondo reale di cui si vogliono memorizzare i dati verrà chiamato *mini-mondo*. Questo viene descritto dal committente del sistema informativo e viene raccolto dal progettista in un documento chiamato *specifiche utente sui dati*. Tipicamente tale documento viene realizzato mediante una o più interviste al committente del sistema informativo. Oltre alla descrizione dei dati che devono essere contenuti nella base di dati vengono fornite dal committente le specifiche su quali operazioni devono essere effettuate sulla base di dati (operazioni di modifica o lettura), con quale frequenza e quali vincoli prestazionali devono essere soddisfatti. L'insieme di queste specifiche viene detto *specifiche funzionali*. Queste verranno date in input al processo di progettazione ed implementazione dei programmi applicativi del sistema. Le specifiche funzionali vengono utilizzate però anche nella progettazione della base di dati. Infatti in funzione della tipologia delle operazioni, della loro frequenza, dei volumi della base di dati e dei vincoli prestazionali (come il tempo massimo o medio di risposta del sistema ad una operazione) si dovranno effettuare opportune scelte progettuali al fine di realizzare una base di dati idonea a rispettare i vincoli

prestazionali richiesti.

La progettazione della base di dati procede generalmente in parallelo alla progettazione dei programmi applicativi come indicato in Fig. 1.2. Per poter soddisfare i requisiti di correttezza ed efficienza la progettazione della base di dati viene suddivisa in tre fasi distinte, la progettazione *concettuale*, la progettazione *logica* e la progettazione *fisica* come indicato in Fig. 1.2.

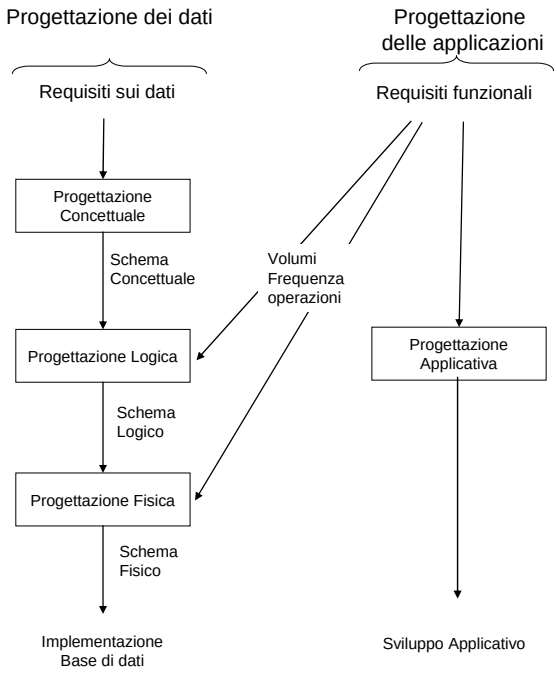


Figura 1.2: Le fasi della progettazione della base di dati

Nella progettazione concettuale e nella progettazione logica vengono utilizzati i modelli, rispettivamente concettuale e logico, i quali forniscono gli strumenti per realizzare la progettazione. Nella progettazione fisica si decidono quali strutture dati utilizzare e come organizzare fisicamente i dati al fine di rendere efficienti le operazioni.

Per comprendere la differenza tra le tre differenti fasi si prenda come esempio un sistema informativo (anche se non informatico) a tutti noto che è quello dell'elenco telefonico.

Il livello fisico di un elenco telefonico è il libro cartaceo che tutti conosciamo. La progettazione fisica, in questo caso consiste nello scegliere il tipo di carta, lo spessore, la dimensione, grandezza e peso dei fogli, ecc.. Significa determinare, ad esempio, il tipo d'inchiostro tipografico, la grandezza delle lettere da stampare e l'ordine in cui verranno stampati i dati degli abbonati sull'elenco. Tutte le scelte che vengono fatte in questa fase, soddisfano esigenze di economicità ed efficienza dell'utilizzo dell'elenco. Ad esempio l'ordine con cui vengono stampate le utenze facilita il tipo di ricerca più comune, che è quello di trovare il telefono dell'abbonato sapendo il suo nome.

Al livello logico, l'elenco telefonico è visto come un insieme di dati (in forma tabellare) come il nome e cognome, l'indirizzo dell'abbonato ed il numero telefonico a lui intestato.

Si osservi come il livello logico ed il livello fisico siano indipendenti. In altre parole l'insieme dei dati degli abbonati esiste indipendentemente da come questi verranno poi stampati, di quale colore sarà l'inchiostro o in quale ordine verranno poi disposti nell'elenco.

Al livello concettuale, l'elenco telefonico è visto come un insieme di utenze telefoniche. Ovvero a questo livello si considera l'insieme concreto, reale, delle utenze telefoniche attualmente attive.

Anche qui il livello concettuale è indipendente da quello logico in quanto che una utenza telefonica esiste, come entità reale, a prescindere da quali dati elementari verranno poi memorizzati per essa.

La progettazione concettuale prende come input l'insieme delle specifiche sui dati e utilizzando un modello concettuale, nel nostro caso il modello *Entity-Relationship*, produce uno schema concettuale della base di dati, rappresentato mediante un diagramma grafico.

Tale diagramma, insieme ad altra documentazione, viene fornito come input alla progettazione logica, la quale traduce il diagramma concettuale in uno schema logico. Per fare ciò utilizza il modello logico *Relazionale*.

Infine lo schema prodotto nella fase di progettazione logica, viene utilizzato nella progettazione fisica, insieme alle informazioni relative ai volumi, alle operazioni ed alla frequenza delle operazioni. In questa fase, anche tenendo conto degli apparati hardware a disposizione, si definiscono quali strutture fisiche utilizzare per l'implementazione della base di dati.

Capitolo 2

Il modello Entità-Associazione

La progettazione concettuale di una base di dati viene realizzata utilizzando il modello Entity-Relationship (che tradurremo in modello Entità-Associazione poiché il termine 'relazione' è utilizzato nel modello relazionale). Tale modello mette a disposizione del progettista un insieme di strumenti i quali consentono di descrivere in modo coinciso e formale le categorie del mondo reale delle quali si vogliono memorizzare i dati.

2.1 Caratteristiche del modello concettuale

Il modello Entità-Associazione è stato proposto nella metà degli anni '70 come strumento per la progettazione delle basi di dati da P.Chen.

L'utilizzo del modello Entità-Associazione nella progettazione concettuale consente di esprimere le specifiche utente sui dati - che sono scritte in linguaggio naturale e perciò prono a interpretazioni soggettive o fraintendimenti - per mezzo di un insieme di costrutti formali e coincisi, i quali non lasciano spazio a fraintendimenti ed interpretazioni. Esistono infatti prodotti software sul mercato che consentono di disegnare il diagramma Entità-Associazione e che forniscono automaticamente, a partire da questo, l'implementazione dello schema della base di dati.

Inoltre l'utilizzo del modello Entità-Associazione prescinde dal sistema fisico che verrà utilizzato per gestire la base di dati e prescinde dal modello logico con cui si vogliono rappresentare i dati. Ciò, consente al progettista, di concentrarsi sull'analisi delle specifiche utente senza dover considerare i dettagli implementativi e le strutture fisiche (strutture dati) e logiche (relazione e tabelle) che dovranno essere utilizzate per l'implementazione del sistema. In altri termini il modello concettuale Entità-Associazione fornisce un insieme di strumenti che consentono, in modo semplice, di stabilire formalmente e senza ambiguità *che cosa* si deve progettare, prima di considerare il *come* realizzarlo. Questo rende più efficace e robusto il processo di progettazione della base di dati. Inoltre il modello Entità-Associazione è fornito di una rappresentazione grafica dei suoi costrutti in modo che, il processo di modellazione consiste nella realizzazione di

uno schema diagrammatico di facile lettura e consultazione.

Di seguito vengono riportate le specifiche utente di due mini-mondi che utilizzeremo come esempi di specifiche utente nei seguenti capitoli.

Specifiche utente applicazione *Cinema*

Una catena di multisale cinematografiche possiede circa 20 multisale. Ogni multisala possiede mediamente 20 sale e ogni sala può contenere mediamente 250 posti numerati. I posti di ciascuna sala sono organizzati in file e all'interno di ogni fila sono numerati da 1 a n dove n è il numero di posti di una fila. Dei film in proiezione si conoscono il titolo, l'anno di produzione, il regista, gli attori principali e la durata. In media si proiettano 400 nuovi film all'anno.
Degli attori si vuole conoscere il nome, la data di nascita e i film interpretati o a cui si è partecipato. Di ogni sala si vuole conoscere il numero di posti totali. La programmazione dei film nelle sale deve avere una finestra temporale di 15 giorni a partire dalla data odierna. Le informazioni relative alla programmazione sono la sala, il film proiettato, la data e ora di inizio proiezione, la durata della proiezione, il numero dei posti disponibili e quello dei posti prenotati o acquistati. Ogni sala effettua in media 5 proiezioni al giorno e per ogni proiezione, la sala è piena mediamente al 60%.
I biglietti possono essere a costo intero o a costo ridotto e vengono distribuiti dal botteghino che stampa sopra ciascuno di essi, al momento dell'emissione, il numero, la data di emissione, la sala, la data e ora di inizio del film, la fila ed il numero del posto acquistato.

Specifiche utente applicazione *Porti*.

I porti (c.a 200) hanno generalmente il nome della città o località che li ospita, hanno una capitaneria di porto con un numero di telefono. Deve essere registrata la regione, la provincia e la profondità minima delle acque di ogni porto. I posti barca (mediamente 500 per porto) possono essere stati acquistati da un cliente oppure sono posti disponibili per i navigli in transito, sono numerati e vengono identificati dal loro numero e dal numero della banchina. In ogni istante, per un dato porto, si deve conoscere il numero di posti barca, disponibili ovvero non occupati da alcun natante, nel porto. Le barche (c.a 80.000) si suddividono in barche a vela e barche a motore. Ogni imbarcazione è identificata da un numero di targa, ha una stazza, una lunghezza fuori tutto ed un pescaggio (la profondità della barca misurata dal livello di galleggiamento) nonché il numero di posti. Per le navi a motore si deve registrare la potenza mentre le barche a vela si dividono in mono albero, bialbero o più di due alberi. Una barca che arriva nel porto deve indicare la data di arrivo e la data di partenza. Per una imbarcazione già transitata si vuole memorizzare la data effettiva di partenza (i dati dei transiti passati

devono essere consultabili in linea per almeno tre anni). I proprietari delle unità da diporto dovranno essere registrati con il nome, residenza, il numero della patente nautica eventualmente posseduto.

2.2 I costrutti del modello Entità-Associazione

2.2.1 Entità

Una entità rappresenta un **insieme** di elementi o di cose o fatti o persone di cui si vogliono memorizzare i dati. Ad esempio, con riferimento all'applicazione Cinema, l'insieme dei biglietti può essere rappresentato in una entità. L'insieme dei film è anche un esempio di entità. Ogni entità ha un nome che indica il tipo o la categoria dell'insieme rappresentato. Pertanto l'insieme dei biglietti darà luogo ad una entità chiamata BIGLIETTO. Parimenti l'insieme dei film proiettati o in proiezione nell'applicazione Cinema darà luogo ad una entità chiamata FILM. Graficamente rappresentiamo una entità mediante un rettangolo all'interno del quale è scritto il nome dell'entità come indicato nella Fig. 2.1.

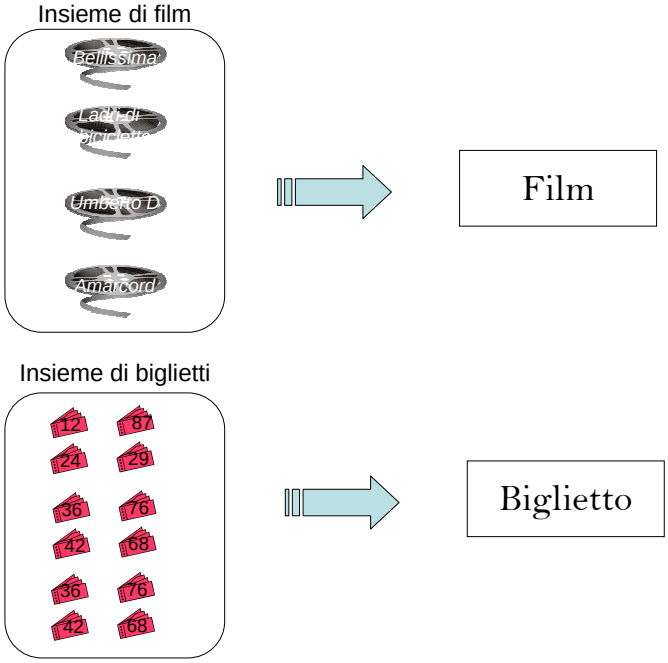


Figura 2.1: Rappresentazione della entità

L'insieme di elementi che compongono una entità è anche chiamato *istanza* dell'entità. Mentre ciascun elemento dell'entità verrà chiamato *occorrenza*. Quindi diremo che un film è un'occorrenza dell'entità FILM e un biglietto è un'occorrenza dell'entità BIGLIETTO. Nel seguito il nome dell'entità indicherà sia la tipologia degli elementi rappresentato, che l'istanza corrente, ovvero un insieme possibile di occorrenze. Si noti che l'insieme di elementi rappresentato dall'entità subisce variazioni nel tempo, dovute all'aumento o alla diminuzio-

ne degli elementi dell'insieme. Ad esempio, nel caso dell'entità BIGLIETTO, l'insieme dei biglietti venduti è previsto aumentare nel tempo.

Analogamente se l'insieme delle barche è rappresentato in una entità BARCA, tale insieme è soggetto a possibili incrementi o diminuzioni poiché nella realtà del mini-mondo l'insieme delle barche può effettivamente aumentare (in caso di nuove barche) o diminuire (in caso che una barca viene smantellata o affonda).

Il costrutto dell'entità consente di modellare la realtà del mini-mondo generando categorie (le entità) ciascuna delle quali rappresenta un **insieme** di elementi tutti dello stesso tipo, omogenei e con caratteristiche comuni tra di loro. Questa è una qualità fondamentale di una entità. Infatti non modelleremo una entità ATTORE_FILM (anzi ciò potrebbe portare ad una cattiva progettazione) dove ogni occorrenza dell'entità rappresenta l'attore ed un film da esso interpretato, poiché in tal caso ogni occorrenza sarebbe composta da concetti - film ed attori - differenti tra di loro.

Un altro aspetto che si deve notare è che, l'insieme degli elementi dell'entità esiste nel mini-mondo, prima ancora di aver deciso quali dettagli delle occorrenze dell'entità memorizzeremo nella base di dati. Ad esempio abbiamo definito l'entità BIGLIETTO poiché ci interessa memorizzare i dati dei biglietti, prima ancora di decidere quali effettivi dettagli vogliamo memorizzare per ciascun biglietto. Infatti saremo certamente interessati a memorizzare il costo e la data di emissione di ciascun biglietto. Ma tali dettagli possono essere individuati dopo aver costituito l'entità BIGLIETTO.

2.2.2 Associazioni

Una volta definite le entità come nel caso di BIGLIETTO, POLTRONA e PROIEZIONE ci interessa modellare il fatto che un certo biglietto è stato acquistato per una certa poltrona e per una certa proiezione. Così come, se abbiamo costituito una entità FILM ed una entità ATTORE, non abbiamo specificato in alcun modo quale attore ha interpretato quale film. Per modellare questi legami che mettono in relazione le occorrenze di diverse entità tra di loro, utilizziamo il costrutto dell'associazione.

Un'*associazione* tra due entità è un **insieme** costituito da un **sottoinsieme** del prodotto cartesiano delle istanze delle entità coinvolte. Ad esempio dovendo rappresentare quali attori hanno interpretato quale film utilizziamo un'associazione INTERPRETA. L'insieme di elementi rappresentato dall'associazione INTERPRETA è costituito da un insieme di coppie ordinate (f, a) dove f è un film e a un attore. Una coppia siffatta è presente nell'associazione INTERPRETA se e solo se l'attore a ha interpretato il film f.

Un altro esempio di associazione è il seguente. Se modelliamo l'insieme delle sale cinematografiche con una entità SALA e l'insieme delle poltrone con un'altra entità POLTRONA allora l'associazione FA_PARTE_DI tre le entità SALA e POLTRONA rappresenta l'insieme delle coppie ordinate (s, p) dove s è una sala e p una poltrona che è presente nella sala s.

Un'associazione tra due entità è rappresentata diagrammaticamente da un rombo unito da linee alle entità che partecipano all'associazione (vedi Fig. 2.2). Anche per questo costrutto, utilizzeremo i termini *istanza*, per indicare l'insieme di elementi di un'associazione in un certo istante di tempo ed *occorrenza* per indicare un singolo elemento dell'associazione. Inoltre utilizzeremo il nome

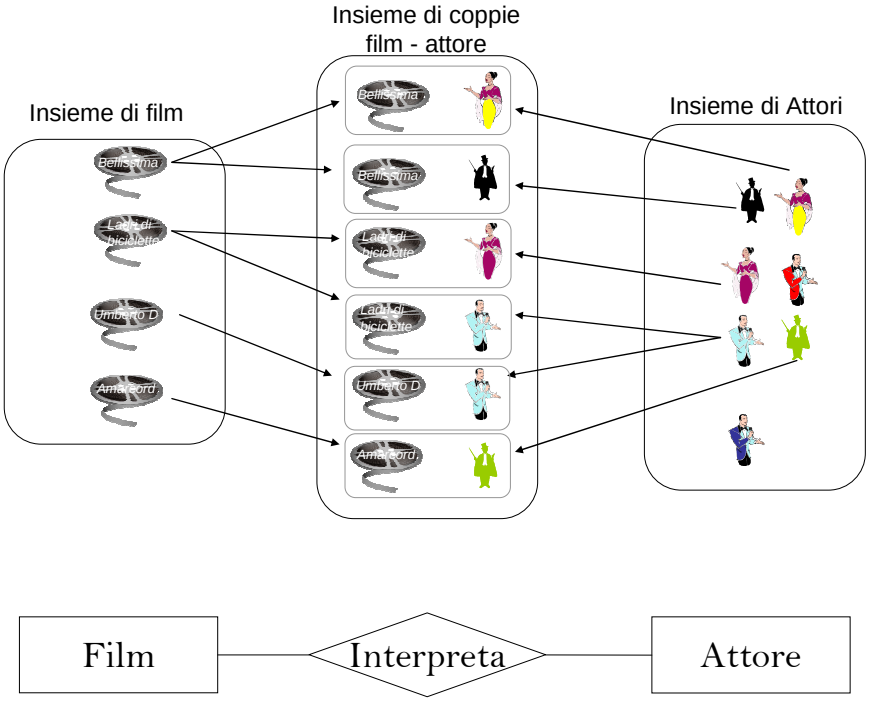


Figura 2.2: Rappresentazione delle associazioni. Associazione INTERPRETA tra FILM ed ATTORE.

dell'associazione, indifferentemente per indicare sia il tipo di associazione che una sua istanza.

Come nel caso dell'entità, gli elementi di un'associazione possono cambiare nel tempo. Ad esempio se f è un nuovo film interpretato dagli attori a_1 ed a_2 allora l'istanza dell'associazione INTERPRETA aumenta di due occorrenze, (f, a_1) ed (f, a_2). In un altro caso, se le poltrone p_1 e p_2 vengono rimosse fisicamente dalla sala s allora le occorrenze (s, p_1) ed (s, p_2) dovranno essere rimosse dall'istanza dell'associazione FA_PARTE_DI.

Possiamo definire associazioni tra più entità. Ad esempio potremmo definire un'associazione tra le entità BIGLIETTO, PROIEZIONE e POLTRONA ad indicare quali biglietti sono associati a quale proiezione e per quale poltrona in quella proiezione. Per fare ciò potremmo costituire un'associazione VISIONE tra queste entità. L'associazione VISIONE avrà come istanza un insieme di terne ordinate (pr, p, b) dove pr è una proiezione per la quale è stato acquistato un biglietto b con poltrona p come indicato in Fig. 2.3.

Più formalmente l'istanza di un'associazione A tra le entità $E_1, E_2, \ldots, E_k$ è costituita da un **sottoinsieme** del prodotto cartesiano $E_1 \times E_2 \times \cdots \times E_k$ delle istanze delle entità coinvolte. Si osservi che non tutti gli elementi del prodotto cartesiano fanno necessariamente parte dell'associazione, in quanto che alcuni di questi elementi non costituiscono un legame effettivamente presente nel mini-mondo.

È piuttosto facile fornire esempi concreti e tangibili di entità. Un film ed

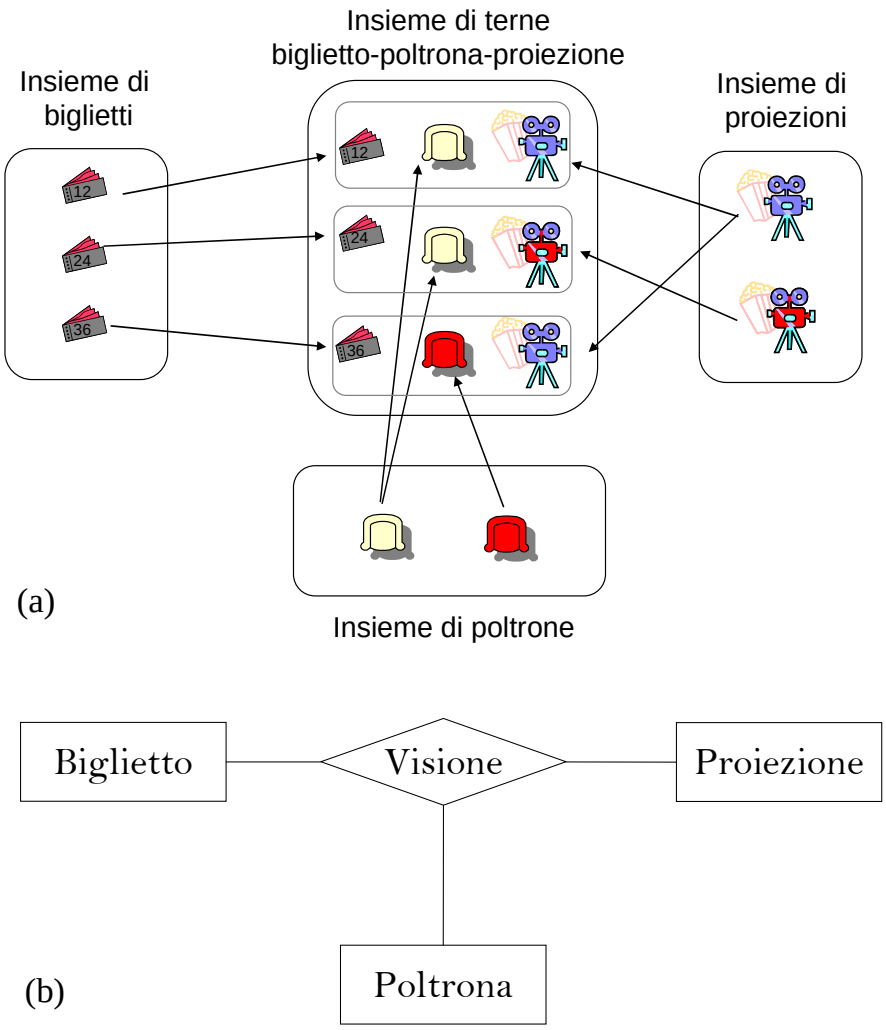


Figura 2.3: (a) Istanza di un associazione ternaria: associazione VISIONE. (b) Rappresentazione diagrammatica

un attore sono elementi concreti e reali. Per il costrutto dell'associazione non troviamo facilmente degli esempi così tangibili. Una coppia (f, a) dove f è un film e a un attore che ha interpretato il film f non esiste tangibilmente ma solo come legame concettuale tra due entità. Però un esempio pratico possiamo trovarlo in quanto segue. Una volta acquistato un veicolo, viene fornito al proprietario un documento denominato *certificato di proprietà*, che attesta la proprietà del veicolo stesso. All'interno di questo documento è necessario che vengano riportati gli estremi di identificazione dell'automezzo posseduto come la targa o il numero seriale dell'automezzo. Parimenti devono essere riportati gli estremi di identificazione del proprietario dell'automezzo come il nome ed il codice fiscale. Ora se modelliamo l'insieme degli automezzi con un entità AUTO e l'insieme dei proprietari di veicolo con una entità PERSONA per modellare concettualmente il fatto che una persona è proprietaria di un dato automezzo

utilizzeremo l'associazione PROPRIETÀ tra le entità AUTO e PERSONA. L'istanza di PROPRIETÀ è un insieme di coppie (a, p) dove a è un automezzo e p una persona che possiede l'automezzo a. L'associazione PROPRIETÀ fornisce il contenuto informativo concettuale equivalente a quello del certificato di proprietà. Conoscere infatti gli insiemi delle persone e l'insieme dei veicoli non ci dice però quale veicolo è intestato a quale persona. Questo fatto del mini-mondo viene pertanto modellato dall'associazione PROPRIETÀ.

Associazioni ricorsive Le entità coinvolte in una associazione non necessariamente devono essere distinte. Infatti possiamo avere associazioni, cosiddette *ricorsive*, tra una entità e se stessa. Ad esempio supponiamo di avere una entità PORTO che rappresenta un insieme di porti. Possiamo modellare le tratte servite da traghetti tra un porto ed un altro, mediante un'associazione TRATTA ricorsiva tra l'entità PORTO e se stessa. Quindi l'associazione TRATTA contiene un sottoinsieme del prodotto cartesiano PORTO $\times$ PORTO. Ovvero ogni occorrenza di TRATTA sarà costituita da una coppia (p_i, p_j) ad indicare una tratta tra il porto p_i ed il porto p_j. Avendo ogni tratta, un porto di partenza ed un porto di arrivo, possiamo assegnare alle coppie di TRATTA un *ruolo*. Ad esempio il primo elemento di ogni coppia ha il ruolo denominato *da* poiché rappresenta il porto di partenza mentre il secondo elemento ha il ruolo di a che rappresenta il porto di arrivo. Graficamente rappresentiamo ciò scrivendo i ruoli ai lati dell'associazione come indicato in Fig. 2.4.

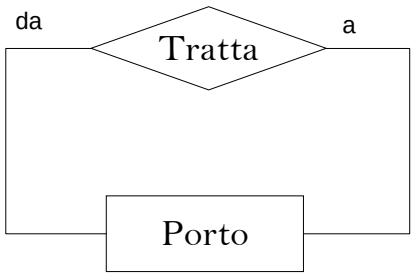


Figura 2.4: Associazioni ricorsive

Corretto utilizzo delle associazioni. Occorre fare attenzione nell'utilizzo del costrutto dell'associazione in quanto che un impiego superficiale di questo può generare dei diagrammi non corretti. Ad esempio supponiamo di avere due entità BARCA e PORTO che rappresentano rispettivamente un insieme di barche ed un insieme di porti. Se vogliamo registrare quali barche sono in sosta in quale porto modelleremo tale fatto mediante una associazione SOSTA tra le due entità. Ogni occorrenza di SOSTA sarà una coppia (p, b) dove p è un porto dove è presente in sosta una barca b.

Ora supponiamo di voler memorizzare anche le soste passate. In altri termini non vogliamo solo sapere quali barche sono in sosta *attualmente* in un porto ma vogliamo anche conservare i dati delle soste passate. Se volessimo modellare tale informazione con un'associazione ORMEGGIO tra BARCA e PORTO avremmo un utilizzo scorretto del costrutto dell'associazione. Infatti, per definizione, l'istanza di una associazione è un sottoinsieme del prodotto cartesiano delle entità

coinvolte. Pertanto se una barca b transita una volta in un porto p questo fatto aggiunge una occorrenza nell'associazione ORMEGGIO costituita dalla coppia (p, b). Se la stessa barca transita una seconda volta nello stesso porto, l'insieme delle occorrenze di ORMEGGIO dovrebbe aumentare di una unità. Ma poiché il secondo transito dovrebbe essere rappresentato con la coppia (p, b) che è la stessa occorrenza del primo transito l'insieme delle occorrenze di ORMEGGIO non aumenterebbe.

In questa situazione la soluzione corretta è quella di utilizzare un'entità TRANSITO per rappresentare l'insieme dei transiti sia attuali che passati e legare tale entità mediante le associazioni APPRODO e ATTRACCO tra PORTO e BARCA come indicato in Fig. 2.5.

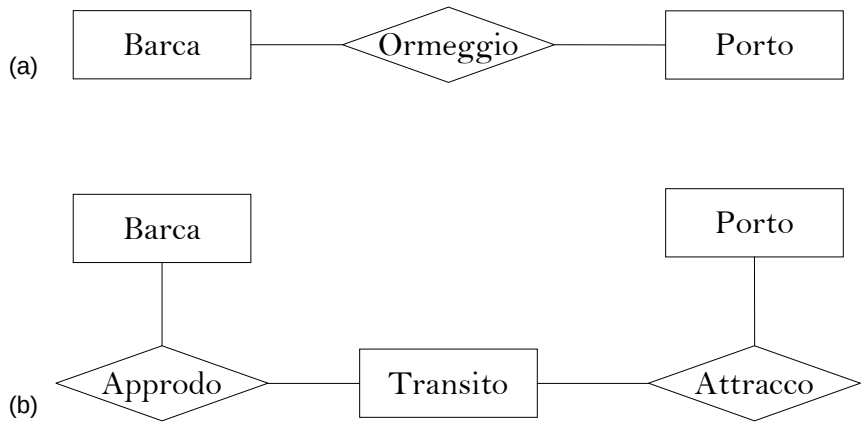


Figura 2.5: (a) Uso scorretto del costrutto dell'associazione per rappresentare i transiti passati. (b) Il diagramma corretto

2.2.3 Cardinalità delle associazioni

Le associazioni sono soggette ad alcuni vincoli rispetto alla quantità di occorrenze, delle entità coinvolte, che possono partecipare nell'associazione. Ad esempio nell'associazione INTERPRETA tra FILM ed ATTORE sappiamo che un attore può aver interpretato molti film e che in un film può esserci o nessun attore (ad esempio un documentario naturalistico) oppure ce ne possono essere più di uno. Per poter rappresentare tali vincoli, ad ogni associazione specificheremo, per ciascuna entità coinvolta, una coppia di numeri (min, max), che chiameremo *cardinalità delle associazioni*. Il primo valore della coppia, chiamato *cardinalità minima*, rappresenta il numero minimo di partecipazioni di ogni occorrenza dell'entità nell'associazione. Ad esempio se abbiamo una associazione PROGRAMMA tra le entità FILM e PROIEZIONE e se ogni film viene programmato in almeno cinque proiezioni allora potremo indicare con $min = 5$ il valore della cardinalità minima per l'entità FILM come in Fig. 2.6(b).

In genere è raro che si conoscano queste informazioni del mini-mondo nella fase di progettazione concettuale e che sia quindi possibile indicare con precisione la cardinalità minima. Pertanto, nella maggior parte dei casi vengono utilizzati i valori 0 ed 1. Nel primo caso parleremo di *partecipazione opzionale* delle occorrenze dell'entità nell'associazione, poiché può esistere una occorrenza dell'entità

che non partecipa all'associazione. Nel secondo caso si indica la *partecipazione obbligatoria* di ciascuna possibile occorrenza dell'entità all'associazione, ovvero ogni entità partecipa ad almeno una occorrenza dell'associazione. Ad esempio l'associazione SOSTA tra BARCA e PORTO di Fig. 2.6(d), che indica quale barca è ormeggiata in quale porto, avrà una cardinalità minima pari a zero poiché è possibile che alcune barche non siano attualmente ormeggiate in un porto.

Il secondo valore, chiamato *cardinalità massima*, indica il massimo numero di occorrenze dell'associazione a cui può partecipare un'occorrenza dell'entità.

Ad esempio se nella realtà del mini-mondo Cinema, un film può essere proiettato in al più duecento proiezioni allora potremmo utilizzare come cardinalità massima per FILM nell'associazione PROGRAMMA il valore $k = 200$ come indicato in Fig. 2.6(b).

Poiché, anche qui, in genere non è noto o non è possibile determinare in fase di progettazione concettuale tale numero, allora si utilizzano in genere due soli valori 1 ed N. Nel primo caso si specifica che nessuna occorrenza dell'entità parteciperà a più di una occorrenza dell'associazione. Nel secondo caso si indica che può esistere almeno una occorrenza dell'entità che partecipa a due o più occorrenze dell'associazione.

Ad esempio se non è conosciuto a priori quante proiezioni verranno programmate per un film ma almeno una occorrenza di film partecipa in più di una proiezione, allora impostiamo la cardinalità massima pari a N come in Fig. 2.6(c).

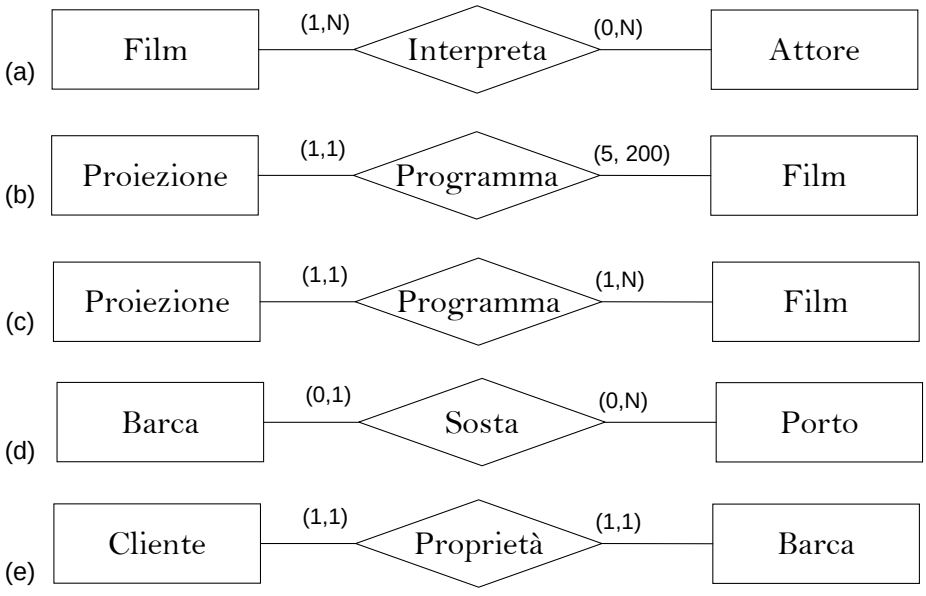


Figura 2.6: Esempi di cardinalità di associazioni. (a) molti-a-molti (b), (c) e (d) uno-a-molti (e) uno-ad-uno

Classificheremo le associazioni binarie in *molti-a-molti* se la cardinalità massima delle due entità che partecipano all'associazione è pari ad N. Parleremo invece di associazione *uno-a-molti* se la cardinalità massima di una delle due entità è pari ad 1 ed è pari ad N per l'altra entità. Mentre invece diremo che un'as-

sociazione è *uno-ad-uno* se la cardinalità massima di entrambe le associazioni è pari ad 1.

2.2.4 Attributi

Le entità e le associazioni consentono di identificare le categorie del mini-mondo ed i legami che intercorrono tra esse, per le quali vogliamo memorizzare i dati. A questo processo di categorizzazione deve seguire un processo per individuare quali dati elementari dobbiamo rappresentare nella base di dati per ciascuna occorrenza delle entità ed associazioni individuate. Per fare ciò viene utilizzato il costrutto dell'*attributo*. Questo ci consente di indicare proprietà elementari di ciascuna occorrenza dell'istanza di una entità o di una associazione.

Ad esempio l'entità BIGLIETTO possiederà degli attributi come, costo, data_emissione, numero ecc., che indicano proprietà elementari di ogni singola occorrenza dell'entità, ovvero di ogni singolo biglietto, che vogliamo vengano rappresentate nella base di dati. Parimenti se vogliamo rappresentare il compenso pagato all'attore a per la recitazione nel film f, utilizzeremo un attributo compenso nell'associazione INTERPRETA tra FILM ed ATTORE. Inoltre aggiungeremo un attributo personaggio, sempre nell'associazione INTERPRETA, per rappresentare il nome del personaggio che ha interpretato l'attore a in un film f.

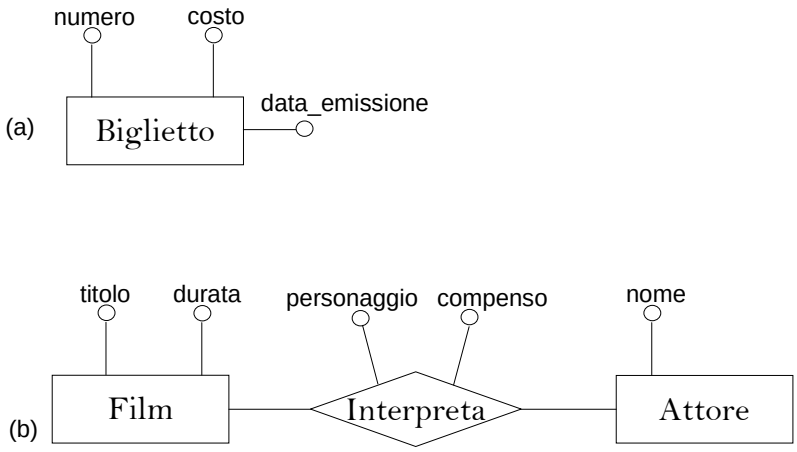


Figura 2.7: Esempi di attributi. (a) Attributi di entità (b) Attributi di entità ed associazioni

Diagrammaticamente disegniamo un attributo mediante una linea congiunta all'entità o all'associazione che termina con un cerchio in corrispondenza del quale verrà indicato il nome dell'attributo come indicato in Fig. 2.7.

Ad ogni attributo A associamo un un dominio, denotato con $dom(A)$ che consiste in un insieme di valori. Formalmente un attributo di una entità o associazione può essere rappresentato come una funzione che prende in input un'occorrenza dell'entità o dell'associazione e restituisce un valore appartenente al dominio dell'attributo.

Ad esempio l'attributo costo dell'entità BIGLIETTO ha un dominio *dom* (costo) che è specificato dall'insieme dei valori presenti nel listino dei costi dei

biglietti. Potremmo supporre, verosimilmente, che il listino dei costi dei biglietti abbia i seguenti valori

Tipo Biglietto	Costo
omaggio	€ 0.0
ridotto feriale	€ 2.5
feriale	€ 5.0
ridotto festivo	€ 5.0
festivo	€ 7.5

Pertanto costo è una funzione che prende in input una occorrenza dell'entità BIGLIETTO - un biglietto - e restituisce un valore appartenente al listino: il costo del singolo biglietto. Analogamente l'attributo compenso, che ha come dominio l'insieme dei razionali positivi, è una funzione che prende come input una occorrenza dell'associazione INTERPRETA - una coppia (f, a) dove a è un attore che ha interpretato il film f - e restituisce un valore numerico che è il compenso rilasciato all'attore a per la sua recitazione nel film f. Si osservi come in quest'ultimo caso non è possibile associare l'attributo compenso né all'entità FILM, poiché ogni film ha in genere più di un attore per il quale viene erogato un compenso, né può essere associato all'entità ATTORE, poiché in genere un attore ha interpretato diversi film, ottenendo un compenso diverso per ciascuno di essi.

Attributi multivalore

Consideriamo l'entità BARCA e supponiamo di avere necessità di inserire un attributo che specifica l'altezza dell'albero. Questo tipo di informazione si applica solo per le barche a vela e non per le barche a motore che sono sprovviste di un albero centrale. Un altro problema che si pone è che esistono barche a vela che posseggono più di un albero. Infatti possono averne due o nel caso dei grandi velieri, più di due. In questo caso vorremmo memorizzare l'altezza di ciascun albero posseduto. Per modellare ciò utilizziamo l'attributo multivalore, altezza_albero. Un attributo *multivalore* è rappresentato come un normale attri-

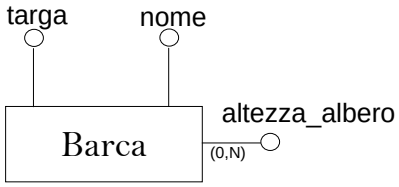


Figura 2.8: Esempio di attributo multivalore altezza_albero

buto solo che in corrispondenza di esso inseriamo due valori (min, max) come indicato in Fig. 2.8.

Tali valori possono essere visti come il corrispettivo della coppia di numeri nella cardinalità delle associazioni. Il numero *min* ammette il valore 0 od 1 ad indicare, nel primo caso, che esistono occorrenze dell'entità per i quali il valore dell'attributo non è esistente oppure non è noto o non è applicabile. In questo caso l'attributo assume un particolare valore chiamato valore NULL. Formalmente un attributo multivalore è una funzione che prende in input un'occorrenza

di entità od associazione e restituisce un insieme di valori tutti appartenenti al dominio dell'attributo. Nell'esempio precedente, il valore del primo numero è 0 poiché esistono occorrenze di barche - le barche a motore - che non hanno albero. Nel caso che *min* invece è 1, significa che per ogni occorrenza di entità tale attributo è presente. Il numero *max* può essere N ad indicare che esistono occorrenze dell'entità che posseggono più di un valore per quell'attributo. La coppia (min, max) viene omessa per gli attributi che ammettono sempre uno ed un solo valore per ogni occorrenza di entità.

2.2.5 Generalizzazioni

Supponiamo di avere una entità BARCA_A_VELA ed una entità BARCA_A_ MOTORE. Supponiamo che, in accordo con la descrizione del mini-mondo, non esistano barche che sono allo stesso tempo a vela e a motore. Queste due entità rappresentano quindi due insiemi distinti di imbarcazioni ed avranno in genere attributi distinti. Ad esempio le barche a vela possono avere un attributo relativo all'altezza dell'albero od alla superficie velica massima. Analogamente le barche a motore possono avere degli attributi specifici come, il tipo e la potenza del motore, l'autonomia in miglia ecc.. Insieme a queste entità supponiamo di avere una entità BARCA, che rappresenta l'insieme di tutte le barche descritte nel mini-mondo, e quindi anche le barche a vela e le barche a motore. Gli attributi dell'entità barca sono attributi comuni a tutte le barche come la targa, la lunghezza, la stazza, il pescaggio ecc..

È chiaro che una occorrenza dell'entità BARCA che è una barca a vela è anche una occorrenza dell'entità BARCA_A_VELA. Lo stesso dicasi per una occorrenza di BARCA che è una barca a motore.

Per rappresentare questo tipo di legame tra le entità BARCA, BARCA_A_VELA e BARCA_A_MOTORE utilizzeremo il costrutto della *generalizzazione*.

Nel costrutto della generalizzazione esiste una entità, chiamata entità *padre*, che generalizza una o più entità, chiamate entità *figlie*, che sono una specializzazione dell'entità padre. Nel caso precedente abbiamo l'entità BARCA che generalizza due entità, BARCA_A_MOTORE e BARCA_A_VELA le quali sono una specializzazione di BARCA. Indicheremo nel diagramma Entità-Associazione il costrutto della generalizzazione utilizzando una freccia che connette le entità generalizzate all'entità padre come indicato in Fig. 2.9. Occorre osservare che, come da definizione di una entità, sia l'entità padre che la entità figlie rappresentano insiemi di elementi. Questi insiemi non sono disgiunti poiché ogni occorrenza di una entità figlia è anche una occorrenza dell'entità padre. In genere non è vero il viceversa, visto che possono esserci generalizzazioni in cui una occorrenza dell'entità padre non è presente come occorrenza in nessuna delle entità figlie. Nell'esempio precedente ogni barca a vela o a motore, è anche una occorrenza di BARCA. Viceversa potrebbe essere che una barca non sia né barca a vela né a motore come nel caso di chiatte o barche a remi e quindi sia presente come occorrenza di BARCA ma non sia presente in nessuna delle entità figlie.

Quando ogni occorrenza dell'entità padre è anche occorrenza di una delle entità figlie, la generalizzazione si dice *totale*. Se invece esistono, come nel precedente esempio, alcune occorrenze dell'entità padre che non sono occorrenze di alcuna entità figlia allora la generalizzazione è *parziale*.

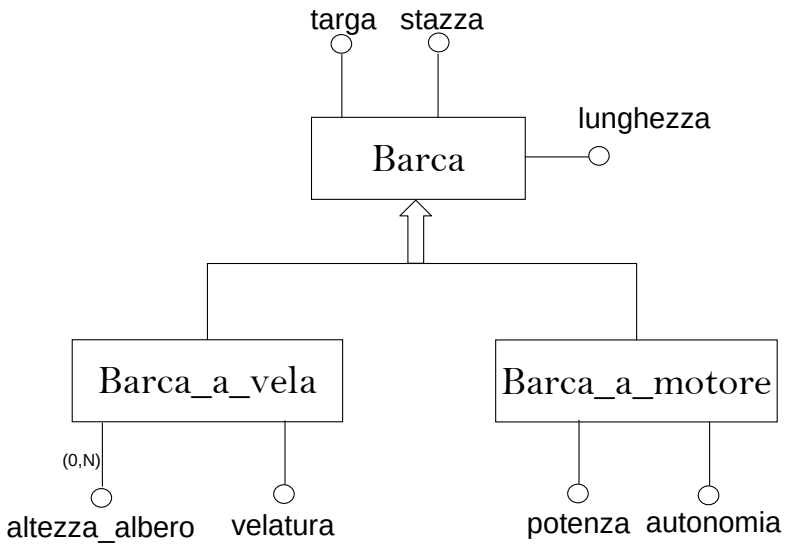


Figura 2.9: Un esempio di generalizzazione

Nel costrutto della generalizzazione tutti gli attributi dell'entità padre sono anche attributi delle entità figlie. Pertanto tali attributi non vengono riportati nelle entità figlie all'interno del diagramma Entità-Associazione.

Supponiamo ora che esistano delle barche a vela che sono anche barche a motore, chiamate anche *motovelieri*. In questo caso avremo che una occorrenza di un motoveliero nell'entità BARCA_A_VELA è anche una occorrenza nell'entità BARCA_A_MOTORE. Quindi le istanze di queste due entità figlie hanno intersezione non vuota. Quando succede che due o più entità figlie in una generalizzazione hanno intersezione non vuota, allora la generalizzazione si dice *sovrapposta*. Quando l'intersezione tra due qualunque entità figlie è sempre vuota allora la generalizzazione si dice *esclusiva*.

Saremo interessati a trattare preferibilmente generalizzazioni esclusive poiché quelle sovrapposte introducono una ridondanza nella rappresentazione. Se in una generalizzazione esistono due entità figlie che hanno intersezione non vuota, allora è possibile trasformare questa in una generalizzazione esclusiva, inserendo una nuova entità figlia che rappresenta l'insieme di occorrenze che appartengono ad entrambe le entità. Ad esempio nel caso di cui sopra possiamo inserire una nuova entità MOTOVELIERO che contiene solo e soltanto le barche che sono motovelieri. Di conseguenza le entità BARCA_A_VELA e BARCA_A_MOTORE rappresenteranno solo e soltanto insiemi di barche a vela pure o a motore senza vele, rispettivamente. Un'altra alternativa è quella di accorpare le occorrenze che compaiono in entrambe le entità in una delle due entità. Ad esempio possiamo catalogare i motovelieri come barche a motore. Ovvero ogni occorrenza di BARCA_A_MOTORE sarà o una barca a motore o un motoveliero. Viceversa ogni occorrenza di BARCA_A_VELA sarà esclusivamente una barca a vela pura.

2.2.6 Identificatori

Un insieme non vuoto $I = \{A_1, \ldots, A_k\}$ di attributi di una entità è un *identificatore* se prese due qualunque possibili distinte occorrenze di entità e_a ed e_b allora esiste un $1 \leq i \leq k$ tale che il valore dell'attributo A_i per la occorrenza e_a è diverso dal valore dell'attributo A_i per l'occorrenza e_b. Ad esempio supponiamo di avere una entità PERSONA che possiede un attributo codice_fiscale. Non esistono due persone per le quali il valore del codice fiscale coincide. Quindi, in accordo con la definizione, l'insieme {codice_fiscale} è un identificatore di persona. Supponiamo che l'entità FILM abbia due attributi titolo e anno, che indicano rispettivamente il titolo del film e la data di produzione. Verosimilmente non esistono due film con lo stesso titolo oppure se hanno lo stesso titolo (ad esempio nel caso di un 'remake') hanno date di produzione differenti. Due film con la stessa data di produzione, non avranno lo stesso titolo e come già detto due film con lo stesso titolo, avranno necessariamente una data di produzione differente. Quindi l'insieme {titolo, anno} rappresenta un identificatore dell'entità FILM.

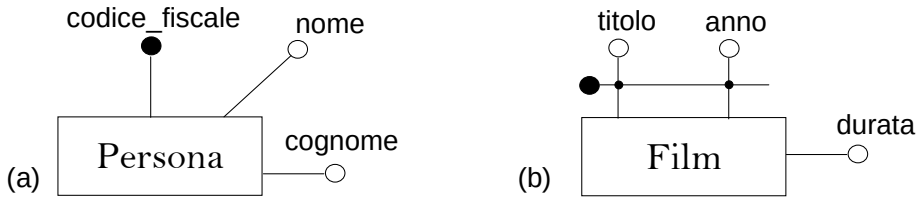


Figura 2.10: (a) L'identificatore {codice_fiscale} composto solo di un attributo (b) L'identificatore {titolo, anno} con più di un attributo

Gli identificatori verranno denotati nel diagramma Entità-Associazione come indicato nella Fig. 2.10(a) se l'identificatore è composto da un solo attributo o come nella Fig. 2.10(b) se l'identificatore è composto da più di un attributo.

A ben vedere un identificatore è semplicemente una proprietà di un insieme non vuoto di attributi di una entità. Nonostante ciò, vista l'importanza che gli identificatori giocano durante la fase di traduzione dal modello Entità-Associazione nel modello relazionale, è uso considerare gli identificatori come costrutti distinti del modello concettuale.

Identificatori esterni

Supponiamo di avere una entità POLTRONA che rappresenta l'insieme delle poltrone del mini-mondo dell'applicazione cinema. Ad ogni poltrona viene assegnata una lettera, che indica la fila a cui appartiene la poltrona ed un numero, che indica la posizione della poltrona all'interno della fila. Quindi l'entità POLTRONA possiede un attributo fila ed un attributo numero. Tali attributi però non sono sufficienti a identificare una poltrona, in quanto che potremmo avere due o più poltrone con stessa fila e stesso numero, appartenenti a sale differenti. Per poter identificare perciò univocamente una poltrona abbiamo bisogno di conoscere la sala a cui appartiene. In questo caso l'entità POLTRONA ha un *identificatore esterno* determinato dall'identificatore dell'entità SALA. In altre parole l'identificatore di poltrona sarà costituito dall'insieme di attributi {fila, numero, codice_sala} dove codice_sala è l'identificatore dell'entità SALA. Questo

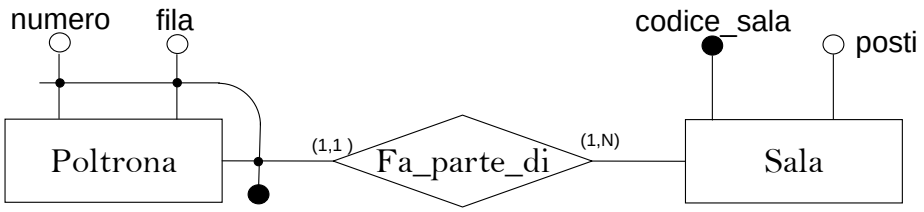


Figura 2.11: Esempio di entità con identificatore esterno

tipo di identificatore viene rappresentato nel diagramma Entità-Associazione nel come indicato nella Fig. 2.11.

2.3 Esercizi

Generare il diagramma Entità-Associazione delle specifiche negli Esercizi dal 2.1 al 2.7

Esercizio 2.1 [Social network] Per un sito internet social network si vogliono memorizzare i dati degli utenti, il nome, il sesso, l'età e un nickname. Ogni utente ha a disposizione uno spazio web dove memorizzare la propria home-page. Nella home page è possibile inserire e modificare diverse pagine html per le foto e per i filmati. Ogni utente possiede una lista di amici. Gli utenti possono formare tra di loro dei gruppi. Un gruppo ha un nome ed una bacheca.

Esercizio 2.2 [Pizzeria] In una pizzeria il personale da tavolo distribuisce l'elenco delle pizze disponibili, dei fritti e delle bevande. I clienti della pizzeria siedono in tavoli numerati. Ogni tavolo ha un certo numero di posti ed effettua una ordinazione tra l'elenco delle pizze dei fritti e delle bevande disponibili. Il personale raccoglie le ordinazioni e le invia alla cucina. Quando le ordinazioni sono pronte il personale consegna le pizze e riscuote il conto.

Esercizio 2.3 [Battaglie storiche] Si vogliono memorizzare i dati delle battaglie storiche e dei popoli / nazioni che le hanno combattute. Occorre memorizzare il nome della nazione o popolo e la sua capitale. Di ogni battaglia effettuata occorre memorizzarne la data il luogo, la quantità truppe che hanno combattuto e il numero dei caduti da entrambe le parti.

Esercizio 2.4 [Gestione logistica] Un insieme di magazzini viene utilizzato per contenere dei prodotti. I prodotti vengono fabbricati e depositati nei magazzini prima della loro vendita. Di ogni magazzino va indicato il codice e l'indirizzo. I trasportatori prelevano i prodotti dai magazzini e li inviano ai negozi per la loro vendita. Per ogni fabbrica va memorizzato il codice, la città e l'indirizzo dove la fabbrica risiede. Di ogni negozio va indicato il nome, il proprietario, la città e l'indirizzo. Di ogni prodotto occorre conoscere la marca, il modello, il numero di serie e la locazione del magazzino dove è depositato.

Esercizio 2.5 [Compagnia aerea] Una compagnia aerea offre voli che possiedono un numero che identifica una tratta (per es. Napoli-Bari) una data un orario di partenza ed un orario di arrivo un aeroporto di partenza ed un aeroporto di

arrivo. Ci sono voli nazionali e voli internazionali. I voli internazionali possono avere più uno o più scali. Dei voli passati è di interesse l'orario reale di partenza e di arrivo. Di quelli futuri è di interesse il numero di posti disponibile.

Esercizio 2.6 [Casa automobilistica] Una casa automobilistica produce veicoli che possono essere automobili, motocicli, camion e trattori. I veicoli sono identificati da un numero di telaio e hanno una cilindrata, una lunghezza ed un colore. Le automobili si suddividono in utilitarie e familiari e vengono classificate in base alla loro cilindrata: piccola (<1200) , media (da 1200 a 2000) e grossa (>2000). I motocicli sono suddivisi in motorini (cc< 125) e moto (cc>125). I camion hanno un peso e possono avere un rimorchio.

Esercizio 2.7 [Anagrafe] Per l'anagrafe di un ipotetico comune vanno memorizzati

- Informazioni sui cittadini nati e residenti nel comune: ogni cittadino è identificato dal codice fiscale e ha nome , cognome, sesso, età e data di nascita. In oltre per i nati in comune sono registrati gli estremi di nascita (numero registro e pagina) per quelli nati in altri comuni il comune di nascita.
- Informazioni sulle famiglie residenti, ognuna delle quali ha uno e un solo capo famiglia e zero o più altri membri per ognuno dei quali è indicato il grado di parentela (coniuge, figlio, ecc.); ogni cittadino appartiene ad una ed una ed una sola famiglia e tutti i membri di una famiglia hanno un solo domicilio (via, numero civico, interno).

Capitolo 3

Il modello Relazionale

Il modello *relazionale*, introdotto da T. Codd negli anni '70, per la rappresentazione logica delle basi di dati, si è affermato come il modello logico più diffuso nella gestione ed implementazione delle basi di dati.

Il modello relazionale ha avuto uno straordinario successo da quando è stato introdotto. Si pensi che la quasi totalità sistemi industriali per la gestione delle basi di dati si basa proprio su questo modello. Tale successo è dovuto alla sua semplicità e flessibilità: infatti per mezzo del modello relazionale è possibile rappresentare efficacemente qualsiasi insieme di dati del mondo reale. Un secondo motivo di successo del modello relazionale è conseguente al fatto che l'utilizzo di questo consente di separare gli aspetti logici della base di dati dagli aspetti implementativi. Ciò permette di progettare la base di dati senza doversi preoccupare degli aspetti relativi all'implementazione fisica della base dati, rendendo la progettazione in questo modo più semplice ed efficiente.

3.1 Descrizione del modello Relazionale

Sia dato un insieme $R = \{A_1, A_2, \ldots, A_n\}$ di stringhe o nomi che chiameremo *attributi*. Ad ogni attributo A_i associamo un insieme finito o infinito ma numerabile che chiameremo *dominio* dell'attributo e che verrà denotato con $dom(A_i)$. L'insieme R e la definizione dei domini di ciascun attributo di R, viene detto *schema di relazione*. Una *ennupla* su R è una funzione t che associa ad ogni attributo di R un valore appartenente al dominio dell'attributo. In altri termini

$$t(A_i) \in dom(A_i) \qquad i = 1, \ldots, n$$

Se X è un sottoinsieme non vuoto di attributi di uno schema R e t una ennupla su R con la notazione $t[X]$ indicheremo una ennupla su X, ovvero definita solo sugli attributi di X e che, su questi attributi, prende lo stesso valore di t. La ennupla $t[X]$ sarà chiamata *restrizione* su X di t.

Esempio 3.1 Ad esempio sia dato l'insieme di attributi BARCA ={targa, nome, lunghezza}. Sia il dominio di targa l'insieme delle stringhe alfanumeriche di lunghezza 10, quello di nome l'insieme delle stringhe alfabetiche di lunghezza 15, e il dominio di lunghezza l'insieme dei razionali non negativi. Allora una ennupla t su schema BARCA può essere

$$\begin{aligned} t(\text{targa}) &= \texttt{RM12345ABC} \\ t(\text{nome}) &= \texttt{Vespucci} \\ t(\text{lunghezza}) &= \texttt{154,32} \end{aligned}$$

La restrizione di t all'insieme di attributi $X = \{\text{nome, lunghezza}\}$ è la ennupla

$$\begin{aligned} t[X](\text{nome}) &= \texttt{Vespucci} \\ t[X](\text{lunghezza}) &= \texttt{154,32} \end{aligned}$$

■

L'insieme di tutte le possibili ennuple su R viene detto *dominio* di R e denotato con $dom(R)$. Una *relazione* r con schema R è un **sottoinsieme** finito di ennuple appartenenti a $dom(R)$ ovvero

$$r \subseteq dom(R)$$

Esempio 3.2 Con riferimento allo schema BARCA sopra definito, una possibile relazione su BARCA potrebbe essere

BARCA	targa	nomebarca	lunghezza
	RM12345ABC	Vespucci	154,32
	GE13345ABC	Rex	127,52
	VE14345ABC	Azzurra	62,73
	BA14375ABC	Azzurra	15,00

■

In genere una relazione può essere rappresentata come nell'esempio precedente, con una struttura tabellare, dove alle colonne corrispondono gli attributi dello schema ed alle righe corrispondono le ennuple della relazione. Si osservi però che diversamente da una tabella, in una relazione l'ordine delle righe non è né prefissato né assume una qualche importanza. Allo stesso modo l'ordine di definizione degli attributi non è importante.

Si può osservare che una relazione può variare nel tempo per rispecchiare i dati del mondo reale. Ad esempio se una barca si aggiunge (a causa di un nuovo varo) oppure una barca si distrugge (ad esempio in caso di smantellamento o naufragio) tali fatti saranno registrati aggiungendo in un caso un'opportuna ennupla nelle relazione BARCA o nell'altro caso, cancellando la corrispondente ennupla.

Una relazione si dice *valida* se tutte le ennuple rispecchiano correttamente tutti e soli i dati della realtà del mini-mondo. Per garantire la correttezza di una base di dati (e in generale di un sistema informativo) dato uno schema di relazione R con dominio $dom(R)$ siamo interessati ai soli sottoinsiemi finiti di $dom(R)$ che siano relazioni valide.

Esempio 3.3 Sia dato uno schema PORTO con attributi città, lat, e lon con domini rispettivamente l'insieme delle stringhe alfanumeriche di lunghezza 50, i numeri decimali con due cifre dopo la virgola compresi tra $-90,00$ e $+90,00$ ed i numeri decimali con due cifre dopo la virgola compresi tra $+180,00$ e $-180,00$. Inoltre sia ORMEGGIO uno schema con attributi {targa, città, data} dove i domini di targa e città coincidono con i domini degli attributi con lo stesso nome in BARCA e PORTO e dove data ha come dominio le stringhe alfanumeriche nella forma `aaaa/mm/gg` dove `aaaa`, `mm` e `gg` sono i numeri di anno, mese e giorno rispettivamente. Una possibile relazione con schema PORTO può essere

PORTO	città	lat	lon
	Genova	44,42	-8,90
	Venezia	45,26	-12,19
	Bari	41,08	-16,52

mentre una relazione con schema ORMEGGIO può essere

ORMEGGIO	città	targa	data
	Genova	GE13345ABC	2008/01/25
	Venezia	VE14345ABC	2009/10/11

■

Un insieme di schemi di relazione $\mathcal{D}$ viene detto *schema di base di dati relazionale*. Un insieme di relazioni è una *base di dati relazionale*. L'insieme {BARCA, PORTO, ORMEGGIO} è uno schema di base di dati relazionale mentre le relazioni degli esempi appena illustrati sono una base di dati relazionale.

3.1.1 Vincoli negli schemi relazionali

Vincolo di chiave Dato uno schema di relazione R, un sottoinsieme di attributi K è chiamato *superchiave* se per una qualunque relazione valida r su R e per ogni coppia di ennuple t_1 e t_2 di r allora $t_1[K] \neq t_2[K]$. Data una superchiave K se nessun sottoinsieme proprio di K è una superchiave allora questa si dice *chiave* di R. Ad esempio l'insieme {targa, nome} dello schema BARCA è una superchiave. Infatti qualunque relazione valida su BARCA non avrà mai due ennuple t_1 e t_2 tali che $t_1[\{\text{targa, nome}\}] = t_2[\{\text{targa, nome}\}]$. Si osservi che {targa, nome} è una superchiave e non una chiave, a causa del fatto che {targa} è una chiave. Invece l'insieme di attributi {città, targa} nello schema di relazione ORMEGGIO è una chiave in quanto che in ogni relazione valida su ORMEGGIO non possono esserci due ennuple che t_1 e t_2 tali che $t_1[\{\text{città, targa}\}]=t_2[\{\text{città, targa}\}]$ e né {città} né {targa} sono chiavi. Il concetto di chiave gioca un ruolo essenziale nel modello relazionale.

Vincoli di integrità referenziale Nella relazione ORMEGGIO dell'Esempio 3.3 abbiamo che per ogni ennupla t il valore t(targa) deve corrispondere ad una targa di una barca della relazione BARCA. Allo stesso modo il valore t(città) deve corrispondere al nome di una città di un porto della relazione PORTO. Possiamo dire che i valori di alcuni attributi della relazione ORMEGGIO sono vincolati, nella realtà del mini-mondo, dai valori della chiave nelle relazioni BARCA e PORTO. Si osservi che tali vincoli valgono per qualsiasi relazione valida su BARCA, PORTO ed ORMEGGIO. L'applicazione di tali vincoli alla base di dati minimizza quanto più possibile la presenza di errori dovuti all'esistenza di ennuple che non rappresentano la realtà del mini-mondo.

Per formalizzare l'esistenza di questi vincoli introduciamo il concetto di *vincolo di integrità referenziale*. Diciamo che, dati due schemi di relazione R ed S, sussiste un vincolo di integrità referenziale tra la sequenza ordinata di attributi $(A_1, A_2, \ldots, A_h)$ di R e la sequenza ordinata di attributi $(B_1, B_2, \ldots, B_h)$ di una chiave di S quando

- $dom(A_i) = dom(B_i)$ per $i = 1, \ldots, h$ e

- per ogni ennupla t di R esiste una ennupla s di S tale che $t(A_i) = s(B_i)$ per $i = 1, \ldots, h$ oppure $t(A_i)$ è NULL per ogni i.

Nell'esempio precedente quindi, esiste un vincolo di integrità referenziale tra l'attributo targa di ORMEGGIO e l'attributo targa di BARCA.

3.2 L'algebra relazionale

Vediamo in sintesi un linguaggio formale, chiamato *algebra relazionale* dal quale poi SQL, che è il linguaggio standard utilizzato nei sistemi industriali, ha ereditato tutta la sua potenza di calcolo. L'algebra relazionale consente di realizzare operazioni unarie o binare su relazioni restituendo in output ancora una relazione. Consente di specificare in modo formale e coinciso un insieme di operazioni che sono utili da effettuare, al livello logico, su di un database relazionale. Vedremo che l'algebra relazionale è un linguaggio al contempo molto semplice ma molto potente a tal punto che l'SQL, che ne eredita in modo completo la potenza espressiva, è un linguaggio universalmente utilizzato (e diventato infatti standard internazionale) per la interrogazione e la manipolazione della basi di dati in ambito industriale.

3.2.1 Gli operatori dell'algebra relazionale

Operatori insiemistici

Essendo le relazioni degli insiemi, possiamo applicare a queste gli operatori insiemistici di *unione*, $\cup$, *intersezione*, $\cap$ e *differenza* $-$, con il vincolo però che le relazioni coinvolte abbiano schemi *compatibili* tra di loro. Due schemi si intendono compatibili se hanno lo stesso numero di attributi e se $\{A_1, A_2, \ldots, A_k\}$ e $\{B_1, B_2, \ldots, B_k\}$ sono rispettivamente gli schemi delle due relazioni allora avremo che $dom(A_i) = dom(B_i)$ per $i = 1, \ldots, k$ ovvero esiste una eguaglianza tra i domini di uno schema con i corrispondenti domini dell'altro schema. Cosicché se r ed s sono due relazioni con schemi compatibili allora $u = r \cup s$ (rispettivamente $r \cap s$ ed $r - s$) è una relazione che ha come schema lo schema di r (oppure di s indifferentemente) e come ennuple l'unione (rispettivamente l'intersezione e la differenza) delle ennuple di r ed s. Per mezzo delle operazioni di unione e differenza si possono esprimere al livello logico, le operazioni di inserimento di una ennupla t in una relazione r con $r \cup \{t\}$ e di cancellazione di una ennupla con $r - \{t\}$.

L'operatore di selezione

Sia $\Theta = \{`=', `\neq', `<', `>', `\geq', `\leq'\}$ un insieme di operatori di confronto binario e sia $\theta \in \Theta$ uno di tali operatori. Data una relazione r con schema R, siano A e B due attributi di R tali che $dom(A) = dom(B)$ e tale che θ sia applicabile ad una arbitraria coppia di elementi di $dom(A)$. Sia infine a un elemento di $dom(A)$. L'operatore di *selezione*, denotato con $\sigma_{A\theta B}(r)$ oppure $\sigma_{A\theta a}(r)$ prende in input una relazione r e restituisce una relazione con lo stesso schema di r le cui ennuple sono

$$\sigma_{A\theta B}(r) = \{t \in r : t[A]\theta t[B]\}$$

nel primo caso e

$$\sigma_{A\theta a}(r) = \{t \in r : t[A]\theta a\}$$

nel secondo caso.

Esempio 3.4 Con riferimento alla relazione BARCA, dell'Esempio 3.2 supponiamo di voler trovare tutte le barche con lunghezza maggiore di 70 metri. Utilizzeremo l'operatore di selezione sulla relazione BARCA come segue:

$$\sigma_{\text{lunghezza}>70}(\text{BARCA})$$

che restituirà la relazione

targa	nomebarca	lunghezza
RM12345ABC	Vespucci	154,32
GE13345ABC	Rex	127,52

■

L'operatore di proiezione

Sia r una relazione con schema R e sia X un sottoinsieme non vuoto di attributi di R. La *proiezione* su X di r, denotata con $\pi_X(r)$, è una relazione con schema X ed ennuple

$$\pi_X(r) = \{t[X] : t \in r\}$$

ovvero l'insieme di ennuple composto dalla restrizione ad X di ciascuna ennupla di r. Si noti che la proiezione contiene in genere, meno ennuple della relazione originale. Se X è una (super)chiave di r allora il numero di ennuple della proiezione sarà uguale al numero di ennuple della relazione r.

Esempio 3.5 Con riferimento alle relazioni dell'Esempio 3.2 e 3.3, supponiamo di voler conoscere le targhe di tutte le barche ormeggiate. Utilizzeremo l'operatore di proiezione come segue

$$\pi_{\text{targa}}(\text{ORMEGGIO})$$

che restituirà la relazione

targa
GE13345ABC
VE14345ABC
RM12345ABC
BA14375ABC

Se invece avessimo voluto sapere i nomi di tutte le barche, allora possiamo ottenere il risultato con la seguente interrogazione

$$\pi_{\text{nomebarca}}(\text{BARCA})$$

che restituisce la seguente relazione

nomebarca
Vespucci
Rex
Azzurra

si osservi che essendo l'attributo nomebarca un campo non chiave, abbiamo che il numero di ennuple ottenute è minore del numero di ennuple della relazione iniziale. ■

L'operatore di natural-join

È il più potente ed importante tra gli operatori dell'algebra relazionale. Questo operatore consente di combinare le ennuple, logicamente collegate, di due relazioni distinte tra di loro. Infatti siano r ed s due relazioni con schemi R ed S rispettivamente. Allora il natural-join di r ed s denotato $r \bowtie s$ è una relazione che ha come schema $R \cup S$ e contiene le ennuple

$$r \bowtie s = \{t \in dom(R \cup S) : \exists t_1 \in r \wedge \exists t_2 \in s \text{ tali che } t[R] = t_1 \wedge t[S] = t_2\}$$

Informalmente il natural-join combina tra di loro le ennuple delle due relazioni che hanno gli stessi valori sugli attributi in comune.

Si osservi che se $R \cap S = \emptyset$ allora possiamo considerare il natural-join come una sorta di prodotto cartesiano delle ennuple di r con le ennuple di s.

Esempio 3.6 Con riferimento alle relazioni dell'Esempio 3.2 e 3.3, supponiamo di voler conoscere quali barche sono ormeggiate. Allora possiamo reperire tali informazioni utilizzando il natural-join tra le relazioni BARCA ed ORMEGGIO come segue:

BARCA ⋈ ORMEGGIO

che ritorna la seguente relazione

nomebarca	lunghezza	città	targa	data
Rex	127,52	Genova	GE13345ABC	2008/01/25
Azzurra	62,73	Venezia	VE14345ABC	2009/10/11

■

L'operatore di ridenominazione

Abbiamo visto che il natural-join consente di effettuare delle interrogazioni combinando i dati di due differenti relazioni. La combinazione tra le relazioni però dipende fortemente dai nomi degli attributi. Si tenga presente che i nomi degli attributi e delle relazioni vengono scelti arbitrariamente dal progettista. Quindi può succedere che due relazioni abbiano attributi che hanno lo stesso significato ma nomi diversi. Ad esempio nello schema di relazione ORMEGGIO in luogo dell'attributo targa avremmo potuto, più espressivamente, utilizzare l'attributo con nome targa_barca. Notiamo però che se avessimo effettuato il natural-join tra BARCA ed ORMEGGIO non avremmo potuto ottenere il risultato corretto perché il natural-join avrebbe considerato gli attributi targa e targa_barca diversi tra di loro. Viceversa ci possono essere relazioni che hanno gli stessi nomi di attributo i quali hanno però significati diversi. Ad esempio l'attributo targa può essere

contenuto in uno schema di relazione AUTOVEICOLO. Questo attributo ha un significato completamente diverso dall'attributo targa dello schema di relazione BARCA.

Per superare queste difficoltà e per consentire nelle interrogazioni di modificare i nomi degli attributi, viene introdotto in algebra relazionale l'operatore di *ridenominazione* ρ. L'operatore di ridenominazione agisce sullo schema della relazione lasciando invariate le ennuple corrispondenti. Quindi se r è una relazione con schema R, A un attributo di R e B un nome di attributo non presente in R, ma con $dom(B) = dom(A)$ allora

$$\rho_{A \leftarrow B}(r)$$

è una relazione con le stesse ennuple di r ma nel suo schema l'attributo A è stato ridenominato in B. In generale se vogliamo cambiare il nome di un insieme $\{A_1, A_2, \ldots, A_k\}$ di attributi di uno schema di relazione nei nuovi nomi $\{B_1, B_2, \ldots, B_k\}$ (a patto che $dom(B_i) = dom(A_i)$, $i = 1, \ldots, k$) scriveremo

$$\rho_{A_1 \leftarrow B_1, \ldots, A_k \leftarrow B_k}(r)$$

Il prodotto cartesiano

Date due relazioni r ed s con schemi rispettivamente R ed S. Il *prodotto cartesiano* di r ed s si ottiene ridenominando ogni attributo A di R ed ogni attributo B di S in $R.A$ ed $S.B$, rispettivamente. Si osservi che dopo quest'operazione gli schemi R' ed S' ottenuti per r ed s, rispettivamente, non hanno attributi in comune. Alle due relazioni con gli attributi così ridenominati, viene applicato il natural-join. In altri termini se $R = \{A_1, A_2, \ldots, A_h\}$ ed $S = \{B_1, B_2, \ldots, B_k\}$ allora $r \times s$ è

$$r \times s = (\rho_{A_1 \leftarrow R.A_1, \ldots, A_h \leftarrow R.A_h}(r)) \bowtie (\rho_{B_1 \leftarrow S.B_1, \ldots, B_k \leftarrow S.B_k}(s))$$

Per mezzo del prodotto cartesiano e dell'operatore di selezione e proiezione possiamo simulare il natural-join. Infatti date due relazioni r ed s con schemi rispettivamente $R = \{A_1, A_2, \ldots, A_h\} \cup \{C_1, C_2, \ldots, C_l\}$ ed $S = \{B_1, B_2, \ldots, B_k\} \cup \{C_1, C_2, \ldots, C_l\}$, dove $\{C_1, C_2, \ldots, C_l\} = R \cap S$. Allora il natural-join tra r ed s può essere espresso come

$$r \bowtie s = \pi_Q(\sigma_{R.C_1 = S.C_1, \ldots, R.C_l = S.C_l}(r \times s))$$

dove $Q = \{R.A_1, \ldots, R.A_k, R.C_1, \ldots, R.C_l, S.B_1, \ldots, S.B_k\}$. Ad esempio l'interrogazione che richiede di sapere i nomi e le targhe delle barche che sono ormeggiate diventa, con l'operatore di prodotto cartesiano

$$\pi_{\text{Barca.nomebarca,Barca.targa}}(\sigma_{\text{Barca.targa = Ormeggio.targa}}(\text{BARCA} \times \text{ORMEGGIO}))$$

L'operatore di prodotto cartesiano non aumenta l'espressività dell'algebra relazionale, ma viene introdotto poiché l'SQL supporta il natural-join per mezzo di tale operatore.

3.3 Esercizi

Esercizio 3.1 Siano dati i seguenti schemi di relazione

PROGRAMMAZIONE(numero, data, ora, film, anno, sala, posti_dispon, posti_occ)

SALA(multisala, numero, posti)

FILM(titolo, anno, regista, genere, durata, produttore)

Per ciascuno degli schemi individuare

- I domini degli attributi
- Le chiavi
- I vincoli di integrità referenziale

Esercizio 3.2 Con riferimento allo schema di database dell'esercizio 3.1 si esprima in algebra relazionale la seguente interrogazione. Si trovino tutti i film prodotti tra il 1986 ed il 1997 del genere "commedia", che abbiano durata superiore a 90 minuti

Esercizio 3.3 Con riferimento allo schema di database dell'esercizio 3.1 si esprima in algebra relazionale la seguente interrogazione: si trovi la sala che proietta il film nel giorno "2010/07/16" alle ore "18:30" titolato "Orizzonti di Gloria".

Esercizio 3.4 Si consideri la base di dati degli Esempi 3.2 e 3.3. Si esprima in algebra relazionale la seguente interrogazione: si trovino i porti con latitudine maggiore di 25^o e longitudine minore di 32^o

Esercizio 3.5 Si consideri la base di dati degli Esempi 3.2 e 3.3. Si trovi un'interrogazione in algebra relazionale per trovare il nome della barca di tutte le barche maggiori di 32 metri ormeggiate nei porti di latitudine compresa tra 20 e 30 gradi; si stampi anche la longitudine del porto.

Esercizio 3.6 Si consideri la base di dati degli Esempi 3.2 e 3.3. Si vogliono trovare tre porti che hanno reciproche differenze di latitudine minori di .5 gradi.

Esercizio 3.7 Si dia la dimostrazione formale che l'operatore di intersezione dell'algebra relazionale può essere espresso per mezzo dell'operatore natural-join.

Esercizio 3.8 Dimostrare che è possibile esprimere tutti gli operatori dell'algebra relazionale utilizzando solamente gli operatori di: proiezione, natural-join, ridenominazione, unione e differenza. (sugg. dimostrare che la selezione può essere espressa con il natural-join)

Capitolo 4

La progettazione della base di dati

In questo capitolo vedremo come utilizzare il modello concettuale Entità-Associazione per effettuare la progettazione concettuale. Vedremo come realizzare a partire dallo schema concettuale la progettazione logica. Inoltre descriveremo l'algoritmo di traduzione del diagramma concettuale nello schema logico relazionale.

4.1 Progettazione concettuale

La prima fase della progettazione concettuale consiste nell'analisi delle specifiche utente. In questa fase occorre individuare, all'interno del testo, i concetti principali, le categorie di elementi di cui ci interessa memorizzare i dati. È consigliabile stilare un glossario di termini per individuare i concetti principali ed evidenziare eventuali ambiguità nel testo dovute a sinonimie od omonimie. Il glossario sarà composto dal nome del concetto, dalla sua descrizione e dall'elenco di eventuali altri termini a cui è collegato.

A partire dal glossario dei termini possiamo scomporre il testo raggruppando tra di loro le frasi riferite allo stesso concetto, rendendo più semplice così l'analisi e la verifica di completezza del lavoro.

Il lavoro di analisi sui requisiti utente, risulta particolarmente utile e imprescindibile, nei progetti di grandi dimensioni, con decine o centinaia di concetti e decine o centinaia di proprietà elementari per ogni singolo concetto. In questi casi è opportuno, vista la quantità di lavoro da svolgere, scegliere una metodologia di progettazione che indirizzi in modo efficiente il susseguirsi delle attività.

Diverse strategie sono possibili: la strategia *top-down*, quella *bottom-up* o più pragmaticamente, una combinazione di queste. La strategia top-down è applicabile quando le specifiche utente sono chiare fin dall'inizio del progetto. Questa consiste nell'individuare i macro concetti e a partire da questi, procedere con raffinamenti successivi, esplicitando di volta in volta nuovi dettagli.

La strategia bottom-up invece suddivide le specifiche utente in tanti sottoinsiemi distinti. Per ogni sottoinsieme di specifiche viene effettuata l'analisi e la progettazione dello schema corrispondente. Una volta effettuato ciò si procede

con uno o più passi di integrazione per unire tra di loro gli schemi ottenuti, fino ad ottenere lo schema finale.

Tale metodologia risulta utile quando non tutte le specifiche utente sono disponibili o quando richiedono un tempo piuttosto lungo per essere raccolte. Quindi, al fine di ottimizzare i tempi di progettazione, si inizia il lavoro non appena sono disponibili i primi requisiti. Questa metodologia consente anche di suddividere le attività tra diversi gruppi di lavoro.

Nella pratica spesso può risultare conveniente realizzare una combinazione delle due suddette strategie.

La costruzione dello schema concettuale Una volta approntata la strategia di progettazione vediamo sinteticamente come può essere impostata la costruzione dello schema Entità-Associazione. Le categorie ed concetti principali individuati nell'analisi delle specifiche utente, verranno rappresentati tramite entità mentre i legami logici tra entità diventeranno associazioni. Se una entità generalizza altre entità allora rappresentiamo questo mediante una generalizzazione. La scelta di promuovere un concetto ad entità deve essere fatta anche considerando la rilevanza dell'insieme da rappresentare in relazione alla realtà del mini-mondo: ad esempio nell'applicazioni Porti, il telefono della capitaneria di porto può essere specificato come un attributo dell'entità PORTO. Viceversa in un'applicazione per la gestione delle linee telefoniche, il numero di telefono può diventare una entità.

In genere non esiste un metodo formale per decidere se un determinato concetto deve essere rappresentato con un'entità oppure con un associazione od un attributo e spesso la decisione di come rappresentare determinati concetti è demandata al giudizio soggettivo ed al buon senso del progettista. Diversi progettisti potrebbero produrre, partendo da eguali specifiche utente, diversi schemi concettuali. Ciò nonostante possiamo al termine della progettazione concettuale effettuare un'attività di verifica della correttezza e qualità dello schema, sia per mezzo di un'attenta ispezione dello stesso sia utilizzando i risultati della teoria delle dipendenze funzionali.

4.1.1 Completezza e minimalità dello schema

Il processo di progettazione, quasi mai è un'attività che procede per scalini a cascata, ma piuttosto un processo iterativo, di raffinamenti successivi, nel quale ad ogni avanzamento occorre verificare che l'attività appena svolta soddisfi le caratteristiche di correttezza ed efficienza richieste e in caso contrario, iterare i passi appena fatti per realizzare un ulteriore raffinamento. Per fare ciò occorre procedere con un attenta analisi del lavoro svolto e verificare la *correttezza*, la *completezza* e la *minimalità* dello schema concettuale prodotto. Parleremo della correttezza nella successiva sezione.

Completezza La completezza dello schema comporta che tutti i concetti presenti nelle specifiche utente siano rappresentati nello schema. Questo può essere fatto ispezionando attentamente le specifiche utente e lo schema concettuale prodotto.

Una volta effettuata questa verifica si prosegue ad accertare che le operazioni elencate nelle specifiche funzionali possano essere realizzate nello schema concettuale.

Ad esempio, nell'applicazione Porti supponiamo di avere, nelle specifiche funzionali, le seguenti operazioni

- **Operazione 1** Stampa il numero di posti barca di un porto, di cui si conosce il nome e la città, che sono disponibili e non occupati da alcuna barca.
- **Operazione 2** Registra l'ingresso in porto di una barca.

Supponiamo che il diagramma Entità-Associazione comprenda il frammento di schema indicato in Fig. 4.1. Per realizzare l'Operazione 1, dobbiamo esami-

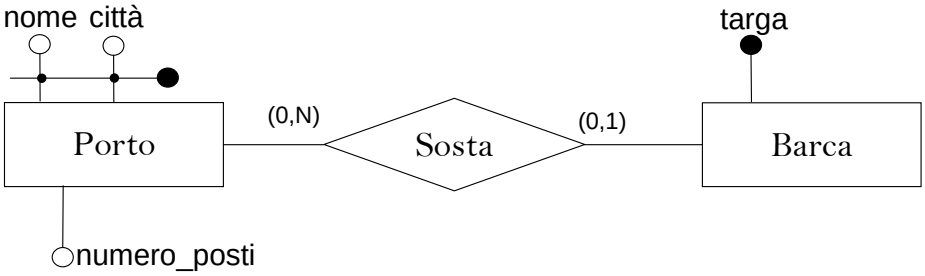


Figura 4.1: Un frammento di schema da verificare

nare tutte le occorrenze di PORTO per trovare quell'occorrenza che corrisponde al nome e alla città richiesta. Utilizzando l'occorrenza di PORTO così ottenuta possiamo elencare tutte le occorrenze dell'associazione SOSTA che contengono l'occorrenza di PORTO testè trovata. Contando tutte le occorrenze di SOSTA così ottenute, siamo in grado di conoscere il numero complessivo dei posti occupati per quel porto. Infine, mediante l'attributo numero_posti, possiamo, per differenza, ottenere il numero complessivo dei posti disponibili.

Supponiamo ora di voler verificare se l'Operazione 2 è effettuabile nello stesso frammento di schema di Fig 4.1. Intanto notiamo che la precedente operazione ci consente di stabilire se esiste o meno un posto nel porto non occupato da alcuna barca. Una volta che abbiamo verificato ciò, l'inserimento della barca in sosta viene effettuato aggiungendo una nuova occorrenza nell'associazione SOSTA, ovvero aggiungendo una coppia $(porto, barca)$ in SOSTA. Al termine di tali verifiche abbiamo che lo schema di Fig. 4.1 è in grado di realizzare le due operazioni ed è pertanto completo rispetto alle operazioni richieste. La precedente verifica va ripetuta per tutte le operazioni delle specifiche funzionali.

Minimalità Un'altra attività volta al controllo della qualità dello schema prodotto è quella della verifica della minimalità dello schema. Uno schema non dovrebbe presentare concetti ripetuti. La forma semplice, ma grave, di ridondanza, è quella di rappresentare come attributi, proprietà o identificatori di altre entità dello schema. Ad esempio supponiamo di avere una entità FILM con un attributo titolo che rappresenta il titolo del film. Se inseriamo un attributo titolo_film nell'entità BIGLIETTO, allora abbiamo un uso scorretto del diagramma Entità-Associazione. La soluzione corretta è quella di inserire un'associazione tra BIGLIETTO e FILM.

Trattiamo ora un secondo tipo di ridondanze. Queste sono rappresentate da attributi che possono essere calcolati o derivati, per mezzo di funzioni numeriche, a partire dai valori di altri attributi. Oppure ridondanze dovute alla presenza di attributi che possono essere calcolati per mezzo di funzioni aggregative di somma o di conteggio, di un insieme occorrenze di entità o associazioni. Un primo esempio di ridondanza è il seguente: l'entità PROIEZIONE contiene gli gli attributi posti_disponibili, posti_totali e posti_occupati. L'attributo posti_disponibili è ottenibile come differenza di posti_totali e posti_occupati; allora uno dei tre attributi è ridondante poiché è calcolabile dagli altri due. Un secondo esempio di ridondanza è dato da quegli attributi che possono essere ricavati da conteggi o da funzioni aggregative su un insieme di occorrenze. Consideriamo l'attributo posti_occupati dell'entità PROIEZIONE, che indica il numero di posti occupati per una proiezione. Tale valore è in realtà calcolabile contando il numero di biglietti associati a quella proiezione e quindi rappresenta una ridondanza nello schema. Parimenti nell'entità FILM, l'attributo incasso di un film può essere calcolato come somma dei costi dei biglietti venduti per tutte le proiezioni di quel film.

Tutte le ridondanze di quest'ultimo tipo, non necessariamente rappresentano un anomalia dello schema. È opportuno però che il progettista ne prenda nota e le documenti al fine di poter effettuare, o nella fase di progettazione logica o nella fase di progettazione fisica opportune analisi di tipo prestazionale. Infatti un valore ridondante da una parte rappresenta uno spreco di spazio e richiede delle operazioni per tenerlo sempre aggiornato. Nello stesso tempo la presenza di un valore ridondante evita, ogni volta che questo viene richiesto dai programmi applicativi, di dover ricalcolare il valore richiesto, ad esempio applicando complesse funzioni numeriche o accedendo ad un numero elevato di occorrenze di entità o associazioni. Queste esigenze contrastanti dovranno essere attentamente soppesate al fine di raggiungere il miglior compromesso tra loro.

4.1.2 Correttezza dello schema, verifica della normalità

Sebbene le procedure descritte nel precedente paragrafo possono dare delle importanti conferme sulla qualità della progettazione svolta, lo strumento formale più efficace per verificare la correttezza e la qualità dello schema prodotto rimane quello della verifica delle *forme normali*. Le forme normali sono state studiate nel contesto del modello relazionale e si basano sul concetto di dipendenza funzionale che di seguito introduciamo. Dato uno schema di relazione R, siano X ed Y due sottoinsiemi non vuoti di attributi di R. Diciamo che esiste una *dipendenza funzionale* $X \to Y$ se per ogni due ennuple t_1 e t_2 di una relazione valida su r su R si ha che

$$\text{se } t_1[X] = t_2[X] \text{ allora } t_1[Y] = t_2[Y]$$

Una dipendenza funzionale è un vincolo derivato dalla descrizione del minimondo e pertanto si applica soltanto ad ogni relazione valida dello schema dato.

Ad esempio si consideri lo schema di relazione ATTORE_FILM(attore, nome_arte, compenso, titolo_film, anno, regista). L'attributo attore identifica univocamente un attore e l'attributo compenso rappresenta la paga dell'attore per la sua recitazione nel film. Sussistono in questo schema le seguenti dipendenze funzionali

titolo_film, anno → regista
attore → nome_arte
titolo_film, anno, attore → compenso

poiché, ogni volta che in due ennuple abbiamo lo stesso valore per gli attributi titolo_film, anno, allora anche il valore dell'attributo regista è uguale. Lo stesso dicasi per la seconda e terza dipendenza funzionale.

Una dipendenza funzionale $X \to Y$ si dice banale quando $Y \subseteq X$ e *non banale* altrimenti. Poiché le dipendenze banali sono valide in ogni relazione saremo interessati a trattare solo con dipendenze funzionali non banali.

Inoltre se $Y = \{A_1, A_2, \ldots, A_n\}$ non è difficile vedere che se vale $X \to Y$ allora deve valere $X \to A_i$, per $1 \leq i \leq n$ e viceversa. Pertanto in quello che segue possiamo supporre che ogni dipendenza funzionale sia del tipo $X \to A$ dove A è un singolo attributo ed $A \notin X$.

Forme normali Le più importanti forme normali sono la *forma normale di Boyce-Codd* (BCNF) e la *terza forma normale* (3NF). Uno schema di relazione R è in BCNF quando per ogni dipendenza funzionale $X \to A$ allora X è una superchiave della relazione. Ad esempio lo schema precedente FILM viola la BCNF poiché {attore} e {titolo, anno} non sono una chiave della relazione e sussistono le dipendenze funzionali titolo, anno → regista e attore → nome_arte.

Uno schema di relazione R è in *terza forma normale* se per ogni dipendenza funzionale $X \to A$ definita su di esso o X è una superchiave oppure A appartiene ad una chiave di R.

Se ogni schema di relazione nello schema rispetta la 3NF o ancora meglio la BCNF, allora questo garantisce formalmente, che la progettazione si è svolta correttamente. Viceversa, se ciò non accade, occorre procedere per mezzo dell'analisi delle dipendenze funzionali ad una opportuna decomposizione degli schemi relazionali in schemi che soddisfino la forma normale richiesta. Si deve osservare che se la progettazione concettuale viene effettuata utilizzando il modello Entità-Associazione, lo schema relazionale che si ottiene alla fine del processo di progettazione, è nella maggior parte dei casi, già in BCNF. Si può dire che la progettazione concettuale mediante il modello Entità-Associazione, fornisce un'*euristica* per la costruzione di schemi di relazioni che soddisfano la BCNF.

La ulteriore trattazione della teoria delle dipendenze funzionali e delle forme normali esula dai compiti di questo testo. La materia viene trattata nel dettaglio, nel primo modulo del corso di Basi di Dati I.

4.1.3 Esempio di progettazione concettuale

Di seguito vediamo due esempi di progettazione concettuale prendendo le specifiche del Capitolo 2. Leggendo attentamente le specifiche del Capitolo 2 relative all'applicazione Cinema possiamo stilare il seguente glossario dei termini

Termine	Descrizione	Collegamenti
Multisala	Un luogo dove sono presenti sale cinematografiche	Sala
Sala	La sala dove si proietta un film	Multisala, poltrona, proiezione
Poltrona	La poltrona fisica del cinema	Sala, biglietto
Biglietto	Il tagliando che consente l'accesso ad una sala per una proiezione	Sala, proiezione, poltrona
Proiezione	Lo svolgersi di una proiezione di un film	Film, sala
Film	La pellicola proiettata	Proiezione , sala
Attore	Gli attori dei file	Film

Possiamo sulla base del glossario stilato raggruppare le frasi intorno ai termini principali, come segue

- **Frasi relative alla Multisala**. Una catena di multisale cinematografiche possiede circa 20 multisale. Ogni multisala possiede mediamente 20 sale.
- **Frasi relative alla Sala**. Ogni sala può contenere mediamente 250 posti numerati.
- **Frasi relative alle Poltrone**. I posti di ciascuna sala sono organizzati in file e all'interno di ogni fila sono numerati da 1 a n dove n è il numero di posti di una fila.
- **Frasi relative alle Proiezioni**. La programmazione dei film nelle sale deve avere una finestra temporale di 15 giorni a partire dalla data odierna. Le informazioni relative alla programmazione sono la sala, il film proiettato, la data e ora di inizio proiezione, la durata della proiezione, il numero di posti totali, quello dei posti disponibili e quello dei posti prenotati o acquistati. Ogni sala effettua in media 5 proiezioni al giorno e per ogni proiezione la sala è piena mediamente al 60%.
- **Frasi relative ai Biglietti**. I biglietti possono essere a costo intero o a costo ridotto e vengono distribuiti dal botteghino che stampa sopra ciascuno di essi, al momento dell'emissione, il numero, la data di emissione, la sala, la data e ora di inizio del film, la fila ed il numero del posto acquistato.
- **Frasi relative ai Film**. Dei film in proiezione si conoscono il titolo, il regista, gli attori principali e la durata. In media si proiettano 400 nuovi film all'anno.
- **Frasi relative agli Attori**. Degli attori si vuole conoscere il nome, la data di nascita e i film interpretati.

Poiché le specifiche utente sono note e complete prima di cominciare la progettazione, questo ci consente di adottare una strategia di progettazione "a macchia d'olio" la quale consiste nel partire dai concetti più evidenti ed elementari e

mano a mano, aggiungere nuovi concetti ed effettuare raffinamenti successivi dello schema concettuale a partire da quelli già esistenti.

Il primo concetto evidente di cui vogliamo memorizzare i dati è quello di biglietto. Quindi creeremo una entità BIGLIETTO. Subito dopo ci accorgiamo che è necessario costituire l'entità SALA e l'entità POLTRONA poiché vorremmo conoscere nel dettaglio quali poltrone fisiche sono occupate per una data proiezione e quali no. Chiaramente l'entità BIGLIETTO e l'entità POLTRONA sono connesse tra di loro poiché ad ogni biglietto corrisponde una poltrona di una sala. Questo fatto, rappresentato dall'associazione ACQUISTO, ci consentirà di stabilire quante e quali poltrone saranno occupate per una certa proiezione. Inoltre l'entità POLTRONA sarà connessa all'entità SALA per indicare quali poltrone sono presenti in quale sala. Quindi abbiamo un primo frammento di schema Entità-Associazione come indicato in Fig. 4.2.

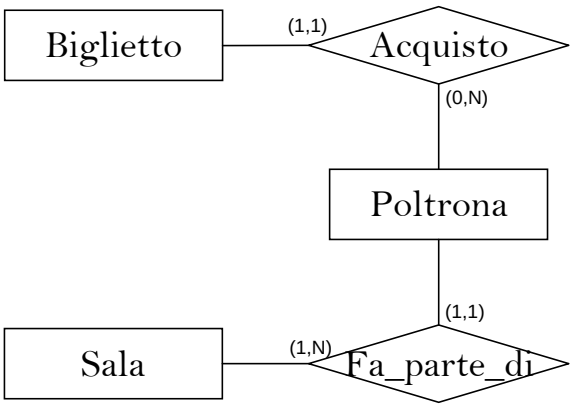


Figura 4.2: Primo frammento di schema dell'applicazione Cinema

Vediamo ora quali sono le cardinalità delle associazioni inserite. Abbiamo che un biglietto è associato ad una ed una sola poltrona, mentre una poltrona è stata o è occupata (in momenti differenti) da molti biglietti. Pertanto, per l'entità BIGLIETTO, sia la cardinalità minima che quella massima dell'associazione ACQUISTO è uno. Mentre per l'entità POLTRONA le cardinalità saranno (0,N) poiché, verosimilmente, una poltrona potrebbe essere mai stata acquistata per un biglietto. Poiché una poltrona appartiene ad una sala, abbiamo che l'entità POLTRONA partecipa all'associazione FA_PARTE_DI con cardinalità $(1, 1)$, mentre l'entità SALA, partecipa con cardinalità (1,N).

Una volta raggiunto questo primo frammento di schema ovviamente ci poniamo il problema di rappresentare il fatto che poltrona verrà occupata in un certo periodo di tempo coincidente con la proiezione del film acquistato. Occorre pertanto inserire un nuovo concetto che è l'entità PROGRAMMA. Questa rappresenta l'insieme delle proiezioni programmate nelle varie sale cinematografiche in una certa data e con un dato film. Si noti che ad esso saranno associate le informazioni relative al numero totale di posti disponibili o di quelli occupati. Quindi il biglietto sarà associato ad una occorrenza di PROGRAMMA. A sua volta un programma è associato ad una sala e ad un particolare film. Pertanto possiamo inserire l'entità FILM ed associarla con l'entità PROGRAMMA mediante

l'associazione PROIEZIONE. Per le cardinalità dell'associazione PROIEZIONE abbiamo che un film può essere proiettato diverse volte mentre un in programma viene proiettato un unico film in un unica sala. Quindi FILM partecipa all'associazione proiezione con cardinalità (0,N) mentre PROGRAMMA partecipa alle associazioni PROIEZIONE e SPETTACOLO con cardinalità $(1,1)$. L'entità SALA partecipa invece all'associazione SPETTACOLO con cardinalità (0,N) poiché una sala dispone diversi spettacoli. Infine, poiché un programma può essere stato acquistato da più persone, avremo che l'entità PROGRAMMA partecipa con cardinalità (0,N) nell'associazione VISIONE, mentre BIGLIETTO parteciperà con cardinalità $(1,1)$.

A questo punto abbiamo realizzato il frammento di schema indicato in Fig. 4.3.

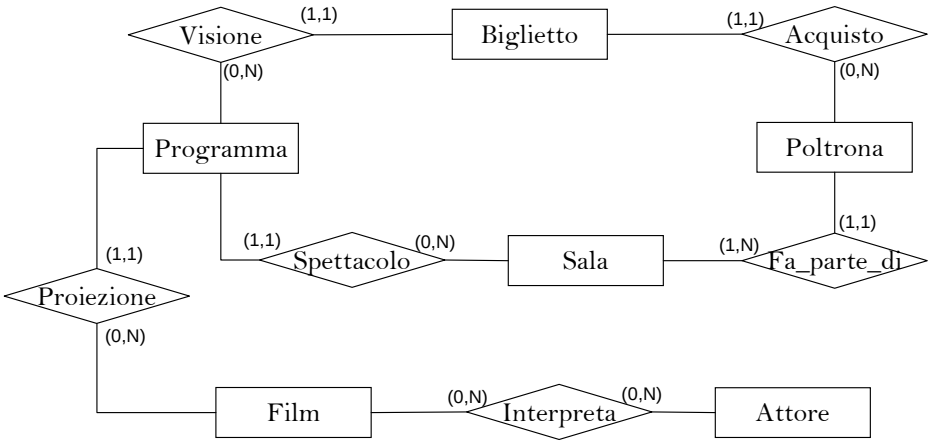


Figura 4.3: Lo schema dell'applicazione Cinema

Possiamo ora integrare lo schema di Fig. 4.3 aggiungendo ad ogni entità ed associazione, gli attribuiti di interesse per le specifiche utente ed individuando, ove possibile, gli identificatori di ciascuna entità. L'entità BIGLIETTO ha come identificatore l'attributo numero. L'entità SALA è identificata dagli attributi numero e multisala. L'entità PROGRAMMA ha un identificatore esterno con l'entità SALA. Infatti non esistono due programmi diversi che hanno la stessa data, l'ora e nella stessa sala. Anche POLTRONA ha un identificatore esterno in SALA, ovvero una poltrona è identificata dal numero, fila e dalla sala. L'entità FILM è identificata dalla coppia di attributi titolo e anno. In questo modo produciamo lo schema finale di Fig. 4.4.

Giunti in questa fase, possiamo procedere nel verificare la completezza di quanto realizzato verificando che sia possibile navigare nello schema per realizzare le operazioni richieste. A tal fine riportiamo in Tabella 4.1 un esempio di specifiche funzionali, che verosimilmente, possono essere richieste per l'applicazione Cinema.

Vediamo a titolo di esempio, come è possibile realizzare l'Operazione 5. Si invita lo studente a verificare, per esercizio le altre operazioni. Per realizzare tale operazione individuiamo, per mezzo dell'attributo numero, l'occorrenza di SALA in cui viene proiettato il film. Mediante l'associazione SPETTACOLO, pos-

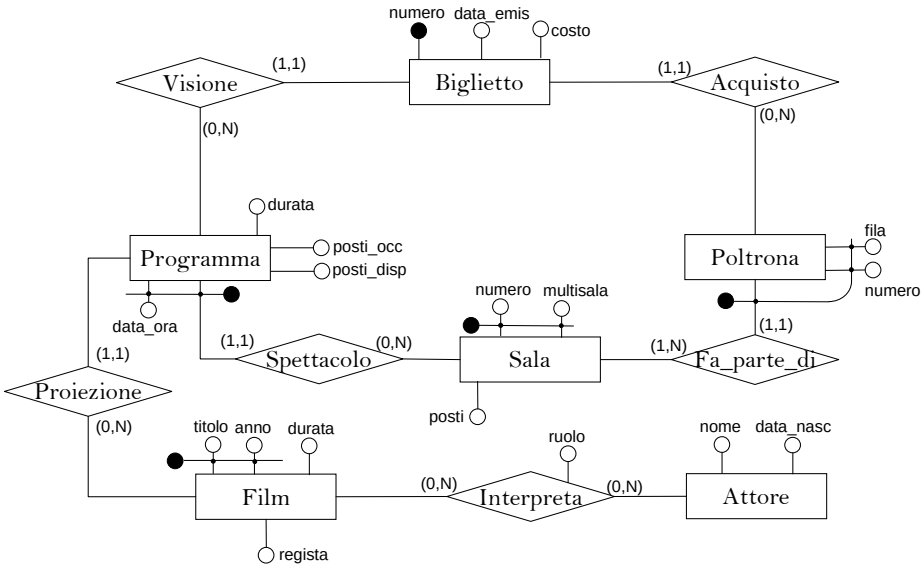


Figura 4.4: Lo schema finale dell'applicazione Cinema

siamo recuperare l'insieme P di tutte le occorrenze di PROGRAMMA in quella sala. Con l'attributo data_ora recuperiamo quindi dall'insieme P, l'insieme di occorrenze di PROGRAMMA presenti nella data richiesta. Poi per mezzo dell'associazione PROIEZIONE, individuiamo l'insieme di film proiettati per ciascun programma. Infine mediante l'attributo regista, selezioniamo solo quei film che sono stati diretti dal regista richiesto. Abbiamo quindi potuto "navigare" nello schema, selezionando insiemi di occorrenze di entità ed associazioni che soddisfavacevano le condizioni logiche poste dall'operazione.

Analizziamo ora le ridondanze presenti nello schema. L'unica ridondanza è dovuta agli attributi posti_occ e posti_disp, calcolabili, data una occorrenza p di PROGRAMMA, contando le occorrenze di VISIONE in cui è presente p nel primo caso e sottraendo al numero di posti della sala, il numero di posti occupati, nel secondo caso.

Si osservi che l'Operazione 3 potrebbe essere realizzata più semplicemente aggiungendo un attributo incasso nell'entità FILM. L'aggiunta di questo attributo o l'eliminazione degli attributi ridondanti, deve essere attentamente valutata sia in questa che nelle fasi successive. Ad esempio, se è vero che l'attributo incasso semplifica l'Operazione 3, è anche vero che tale attributo deve essere costantemente aggiornato ogni volta che viene effettuata l'operazione di vendita di un biglietto. Quindi se da una parte l'Operazione 3 si semplifica, dall'altra l'Operazione 1 diventa più complessa.

4.2 Progettazione logica

Una volta effettuata la progettazione concettuale, lo schema ottenuto al termine di questa, viene tradotto nello schema logico relazionale, mediante un ben

N	Operazione	Freq. giorno
1	Registrare l'acquisto di un biglietto per una determinata poltrona e per una certa proiezione.	67.000
2	Modificare la programmazione per i prossimi 15 giorni.	1
3	Stampare i tre film che hanno registrato l'incasso maggiore e per ciascun film stampare il nome del regista, il nome degli attori e l'importo complessivo dell'incasso.	50
4	Calcolare il totale dei posti disponibili di una sala per una data proiezione.	700
5	Trovare il titolo di un film di cui si conosce il regista, la sala di appartenenza e la data di programmazione.	32.000
6	Trovare all'interno di una sala i numeri dei posti e le file che contengono tre poltrone consecutive nella stessa fila non occupate e non prenotate per una data proiezione.	3.000

Tabella 4.1: Elenco di operazioni per l'applicazione Cinema

preciso algoritmo di traduzione. Tale algoritmo viene in effetti implementato in numerosi prodotti CASE (Computer Aided Software Engineering) presenti sul mercato, i quali consentono di disegnare lo schema Entità-Associazione e producono da questo, automaticamente, lo schema relazionale. Spesso tali prodotti vengono utilizzati in stretto contatto con i DBMS relazionali commerciali e il risultato concreto dell'algoritmo di traduzione è un insieme di comandi SQL, i quali quando eseguiti in un DBMS commerciale, creano lo schema della base di dati. Prima di effettuare la traduzione nello schema logico, è necessario effettuare alcune modifiche allo schema concettuale. Questo perché in questa fase della progettazione cominciano a prevalere gli aspetti implementativi. L'obiettivo di queste modifiche sarà quello di produrre uno schema logico che sia quanto più performante ed efficiente. Inoltre occorre trasformare lo schema concettuale per eliminare le generalizzazioni e gli attributi multivalore che non possono direttamente essere tradotti nello schema relazionale dall'algoritmo di traduzione. Le operazioni da effettuare saranno:

1. Partizionamento ed accorpamento di entità ed associazioni
2. Eliminazione delle generalizzazioni
3. Eliminazione degli attributi multivalore
4. Scelta degli identificatori principali
5. Traduzione dello schema Entità-Associazione nello schema relazionale

4.2.1 Analisi delle prestazioni

Prima ancora di tradurre lo schema concettuale nello schema logico relazionale è opportuno analizzare lo schema Entità-Associazione, al fine di apportare delle modifiche che possono rendere più efficiente l'implementazione finale. A questo livello è piuttosto difficile fare delle scelte in funzione delle prestazioni poiché non è noto né il sistema di gestione delle basi di dati che verrà utilizzato, né

lo schema fisico della base di dati. Ciò nonostante è possibile effettuare delle considerazioni di tipo prestazionale. Per poter fare ciò, occorre conoscere quali operazioni dovranno essere implementate dai programmi applicativi e con quale frequenza queste operazioni verranno eseguite sulla base di dati a regime. Inoltre occorre conoscere quali volumi - ovvero la cardinalità delle istanze di entità ed associazioni - che la base di dati deve contenere. Si costruiscono pertanto due tabelle : la *Tavola delle operazioni* contenente l'elenco delle operazioni e la loro frequenza e la *Tavola dei volumi* nella quale si riportano le cardinalità delle entità ed associazioni dello schema.

Con tali informazioni si può analizzare lo schema concettuale al fine di individuare e realizzare quelle modifiche che possono portare dei miglioramenti alle prestazioni. Si deve però sempre tenere presente che lo schema così modificato non è più uno schema concettuale in senso proprio, ma una particolare versione, in funzione delle operazioni considerate e dei volumi della base di dati. Se la tipologia e la frequenza delle operazioni o i volumi della base di dati dovessero mutare nel tempo allora, mentre lo schema concettuale originale rimarrebbe ancora valido, quello costruito durante questa fase potrebbe essere inefficiente ed inadeguato alle mutate condizioni.

4.2.2 Partizionamento ed accorpamento di entità ed associazioni

Un semplice metodo, ma a volte estremamente efficace, per migliorare le prestazioni della base di dati è quello del *partizionamento* e dell'*accorpamento* di entità od associazioni. Il principio che è dietro tale metodologia è quello di tenere separati quei dati che vengono acceduti da differenti operazioni (partizionamento dei dati) oppure accorpare quei dati che devono essere acceduti da una stessa operazione (accorpamento dei dati). Il partizionamento od accorpamento è di due tipi: *verticale* od *orizzontale*.

Nel partizionamento verticale di una entità separiamo un insieme di attributi dell'entità, quelli che vengono acceduti più frequentemente, generando due entità legate tra di loro per mezzo di una nuova associazione. A volte un'entità può avere centinaia di attributi e quindi richiedere molto spazio per memorizzare i dati afferenti ad una singola occorrenza di entità (chiameremo *record* l'elemento fisico in cui vengono memorizzati i dati di una singola occorrenza di entità). Viceversa, con pochi attributi, le dimensioni dei record sono molto piccole. Il vantaggio in questo caso è che se il record è piccolo allora molti record possono essere recuperati da una singola operazione di lettura o scrittura rendendo tali operazioni molto efficienti.

Ad esempio possiamo avere nell'entità TRAGHETTO due tipologie di attributi (vedi Fig. 4.5). Una afferente alle caratteristiche nautiche del traghetto, come la lunghezza, la stazza, la potenza del motore e così via. Altri attributi possono riferirsi alle capacità commerciali del traghetto come il numero posti_ponte, il numero di posti_auto o quello per i posti_TIR ecc.. Allora in un'applicazione per la vendita di biglietti, visto che gli attributi relativi alle capacità commerciali, saranno acceduti più frequentemente, potrebbe essere utile partizionare traghetto in due entità: TRAG_TECH e TRAG_COMM uniti da un'associazione come indicato in Fig. 4.5.

In tal modo l'accesso ai dati commerciali risulta agevolato e più efficiente. Inoltre eventuali operazioni che richiedono il reperimento dei dati tecnici di un

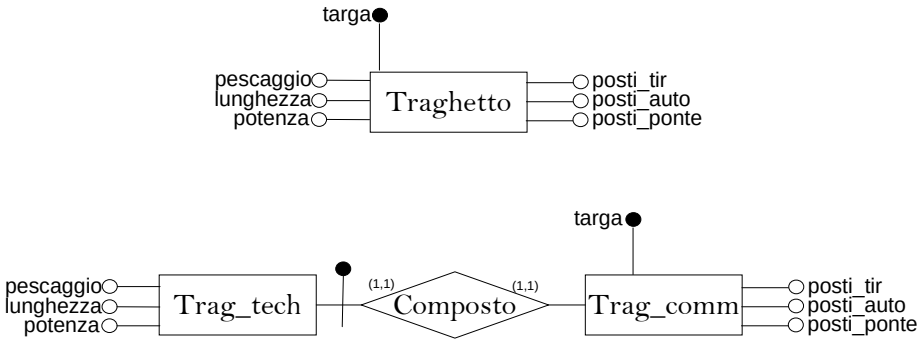


Figura 4.5: Esempio di partizionamento verticale

traghetto non interferiranno con le altre operazioni commerciali, rendendo così la base di dati più robusta ed efficiente.

Il partizionamento orizzontale di una entità consiste invece nel suddividere l'insieme delle occorrenze di una entità in due o più sottoinsiemi disgiunti.

Ad esempio prendiamo l'entità BIGLIETTO dell'applicazione Cinema. I biglietti venduti in giornata saranno quelli che sono consultati se vogliamo conoscere, ad esempio, quanti posti sono disponibili per una data proiezione. Viceversa, i biglietti acquistati il giorno prima o in un momento antecedente la giornata odierna, verranno acceduti con molta minore frequenza e da un diverso tipo di operazioni, ad esempio per sapere quanti biglietti sono stati venduti in un certo lasso di tempo o l'incasso complessivo di una giornata, di un mese ecc.. Pertanto è sensato partizionare orizzontalmente l'entità BIGLIETTO in due entità distinte, BIGLIETTO e BIGLIETTO_STORICO, dove nella prima saranno rappresentati solo i biglietti venduti e validi nella giornata odierna, mentre nella seconda i biglietti validi dei giorni precedenti[1]. Se ad esempio i biglietti dovranno essere memorizzati per almeno tre anni avremo che l'entità BIGLIETTO avrà circa un millesimo delle occorrenze dell'entità BIGLIETTO_STORICO, con un evidente vantaggio per le prestazioni applicative.

L'operazione inversa del partizionamento è quella dell'accorpamento. Anche qui l'accorpamento può essere verticale od orizzontale. In caso di accorpamento verticale, se le due entità sono legate da una (o più) associazioni di tipo uno-a-molti o molti-a-molti, bisogna fare attenzione nel procedere, poiché lo schema risultante potrebbe contenere delle ridondanze e lo schema logico relazionale che poi ne deriva, potrebbe violare le forme normali. La regola è quella di accorpare, se ritenuto vantaggioso, quelle entità che hanno tra di loro solo associazioni con cardinalità uno-a-uno.

L'accorpamento orizzontale di entità può risultare molto efficace, dal punto di vista prestazionale, quando le occorrenze di diverse entità devono essere accedute da una stessa operazione. Ad esempio supponiamo che si voglia accedere alle occorrenze delle entità TRAGHETTO e PESCHERECCIO, e si vogliano elencare

[1] Questo tipo di partizionamento richiederà nella pratica, un'attività giornaliera di *svecchiamento* dei dati, consistente nel trasferire a fine giornata i dati da BIGLIETTO a BIGLIETTO_STORICO

tutte queste barche in ordine di stazza. Per fare ciò, ogni volta occorre fondere le due istanze tra di loro ed effettuare un ordinamento complessivo. Questa può rilevarsi un'operazione alquanto dispendiosa. Viceversa unendo le due entità tra di loro, ed utilizzando un indice sull'attributo stazza, si renderebbe tale operazione estremamente più efficiente.

Quanto detto sopra per le entità vale anche per le associazioni. Ad esempio supponiamo di voler partizionare orizzontalmente l'associazione INTERPRETA tra ATTORE e FILM in due associazioni INTERPRETA_PROTAGONISTA ed INTERPRETA_ALTRO se ad esempio, l'operazione che richiede di elencare gli attori ed i film nei quali gli attori hanno recitato come protagonisti è prevalente nella base di dati.

4.2.3 Eliminazione delle generalizzazioni

Le generalizzazioni non possono essere direttamente tradotte dall'algoritmo di traduzione nel modello relazionale. Pertanto occorre trasformare le generalizzazioni presenti nello schema Entità-Associazioni in una combinazione di entità ed associazioni. Abbiamo tre metodi per eliminare le generalizzazioni presenti nello schema.

1. Accorpamento delle entità figlie nel genitore
2. Accorpamento dell'entità genitore nelle entità figlie
3. Rappresentazione della generalizzazione tramite associazioni

Accorpamento delle entità figlie nel genitore In questo caso si eliminano tutte le entità figlie e si aggiungono gli attributi propri di ciascuna entità eliminata, nell'entità padre. Poi è necessario aggiungere un ulteriore attributo tipologia nell'entità padre, che avrà nel suo dominio tanti elementi quante sono le entità figlie eliminate. Per mezzo di tale attributo sarà possibile stabilire, per ciascuna occorrenza dell'entità padre, a quale entità figlia apparteneva. Un'eventuale associazione tra una entità figlia ed un'altra entità E, diventerà un associazione tra l'entità padre e l'entità E con la stessa cardinalità massima della precedente associazione ma con cardinalità minima pari a zero.

Ad esempio in una generalizzazione tra BARCA, TRAGHETTO e PESCHERECCIO supponiamo di voler accorpare le entità TRAGHETTO e PESCHERECCIO nell'entità padre BARCA (vedi Fig. 4.6). Allora aggiungeremo a BARCA gli attributi propri di TRAGHETTO e PESCHERECCIO, più un attributo tipo_barca che avrà, nel suo dominio, due valori: "Peschereccio" e "Traghetto". Con riferimento alla Fig. 4.6, l'associazione SCARICO tra PESCHERECCIO e PORTO, diventerà un'associazione tra BARCA e PORTO ma con cardinalità minima pari a zero. Inoltre, nell'entità BARCA, gli attributi posti_ponte e capacità_carico, possono avere valori nulli.

Accorpamento dell'entità genitore nelle entità figlie Questo caso è applicabile solo se la generalizzazione è totale, poiché altrimenti alcune occorrenze che sono solo nell'entità padre non sarebbero più rappresentate.

Per accorpare il genitore nelle figlie, si procede trasferendo gli attributi dell'entità padre in ciascuna delle entità figlie ed eliminando la entità padre. Se esisteva una associazione R che coinvolgeva l'entità padre con un'altra entità E

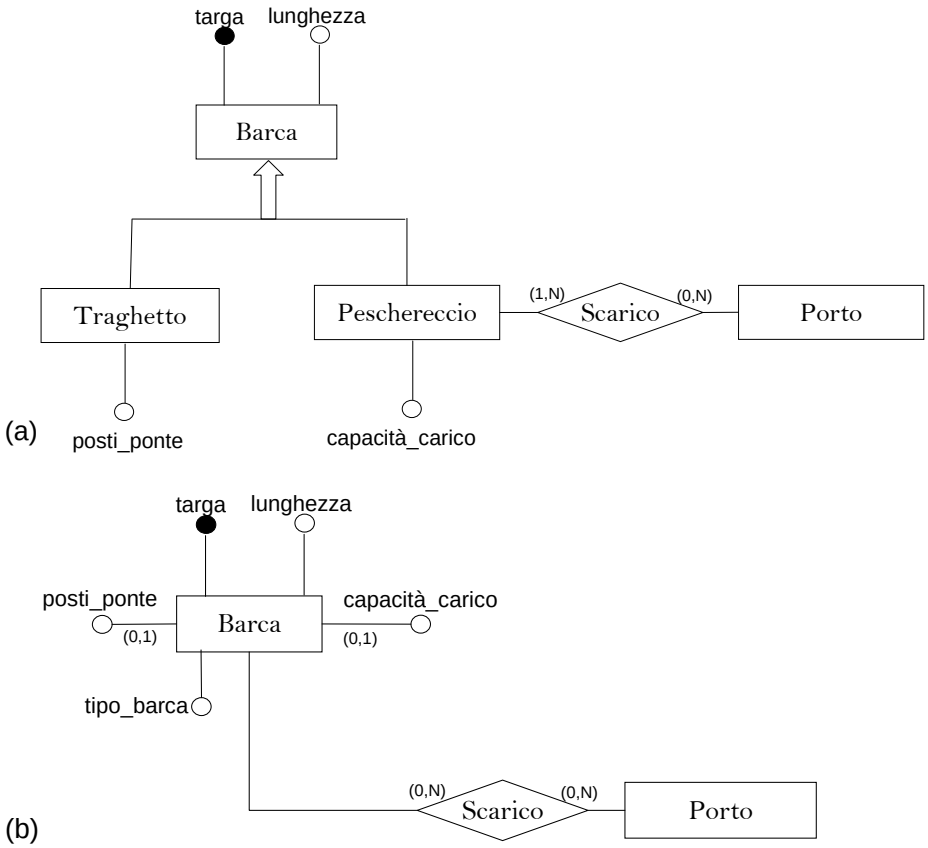


Figura 4.6: (a) La generalizzazione iniziale (b) Dopo l'accorpamento delle figlie nel padre

allora occorre procedere come segue. Siano $E_1, E_2, \ldots, E_k$ le entità figlie della generalizzazione. Per ogni entità E_i, $i = 1, \ldots, k$, si aggiunge un'associazione R_i tra E_i ed E che ha lo stesso significato di R.

Ad esempio, con riferimento alla Fig. 4.7, in una generalizzazione tra BARCA, TRAGHETTO, PESCHERECCIO, nell'ipotesi che nella realtà del mini-mondo tale generalizzazione sia totale, gli attributi dell'entità padre BARCA vengono trasferiti nelle entità figlie. Poi dopo aver eliminato la entità padre BARCA, si aggiungono le associazioni VARO_PESCH e VARO_TRAG tra PESCHERECCIO ed CANTIERE e tra TRAGHETTO ed CANTIERE.

Rappresentazione della generalizzazione tramite associazioni Questo caso è applicabile indifferentemente sia che la generalizzazione sia totale o parziale. La rappresentazione della generalizzazione tramite associazioni consiste nel creare un'associazione tra una entità figlia e l'entità padre per tutte le entità figlie della generalizzazione. Inoltre ciascuna entità figlia avrà come identificatore esterno l'identificatore dell'entità padre.

Ad esempio nella generalizzazione tra BARCA, TRAGHETTO e PESCHERECCIO aggiungeremo le associazioni TIPO_TRAGH e TIPO_PESCH tra TRAGHETTO

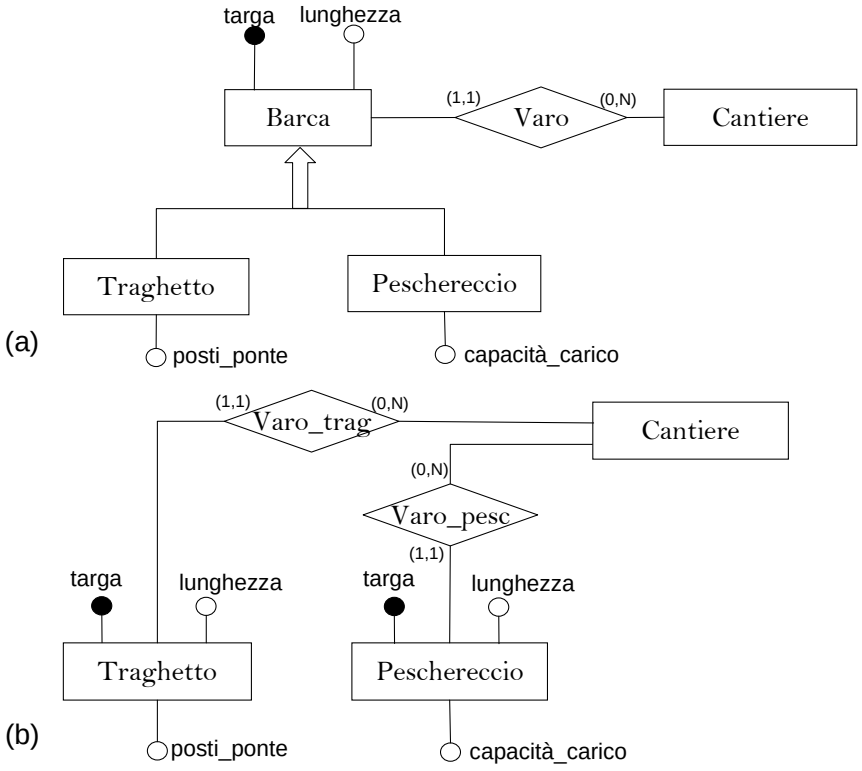


Figura 4.7: (a) La generalizzazione iniziale (b) Dopo l'accorpamento del padre nelle figlie

e PESCHERECCIO, rispettivamente e BARCA. Infine le entità TRAGHETTO e PESCHERECCIO avranno come unico identificatore esterno quello dell'entità BARCA come indicato in Fig. 4.8.

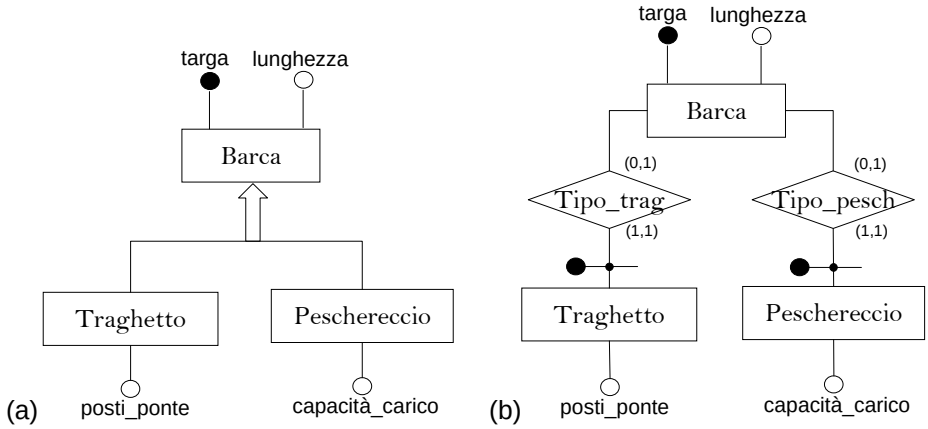


Figura 4.8: (a) La generalizzazione iniziale (b) La rappresentazione tramite associazioni

Linee guida per l'eliminazione delle generalizzazioni. La scelta tra le varie alternative proposte per eliminare le generalizzazioni può essere fatta attraverso un'analisi simile a quella fatta per decidere se accorpare o partizionare, orizzontalmente o verticalmente, entità od associazioni. Infatti l'operazione di unire le figlie nel genitore corrisponde effettivamente ad un accorpamento di entità. Viceversa l'operazione inversa di accorpamento del genitore nelle figlie corrisponde ad un partizionamento dell'entità padre nelle entità figlie.

In altre parole possiamo immaginare il processo di partizionamento orizzontale di una entità, come l'introduzione di una generalizzazione dell'entità con tante figlie quante sono le partizioni da effettuare, seguita da un accorpamento del padre nelle figlie.

Infine, la scelta di rappresentare la generalizzazione tramite associazioni corrisponde, intuitivamente, ad un doppio partizionamento sia verticale che orizzontale. Infatti, in questo caso, le entità figlie mantengono gli attributi propri e questo corrisponde ad una sorta di operazione di partizionamento verticale degli attributi di tutte le entità della generalizzazione. Infine l'insieme delle entità figlie corrisponde, come già detto, ad un partizionamento orizzontale dell'entità padre.

In conclusione l'analisi e la scelta del metodo per l'eliminazione delle generalizzazioni, può essere fatta in modo analogo alla scelta dei partizionamenti ed accorpamenti.

4.2.4 Eliminazione degli attributi multivalore

Per ogni attributo multivalore A appartenente ad una entità E genereremo una entità con nome E_A che ha come unico attributo l'attributo A. L'entita E_A sarà associata mediante una relazione R_A all'entità E che diventa, insieme con l'attributo A, identificatore esterno per E_A. La cardinalità con cui l'entità E partecipa all'associazione R_A è pari alla cardinalità dell'attributo multivalore. Ad esempio supponiamo che in una entità BARCA ci sia un attributo multivalore altezza_albero con cardinalità (0,N). Allora si creerà una entità ALTEZZA_ALBERO con attributo altezza . In più aggiungeremo un'associazione POSSIEDE tra BARCA e ALTEZZA_ALBERO nella quale l'entità BARCA partecipa con cardinalità (0,N) ed è l'identificatore esterno di ALTEZZA_ALBERO.

4.2.5 Scelta degli identificatori principali

Prima di tradurre lo schema concettuale nello schema relazionale occorre specificare per ogni entità dello schema un *identificatore principale*. Questa è un'attività fondamentale per la traduzione dello schema nel modello relazionale. Una entità può contenere più di un identificatore principale oppure può non possedere alcun identificatore. La scelta dell'identificatore principale deve essere fatta scegliendo quello che ha la dimensione più piccola possibile. Quindi un identificatore costituito da uno o pochi attributi è da preferirsi rispetto ad uno che ne ha molti. In alcuni casi può essere opportuno definire ad hoc un identificatore principale in un'entità, o perché gli identificatori presenti sono troppo lunghi o perché l'entità non ha nessun identificatore come nel seguente esempio.

Una entità STUDENTE, con attributi nome, età e data_iscrizione è priva di un identificatore. In questo caso si aggiunge all'entità un nuovo attributo che ha le

funzioni di identificatore. Nel caso specifico viene aggiunto l'attributo matricola, che diventa l'identificatore principale dell'entità.

4.2.6 L'algoritmo di traduzione dello schema Entità- Associazione nello schema relazionale

Di seguito descriviamo nel dettaglio i passi per la traduzione dello schema concettuale nello schema logico relazionale. L'algoritmo di traduzione prende in input uno schema Entità-Associazione nel quale per tutte le entità presenti è già stato specificato un identificatore e nel quale non siano presenti né generalizzazioni né attributi multivalore.

Passo 1: Traduzione di Entità

Ciascuna entità dello schema viene tradotta in uno schema di relazione che ha come nome, il nome dell'entità e come attributi, gli attributi dell'entità. Inoltre l'identificatore dell'entità diventa la chiave principale dello schema di relazione. Ad esempio l'entità PORTO di Fig. 4.9, si tradurrà nel seguente schema di

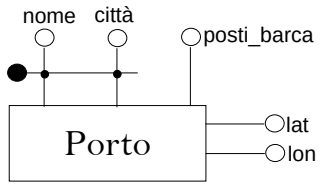


Figura 4.9: L'entità PORTO

relazione

PORTO(città, nome, lat, lon, posti_barca)

Per prime verranno tradotte le entità che non hanno identificatori esterni. Se l'entità ha identificatori esterni allora occorre aggiungere allo schema di relazione tutti gli attributi degli identificatori delle entità identificanti.

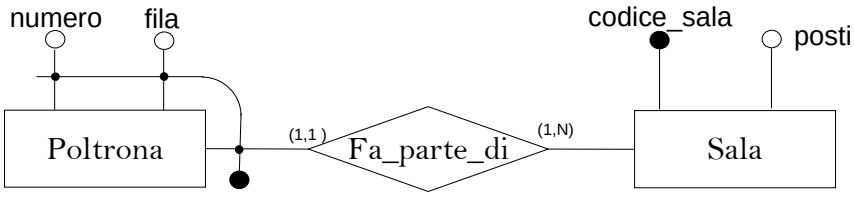


Figura 4.10: Entità con identificatore esterno

Ad esempio tradurremo l'entità POLTRONA della Fig. 4.10 come

POLTRONA(fila, numero, codice_sala)

aggiungendo nello schema di POLTRONA l'identificatore di SALA.

Passo 2: Traduzione di Associazioni

Consideriamo dapprima il caso di associazioni binarie, che coinvolgono due entità (non necessariamente distinte) e poi estendiamo la trattazione al caso di più di due entità. Distinguiamo diversi casi: il caso di associazioni molti-a-molti, il caso di associazioni uno-a-molti ed il caso di associazioni uno-ad-uno.

Caso molti-a-molti La traduzione dell'associazione comporta la creazione di uno schema di relazione che ha il nome dell'associazione ed ha come attributi gli attributi propri dell'associazione e gli attributi degli identificatori di ciascuna delle due entità coinvolte. Ad esempio l'associazione INTERPRETA tra ATTORE

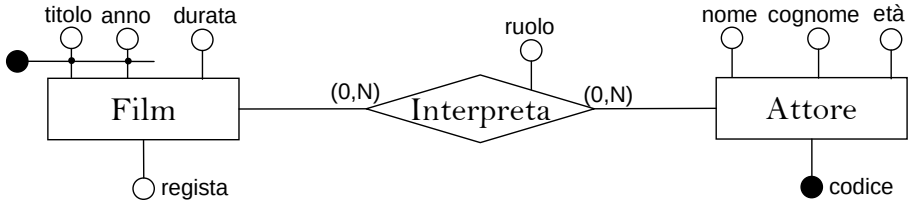


Figura 4.11: Associazione molti-a-molti

e FILM di Fig. 4.11, verrà tradotta in uno schema di relazione nel quale è presente l'attributo ruolo e gli identificatori degli schemi di relazione FILM e ATTORE come di seguito indicato

ATTORE(codice, nome, cognome, età)
FILM(titolo, anno, regista, durata)
INTERPRETA(codice, titolo, anno, ruolo)

Avremo un vincolo di integrità referenziale tra gli attributi codice di INTERPRETA e codice di ATTORE e tra la coppia di attributi {titolo, anno} di INTERPRETA e la chiave di FILM, {titolo, anno}. È conveniente però rinominare gli attributi con nomi più espressivi. Ad esempio tradurremo l'associazione INTERPRETA come:

INTERPRETA(attore, film, anno, ruolo)

con vincolo di integrità referenziale tra gli attributi attore di INTERPRETA e codice di ATTORE e tra la coppia di attributi {film, anno} di INTERPRETA e la chiave di FILM, {titolo, anno}.

In caso di associazione ricorsiva occorre necessariamente rinominare, nello schema di relazione risultante, i nomi degli attributi di una (o di tutte due) le chiavi dell'entità coinvolta nell'associazione.

Ad esempio l'associazione ricorsiva TRATTA da PORTO a PORTO verrà tradotta come segue

PORTO(nome, città, lat, lon, posti_barca)
TRATTA(nome_part, città_part, nome_arr, città_arr, durata)

dove le coppie di attributi {nome_part, città_part} e {nome_arr, città_arr} hanno un vincolo di integrità referenziale con la coppia di attributi {nome, città} dello schema di relazione PORTO.

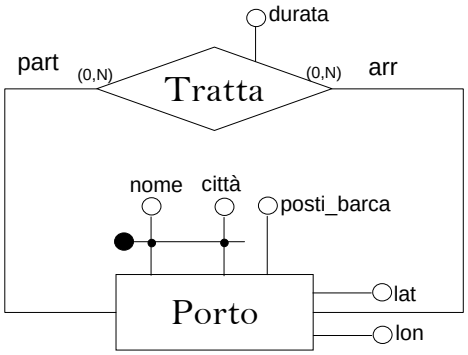


Figura 4.12: Associazione ricorsiva

La chiave dello schema di relazione corrispondente all'associazione è data dall'unione degli identificatori delle due entità coinvolte. Ad esempio nel primo caso, la chiave di INTERPRETA è la combinazione degli identificatori di FILM ed ATTORE, mentre nel secondo caso la chiave di tratta è data dalla combinazione di attributi {nome_part, città_part, nome_arr, città_arr}.
Si deve tenere presente che questo metodo è valido in generale qualunque sia la cardinalità dell'associazione.

Caso uno a molti In questo caso abbiamo due alternative: la prima è procedere come nel caso molti a molti osservando però quanto segue. Le occorrenze dell'associazione da tradurre sono in corrispondenza uno ad uno con (un sottoinsieme de) le occorrenze dell'entità che partecipa nell'associazione con cardinalità massima pari ad uno. Quindi la chiave della relazione risultante sarà pari alla chiave della relazione corrispondente a questa entità.

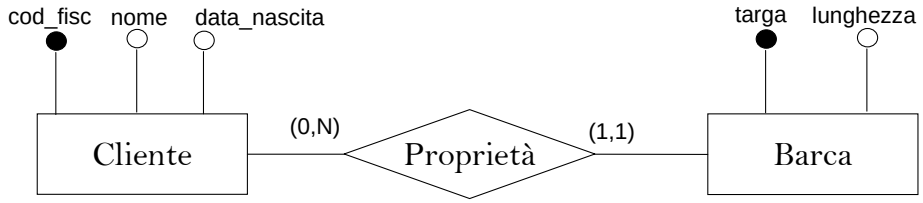


Figura 4.13: Traduzione uno-a-molti

Ad esempio nell'associazione PROPRIETÀ tra CLIENTE e BARCA, l'entità BARCA partecipa con cardinalità (1,1) (Fig. 4.13). La prima alternativa è quella di tradurre l'associazione con uno schema di relazione separato come segue

CLIENTE(cod_fisc, nome, pat_nautica, data_nascita)
BARCA(targa, lunghezza)
PROPRIETÀ(targa, cod_fisc)

Si noti che la chiave nello schema di relazione PROPRIETÀ è data dall'attributo targa poiché ogni barca è posseduta da esattamente un solo cliente.

La seconda alternativa è quella di accorpare lo schema di relazione PROPRIETÀ nello schema di relazione BARCA.

CLIENTE(cod_fisc, nome, pat_nautica, data_nascita)
BARCA(targa, lunghezza, proprietario)

Si noti come l'attributo cod_fisc viene rinominato più espressivamente in proprietario nello schema di relazione BARCA. Ci sarà pertanto, un vincolo di integrità referenziale tra l'attributo proprietario di BARCA e l'attributo cod_fisc di CLIENTE. Quest'ultima alternativa è da preferire sempre perché in questo modo si risparmia spazio e si semplifica lo schema risultante.

Se la cardinalità minima di BARCA fosse stata invece, pari a zero, e avessimo tradotto accorpando nello schema di relazione barca l'attributo proprietario allora ci sarebbero alcune occorrenze di BARCA, quelle che non sono possedute da alcun cliente, con valore nullo per questi attributi. Indicheremo ciò aggiungendo un asterisco in corrispondenza degli attributi che possono essere NULL.

CLIENTE(cod_fisc, nome, pat_nautica, data_nascita)
BARCA(targa,lunghezza, proprietario*)

Caso uno a uno Anche qui abbiamo diverse alternative. Possiamo comportarci come nel caso uno a molti. L'unica differenza è che il ruolo di una delle due entità è simmetrico se entrambe le cardinalità minime sono uguali. Se invece una delle due entità partecipa all'associazione con cardinalità minima pari zero e l'altra con cardinalità minima pari ad uno, è allora preferibile scegliere di accorpare tutti gli attributi nella relazione di questa ultima entità, evitando perciò i valori nulli. Ad esempio si consideri l'associazione uno-a-uno PRESIDIO

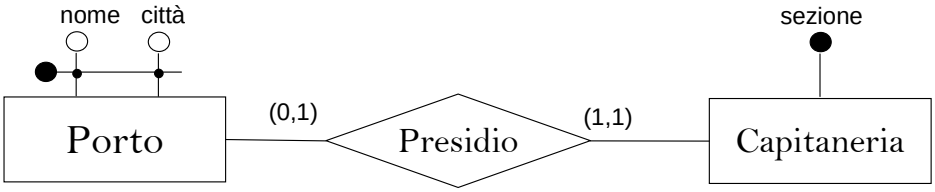


Figura 4.14: Traduzione uno-a-uno

tra PORTO e CAPITANERIA. Se entrambe le cardinalità minime fossero pari ad uno allora la scelta di accorpare l'associazione in uno schema oppure nell'altro è arbitraria. Se invece abbiamo come in Fig. 4.14, che PORTO partecipa con cardinalità minima pari a zero, allora è preferibile la seguente traduzione, che evita i valori nulli:

PORTO(nome, città)
CAPITANERIA(sezione, porto, città)

con vincolo di integrità referenziale tra la coppia {porto, città} e la chiave dello schema di relazione PORTO.

Traduzione di associazioni tra più di due entità Supponiamo di avere ora un associazione ternaria CROCIERA tra NAVE, ARMATORE e REGIONE per indicare di ogni crociera, il tipo di nave, l'armatore e la regione (mediterraneo, caraibi, ecc.) dove questa si svolge.

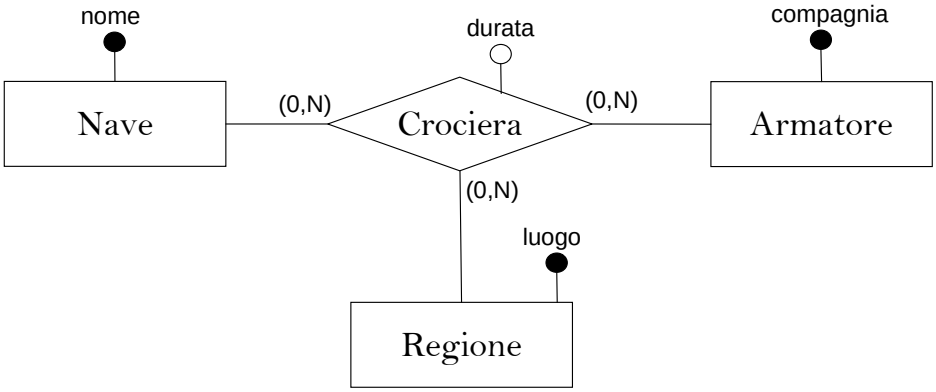


Figura 4.15: Traduzione associazioni ternarie

Traduciamo l'associazione in modo analogo al caso molti-a-molti. Si crea uno schema di relazione CROCIERA che ha come chiave l'insieme delle chiavi dell'entità coinvolte e gli attributi propri dell'associazione.

NAVE(nome)
ARMATORE(compagnia)
REGIONE(luogo)
CROCIERA(nave, armatore, regione, durata)

con il vincolo di integrità referenziale tra nave, armatore e regione di CROCIERA, rispettivamente con nome di NAVE, compagnia di ARMATORE e luogo di REGIONE.

Se una entità partecipa in un associazione ternaria con cardinalità massima pari ad uno allora è possibile tradurre l'associazione come nel caso uno-a-molti. Ad esempio nell'associazione VISIONE tra POLTRONA, PROGRAMMA e BIGLIETTO di Fig 2.3, l'entità BIGLIETTO partecipa con cardinalità massima pari ad uno. Possiamo quindi accorpare nello schema di relazione di BIGLIETTO gli identificatori delle altre entità.

PROGRAMMA(codice)
POLTRONA(identificativo)
BIGLIETTO(numero, poltrona, proiezione)

con il vincolo di integrità referenziale tra poltrona e proiezione, rispettivamente con identificativo di POLTRONA e codice di PROGRAMMA.

Nelle associazioni binare molti-a-molti la chiave dello schema di relazione corrispondente è sempre l'insieme delle chiavi delle due entità coinvolte. Viceversa nel caso di associazioni ternarie questo non è sempre vero. Ad esempio si consideri l'associazione TIMONIERE, tra le entità SKIPPER, BARCA e REGATA di Fig. 4.16. Si suppone che uno skipper possa timonare diverse barche in diverse regate, ma in una stessa regata lo skipper non possa condurre due barche differenti. L'associazione è molti-a-molti. Si osservi che una barca ed una regata determinano univocamente lo skipper oppure una regata ed uno skipper determinano univocamente la barca. Quindi gli insiemi {barca, regata} e {skipper, regata} sono chiavi minimali dello schema di relazione TIMONIERE.

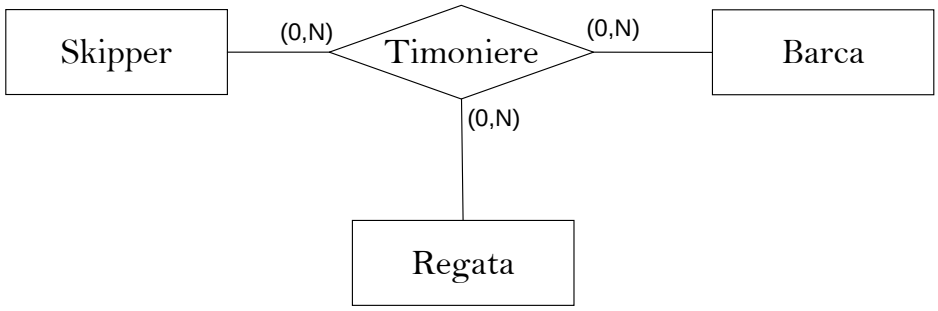


Figura 4.16: Identificatori minimali in associazioni ternarie

Associazioni multiple tra due entità La regola generale è che la traduzione di un'associazione comporta la creazione di uno schema di relazione che ha come attributi le chiavi delle entità coinvolte, più gli attributi propri dell'associazione. Nel caso uno-a-molti e in quello uno-a-uno preferiamo tradurre l'associazione inserendo, nell'entità che partecipa con cardinalità massima pari ad uno, gli attributi dell'identificatore dell'altra entità. Nel caso che ci siano più associazioni tra le due entità, l'operazione precedente va ripetuta per ogni associazione, ma necessariamente occorre rinominare, di volta in volta l'identificatore della seconda entità. Ad esempio si consideri lo schema di Fig. 4.17. L'entità CLIENTE partecipa alle associazioni NASCITA e RESIDENZA con

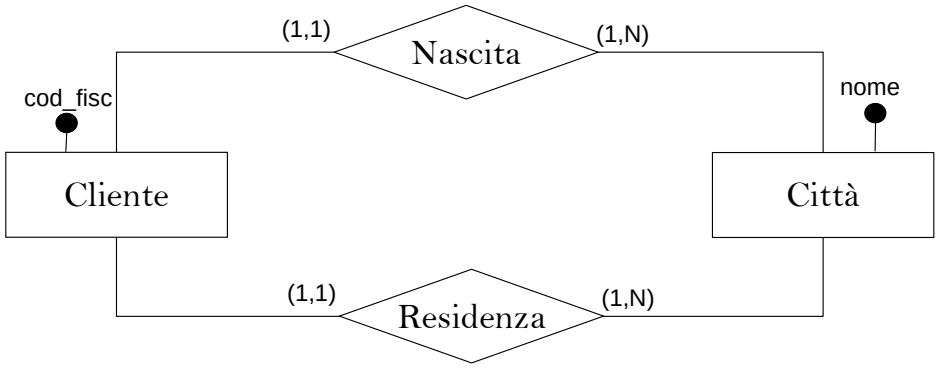


Figura 4.17: Traduzione di associazioni multiple

cardinalità massima pari ad uno. Se vogliamo rappresentare tali associazioni accorpando l'identificatore di CITTÀ in CLIENTE allora dobbiamo aggiungere due volte l'attributo nome di CITTÀ nello schema di relazione di CLIENTE. Dobbiamo però necessariamente rinominare tale attributo per ciascuna associazione in aggiunta alla prima. Quindi lo schema risultante sarà

CITTÀ(nome)
CLIENTE(cod_fisc, nato, residente)

dove entrambi gli attributi nato e residente hanno un vincolo di integrità referenziale con l'attributo nome di CITTÀ.

4.2.7 Esempio di progettazione logica

Continuiamo la progettazione dell'applicazione Cinema effettuando la progettazione logica a partire dallo schema Entità-Associazione di Fig. 4.4.

-Tavola dei volumi Il primo passo da effettuare è compilare la tavola dei volumi dell'applicazione. Sulla base delle specifiche utente ricaviamo le cardinalità delle entità coinvolte. Abbiamo che il numero delle sale è pari a 400 e poiché ogni sala mediamente contiene 250 poltrone, allora ci saranno complessivamente 100.000 poltrone. Per quanto riguarda i programmi, sappiamo che ogni sala effettua mediamente 5 proiezioni al giorno. Pertanto moltiplicando questo numero per il numero delle sale, otteniamo un totale di 2.000 programmazioni giornaliere. Dopo 3 anni di servizio (un tempo di vita, verosimile, del sistema) avremmo accumulato $2.000 \times 365 \times 3 = 2.190.000$ programmazioni. Infine se per ogni programmazione la sala è piena al 60% abbiamo che il numero di biglietti che devono essere registrati è $2.190.000 \times 250 \times 0,6 = 328.500.000$. I film nuovi che vengono proiettati in un anno sono 400 per un totale di 1.200 film in 3 anni. Se per ogni film ci sono mediamente 10 attori principali e se ogni attore in media ha recitato in 3 film possiamo ragionare come segue. Ogni occorrenza di film partecipa all'associazione INTERPRETA mediamente 10 volte. Quindi l'associazione interpreta ha un numero complessivo di 12.000 occorrenze. Se ogni attore ha recitato mediamente in 3 film, allora devono esserci, all'incirca, $12.000/3 = 4.000$ attori registrati nella base di dati.

La numerosità delle associazioni uno-a-uno o uno-a-molti è facilmente calcolata poiché è pari al volume dell'entità che partecipa con cardinalità massima pari ad uno. Riportiamo i valori di questa tavola in Tabella 4.2.

Tipo	**Costrutto**	**Volume**
E	Sala	400
E	Poltrona	100.000
E	Programma	2.190.000
E	Biglietto	328.500.000
E	Film	1.200
E	Attore	4.000
A	Acquisto	328.500.000
A	Fa_parte_di	100.000
A	Spettacolo	2.190.000
A	Visione	328.500.000
A	Proiezione	2.190.000
A	Interpreta	12.000

Tabella 4.2: Tavola dei volumi applicazione Cinema

-Generalizzazioni, Partizionamenti ed Accorpamenti, Attributi multivalore Nello schema non esistono generalizzazioni, né esistono attributi multivalore. Possiamo osservare però che l'entità biglietto è effettivamente molto grande. Esistono infine molte operazioni che insistono direttamente o indirettamente in questa entità. L'operazione 1 richiede di inserire un biglietto, la 3 che richiede di trovare 3 posti liberi consecutivi in una fila, la 4 che richiede di sapere i posti disponibili di una proiezione e la 6 che richiede di trovare i film che hanno avuto

l'incasso maggiore. Poiché le prime tre delle suddette operazioni riguardano solo i biglietti della giornata in corso, ciò suggerisce la possibilità di partizionare l'entità biglietto in due entità BIGLIETTO e BIGLIET_STOR. La prima contenente i biglietti della giornata in corso e l'altra tutti biglietti dei giorni precedenti. In questo modo l'entità BIGLIETTO conterrà soltanto $2.000 \times 250 \times 0,6 = 300.000$ occorrenze al massimo. L'introduzione di questa partizione modificherà lo schema concettuale come indicato in Fig. 4.18[2].

-Identificatori principali Si devono ora individuare gli identificatori principali dello schema. Notiamo che l'entità ATTORE non possiede un identificatore poiché nome e data_nasc potrebbero essere uguali per due attori differenti. Aggiungiamo quindi a questa entità un attributo codice che identificherà ogni attore.

Osserviamo che, essendo l'identificatore di SALA la coppia {numero, multisala} ed essendo l'entità SALA identificatore esterno per POLTRONA e PROGRAMMA, ciò comporta che in PROGRAMMA e POLTRONA l'identificatore sarà composto da molti byte. A peggiorare le cose c'è il fatto che gli identificatori sia di PROGRAMMA che di POLTRONA verranno utilizzati per la traduzione nel modello relazionale delle associazioni VISIONE, ACQUISTO, VIS_STOR e ACQ_STOR che possiedono complessivamente centinaia di milioni di occorrenze. Questo comporterebbe da una parte uno spreco di spazio e dall'altra, un rallentamento delle operazioni di join tra le entità coinvolte. Per ovviare a ciò si introducono tre nuovi attributi cod_sala, cod_prog e cod_poltr, rispettivamente nelle entità SALA, POLTRONA e nell'entità PROGRAMMA che diventeranno gli identificatori principali di queste entità. Lo schema ottenuto al termine di questa fase è indicato in Fig. 4.18.

-Traduzione nello schema relazionale A questo punto possiamo tradurre lo schema di Fig. 4.18 nello schema relazionale. Iniziamo a tradurre le entità. La traduzione creerà i seguenti schemi di relazione

BIGLIETTO(numero, data_emis, costo)
BIGLIET_STOR(numero, data_emis, costo)
POLTRONA(cod_poltr, fila, numero)
SALA(cod_sala, numero, multisala, posti)
PROGRAMMA(cod_prog, data_ora, durata, posti_occ, posti_disp)
FILM(titolo, anno, durata, regista)
ATTORE(codice, nome, data_nasc)

A questo punto cominciamo a tradurre le associazioni. Notiamo che tutte le associazioni, tranne INTERPRETA, sono uno-a-molti. Tradurremo queste associazioni accorpando l'identificatore dell'entità che partecipa con cardinalità massima pari ad N nello schema di relazione corrispondente all'entità che partecipa con cardinalità massima pari ad 1. Mentre l'associazione INTERPRETA verrà tradotta in uno schema di relazione separato. Lo schema di database finale è quindi quello mostrato in Fig. 4.19.

In più occorre specificare i vincoli di integrità referenziale di seguito indicati

BIGLIETTO(programma) $\rightarrow$ PROGRAMMA(cod_prog)
BIGLIETTO(poltrona) $\rightarrow$ POLTRONA(cod_poltr)

[2]Per semplicità viene omesso ma si dovrebbe valutare se partizionare l'entità PROGRAMMA nello stesso modo in cui abbiamo partizionato BIGLIETTO. Si invita lo studente a verificare la convenienza o meno di tale operazione

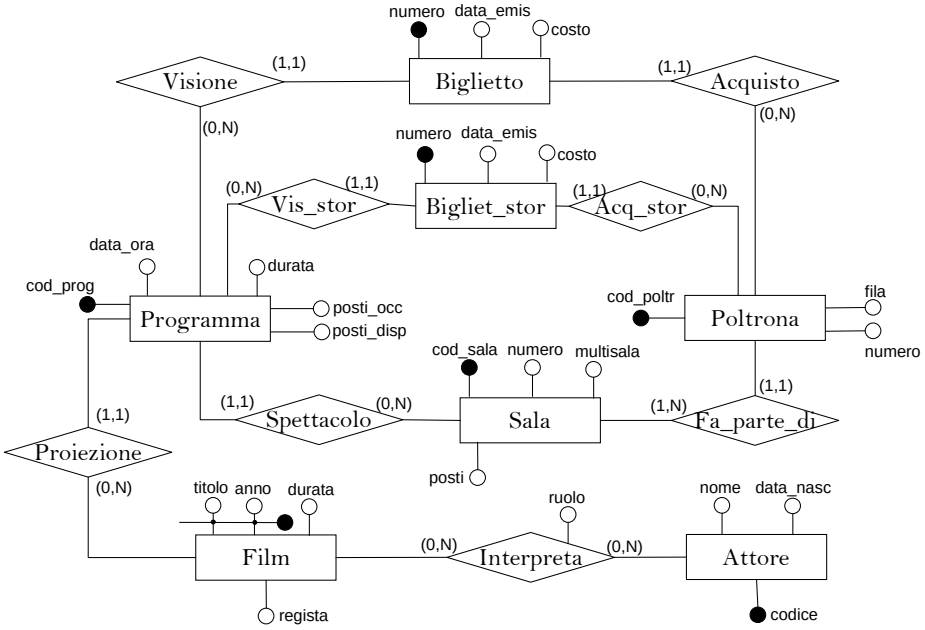


Figura 4.18: Lo schema concettuale dopo la fase della progettazione logica

BIGLIETTO(numero, data_emis, costo, programma, poltrona)
BIGLIET_STOR(numero, data_emis, costo, programma, poltrona)
POLTRONA(cod_poltr, fila, numero, sala)
SALA(cod_sala, numero, multisala, posti)
PROGRAMMA(cod_prog, data_ora, durata, posti_occ, posti_disp, film, anno, sala)
FILM(titolo, anno, durata, regista)
ATTORE(codice, nome, data_nasc)
INTERPRETA (attore, film, anno, ruolo)

Figura 4.19: Lo schema relazionale dell'applicazione Cinema

BIGLIET_STOR(programma) → PROGRAMMA(cod_prog)
BIGLIET_STOR(poltrona) → POLTRONA(cod_poltr)
POLTRONA(sala) → SALA(cod_sala)
PROGRAMMA(film, anno) → FILM(titolo, anno)
PROGRAMMA(sala) → SALA(cod_sala)
INTERPRETA (film, anno) → FILM(titolo, anno)
INTERPRETA (attore) → ATTORE(codice)

-*Verifica della correttezza dello schema* Infine verifichiamo che nello schema di database nessun schema relazionale violi la BCNF. Occorre individuare sulla base delle specifiche del mini-mondo le dipendenze funzionali che sussistono in ciascuno schema. Una volta fatto ciò dobbiamo individuare se esiste qualche dipendenza funzionale che viola la BCNF.

Nel nostro esempio abbiamo che

- Gli schemi BIGLIETTO e BIGLIETTO_STOR hanno chiave {numero} e nessun'altra dipendenza funzionale è in essi presente
- Lo schema POLTRONA ha chiavi {fila, numero, sala} e {cod_poltr} e nessun'altra dipendenza funzionale è in esso presente
- Lo schema SALA ha chiavi {cod_sala} e {numero, multisala} e nessun'altra dipendenza funzionale è in esso presente
- Lo schema FILM ha chiave {titolo, anno} e nessun'altra dipendenza funzionale è in esso presente
- Lo schemaATTORE ha chiave {codice} e nessun'altra dipendenza funzionale è in esso presente
- Lo schema PROGRAMMA ha chiavi {cod_prog} e {data_ora, sala}. Potrebbe sussistere una dipendenza funzionale titolo, anno→ durata che viola la BCNF poiché {titolo, anno} non è una chiave di PROGRAMMA. Se così fosse per eliminare tale violazione sarebbe sufficiente introdurre un'attributo durata_proiez. nello schema FILM ed eliminare l'attributo durata dallo schema PROGRAMMA
- Lo schema INTERPRETA ha chiave {attore, film, anno} e nessun'altra dipendenza funzionale è in esso presente

Lo schema soddisfa la BCNF e questo conferma la bontà della progettazione effettuata.

4.3 Esercizi

Esercizio 4.1 Eseguire la progettazione logica degli schemi ottenuti dalle specifiche degli esercizi dal 2.1 al 2.7.

Esercizio 4.2 Eseguire la progettazione concettuale e logica dell'applicazione Porti.

Esercizio 4.3 Traccia d'esame dell'AA06-07. Effettuare la progettazione concettuale e logica. Suggerimento: si utilizzi il costrutto della generalizzazione; si utilizzino opportune associazioni tra le entità figlie della generalizzazione.

Specifiche sui dati

Si vuole costruire un database per la catalogazione degli oggetti celesti e per la loro osservazione. Gli oggetti celesti si distinguono in satelliti artificiali e corpi naturali. I corpi naturali si distinguono a loro volta in : asteroidi e comete, satelliti, pianeti, stelle e galassie. Ci sono altri corpi astrali che devono essere considerati (c.a 10.000) come come nebulose, stelle a neutroni, nane bianche, buchi neri, ecc.. Ogni oggetto astrale possiede un codice identificativo (ad es. M12 o SN01A) ed un eventuale nome (Marte, Sirio, Ceres, ecc.), la sua brillantezza apparente (indicata da un numero) e le sue coordinate astrali : Ascensione Retta (Right Ascension) e Declinazione (Declination), entrambe espresse in gradi, minuti e secondi. Occorre memorizzare, per gli oggetti fuori dal sistema solare, anche il valore numerico $\mathbb{Z}$ che indica il Redshift del corpo luminoso (Redshift

= spostamento verso il rosso). Deve essere indicato il diametro e le coordinate (x, y, z), rispetto ad un sistema di riferimento che ha come centro il sole, di comete e asteroidi (circa 370.000 tra comete ed asteroidi). Occorre indicare il pianeta orbitato, il periodo e l'angolo di inclinazione dell'orbita dei satelliti naturali (circa 850 a catalogo). I pianeti (circa 350 comprensivi degli otto pianeti del sistema solare) orbitano intorno ad una stella e posseggono dei satelliti che a loro volta orbitano il pianeta considerato (ad es. la terra ha un satellite, la luna, ed orbita intorno ad una stella, il sole). Bisogna indicare se il pianeta è gassoso o solido, il periodo dell'orbita e la sua eccentricità quest'ultimo indicato da un numero. Le stelle (circa 10 milioni) possiedono un codice, una classe e la massa. A loro volta le stelle hanno dei pianeti orbitanti e sono parte di una galassia di provenienza. Sono classificate come stelle solo quelle che hanno una massa superiore ad 1/2 di quella del sole. Le stelle con massa inferiore ad 1/2 di quella del sole sono classificate come nane bianche. Le galassie (2.000.000) sono classificate secondo la loro forma (a spirale, ellittica, irregolare, ecc.). Infine si vuole sapere l'anno di messa in orbita dei satelliti artificiali (c.a 1.500). Si vogliono altresì registrare le osservazioni effettuate dagli astronomi (c.a 50.000) - per i quali va registrato il nome, la qualifica e l'istituto o università di appartenenza - su questi corpi celesti dai vari osservatori sparsi sulla terra. Per ogni osservazione di un oggetto luminoso va indicata la data, l'ora ed il minuto in cui inizia l'osservazione e la sua durata, il corpo celeste osservato e l'osservatorio da cui è stata realizzata l'osservazione, l'ascensione retta e la declinazione dell'oggetto osservato. Al fine di poter tracciare eventi transitori, per ogni oggetto celeste si deve conoscere in ogni istante il numero di volte che è stato sottoposto ad osservazione. Tutte le osservazioni registrate nel database dovranno rimanere in linea per tutta la durata in vita del sistema che si presume sia di almeno cinque anni. Si noti che la istituzione proprietaria di un telescopio (c.a 600) - per la quale occorre memorizzare il nome, la nazionalità ed il numero di telefono - potrebbe coincidere con l'istituto o università di appartenenza di astronomi che effettuano osservazioni anche su telescopi diversi da quelli dell'istituzione a cui appartengono.

Specifiche sulle operazioni

1. Registrare una osservazione di un oggetto astronomico. (frequenza 14.000 volte al giorno).

2. Modificare l'ascensione retta, la declinazione e le coordinate per gli oggetti celesti in movimento come comete, asteroidi, satelliti artificiali, ecc. (3000 volte al giorno)

3. Trovare le terne di asteroidi o comete la cui reciproca distanza è minore di un fissato parametro. (3 volte alla settimana) (Si definisca la distanza tra due oggetti o_1 e o_2 con coordinate rispettivamente (x_1, y_1, z_1) e (x_2, y_2, z_2) come $d(o_1, o_2) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2}$)

4. Stampare i dati delle galassie che contengono almeno una stella orbitata da almeno due o più pianeti.(4 volte al mese)

5. Calcolare, sulla base dei dati storici, il numero di telescopi, che erano inutilizzati una qualunque settimana dell'anno precedente. (1 volta alla settimana)

6. Stilare la lista delle 5.000 galassie più luminose. (70 volte al giorno)
7. Trovare la o le stelle che abbiano un data massa. (3.000 volte al giorno)
8. Calcolare per una data galassia rapporto tra il numero di stelle osservate appartenenti alla galassia ed il numero di stelle presenti a catalogo per quella galassia. (700 volte al giorno)
9. Calcolare il numero complessivo delle osservazioni dell'ultimo anno suddiviso per istituzione che possiede il telescopio dal quale è stata effettuata la osservazione. (10 volte al mese)
10. Calcolare, sulla base dei dati storici, il tempo medio di utilizzo di un telescopio. (10 volte al mese)

Esercizio 4.4 Traccia d'esame dell'AA08-09. Effettuare la progettazione concettuale e logica.

Specifiche sui dati

Una software house deve realizzare un browser-game multiutente di strategia on-line. Il gioco prevede la registrazione prevalentemente gratuita e punta ad ottenere nei due anni di vita della prima release, 100.000 utenti.

Il gioco consiste nel realizzare ed ingrandire quanti più villaggi sia attraverso azioni di sviluppo che attraverso combattimenti con altri avversari. Ogni giocatore, per il quale occorre generare un nome utente, un nickname, una password e per cui occorre registrare sesso, data di nascita e città di residenza, possiederà un villaggio al momento della sua registrazione. Il compito di ciascun utente sarà quello di sviluppare il proprio villaggio, difenderlo dagli eventuali attacchi degli altri giocatori o conquistare villaggi di altri giocatori.

Ciascun villaggio possiede un nome e possiede delle infrastrutture per la produzione di beni: *campi di grano*, *miniere di argilla*, *miniere di ferro* e *foreste*. Ci saranno delle infrastrutture civili come: il *centro città*, il *municipio*, il *granaio*, il *magazzino*, il *forno*, il *fabbro* ed il *mercato*. Infine ci sono delle infrastrutture militari: la *caserma*, la *scuderia*, l'*armeria* e l'*arena* ed il *castello*.

Ciascuna delle infrastrutture di cui sopra dovrà essere sviluppata dal livello 0 (quello in cui infrastruttura è assente), fino al livello 20. Per poter sviluppare un'infrastruttura al livello successivo è necessario possedere e spendere una determinata quantità di risorse tra argilla, grano, ferro e legno. La quantità di risorse necessarie per passare da un livello all'altro è diversa per ciascun livello (quanto più aumenta il livello tante più risorse occorrono per passare al livello successivo) e ogni infrastruttura ha bisogno di quantità specifiche e diverse dalle altre infrastrutture.

Ad ogni infrastruttura di livello l viene assegnato un numero di abitanti pari a $(3 + 2 \times l)$. Ogni villaggio avrà un numero di abitanti pari alla somma degli abitanti di ciascuna infrastruttura. Occorre registrare la data ora e minuto dell'ultimo passaggio di livello di ciascuna infrastruttura.

Alla registrazione di un utente o alla generazione di un nuovo villaggio, nel magazzino sarà disponibile una quantità pari a 200 di grano, 200 di ferro, 200 di argilla e 200 di legno. Ci saranno inoltre 4 campi di grano, 4 miniere di ferro 4 di argilla e 4 foreste, tutte al livello 1. Inoltre ci sarà un municipio ed un centro città entrambi a livello 1. Ogni campo di grano, miniera di ferro e argilla

e ogni foresta fornirà un produzione oraria pari a $(100 + l^2 \times 50)$ unità dove l è il livello corrente di sviluppo dell'infrastruttura. Ogni altra infrastruttura, civile o militare, dovrà essere costruita e sviluppata ex-novo. Per ogni villaggio si potranno creare fino a un totale 6 campi di grano, 6 miniere di ferro, 6 di argilla e 6 foreste e fino a 3 infrastrutture dello stesso tipo (ad es. 3 magazzini, 3 municipi, ecc.).

Ciascun villaggio sarà casualmente posizionato, all'atto della registrazione dell'utente, su di una griglia discreta di 750×750 caselle dove a ciascuna casella corrisponderà o un villaggio oppure un territorio libero per fondare nuovi villaggi. Ad ogni villaggio verrà assegnata pertanto una coppia di coordinate comprese tra $+375$ e -375. La distanza tra due villaggi è ottenuta dalla normale distanza euclidea sul piano misurata in numero di caselle e arrotondata all'intero superiore.

Le infrastrutture della caserma, della scuderia e dell'armeria consentono di generare truppe. La caserma genera le truppe di fanteria: *legionari*, *pretoriani* e *imperiani*. La scuderia consente di generare le truppe di cavalleria: *l'imperiano a cavallo* ed il *legionario a cavallo*. L'armeria consente di generare *catapulte* ed *arieti*. Per generare una unità si richiede di possedere nel villaggio e spendere un certo numero di risorse in argilla, grano, ferro e legno diverse per ciascuna unità militare.

Ciascuna unità militare ha un punteggio di attacco ed un punteggio di difesa. Per conquistare un villaggio di un altro avversario occorre prima di tutto distruggere tutte le truppe presenti nel villaggio. Questo può essere fatto portando uno o più attacchi. L'attacco avviene selezionando le truppe prescelte e le coordinate del villaggio da attaccare. Occorre memorizzare i dati di tutti gli attacchi effettuati nel gioco. Per ciascun attacco si vuole conoscere il tipo e la quantità di truppe attaccanti, il tipo e la quantità di truppe difendenti, la data ed ora dell'attacco ed il tipo ed il numero di truppe perse sia dell'attaccante che del difendente.

In ogni istante, per ogni villaggio, occorre conoscere il numero complessivo di abitanti e il numero di truppe militari presenti nel villaggio.

Specifiche sulle operazioni

1. Registrare un nuovo utente e la creazione di un nuovo villaggio (580 volte/ giorno)
2. Registrare l'esecuzione di un attacco (100.000 volte / giorno)
3. Registrare l'avvivo del passaggio di livello di una nuova infrastruttura (40.000 volte / giorno)
4. Registrare la creazione di un nuova unità militare (500.000 volte / giorno)
5. Stampare la somma delle quantità di di grano, ferro argilla e legno presenti in tutti i magazzini di un villaggio (16.000 volte / giorno)
6. Trovare i gruppi di 4 giocatori che hanno villaggi la cui reciproca distanza è minore di 3 caselle (3 volte/ mese)
7. Trovare il numero di utenti suddiviso per sesso, età e città di residenza (500 volte/anno)

8. Trovare i dieci giocatori che hanno il più alto numero di abitanti (300 volte/gg)
9. Trovare il livello medio di sviluppo di una infrastruttura (sia essa civile o militare) per tutti gli utenti attivi del gioco (3 volte al mese)
10. Per un dato giocatore stampare il tipo e la quantità di truppe presenti in un suo villaggio (50.000 volte / giorno)

Capitolo 5

Il linguaggio SQL

Questo capitolo introdurremo il linguaggio SQL. Tale linguaggio ha avuto un ampia diffusione nei sistemi industriali e grazie alla definizione di uno standard internazionale, è oggi comunemente utilizzato nei sistemi di gestione delle basi di dati commerciali. L'obiettivo di questo capitolo è quello di fornire allo studente una panoramica degli elementi essenziali e maggiormente utilizzati del linguaggio, mantenendo la trattazione semplice ed essenziale. Gli ulteriori approfondimenti della sintassi dell'SQL possono essere trovati nei manuali dei prodotti DBMS o facendo riferimento ai testi riportati nella bibliografia.

5.1 Tipi di dato

I tipi di dati elementari supportati da SQL sono simili ai tipi di dato supportati dai più diffusi linguaggi di programmazione. Abbiamo tipi di dati numerici, di testo, tipi per indicare data ed ora ed altri tipi per supportare le recenti applicazioni multimediali o per la gestione dei siti web.

Tipi di dati numerici. I tipi di dati numerici comprendono i valori interi con diverse grandezze come `INTEGER` o `SMALLINT` o `BIGINT`. È possibile specificare dati numerici con una precisione esatta con `NUMERIC(`*precisione*, *scala*`)` dove il valore *precisione* indica il numero complessivo di cifre del dato e la *scala* indica il numero di cifre decimali. Ad esempio `lunghezza NUMERIC(10,4)` dichiara l'attributo `lunghezza` come tipo di dato numerico con 10 cifre complessive di cui 4 decimali. Inoltre esistono i tipi per rappresentare i numeri in virgola mobile come `FLOAT` e `DOUBLE PRECISION`. SQL supporta il tipo di dato `BIT` per rappresentare un dato booleano oppure `BIT(`n`)` per rappresentare una stringa di n bit.

Tipi di dati per la gestione di stringhe. Per la memorizzazione delle stringhe abbiamo i tipi `CHARACTER` (oppure `CHAR`) per un singolo carattere oppure `CHARACTER(`n`)` o anche più brevemente `CHAR(`n`)` per rappresentare una stringa a lunghezza fissa di n caratteri. Per le stringhe di lunghezza variabile si usa `CHARACTER VARYING` o semplicemente `VARCHAR`. Se si vuole comunque limitare la grandezza della stringa si può utilizzare `VARCHAR(`n`)` per rappresentare le stringhe di lunghezza variabile ma minore od uguale ad n.

Tipi di dati per rappresentare data ed ora. Il linguaggio SQL supporta i tipi di dato `DATE` per rappresentare le date. Il formato prevede la memorizzazione di otto posizioni nella forma `aaaa/mm/gg` dove `aaaa` sono quattro cifre per l'anno, `mm` due cifre per il mese e `gg` due cifre per il giorno. Viene supportato il tipo di dato `TIME` per indicare l'ora. Tale tipo prevede sei posizioni nella forma `hh:mm:ss` dove `hh` sono due cifre per l'ora `mm` due cifre per il minuto e `ss` due cifre per il secondo. È opzionalmente possibile specificare una precisione p attraverso `TIME(`p`)` per indicare il numero di cifre da utilizzare per la memorizzazione delle frazioni di secondo. Ad esempio `arrivo TIME(3)` dichiara un attributo `arrivo` come tipo di dato `TIME` nel quale è possibile memorizzare ora, minuto e secondo e le frazioni di secondo fino ai millesimi. Il tipo di dato `TIMESTAMP` consente di memorizzare contemporaneamente la data e l'ora. Quindi in `TIMESTAMP` abbiamo una combinazione dei formati `DATE` e `TIME`. Come nel formato `TIME` è possibile specificare una precisione p attraverso `TIMESTAMP(`p`)` per indicare il numero di cifre massimo per memorizzare le frazioni di secondo. SQL mette a disposizione il tipo di dato `INTERVAL` per gestire intervalli di tempo. È possibile opzionalmente aggiungere il tipo di intervallo con le clausole `year, month, day, hour, minute` e `second`, ad indicare se l'intervallo è di anni, mesi ecc.. In questo modo è possibile gestire le operazioni di differenza tra due variabili temporali. Ad esempio `durata INTERVAL MINUTE` dichiara l'attributo durata come un tipo intervallo di minuti.

Tipi di dato per applicazioni multimediali e web. Recenti applicazioni di tipo multimediale e web hanno reso necessaria l'introduzione di nuovi tipi di dato per supportare file multimediali come brani audio, immagini, video e filmati. Per questo è stato introdotto il tipo `BLOB` che stà per Binary Large Object che consente di dichiarare attributi per la memorizzazione di dati binari non strutturati di grandi dimensioni, nell'ordine di centinaia o migliaia di MB.

5.2 Il comando `CREATE TABLE`

Lo standard SQL fornisce un insieme di comandi ed istruzioni per supportare il modello logico relazionale. I DBMS che implementano lo standard SQL sono anche detti sistemi di gestione delle basi di dati relazionali. In questi ambienti in luogo dei termini relazione, attributo ed ennupla vengono utilizzati propriamente i termini *tabella*, *colonna* e *riga* rispettivamente.

La più importante differenza tra una relazione ed una tabella è che mentre ogni relazione è un insieme finito di ennuple, una tabella, per motivi di praticità ed efficienza, *non* è sempre un insieme di righe. Possono infatti esistere in una tabella due o più righe che contengono gli stessi valori ovvero possono esserci righe *duplicate*. Vedremo tra breve che questo scostamento dal modello relazionale può essere però facilmente ovviato poiché in SQL è possibile vincolare una tabella in modo che non esistano mai due righe duplicate. Con la presenza di tali vincoli è possibile trattare equivalentemente una tabella come fosse una relazione.

Nel seguito useremo la convenzione che, quanto posto tra i simboli $< >$, indichi un argomento obbligatorio, mentre quello posto tra i simboli [ ] un argomento opzionale. Per definire una tabella viene utilizzato il comando `CREATE TABLE`. La sintassi del comando è la seguente:

```
CREATE TABLE <nome_tabella> (
   <nome_attr1> <tipo_dato1> [vincoli1] ,
   <nome_attr2> <tipo_dato2> [vincoli2] ,
   ...
   <nome_attrk> <tipo_datok> [vincolik] ,
[altri_vincoli]
);
```

Con il comando `CREATE TABLE` creiamo una tabella con nome <*nome_tabella*> ed attributi nome_attr$_1$,nome_attr$_2$, ... , nome_attr$_k$ ai quali sono associati i tipi di dato *tipo_dato*$_1$, *tipo_dato*$_2$, *tipo_dato*$_k$ ed ai quali sono associati i vincoli *vincoli*$_1$, *vincoli*$_2$, *vincoli*$_k$, rispettivamente. I vincoli più importanti che possono essere espressi nel comando `CREATE TABLE` sono i vincoli di *chiave primaria* ed i vincoli di *chiavi esterne*.

Il vincolo di chiave primaria consente di specificare un insieme di attributi che rappresenterà la chiave primaria della tabella. In questo modo la tabella non potrà mai contenere righe duplicate. Il vincolo di chiave può essere specificato in due modi distinti in dipendenza del fatto se la chiave primaria della tabella è composta da uno o più attributi. Se la chiave della tabella è composta da un singolo attributo, allora il vincolo di chiave può essere specificato utilizzando la clausola `PRIMARY KEY` in corrispondenza della dichiarazione dell'attributo. Se invece la chiave è composta da più attributi <*nome_attr*$_{i_1}$, ..., *nome_attr*$_{i_h}$> ciò può essere specificato come [*altri_vincoli*] con la sintassi

```
PRIMARY KEY ( <nome_attri1, ..., nome_attrih> )
```

Ad esempio supponiamo di voler creare una tabella corrispondente allo schema di relazione Barca(targa, nome, lunghezza). Allora potremmo utilizzare il comando

```
CREATE TABLE Barca (
   targa CHAR(10) PRIMARY KEY,
   nome VARCHAR(25),
   lunghezza NUMERIC(5,2)
);
```

Se vogliamo creare una tabella Porto(nome, città, numero_posti) nel quale la chiave primaria è specificata dall'insieme di attributi {nome, città} allora utilizzeremo il seguente

```
CREATE TABLE Porto (
   nome CHAR(20),
   città CHAR(20),
   posti_barca INTEGER,
   PRIMARY KEY (nome, città)
);
```

Il vincolo di integrità referenziale tra uno o più attributi di una tabella e i corrispondenti attributi della tabella referenziata può essere specificato in dipendenza del fatto se gli attributi referenziati sono uno o più di uno. Nel primo caso indicheremo la tabella e l'attributo referenziato, preceduto dalla clausola

REFERENCES, accanto alla definizione dell'attributo referenziante, come di seguito indicato

REFERENCES <*nome_tabella_referenz.*> (<*nome_attr*>)

Ad esempio supponiamo di voler creare la tabella ORMEGGIO(targa, città, nome, data_arrivo). Allora utilizzeremo il seguente comando

```
CREATE TABLE Ormeggio (
   barca CHAR(10) REFERENCES Barca(targa),
   città CHAR(20),
   porto CHAR(20),
   data_arrivo DATE,
   PRIMARY KEY (porto, città, barca),
   FOREING KEY (porto, città) REFERENCES Porto(nome, città)
);
```

Si noti che se il vincolo di integrità referenziale coinvolge due o più attributi allora il modo per specificare ciò, è quello di utilizzare la clausola

FOREING KEY ($<A_1,\ldots,A_h>$)
REFERENCES <*tab_referenziata*>($<B_1,\ldots,B_h>$)

Dove $A_1,\ldots,A_h$ è la sequenza di attributi della tabella referenziante mentre <*tab_referenziata*> e $B_1,\ldots,B_h$ è il nome e l'elenco di attributi della tabella referenziata.

È possibile, nella dichiarazione di un attributo aggiungere ulteriori vincoli come il vincolo NOT NULL per specificare se un attributo può o non può ammettere valori NULL. Il vincolo UNIQUE consente di imporre che non esistano due righe della tabella con valori uguali su uno o più attributi specificati. Anche qui abbiamo due possibilità nell'utilizzare tale vincolo. Se vogliamo imporre che non esistano nella tabella due righe che prendono lo stesso valore su di un singolo attributo, allora si indica la clausola UNIQUE nella riga di dichiarazione dell'attributo. Ad esempio

```
CREATE TABLE Porto (
   nome CHAR(20) NOT NULL,
   città CHAR(20) NOT NULL,
   nome_turistico VARCHAR (30) UNIQUE NOT NULL,
   posti_barca INTEGER,
   UNIQUE (nome, città)
);
```

Con questo comando creeremo una tabella PORTO dove il nome_turistico di ogni porto deve essere unico e diverso da NULL. Inoltre si specifica che non debbano esistere due righe della tabella che hanno gli stessi valori nella coppia di attributi (nome, città). In effetti la combinazione di UNIQUE e NOT NULL corrisponde esattamente ad un vincolo di chiave. La differenza tra UNIQUE e PRIMARY KEY consiste nel fatto che nella prima sono ammessi i valori nulli su uno o più attributi specificati, mentre nella seconda i valori nulli non sono ammessi.

In aggiunta ad i vincoli finora visti è possibile specificare una tipologia di vincoli più complessi con la clausola CHECK. Tipicamente tale clausola può essere

Barca	targa	nome	lunghezza
	AB1234CD	Azzurra	42,34
	AC2347LM	Fila	123,45
	RD3456ST	Rex	154,98
	BD2345DB	Nina	48,33
	UV9876VU	Vespucci	132,56
	EF9876VU	Azzurra	12,56

Porto	nome	città	posti_barca	lat	lon
	VECCHIO	NAPOLI	234	40,81	-14,20
	DARSENA FIUMIC.	ROMA	123	41,72	-12,21
	MESTRE	VENEZIA	532	45,60	-12,23
	EMPEDOCLE	AGRIGENTO	187	37,31	-13,52
	FIUMARA	ROMA	654	41,71	-12,25
	PORTOFERRAIO	PORTOFERRAIO	57	42,82	10,30
	NETTUNO	NETTUNO	362	41,60	12,62

Ormeggio	nome	città	targa	data_arrivo
	VECCHIO	NAPOLI	EF9876VU	15/03/2010
	VECCHIO	NAPOLI	AB1234CD	27/06/2010
	DARSENA FIUMIC.	ROMA	AC2347LM	17/04/2010
	DARSENA FIUMIC.	ROMA	UV9876VU	03/04/2010
	MESTRE	VENEZIA	RD3456ST	27/12/2008
	EMPEDOCLE	AGRIGENTO	BD2345DB	05/05/2010

Transito	nome	città	targa	data_arrivo	data_partenza
	VECCHIO	NAPOLI	AB1234CD	17/12/2006	29/12/2006
	DARSENA FIUMIC.	ROMA	AC2347LM	15/03/2007	06/04/2008
	MESTRE	VENEZIA	RD3456ST	04/05/2005	26/12/2006
	EMPEDOCLE	AGRIGENTO	BD2345DB	01/09/2008	13/09/2008

Figura 5.1: Una base di dati

utilizzata per limitare l'insieme di valori possibili che un attributo può prendere. Ad esempio poiché una barca non avrà mai lunghezze negative potremmo vincolare l'attributo lunghezza a prendere soltanto valori interi maggiori di zero come nel seguente comando

```
CREATE TABLE Barca (
   targa CHAR(10) PRIMARY KEY,
   nome VARCHAR(25),
   lunghezza NUMERIC(5,2) CHECK lunghezza>0
);
```

5.3 Il comando SELECT-FROM-WHERE

Per poter accedere in lettura alle tabelle della base di dati si utilizza il comando SELECT-FROM-WHERE. La sua sintassi breve è la seguente:

SELECT *<elenco attributi>*
FROM *<lista tabelle>*
WHERE *<condizioni>*

Interrogazioni semplici Con riferimento alla tabella Barca di Fig. 5.1 supponiamo di voler reperire tutte le barche di lunghezza maggiore di 50 metri

e stampare il nome e la targa della barca. Allora possiamo formulare la domanda come segue

```
SELECT     nome, targa
FROM       Barca
WHERE      lunghezza>=50
```

Il risultato di questa interrogazione è la tabella di Fig. 5.2. Quando nella clau-

targa	nome	lunghezza
AC2347LM	Fila	123,45
RD3456ST	Rex	154,98
UV9876VU	Vespucci	132,56

Figura 5.2: Risultato dell'interrogazione

sola `FROM` compare una sola tabella R allora il comando `SELECT-FROM-WHERE` è molto simile ad un operazione di selezione e proiezione dell'algebra relazionale. Infatti in algebra relazionale avremmo potuto esprimere la precedente interrogazione come

$$\pi_{nome,targa}(\sigma_{lunghezza \geq 50}(\text{BARCA}))$$

Il comando `SELECT-FROM-WHERE` e l'operatore di proiezione non sono equivalenti poiché in SQL il comando `SELECT-FROM-WHERE` può ritornare righe duplicate. Ad esempio se eseguiamo il seguente comando

```
SELECT     nome
FROM       Barca
```

Il risultato può contenere anche nomi duplicati se nella tabella BARCA esistono più di due barche con lo stesso nome. Per eliminare dal risultato di una `SELECT` le righe duplicate occorre specificare la clausola `DISTINCT` come di seguito indicato

```
SELECT DISTINCT nome
FROM            Barca
```

Utilizzando i connettivi logici `AND`, `OR` e `NOT` possiamo esprimere nella clausola `WHERE` condizioni logiche complesse. Ad esempio se volessimo conoscere i nomi di tutti i porti di ROMA che hanno un numero di posti maggiore di 250 allora possiamo fare ciò con la seguente interrogazione

```
SELECT     nome
FROM       Porto
WHERE      città = 'ROMA' AND posti_barca>250
```

Quando vogliamo specificare nella clausola `SELECT` tutti gli attributi della tabella o delle tabelle presenti nella clausola `FROM` in SQL possiamo utilizzare il simbolo `*`. Ad esempio

```
SELECT     *
FROM       Porto
WHERE      città = 'ROMA' AND posti_barca >50
```

seleziona tutti gli attributi della tabella PORTO.

Interrogazioni con più tabelle Se vogliamo reperire informazioni da diverse tabelle allora dobbiamo specificarle nella clausola `FROM`. Ad esempio supponiamo di voler conoscere il nome e la data di arrivo delle barche ormeggiate nel porto MESTRE di VENEZIA. Ciò può essere fatto con la seguente interrogazione

```
SELECT    Barca.nome, Ormeggio.nome, Ormeggio.data_arrivo
FROM      Barca, Ormeggio
WHERE     Barca.targa = Ormeggio.targa
          Ormeggio.nome ='MESTRE' AND
          Ormeggio.città = 'VENEZIA'
```

Il cui risultato è

Barca.nome	Ormeggio.nome	Ormeggio.data_arrivo
Rex	MESTRE	27/12/2008

Si osservi innanzitutto che in SQL il nome di un attributo può essere specificato in modo non ambiguo ponendo di fronte ad esso, il nome della tabella a cui appartiene. Nell'interrogazione precedente infatti l'attributo targa è presente sia nella tabella `Ormeggio` che nella tabella `Barca` e ciò crea ambiguità. Se però non vi sono ambiguità tra gli attributi di diverse tabelle oppure se nella clausola `FROM` è presente una sola tabella allora possiamo omettere il nome della tabella davanti all'attributo come abbiamo fatto nelle precedenti interrogazioni.

Il significato del comando `SELECT-FROM-WHERE` in presenza di più di una tabella nella clausola `FROM` è il seguente. Per prima cosa viene generata una tabella che è il prodotto cartesiano delle tabelle presenti nella clausola `FROM`. A questa tabella viene applicata la condizione logica presente nella clausola `WHERE` in modo che solo le righe che soddisfano la condizione della clausola `WHERE` vengono selezionate. La tabella risultante viene quindi proiettata sugli attributi specificati nella clausola `SELECT`. Come già detto sopra bisogna tenere conto che il comando `SELECT-FROM-WHERE` non elimina le righe duplicate. L'eliminazione delle righe duplicate può essere sempre realizzata con l'uso della clausola `DISTINCT`. Se non ci sono righe duplicate allora il significato dell'istruzione

```
SELECT    <elenco attributi>
FROM      R1, R2, ... , Rh
WHERE     <condizioni>
```

è equivalente alla seguente espressione di algebra relazionale

$$\pi_{<elenco\ attributi>}(\sigma_{<condizioni>}(R_1 \times R_2 \times \cdots \times R_h))$$

Abbiamo visto che per risolvere l'ambiguità generata dalla presenza di attributi che hanno lo stesso nome, è necessario specificare per ciascun attributo, il nome della tabella a cui appartiene. Nel caso però che sia presente nella clausola `FROM`, ripetuta più di una volta la stessa tabella, allora il metodo proposto non è più applicabile poiché anche il nome della tabella sarebbe ripetuto. Ad esempio supponiamo di voler elencare i nomi e le città delle coppie di porti che abbiamo una differenza in latitudine minore di 0,1 gradi. Allora possiamo fare ciò con il seguente comando

```
SELECT    P1.nome, P1.città, P2.nome, P2.città
```

```
FROM      Porto AS P1, Porto AS P2
WHERE     P1.lat - P2.lat < 0,1
          AND P1.nome <> P2.nome
```

Il cui risultato è il seguente

P1.nome	P1.città	P2.nome	P2.città
DARSENA FIUMIC.	ROMA	FIUMARA	ROMA
FIUMARA	ROMA	DARSENA FIUMIC.	ROMA

In questo caso la tabella PORTO compare due volte nella clausola `FROM`. Per evitare l'ambiguità si deve assegnare ad ogni tabella nella clausola `FROM` il nome di un alias, preceduto opzionalmente dalla clausola `AS`, da usare in tutti gli altri punti dell'interrogazione. Nel caso specifico abbiamo utilizzato gli alias `P1` e `P2`.

Si osservi che nella clausola `WHERE` abbiamo utilizzato la funzione matematica di differenza. In realtà possiamo utilizzare diverse funzioni matematiche che coinvolgono uno o più attributi numerici ed applicare al risultato della funzione uno degli operatori di confronto binario come "=", "≤", ecc..

Utilizzo di interrogazioni annidate È possibile utilizzare nella clausola `WHERE` altre interrogazioni per specificare condizioni complesse e per rendere, in alcuni casi, più leggibile l'interrogazione. In altri termini, il valore di un attributo è confrontato con i risultato di una istruzione `SELECT-FROM-WHERE` *annidata* nella clausola `WHERE`. Fino ad ora abbiamo visto che, nella clausola `WHERE`, è possibile confrontare il valore di un attributo con il valore di un'altro attributo o di una costante. Viceversa, in genere, una `SELECT-FROM-WHERE` restituisce una *collezione* di valori o di righe. Quindi per utilizzare le interrogazioni annidate, SQL fornisce due operatori `IN` ed θ-`ALL` dove θ è uno degli operatori di confronto come $\{=, <, \leq, >\geq, \neq\}$. L'operatore `IN` confronta il valore della parte sinistra con la collezione di valori della sotto-interrogazione nella parte destra. Se esiste almeno un valore nella parte destra uguale al valore nella parte sinistra allora la riga corrispondente viene selezionata. Ad esempio si supponga di voler trovare il nome e la targa delle barche ormeggiate in uno qualunque dei porti della città di VENEZIA. Per fare ciò, possiamo utilizzare l'operatore `in` come segue

```
SELECT     nome, targa
FROM       Barca
WHERE      targa IN    (SELECT targa
                        FROM Ormeggio
                        WHERE città ='VENEZIA')
```

L'operatore θ-`ALL` invece confronta il valore della parte sinistra utilizzando l'operatore θ con tutti i valori della sotto-interrogazione della parte destra. Se il confronto è soddisfatto per tutti i valori della parte destra allora la riga corrispondente viene selezionata. Ad esempio supponiamo di voler ottenere tutte le barche che hanno lunghezza più grande di tutte le altre. Possiamo specificare quest'interrogazione come segue

```
SELECT     *
FROM       Barca
WHERE      lunghezza >= ALL (SELECT lunghezza
                             FROM Barca)
```

il cui risultato è

targa	nome	lunghezza
RD3456ST	Rex	154,98

Operatori aggregati Il linguaggio SQL è dotato di un insieme di *operatori aggregati* come `COUNT`, `SUM`, `MIN`, `MAX` ed `AVG`. Questi operatori, data una collezione di righe ed, opzionalmente per l'operatore `COUNT`, un campo di tipo numerico, calcolano il valore dell'operatore su tutti i valori numerici del campo per le righe selezionate.

Cominciamo con l'illustrare l'operatore `COUNT`. Supponiamo di voler sapere quante barche sono ormeggiate nella DARSENA di FIUMICINO a ROMA. Per fare ciò utilizziamo l'operatore `COUNT` come nella seguente interrogazione

```
SELECT     COUNT(*)
FROM       Ormeggio
WHERE      Ormeggio.città = 'ROMA' AND
           Ormeggio.nome ='DARSENA FIUMIC.'
```

Nella precedente interrogazione vengono contate tutte le righe selezionate dalle condizioni presenti nelle clausole `FROM` e `WHERE` ovvero tutte le barche ormeggiate nella DARSENA di FIUMICINO. Il risultato è

COUNT(*)
2

Per gli altri operatori aggregati il calcolo viene effettuato nel seguente modo. Si ottiene, inizialmente, la collezione di righe selezionata dalla clausola `WHERE`. Si applica poi l'operatore aggregato alla collezione di valori numerici dell'attributo selezionato. Ad esempio la seguente interrogazione fornisce la somma dei posti barca nei porti di ROMA e NAPOLI ed utilizza l'operatore aggregato `SUM`.

```
SELECT     SUM(posti_barca)
FROM       Porto
WHERE      città = 'ROMA' OR città = 'NAPOLI'
```

Il risultato è

SUM(posti_barca)
1.011

In quest'interrogazione viene selezionato il valore dell'attributo `posti_barca` per tutte le righe della tabella PORTO delle città di NAPOLI e ROMA (in questo caso senza eliminare i duplicati). Infine i valori così ottenuti vengono sommati tra loro fornendo il risultato.

La clausola `GROUP BY`. Al comando `SELECT` è possibile applicare una clausola `GROUP BY` in modo da facilitare l'applicazione degli operatori aggregati. Questa clausola, infatti, consente di applicare uno o più operatori aggregati a gruppi di righe che hanno gli stessi valori su una o più colonne. Supponiamo infatti di voler conoscere il numero delle barche ormeggiate e la media della lunghezze di queste barche in tutti i porti delle città con latitudine superiore a 40 gradi. Questa interrogazione può essere espressa come segue:

	nome	città	lat	targa	lunghezza
1° gruppo	VECCHIO	NAPOLI	40,81	EF9876VU	12,56
	VECCHIO	NAPOLI	40,81	AB1234CD	42,34
2° gruppo	DARSENA FIUMICINO	ROMA	41,72	AC2347LM	123,45
	DARSENA FIUMICINO	ROMA	41,72	UV9876VU	132,56
3° gruppo	MESTRE	VENEZIA	45,6	RD3456ST	154,98

(a)

nome	città	COUNT(*)	AVG(B.lunghezza)
VECCHIO	NAPOLI	2	27,45
DARSENA FIUMICINO	ROMA	2	128,00
MESTRE	VENEZIA	1	154,98

(b)

Figura 5.3: Esempio di esecuzione della clausola `GROUP BY`. (a) Il raggruppamento delle righe (b) il risultato finale

```
SELECT     P.nome, P.città, COUNT(*), AVG(B.lunghezza)
FROM       Porto P, Ormeggio O, Barca B
WHERE      P.lat > 40 AND
           P.nome = O.nome AND P.città = O.città AND
           O.targa = B.targa
GROUP BY   P.nome, P.città
```

L'operatore `GROUP BY` <*elenco attributi*> raccoglie tutte le righe selezionate in gruppi che hanno gli stessi valori per l'elenco di attributi specificato. Nell'interrogazione precedente l'elenco di attributi è il nome e la città della tabella PORTO (vedi Fig. 5.3).

A ciascun gruppo così ottenuto applica le funzioni aggregate `COUNT(*)` e `AVG(B.lunghezza)`. Il risultato ottenuto, indicato in Fig. 5.3(b) è una tabella che ha una riga per quante sono le occorrenze univoche di nome e città e ciascuna riga riporta il conteggio delle barche ormeggiate in quel porto e la media della lunghezza di queste barche.

La sintassi standard del comando richiede che gli attributi che seguono la clausola `SELECT` e che non sono funzioni di aggregazione, devono essere gli attributi specificati nella `GROUP BY`.

Anche con l'operatore `GROUP BY` il risultato del comando `SELECT` è sempre una tabella. Pertanto supponiamo di voler applicare dei filtri al risultato della `GROUP BY` che coinvolgano i valori degli operatori aggregati. Non possiamo specificare tali condizioni nella clausola `WHERE` poiché queste vengono applicate alle righe e non al valore dell'operatore aggregato. Per fare ciò, SQL mette a disposizione la clausola `HAVING` come indicato nella seguente interrogazione. Si supponga di voler conoscere somma dei posti barca di tutti i porti di una città ma solo di quelle città che hanno un numero complessivo di posti barca maggiore a 150 posti. Questo può essere realizzato con la seguente interrogazione

```
SELECT     città, SUM(posti_barca)
FROM       Porto
GROUP BY   città
HAVING     SUM(posti_barca) >=150
```

Ordinamento dei risultati Attraverso comando il `SELECT-FROM-WHERE` è possibile ordinare i risultati ottenuti attraverso la clausola

`ORDER BY` <*elenco attributi*> [*asc*|*desc*]

L'ordinamento numerico viene utilizzato per i dati numerici come `FLOAT`, `INTEGER`, ecc.. Per le stringhe viene utilizzato l'ordinamento lessicografico. L'opzione [*asc*|*desc*] consente di specificare se l'ordine deve essere ascendente o discendente. Supponiamo ad esempio di voler conoscere il nome e la lunghezza di tutte le barche ormeggiate presso il porto di FIUMARA a ROMA e di voler elencarle in ordine discendente di lunghezza. Allora ciò viene realizzato dalla seguente interrogazione

```
SELECT     B.nome, B.lunghezza
FROM       Barca B, Ormeggio B
WHERE      O.città= 'ROMA' AND O.nome ='FIUMARA' AND
           O.targa = B.targa
ORDER BY   lunghezza DESC
```

5.4 Gli operatori insiemistici

L'algebra relazionale è *completa* in quanto che questa ha la stessa potenza espressiva del *calcolo relazionale sulle tuple*. Abbiamo visto che il comando `SELECT-FROM-WHERE` (con la clausola `DISTINCT`) è equivalente all'operazione *selezione-proiezione-join* dell'algebra relazionale. Per rendere completo l'SQL rispetto all'algebra relazionale e quindi rispetto al calcolo relazionale delle tuple, in esso sono stati incorporati gli operatori insiemistici di unione, intersezione e differenza. È possibile tramite gli operatori `UNION`, `INTERSECT` e `MINUS` (oppure `EXCEPT`) realizzare l'unione, l'intersezione e la differenza, rispettivamente, dei risultati di due istruzioni `SELECT-FROM-WHERE`. L'unica condizione è che gli schemi risultanti dai due comandi siano compatibili tra di loro; ovvero abbiano lo stesso numero di attributi ed i domini degli attributi siano compatibili tra di loro. Ad esempio supponiamo di voler conoscere il nome e la targa di tutte le barche ormeggiate nel porto di EMPEDOCLE ad AGRIGENTO e di tutte quelle che hanno effettuato una sosta il mese precedente nel porto VECCHIO di NAPOLI. Allora utilizzeremo la seguente interrogazione

```
SELECT     B.nome, B.targa
FROM       Barca B, Ormeggio B
WHERE      O.città= 'AGRIGENTO' AND O.nome ='EMPEDOCLE'
           AND O.targa = B.targa
UNION
SELECT     B.nome, B.targa
FROM       Barca B, Transito T
WHERE      T.città= 'NAPOLI' AND T.nome ='VECCHIO' AND
           T.targa = T.targa
```

L'operatore `UNION` (cosi come gli altri operatori insiemistici) fornisce un risultato nel quale, a differenza del comando `SELECT-FROM-WHERE`, non sono presenti

mai righe duplicate. Se si vogliono comunque mantenere le righe duplicate occorre utilizzare dopo la clausola `UNION` (rispettivamente `INTERSECT, MINUS`) la clausola `ALL`.
Il comando `MINUS` è fondamentale in alcune situazioni. Ad esempio supponiamo di voler trovare tutte quelle barche che sono più lunghe di tutte le altre. Se B è l'insieme di tutte le barche ed A l'insieme di quelle più lunghe di tutte le altre allora l'insieme A contiene una barca b se $\forall c \in B$, $b(lunghezza) \geq c(lunghezza)$. La proposizione logica $\forall c\ P(c)$ dove $P(c)$ è un predicato come quello appena visto può essere riscritta come $\neg\exists c(\neg P(c))$. Quindi possiamo dire che A contiene una barca b se $\neg\exists c \in B$ tale che $b(lunghezza) < c(lunghezza)$. Allora l'insieme A può essere ottenuto con la seguente interrogazione nel quale utilizziamo la clausola `MINUS`

```
SELECT     nome, targa, lunghezza
FROM       Barca
MINUS
SELECT     B1.nome, B1.targa, B1.lunghezza
FROM       Barca B1, Barca B2
WHERE      B1.lunghezza < B2.lunghezza
```

La seconda `SELECT` della precedente interrogazione, seleziona tutte quelle barche per le quali esiste almeno un'altra barca di lunghezza maggiore. L'insieme voluto è quindi ottenuto eliminando dall'insieme di tutte le barche, quelle barche per le quali né esiste almeno un'altra di lunghezza maggiore.

5.5 I comandi per inserire, modificare e cancellare righe da una tabella

Il linguaggio SQL fornisce i comandi `INSERT`, `DELETE` ed `UPDATE` per inserire cancellare e modificare righe o insiemi di righe di una tabella.

Il comando `INSERT`. La sintassi del comando `INSERT` è la seguente

`INSERT INTO` <*nome tabella*> `VALUES (` <*elenco di valori*> `)`

Se vogliamo ad esempio inserire i dati di una nuova barca nella tabella Barca allora possiamo realizzare ciò con il seguente comando

```
INSERT INTO Barca VALUES
            ('BR09CGRM', 'azzurra', 34.5)
```

È possibile inserire anche diverse righe in una volta. Se volessimo inserire nella tabella Porto i porti di RIVA DI TRAIANO e di PORTO VECCHIO a GENOVA allora possiamo utilizzare il comando

```
INSERT INTO Porto VALUES
       ('RIVA DI TRAIANO', 'CIVITAVECCHIA', 231, 42.34, 12.23),
       ('PORTO VECCHIO', 'GENOVA', 342, 42.34, 12.23)
```

Nei due precedenti casi dopo la clausola `VALUES` specifichiamo una o più sequenze di valori ed espressioni. Il numero e l'ordine di questi valori deve corrispondere

al numero ed all'ordine di dichiarazione degli attributi della tabella così come effettuato nel comando `CREATE TABLE`. Se invece non vogliamo specificare tutti i valori e/o vogliamo indicare un diverso ordinamento dei valori inseriti possiamo specificare l'insieme di attributi oggetto di inserimento con la seguente sintassi

`INSERT INTO` <*nome tabella*> (<*elenco di attributi*>)
`VALUES` (<*elenco di valori*>)

Se per esempio, vogliamo inserire nella tabella PORTO solo il nome e la città di due nuove righe allora possiamo eseguire il seguente comando

```
INSERT INTO Porto (nome, città) VALUES
           ('EMPEDOCLE', 'AGRIGENTO'),
           ('PORTOFERRAIO', 'PORTOFERRAIO')
```

I valori dei campi non specificati verranno impostati a `NULL` oppure al valore di default. È possibile utilizzare, all'interno di una `INSERT` una sotto-interrogazione per inserire i dati restituiti da un comando `SELECT-FROM-WHERE`. Si supponga di avere una tabella ORMEGGIO_ROMA(targa, nome, lunghezza) in cui si voglia inserire il nome, la targa e la lunghezza delle barche ormeggiate presso il porto di OSTIA a ROMA. Allora possiamo realizzare ciò con il comando

```
INSERT INTO Ormeggio_roma
SELECT     B.targa, B.nome, B.lunghezza
FROM       Barca B, Ormeggio O
WHERE      B.targa =O.targa AND
           O.città = 'ROMA' OR O.porto = 'FIUMARA'
```

Il comando `DELETE`. Con questo comando possiamo cancellare le righe di una tabella. La sua sintassi è

`DELETE FROM` <*nome tabella*> [`WHERE` <*condizioni*>]

Ad esempio supponiamo di voler cancellare dalla tabella ORMEGGIO la riga che ha una targa = 'AB123CD'. Allora il comando è il seguente

```
DELETE FROM Ormeggio WHERE targa = 'AB123CD'
```

Se invece volessimo cancellare tutte le righe dalla tabella ORMEGGIO possiamo utilizzare

```
DELETE FROM Ormeggio
```

Il comando `UPDATE`. Per poter modificare i valori di una o più colonne di un insieme di righe di una tabella SQL mette a disposizione il comando `UPDATE`. La sua sintassi è

`UPDATE` <*nome tabella*> `SET` <*nome_colonna = espressione*>

Ad esempio supponiamo di voler modificare il valore della colonna data_arrivo della relazione ORMEGGIO per le barche ormeggiate nel porto di NETTUNO a NETTUNO e di volerla impostare al '03/06/2010'. Allora otterremo ciò con il comando

```
UPDATE Ormeggio SET data_arrivo = '2010/06/03'
WHERE nome ='NETTUNO' AND città = 'NETTUNO'
```

Possiamo eventualmente effettuare la modifica di più colonne contemporaneamente come nel seguente esempio. Se vogliamo modificare il nome e la lunghezza della barca con targa 'AB1234DC' allora utilizzeremo il seguente comando

```
UPDATE Barca SET nome = 'NINA', lunghezza =31.5
WHERE targa ='AB1234DC'
```

5.6 Viste

Abbiamo visto che in SQL il risultato di un comando `SELECT-FROM-WHERE` è sempre una tabella. In effetti possiamo scrivere un intero comando `SELECT-FROM -WHERE` all'interno della una clausola `FROM` di un altro comando. Questo metodo di riutilizzare il risultato di un `SELECT-FROM-WHERE` è però poco pratico e leggibile. SQL mette a disposizione il comando `CREATE VIEW` per associare un'interrogazione ad un nome. L'oggetto così creato viene chiamato *vista* (view). In questo modo è possibile utilizzare il nome della vista come se fosse una tabella fisica del database. In realtà ogni volta che viene utilizzata una vista viene eseguita l'interrogazione associata e il risultato viene memorizzato in una tabella temporanea (in modo del tutto trasparente per l'utente).

Nell'esempio che segue creiamo un vista di tutte le barche ormeggiate nei porti di ROMA, riportando la targa ed il nome della barca.

```
CREATE VIEW barche_a_roma (targa_barca, nome_barca) AS
SELECT     B.targa, B.nome
FROM       Barca B, Ormeggio O
WHERE      O.città= 'ROMA' AND
           O.targa = B.targa
```

Possiamo ora utilizzare la vista BARCHE_A_ROMA come se fosse una tabella della base di dati con schema {targa_barca, nome_barca}.

5.7 Le operazioni dell'applicazione Cinema

Vediamo quali comandi SQL sono necessari per creare la base di dati di Fig. 5.1 e per realizzare le operazioni di Tab. 4.1 relative all'applicazione Cinema.

Comandi per la creazione delle tabelle Per prima cosa creiamo lo schema della base di dati con i seguenti comandi

```
CREATE TABLE film (
     titolo VARCHAR(50),
     anno DATE,
     durata SMALLINT NOT NULL,
     regista VARCHAR(50),
     PRIMARY KEY (titolo, anno)
);

CREATE TABLE attore (
     codice INTEGER PRIMARY KEY,
     nome VARCHAR(50) NOT NULL,
     data_nasc DATE NOT NULL
);
```

```
CREATE TABLE sala (
     cod_sala INTEGER PRIMARY KEY,
     numero SMALLINT NOT NULL,
     multisala VARCHAR(50) NOT NULL,
     posti INTEGER NOT NULL
);

CREATE TABLE poltrona (
     cod_poltr INTEGER PRIMARY KEY,
     fila char(1) NOT NULL,
     numero SMALLINT NOT NULL,
     sala INTEGER NOT NULL REFERENCES
          sala(cod_sala)
);

CREATE TABLE programma (
    cod_prog INTEGER PRIMARY KEY,
    data_ora TIMESTAMP NOT NULL,
    durata SMALLINT NOT NULL,
    posti_occ INTEGER  NOT NULL,
    posti_disp INTEGER NOT NULL,
    film VARCHAR(50) NOT NULL,
    anno DATE NOT NULL,
    sala INTEGER NOT NULL,
    FOREIGN KEY (film, anno) REFERENCES film (titolo, anno),
    FOREIGN KEY (sala) REFERENCES sala (cod_sala)
);

CREATE TABLE biglietto (
     numero INTEGER PRIMARY KEY,
     data_emis TIMESTAMP NOT NULL,
     costo numeric(4,2) NOT NULL,
     programma INTEGER NOT NULL
          REFERENCES programma (cod_prog),
     poltrona INTEGER NOT NULL
          REFERENCES  poltrona (cod_poltr)
);

CREATE TABLE bigliet_stor (
     numero INTEGER PRIMARY KEY,
     data_emis TIMESTAMP NOT NULL,
     costo numeric(4,2) NOT NULL,
     programma INTEGER NOT NULL
          REFERENCES programma (cod_prog),
     poltrona INTEGER NOT NULL
          REFERENCES  poltrona (cod_poltr)
);

CREATE TABLE interpreta (
     film VARCHAR (50) NOT NULL,
     anno DATE NOT NULL,
     attore INTEGER NOT NULL
          REFERENCES attore (codice),
     ruolo VARCHAR(50),
     FOREIGN KEY (film, anno) REFERENCES film (titolo, anno)
);
```

A questo punto inseriamo nella base di dati un piccolo insieme di dati di prova, su cui testare le operazioni come di seguito indicato.

```
INSERT INTO film VALUES
('bellissima','1951/10/10',113 ,'Luchino  Visconti'),
```

```
('ladri di biciclette','1948/10/10',93,'Vittorio De Sica'),
('umberto d','1952/10/10',89  ,'Vittorio De Sica');

INSERT INTO attore VALUES
(1,'Anna Magnani','1940/10/01'),
(2,'Walter Chiari','1940/10/02'),
(3,'Carlo Battisti','1912/02/01'),
(4,'Lamberto Maggiorani','1925/12/01');

INSERT INTO interpreta VALUES
('bellissima','1951/10/10',1,'Anna'),
('bellissima','1951/10/10',2,'Mario'),
('ladri di biciclette','1948/10/10',3,'Luchino'),
('umberto d','1952/10/10',4,'Umberto');

INSERT INTO sala VALUES
(1,1,'cineplex',250),
(2,2,'cineplex',240);

INSERT INTO poltrona VALUES
(1,'A',1,1), (2,'B',1,1),  (3,'C',1,1),  (4,'D',1,1),
(5,'A',2,1), (6,'B',2,1),  (7,'C',2,1),  (8,'D',2,1),
(9,'A',3,1), (10,'B',3,1), (11,'C',3,1), (12,'D',3,1),
(13,'A',1,2), (14,'B',1,2),(15,'C',1,2), (16,'D',1,2),
(17,'A',2,2), (18,'B',2,2),(19,'C',2,2), (20,'D',2,2),
(21,'A',3,2), (22,'B',3,2),(23,'C',3,2), (24,'D',3,2);

INSERT INTO programma VALUES
(1,'2009/12/22 18:30:00',140 , 150,100,'bellissima','1951/10/10',1),
(2,'2009/12/22 20:30:00',135 ,200,40,'bellissima','1951/10/10',2),
(3,'2009/12/23 18:30:00',110 ,25,225,'ladri di biciclette','1948/10/10',1),
(4,'2009/12/23 00:00:00',110 ,50,190,'umberto d','1952/10/10',2);

INSERT INTO biglietto VALUES
(13,'2009/12/23',7.5,3,1), (15,'2009/12/23',7.5,3,3),
(16,'2009/12/23',7.5,3,5), (17,'2009/12/23',7.5,3,7),
(19,'2009/12/23',7.5,3,9), (20,'2009/12/23',7.5,4,13),
(21,'2009/12/23',7.5,4,14), (22,'2009/12/23',7.5,4,17),
(23,'2009/12/23',7.5,4,19), (24,'2009/12/23',7.5,4,21);

INSERT INTO bigliet_stor VALUES
(1,'2009/12/22',7.5,1,1), (2,'2009/12/22',7.5,1,2),
(3,'2009/12/22',7.5,1,3), (4,'2009/12/22',7.5,1,5),
(5,'2009/12/22',7.5,1,6), (6,'2009/12/22',7.5,1,6),
(7,'2009/12/22',7.5,1,9), (8,'2009/12/22',7.5,2,13),
(9,'2009/12/22',7.5,2,14), (10,'2009/12/22',7.5,2,17),
(11,'2009/12/22',7.5,2,19), (12,'2009/12/22',7.5,2,21);
```

Esecuzione delle operazioni Vediamo ora come è possibile realizzare in SQL le Operazioni di Tabella 4.1. Mostreremo le Operazioni 3 e 6. Lo studente è invitato a realizzare come esercizio le restanti operazioni.

Per l'Operazione 3, creiamo una vista `biglietti` che comprende tutti i biglietti venduti, sia quelli della giornata odierna che i biglietti storici. Si crea una vista `incasso_film` dove per ogni film si calcola l'incasso complessivo, ottenuto dalla somma dei costi di tutti i biglietti venduti per ogni film proiettato. Infine si effettua una join con le tabelle `film, interpreta` e `attore` per reperire le informazioni richieste sugli attori e sul regista del film.

```
--================*
-- Operazione 3   *
```

```
--================*

CREATE VIEW biglietti as
SELECT   *
FROM     biglietto
UNION
SELECT   *
FROM     bigliet_stor;

CREATE   VIEW  incasso_film (titolo, anno, incasso) as
SELECT   f.titolo, f.anno , sum(b.costo)
FROM     film f, programma pr, biglietti b
WHERE    b.programma = pr.cod_prog AND
         pr.film = f. titolo AND
         pr.anno = f.anno
GROUP BY f.titolo, f.anno ;

SELECT   f.titolo, f.anno, f.regista, c.incasso, a.nome
FROM     attore a, interpreta i, film f, incasso_film c
WHERE    a.codice = i.attore AND
         i.film = f.titolo AND i.anno = f.anno AND
         c.titolo = f.titolo AND c.anno = f.anno
ORDER BY incasso desc;
```

L'output dell'esecuzione dei precedenti comandi in PostgreSql è il seguente

```
       titolo        |    anno    |      regista      | incasso |        nome
---------------------+------------+-------------------+---------+---------------------
 bellissima          | 1951-10-10 | Luchino  Visconti |   90.00 | Anna Magnani
 bellissima          | 1951-10-10 | Luchino  Visconti |   90.00 | Walter Chiari
 ladri di biciclette | 1948-10-10 | Vittorio De Sica  |   37.50 | Carlo Battisti
 umberto d           | 1952-10-10 | Vittorio De Sica  |   37.50 | Lamberto Maggiorani
(4 rows)
```

Per l'Operazione 6 creiamo una vista `p_lib_pro` che contiene le poltrone libere di ciascuna proiezione[1]. Questa viene realizzata cancellando dall'insieme di tutte le poltrone di una proiezione, quelle per cui è stato acquistato un biglietto. Poi viene realizzata una vista, `tre_poltrone` che contiene le terne di poltrone che per ogni proiezione sono libere e consecutive in una fila. Viene infine dato un esempio di utilizzo di questa vista.

```
--================*
-- Operazione 6   *
--================*

CREATE  VIEW p_lib_pro (programma, poltrona, fila, numero) as
SELECT  cod_prog, cod_poltr, fila, numero
FROM    programma pr, poltrona p
WHERE   pr.sala = p.sala
EXCEPT
SELECT  cod_prog, cod_poltr, fila, p.numero
FROM    programma pr, biglietto b, poltrona p
WHERE   b.programma = pr.cod_prog AND
        b.poltrona = p.cod_poltr;

CREATE VIEW tre_posti (programma, poltrona1, poltrona2, poltrona3) as
SELECT  p1.programma, p1.poltrona, p2.poltrona, p3.poltrona
FROM    p_lib_pro p1, p_lib_pro p2, p_lib_pro p3
WHERE   p1.fila = p2.fila AND p2.fila = p3.fila AND
```

[1] Lo si è omesso per semplicità ma sarebbe opportuno, per motivi prestazionali, limitare la vista alle sole proiezioni della giornata odierna

```
        p1.programma = p2.programma AND p2.programma = p3.programma AND
        p1.numero = p2.numero-1 AND p2.numero = p3.numero -1;

SELECT  poltrona1, poltrona2, poltrona3
FROM    tre_posti
WHERE   programma = 3;
```

e l'output è il seguente

```
 poltrona1 | poltrona2 | poltrona3
-----------+-----------+-----------
         2 |         6 |        10
         4 |         8 |        12
(2 rows)
```

5.8 Esercizi

Esercizio 5.1 In relazione alla base di dati relazionale di Fig. 5.4 creare e popolare le tabelle. Scegliere gli appropriati domini per gli attributi di ciascuna tabella. Creare gli appropriati vincoli di chiave e di integrità referenziale.

Esercizio 5.2 *Interrogazioni semplici.* Creare ed eseguire i comandi SQL relativi alle interrogazioni di seguito indicate, sulla base di dati di Fig. 5.4.

- Trovare il nome delle barche fabbricate nel 2003
- Trovare il nome del porto e la targa delle barche sostate nel 2004
- Trovare le barche più lunghe di 10 metri costruite prima del 1900
- Elencare tutti i posti barca del porto di Roma
- Elencare la banchina, il numero di posto e la targa delle barche in sosta
- Stampare la lunghezza in "piedi" delle barche (un metro equivale a 3,2808 piedi)
- Inserisci la data di partenza della barca con targa 'genoa234'
- Elimina da 'postobarca' tutti i posti barca del porto di 'venezia'

Esercizio 5.3 *Interrogazioni con più tabelle.* Creare ed eseguire i comandi SQL relativi alle interrogazioni di seguito indicate, sulla base di dati di Fig. 5.4.

- Trovare il costo giornaliero delle barche che hanno sostato presso un determinato PB
- Trovare le barche che hanno ormeggiato in posti barca adiacenti (per adiacenti si intende due PB con numeri consecutivi)
- Trovare il nome del porto e la regione dei posti barca non occupati in una certa data
- Stampare i nomi delle barche in ordine della data di arrivo in un porto.
- Trovare le terne di barche che hanno transitato in uno stesso porto e stampare solo e soltanto il nome della barca ed il porto di transito.

BARCA	nome	targa	lunghezza	anno_varo
	Nettuno	Roma2134	12	1998
	Vento	Roma3578	13,5	1954
	Luna Rossa	Genoa234	10	1490
	Azzurra	Genoa725	18,7	1600
	Leggera	Bari89212	15	1995
	Delfino	Napoli98	22	1997
	Vespucci	Venezia12	87	1894
	Ninetta	Genoa156	54	1978
	Garibaldi	Napoli32	123	2003

Sosta	barca	porto	posto	banchina	data_arrivo	data_part
	Roma2134	Roma	2	AS	01/12/2004	03/12/2004
	Roma3578	Napoli	4	C	05/06/2006	null
	Genoa234	Roma	6	F	08/01/2004	17/03/2004
	Genoa725	Genova	3	C	05/11/2005	02/02/2006
	Bari89212	Genova	5	AS	04/08/2005	12/11/2005
	Genoa234	Venezia	3	R	03/07/2006	null
	Genoa725	Venezia	2	S	15/12/2007	17/12/2007
	Bari89212	Roma	7	AS	25/03/2006	04/04/2006

POSTOBARCA	porto	numero	banchina	mensile	giornaliero
	Roma	2	AS	500	20
	Roma	7	S	572	40
	Roma	6	F	620	43
	Genova	3	C	150	47
	Genova	5	AS	300	81
	Venezia	3	R	400	98
	Venezia	2	S	450	50
	Napoli	1	C	321	15
	Napoli	4	C	745	95

PORTO	nome	regione	posti
	Cagliari	Sardegna	521
	Civitavecchia	Lazio	345
	Palermo	Sicilia	720
	Roma	Lazio	182
	Napoli	Campania	905
	Venezia	Veneto	650
	Genova	Liguria	832

Figura 5.4: La base di dati dell'esercizio 5.1

Esercizio 5.4 *Interrogazioni con operatori aggregati.* Creare ed eseguire i comandi SQL relativi alle interrogazioni di seguito indicate, sulla base di dati di Fig. 5.4.

- Calcolare il numero di giorni di sosta per le barche sostate nei porti di Genova - Venezia - Roma
- Stampare il numero complessivo delle barche in sosta o transitate nei porti

di Genova e Napoli

- Calcolare la media della lunghezza di tutte le barche
- Trovare la somma dei costi mensili di affitto dei posti barca dei porti di Roma e Venezia
- Trovare la media dei giorni di permanenza in un porto e per ogni porto stampare il risultato

Esercizio 5.5 Creare le tabelle ed eseguire le operazioni indicate nell'Esercizio 4.3

Esercizio 5.6 Creare le tabelle ed eseguire le operazioni indicate nell'Esercizio 4.4

c_1	c_2	d
napoli	roma	119
napoli	milano	219
napoli	palermo	99
roma	milano	180
roma	palermo	43
milano	palermo	96
roma	napoli	119
milano	napoli	219
palermo	napoli	99
milano	roma	180
palermo	roma	43
palermo	milano	96

Figura 5.5: La tabella delle distanze tra le città dell'esercizio 5.7

Esercizio 5.7 Si risolva in SQL il noto problema del *Commesso Viaggiatore* (in inglese Travelling Salesman Problem) di seguito definito. Dato un insieme di città $C = \{c_1, c_2, \ldots, c_n\}$ e un numero k trovare un ordinamento delle città $(c_{i_1}, c_{i_2}, \ldots, c_{i_n})$ tale che

$$\sum_{j=1,\ldots,n-1} d(c_{i_j}, c_{i_j+1}) < k$$

dove $d(c_i, c_j)$ è la distanza tra le città c_i e c_j. Si utilizzi per le distanze e le città la tabella DIS di Fig 5.5 (le distanze sono state generate casualmente) dove $n = 4$. (Sugg.: si utilizzi il prodotto cartesiano della tabella DIS per se stessa $n - 1$ volte)

Capitolo 6

Organizzazione fisica dei dati

Lo stato attuale della tecnologia mette a disposizione diversi strumenti ed apparati per la memorizzazione fisica dei dati nei sistemi informativi. In funzione degli apparati utilizzati è necessario implementare procedure e strutture dati che ottimizzino l'utilizzo di queste apparecchiature al fine di rendere quanto più rapide ed efficienti le operazioni richieste. Descriveremo, in questo capitolo, sia i sistemi tecnologici utilizzati per la memorizzazione permanente delle informazioni, che le strutture dati e gli algoritmi utilizzati per l'accesso e la gestione efficiente della base di dati.

6.1 Gerarchie di memoria

L'architettura di un computer prevede che la CPU (Central Processing Unit, l'unita centrale di calcolo) utilizzi, per leggere programmi ed elaborare dati, esclusivamente la *memoria principale*, chiamata anche *memoria RAM* (Random Access Memory o memoria ad accesso casuale).

L'accesso a questo tipo di memoria è molto rapido ma il contenuto di questa svanisce quando viene tolta l'alimentazione elettrica. Vista la difficoltà ed i costi connessi ad assicurare che un sistema rimanga sempre alimentato elettricamente (un guasto all'alimentazione elettrica è sempre possibile e non così infrequente) sono state prodotte delle apparecchiature che hanno la capacità di memorizzare i dati in modo permanente, indipendentemente dalla loro alimentazione elettrica. Tali apparati consentono, sia di trasferire i dati in essi memorizzati verso la memoria RAM, sia ricevere i dati presenti dalla memoria RAM per essere poi memorizzati. Poiché i dati presenti in questi dispositivi non sono disponibili immediatamente per essere elaborati dalla CPU allora ci riferiremo a questo complesso di apparecchiature come la *memoria secondaria* o anche *memoria di massa* del sistema.
Le apparecchiature più diffuse in commercio per la memorizzazione permanente di dati e programmi sono i *dischi magnetici* (comunemente detti Hard Disk), i *dischi ottici* come i CD-ROM, i DVD e recentemente i BluRay Disk e i *nastri magnetici*. La tecnologia dei dischi magnetici è quella più frequentemente utilizzata per memorizzare le basi di dati ed i programmi.

I dischi magnetici Un disco magnetico ha una superficie di materiale magnetizzabile. La magnetizzazione che viene indotta su di una porzione della superficie, da un dispositivo, detto testina di scrittura, è in grado di rappresentare il valore (zero od uno) di un singolo bit. Una testina, detta di lettura, che passa vicino al materiale magnetizzato è in grado di leggere successivamente le sequenze di bit memorizzate. In pratica sulla superficie di un disco del diametro

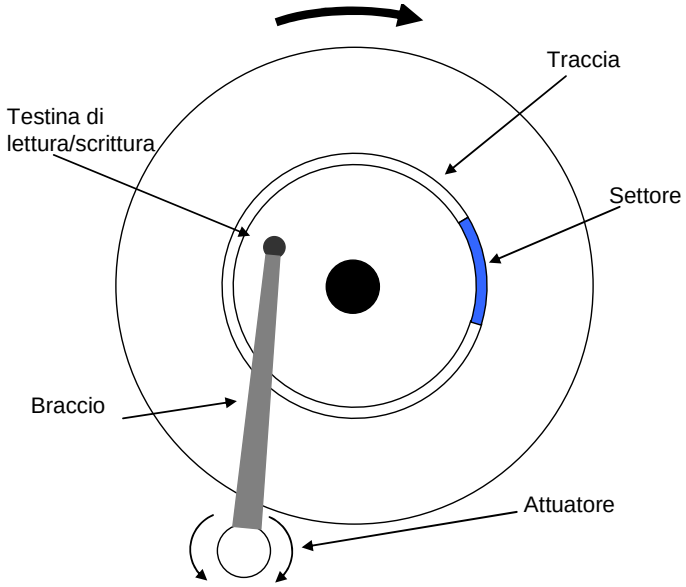


Figura 6.1: Struttura di un disco magnetico.

di 5 cm circa, viene steso il materiale magnetico protetto da un sottile film di acrilico. Tale disco viene fatto girare a velocità costante. Tipiche velocità variano da 8.000 giri/minuto (g/m) fino a 15.000 g/m. Al di sopra del disco viene fatta scorrere la testina di lettura e scrittura fissata rigidamente ad un braccio meccanico attuato da un motore elettrico che consente alla testina di muoversi dal centro fino al bordo del disco e viceversa. Spesso nella pratica, entrambe le facce del disco vengono utilizzate per la registrazione dei dati e molti dischi (da 4 a 8) vengono fissati in un unico assemblaggio. Il braccio meccanico comanda quindi, tante testine di lettura/scrittura quante sono le facce dei dischi presenti. La testina incorpora l'elettronica sia per la lettura e la scrittura ed è comandata dall'unità di controllo del disco, la quale è connessa, per mezzo di una opportuna interfaccia, con il server o l'unità centrale, il quale vede il disco come una unità logica di input ed output.

La quantità di bit che può essere memorizzata sulla superficie del disco per centimetro quadro, viene detta *densità del disco*. I dischi attuali hanno densità che arrivano anche a 1800Gigabit/cm^2.

La quantità di bit memorizzata nel disco viene misurata in *byte*, che corrisponde a 8 bit, in *kilobyte* (KB) che corrisponde a 2^{10} byte, in *Megabyte* (MB), *Gigabyte* (GB) e *Terabyte* (TB) che corrispondono rispettivamente a 2^{20}, 2^{30} e 2^{40} byte.

Il disco è suddiviso in tanti cerchi concentrici e ciascuno di questi è chiamato *traccia* (vedi Fig. 6.1). A sua volta ogni traccia è suddivisa in settori.

Ad esempio un settore corrisponde ad un preciso angolo sul disco. L'inizio di ciascun settore viene marcato fisicamente sul disco. I settori a loro volta vengono suddivisi in blocchi dal sistema operativo, al momento dell'inizializzazione (formattazione) del disco. I blocchi possono avere dimensioni che vanno da 512 byte a 8196 byte.

Pertanto per accedere ad un blocco sul disco occorre specificare la traccia, il settore ed il numero del blocco all'interno del settore. I blocchi sul disco possono essere numerati complessivamente da 1 a n. Tale numero che viene chiamato *numero logico del blocco* (LBN) ed è cura del controller del disco tradurre il LBN di un blocco nell'indirizzo fisico corrispondente.

Per poter accedere ad un blocco sul disco, occorre prima spostare la testina di lettura e scrittura sulla traccia richiesta. Il tempo necessario per fare questo è mediamente compreso tra $8 \div 10$ millisecondi nei dischi più lenti fino a $4 \div 6$ millisecondi nei dischi più veloci. Il tempo richiesto per spostare la testina di lettura e scrittura sulla traccia richiesta è detto *tempo di posizionamento* TP (seek time). Una volta posizionata la testina sulla traccia occorre attendere che il blocco richiesto passi sotto la testina per essere letto o scritto. Se un disco ruota a 10.000 g/m per effettuare mezzo giro occorrono $60.000/20.000 = 3$ millisecondi. Il tempo speso in questa fase è detto *tempo di latenza* TL (latency). A questo punto inizia l'accesso (in lettura o scrittura) al blocco. Il numero di byte che vengono letti o scritti nell'unità di tempo viene detto *rateo di trasferimento* (RT). In funzione della velocità del disco e della densità di memorizzazione abbiamo oggi ratei di trasferimento che possono arrivare fino a 170 MB/sec. Quindi, per calcolare il tempo necessario per accedere ad un blocco occorre sommare il tempo di posizionamento, il tempo di latenza e infine, sommare il tempo di trasferimento. È facile vedere che il tempo di posizionamento e di latenza sono di gran lunga superiori al tempo di trasferimento.

Si osservi infine che il tempo di lettura medio di un byte sul disco magnetico è diversi ordini di grandezza più lento del tempo che occorre per accedere ad un byte della memoria principale, visto che attualmente i tempi di lettura della memoria RAM sono dell'ordine di miliardesimi di secondo.

Quindi l'accesso ai blocchi in memoria secondaria è il collo di bottiglia per le operazioni sulle basi di dati. Terremo conto di questo aspetto e vedremo che le strutture dati e gli algoritmi implementati per la memorizzazione fisica delle tabelle è tesa a minimizzare il più possibile il numero di blocchi che devono essere acceduti, dalle operazioni, nella memoria secondaria.

Dischi ottici e nastri magnetici I dischi ottici hanno grandi capacità di memorizzazione dei dati ma hanno meno flessibilità dei dischi magnetici i quali, abbiamo visto, consentono una facile scrittura dei dati con tempi accettabili. Viceversa i dischi ottici non si prestano ad essere (ri)scritti facilmente. Per questo, l'impiego dei dischi ottici, rimane limitato a quelle applicazioni che non richiedono scritture di dati ovvero per la memorizzazione di grandi quantità di dati che non cambiano significamente nel tempo.

I nastri magnetici consentono di memorizzare una grande quantità di informazioni e hanno un costo molto contenuto. Come i dischi magnetici possono essere facilmente riscritti. Il problema di questi dispositivi è che consentono un accesso *sequenziale* ai dati che ne limita l'applicazione a specifici contesti: quelli in cui la lettura e scrittura dei dati è di tipo sequenziale. Ad esempio i nastri

sono impiegati in particolar modo nelle applicazioni di *back up* (salvataggio dei dati).

6.2 File e record

Studieremo di seguito le strutture dati utilizzate per supportare il modello relazionale fornito dall'SQL. In SQL abbiamo che ogni tabella (che è il corrispondente di una relazione) è composta logicamente da righe e ciascuna riga è suddivisa in campi o colonne.

Una tabella viene memorizzata permanentemente su dischi magnetici. Ad essa vengono riservati un certo numero di blocchi fisici del disco. L'insieme dei blocchi fisici allocati per una tabella viene chiamato *file*. Per individuare un file occorre quindi conoscere l'insieme degli indirizzi logici dei blocchi ad esso allocati oppure, se i blocchi sono concatenati uno con l'altro come una lista, occorre conoscere l'indirizzo del primo blocco ed eventualmente il numero complessivo dei blocchi del file.

Record Ad ogni riga di una tabella, corrisponde fisicamente un *record*, che è una sequenza di byte nel quale sono memorizzati sia i valori dei campi della riga, sia altri valori che consentono, ai programmi applicativi, di accedere in lettura e scrittura ai valori del record. Possiamo avere record di lunghezza fissa oppure di lunghezza variabile. I record di lunghezza variabile possono esistere perché ad esempio, nella tabella è dichiarato un tipo di dato variabile come `VARCHAR` oppure perché alcuni campi del record possono contenere valori nulli.

Nelle tabelle con record di lunghezza fissa si memorizza, all'inizio del file, la lunghezza del record e la lunghezza di ciascun campo della tabella in modo tale che sia possibile ottenere la posizione iniziale e finale all'interno del record di ciascun campo. In quelle tabelle che hanno record a lunghezza variabile, al fine di conoscere l'inizio e la fine di un campo nel record, sono possibili diverse alternative. Una è quella di indicare all'inizio di ciascun campo a lunghezza variabile del record, il numero di byte utilizzato per quel campo. In alternativa è possibile inserire dei byte speciali non utilizzati per la memorizzazione dei dati per marcare l'inizio e la fine di un campo a lunghezza variabile.

Organizzazione dei record nei blocchi I record possono essere memorizzati nei blocchi consecutivamente. All'interno del blocco può esserci una directory che indica, per ciascun record, la sua posizione nel blocco. In più nel blocco si può mantenere per ciascun record in esso presente, un bit di cancellazione che indica se il record è valido oppure è cancellato. Ci può essere un puntatore che indica la prima posizione libera all'interno del blocco di modo che si possa facilmente inserire un nuovo record nel blocco. Eventualmente il blocco può contenere un puntatore al blocco precedente e/o successivo se i blocchi del file sono concatenati l'uno con l'altro. Un blocco può contenere solo record interi oppure consentire, per ottimizzare lo spazio che un record possa essere memorizzato in due o più blocchi. Quest'ultima opzione è necessaria se la grandezza del record è maggiore della grandezza dello spazio utile nel blocco. Se un record è spezzato in due blocchi occorre mantenere un puntatore al blocco in cui è memorizzata la parte restante del record.

6.3 Organizzazione dei record di una tabella

Abbiamo visto che il tempo necessario al disco magnetico per leggere o scrivere un blocco è diversi ordini di grandezza maggiore del tempo che viene speso per elaborare le informazioni all'interno del blocco, una volta che questo è stato trasferito dal disco alla memoria principale. Sebbene esistano dei meccanismi di *caching* che posticipano quanto più possibile la scrittura di un blocco presente in memoria RAM, sul disco, di modo che questo può essere utilizzato più volte dalla CPU, in quello che segue analizzeremo le prestazioni delle organizzazioni dei record e dei blocchi di un file, utilizzando come costo delle operazioni analizzate, il numero di blocchi che devono essere letti o scritti nella memoria principale. Questa assunzione ci consentirà di determinare le prestazioni *garantite* di ciascuna delle alternative considerate. Le operazione elementari di cui vogliamo misurare l'efficienza rispetto ad un certo tipo di organizzazione di file sono le seguenti

1. La *ricerca* di un record nel file che soddisfa le condizioni richieste.
2. L'*inserimento* di un record nel file.
3. La *cancellazione* di un record nel file che soddisfa le condizioni di ricerca.
4. La *modifica* di un record nel file che soddisfa le condizioni di ricerca.

6.4 File heap

La più semplice organizzazione dei record di una tabella, ma di fondamentale importanza, è l'organizzazione *heap* (dall'inglese: un insieme di cose accatastate le une sopra le altre). Nel file heap i record vengono scritti nei blocchi nell'ordine in cui si presentano. L'inserimento di un nuovo record viene effettuato trasferendo nella memoria principale l'ultimo blocco del file ed inserendo il record in questo blocco. Se il blocco è pieno viene allocato un nuovo blocco su cui viene scritto il nuovo record.

Il file heap ha come punto di forza la sua estrema semplicità ed è molto efficiente nell'operazione di inserimento di un nuovo record. Viceversa per trovare un record che soddisfa le condizioni di ricerca, occorre caricare in memoria principale i blocchi del file partendo dal primo fino a che non si trova il blocco con il record cercato. In media quest'operazione richiede l'accesso e la lettura della metà dei blocchi esistenti e nel caso peggiore, ad esempio quando non esiste un record che soddisfa le condizioni di ricerca, richiede l'accesso a tutti i blocchi del file.

Di conseguenza la cancellazione e la modifica di un record comporta il fatto di dover prima trovare in quale blocco si trova il record da cancellare. Perciò queste operazioni richiedono in media, l'accesso alla metà dei blocchi del file. Dopo aver cancellato o modificato il record dal blocco si riscrive il blocco nella memoria secondaria.

6.5 File ordinati

Per rendere più efficienti le operazioni di ricerca è possibile ordinare i record nei blocchi in funzione dei valori presenti in uno campo della tabella ed ordinare

conseguentemente i blocchi del file. Il campo C sulla base del quale il file viene ordinato verrà indicato nel seguito come *chiave* di ricerca del file. Non necessariamente però questo campo deve corrispondere al concetto di chiave così come è definita nel modello relazionale. Ovvero, in quello che segue in un campo chiave possono anche esserci valori ripetuti.

In un file ordinato, la ricerca di un record che contiene un particolare valore per la chiave C può essere fatta per mezzo della *ricerca binaria*. La ricerca binaria è simile al modo che comunemente utilizziamo per cercare una voce all'interno di un dizionario. Si apre il dizionario all'incirca a metà e si confronta la voce da cercare con le voci della pagina corrente. Se la voce si trova nella pagina corrente allora la ricerca è terminata. Se invece la voce da cercare è minore della prima voce nella pagina corrente si prosegue la ricerca nella prima metà del dizionario, altrimenti si prosegue la ricerca nella seconda metà del dizionario. L'algoritmo di ricerca binaria è indicato in Fig. 6.2. La ricerca

Algoritmo RicercaBinaria
Input: un valore v ed il numero n di blocchi del file
Output: **True** se esiste un record con chiave v, **False** altrimenti
begin
  $I \leftarrow 1$;
  $F \leftarrow n$;
  do repeat
    $i \leftarrow \lceil (I + F)/2 \rceil$;
    if v è in un record nel blocco B_i **then** esci e torna **True**;
    sia K_i la chiave del primo record nel blocco;
    if $v < K_i$ **then** $F \leftarrow i - 1$;
    if $v > K_i$ **then** $I \leftarrow i + 1$;
  until $I \leq F$
  torna **False**;
end

Figura 6.2: Algoritmo di ricerca binaria

binaria richiede, nel caso peggiore, la lettura di $\lfloor log_2(n) \rfloor + 1$, blocchi per trovare il record con valore della chiave richiesto dove $\lfloor x \rfloor$ indica la parte intera di x e $\lceil x \rceil$ indica il più piccolo intero maggiore o uguale a x. Questo perché al più occorrono $\lfloor log_2(n) \rfloor + 1$ dimezzamenti del numero di blocchi in cui cercare la data chiave[1].

L'inserimento di un record risulta meno efficiente dell'inserimento in un file heap. Infatti, si deve inizialmente cercare il blocco corretto in cui inserire il nuovo record al fine di rispettare l'ordinamento esistente. Se nel blocco trovato c'è spazio si inserisce il record spostando opportunamente i record presenti nel blocco. Se invece il blocco trovato è pieno, è possibile seguire diverse strategie. Si può cercare nel blocco precedente o in quello successivo per vedere se c'è spazio ed eventualmente distribuire i record nei due blocchi. Se così non è occorre allocare un nuovo blocco e distribuire i record del vecchio blocco tra il vecchio ed il nuovo blocco. Quindi l'inserimento di un record comporta un accesso a un massimo di $\lfloor log_2(n) \rfloor + 1$ blocchi.

[1] Lo studente è invitato a dare per esercizio una dimostrazione formale di ciò

La cancellazione di un record che contiene una data chiave richiede la ricerca del record per individuare il blocco che lo contiene. Il record viene quindi cancellato dal blocco. Se questo era l'ultimo record presente nel blocco allora è possibile rimuovere l'intero blocco dal file.

Anche la modifica richiede la ricerca del record. Se la modifica da effettuare non coinvolge il campo chiave allora si legge il blocco in memoria, si modifica il record richiesto e si riscrive il blocco sul disco. Se invece la modifica coinvolge il campo chiave occorre, per rispettare l'ordine del file, trovare il blocco nel quale dovrebbe essere correttamente inserito il record modificato. In questo caso la modifica consiste in una cancellazione seguita da un inserimento. In ogni caso la modifica richiede al più di accedere a $O(log_2(n))$ blocchi.

Si osservi che il file ordinato non fornisce alcun aiuto se le operazioni di ricerca vengono fatte su di un campo che non è il campo chiave dell'ordinamento. In questo caso infatti per cercare un record in base al valore di un campo diverso dalla chiave, occorre scorrere tutti i blocchi del file.

Esempio 6.1 Supponiamo di avere un file di $R = 1.500.000$ record nel quale ogni record ha lunghezza $l_r = 120$ byte. Supponiamo che la dimensione di un blocco sia di $b = 4.096$ byte. Supponendo che i blocchi contengano solo record interi abbiamo che un blocco può contenere $e = \lfloor b/l_r \rfloor = 34$ record. Per memorizzare tutti i record del file occorreranno quindi $B = \lceil R/e \rceil = 44.118$ blocchi. Quindi una ricerca binaria richiederà $\lfloor log_2(B) \rfloor + 1 = 16$ accessi ai blocchi. Un grande vantaggio rispetto ai 22.059 accessi che sarebbero stati richiesti in media per trovare un record se il file non fosse ordinato. ■

6.6 File hash

Al fine di rendere l'operazione di ricerca di un record con una data chiave il più efficiente possibile si può utilizzare l'organizzazione *hash*. Nel file hash viene predisposto un vettore di M indirizzi di blocchi detti *bucket* (secchio o contenitore). Una funzione

$$h : dom(C) \rightarrow \{0, 1, \ldots, M-1\}$$

detta *funzione hash* prende il valore v del campo chiave C di un record e restituisce un numero $h(v)$ compreso tra 0 ed $M-1$. Il valore hash $h(v)$ restituisce pertanto il numero del bucket in cui quel record verrà memorizzato. Se e è il numero di record che un blocco può contenere, allora si sceglie M circa pari ad R/e dove R è il numero dei record del file. Generalmente si sceglie M in modo che sia un numero primo. Una tipica funzione di hash è quella che trasforma la sequenza di byte del campo chiave in un numero intero ed applica a tale numero la funzione *modulo* M come indicato in Fig. 6.3. In altri termini data la rappresentazione binaria v della chiave, il valore hash corrispondente per la chiave v viene dato da

$$h(v) = v \ \ mod \ \ M$$

Si noti che poiché R è più grande di M, chiavi diverse avranno lo stesso valore della funzione hash e i record corrispondenti verranno quindi memorizzati nello stesso bucket. La funzione hash deve pertanto essere scelta in modo che i record vengano distribuiti quanto più uniformemente possibile nei bucket. Se ciò non

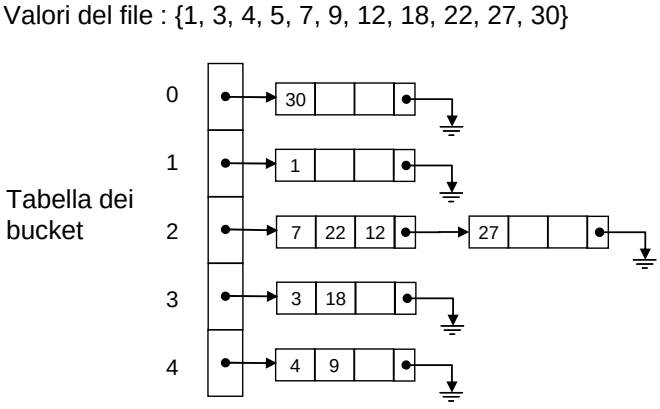


Figura 6.3: Un esempio di organizzazione hash con $M = 5$

avviene può succedere che un bucket diventi pieno. In questo caso si dice che il bucket ha un *trabocco* (overflow). Se un record viene successivamente inserito nello stesso bucket, allora occorre allocare un nuovo blocco e collegare questo nuovo blocco al bucket iniziale. In questo modo ogni bucket sarà formato da una lista, di blocchi collegati tra di loro, organizzata come un file heap. Ad esempio, in Fig. 6.3, il bucket 2 ha un trabocco, ed è composto da una lista di due blocchi.

La ricerca di un record che contiene una data chiave viene effettuata calcolando mediante la funzione hash, il numero del suo bucket. Se in media ad ogni bucket corrisponde un singolo blocco allora il tempo richiesto per la ricerca di un record è pari all'accesso ad un singolo blocco. In caso contrario occorre scorrere tutti i blocchi della lista associata al bucket, fino a che non viene trovato il blocco che contiene il record cercato.

La cancellazione di un record richiede di trovare il bucket corrispondente e il blocco che contiene il record. Eventualmente se il blocco diventa vuoto allora è possibile restituirlo al sistema ed accorciare la lista del bucket.

Per effettuare la modifica, occorre prima cercare il bucket che contiene il record da modificare. Se la modifica del record non coinvolge il campo chiave, allora il record viene modificato e il blocco viene riscritto in memoria secondaria. Se invece la modifica coinvolge il campo chiave, allora potrebbe essere necessario trovare il nuovo bucket del record e spostare in questo bucket il record modificato. In effetti la modifica corrisponde ad una cancellazione seguita da un inserimento. Se ogni bucket contiene in media un solo blocco, sono necessari al più due accessi per modificare un record.

L'organizzazione file hash è molto efficiente per la ricerca di un record con una data chiave. Lo svantaggio di tale organizzazione è che non è possibile recuperare i record nell'ordine della chiave. Ad esempio se si vogliono recuperare tutti i record che hanno un valore per la chiave compreso tra l ed u ciò non può essere fatto efficientemente poiché ogni record che soddisfa le condizioni potrebbe trovarsi in un bucket differente dagli altri. Invece con i file ordinati, questo tipo di operazione è più efficiente poiché i record che soddisfano le condizioni sono memorizzati in blocchi consecutivi del file.

6.7 Indici

Un altro metodo di organizzazione dei dati, teso a rendere ancora più efficienti le operazioni di ricerca, è quello dell'utilizzo di strutture dati chiamate *indici*, da affiancare al file principale.

Queste strutture dati sono simili allo stradario di una città. Lo stradario contiene le mappe di tutte le strade della città. Al fondo dello stradario però, c'è l'elenco in ordine alfabetico di tutte le strade, con accanto indicato, il numero della pagina nel quale è presente la mappa e le coordinate all'interno della mappa del blocco che contiene la strada. In questo modo sapendo il nome della strada è possibile trovare, molto rapidamente, la sua locazione nella mappa.

Un *file indice*, è costituito da un insieme di coppie $< k(r), p(r) >$ dove $k(r)$ è il valore della chiave di un record r e $p(r)$ è un puntatore al blocco del file che contiene il record r. Il numero di coppie presenti nel file indice può essere pari al numero di record del file oppure può essere pari al numero di blocchi del file. Nel primo caso l'indice viene detto *denso*, mentre nel secondo l'indice viene detto *sparso*. Se il file principale è ordinato secondo i valori di un dato campo C

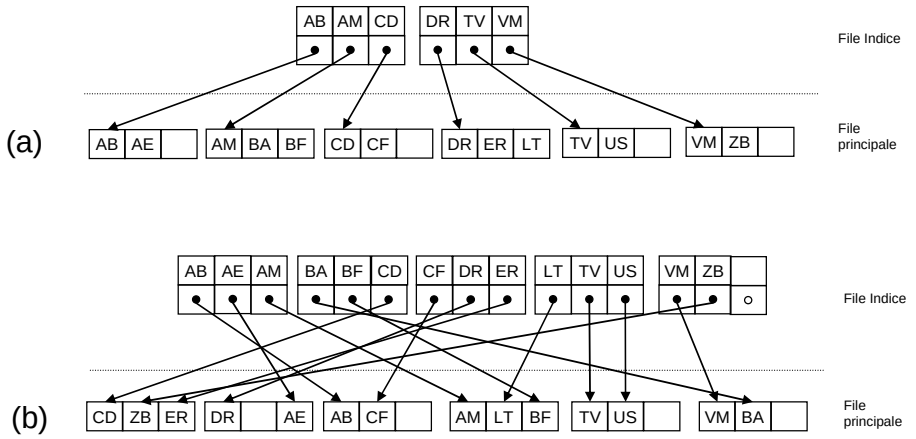


Figura 6.4: File con indice. (a) Indice sparso con file ordinato (b) Indice denso con file heap

(che assumiamo che corrisponda ad una chiave della tabella), e si vuole costruire un file indice sulla chiave C, allora l'indice conterrà una coppia $< k(i), p(i) >$ per ogni blocco del file come indicato ad esempio nella Fig. 6.4(a). In questo caso l'indice è sparso. Il valore $k(i)$ relativo alla coppia i-esima dell'indice, è pari al valore dalla chiave del primo record del blocco i. Le coppie del file indice sono ordinate in funzione dei valori presenti in $k(i)$. Per ricercare un record che contiene la chiave v, possiamo effettuare una ricerca binaria sull'indice. La ricerca binaria termina quando si è trovata una coppia di chiavi dell'indice tali che $k(i) \leq v < k(i+1)$. In questo modo possiamo leggere il puntatore $p(i)$ corrispondente alla chiave $k(i)$ e tramite il puntatore $p(i)$, leggere il blocco del file principale, all'interno del quale viene ricercato il record che contiene la chiave v.

Esempio 6.2 Prendiamo i dati del file dell'Esempio 6.1. Supponiamo che il campo chiave del file sia lungo 8 byte e supponiamo che un puntatore ad un

blocco richieda 4 byte. Poiché il file occupa 44.118 blocchi allora abbiamo bisogno di altrettante coppie $< k(i), p(i) >$ nell'indice. Poiché ogni coppia occupa 12 byte avremo un numero di coppie per blocco pari a $d = \lfloor B/12 \rfloor = 341$. Per memorizzare quindi l'intero indice ci occorrono 120 blocchi. Quindi una ricerca binaria sull'indice richiede $\lfloor log_2(120) \rfloor + 1 = 8$ accessi, più un accesso al file principale, per un totale di 9 accessi. Un buon vantaggio rispetto ai 16 accessi che sarebbero stati richiesti se il file ordinato non avesse avuto un indice. ■

L'inserimento di un record in un file ordinato a cui è associato un indice, è ancora più complesso dell'inserimento di un record in un file ordinato senza indice. Infatti per prima cosa occorre inserire il record nel file ordinato. Se l'inserimento richiede l'allocazione di un nuovo blocco del file principale, allora occorre inserire una nuova voce nell'indice $< k(o), p(o) >$ dove $p(o)$ punta al blocco appena creato e $k(o)$ è la chiave del primo record nel nuovo blocco. Ciò comporta il dover fare spazio nell'indice per la nuova coppia. Nel caso peggiore, se il blocco dell'indice non ha spazio a sufficienza per inserire la nuova coppia, occorre allocare un nuovo blocco anche nel file indice. Questa complessità è attenuata dal fatto che per trovare il blocco nel file principale in cui dobbiamo inserire il nuovo record possiamo utilizzare l'indice stesso con conseguente risparmio di accessi.

Anche nella cancellazione possiamo sfruttare l'indice per cercare il record da cancellare ma, nel caso in cui un blocco contenente il record da cancellare viene disallocato dal file occorre modificare l'indice, cancellando la voce corrispondente. Se il blocco dell'indice, dopo la cancellazione della coppia, diventa vuoto, allora è possibile disallocarlo accorciando così la grandezza dell'indice stesso.

La modifica di un record, per un campo diverso da quello dell'indice, comporta una ricerca del record e la sua modifica. Se invece è il campo dell'indice che viene modificato, allora l'operazione di modifica corrisponderà ad una cancellazione seguita da un inserimento.

Indici densi Supponiamo ora che il file non sia ordinato oppure sia ordinato ma si voglia creare su di esso un indice per un campo diverso da quello utilizzato per l'ordinamento corrente. Supponiamo, per semplicità, che il campo su cui vogliamo creare l'indice abbia tutti valori univoci nel file, e sia pertanto una chiave della tabella. In questo caso o perché il file principale non è ordinato o perché è ordinato ma in base ai valori di un altro campo, nell'indice non abbiamo altra alternativa che inserire una voce per ogni record del file. In questo caso abbiamo un indice denso. Nella Fig. 6.4(b) c'è un esempio di indice denso con file principale organizzato ad heap.

Anche qui per cercare un record possiamo effettuare sempre una ricerca binaria sull'indice. Questo però conterrà tanti valori, quanti sono i record del file. In questo caso la ricerca sul file indice risulterà leggermente più lenta di quella che avremmo effettuato se il file principale fosse stato ordinato.

Esempio 6.3 Prendiamo i dati del file dell'Esempio 6.1 e 6.2. Il file contiene $R = 1.500.000$ record e supponiamo che i record sono in un file heap e perciò non ordinati. Quindi il file indice conterrà una voce per ogni record. Poiché il numero di voci dell'indice che entrano in un in un blocco è $d = 341$ abbiamo

bisogno di non meno di $\lceil R/d \rceil = 4.399$ blocchi per memorizzare l'indice. Una ricerca binaria sul file indice richiede l'accesso a $\lfloor log_2(4.399) \rfloor + 1 = 13$ blocchi. Un rallentamento rispetto all'indice sparso ma sempre un vantaggio rispetto al file ordinato senza indice. Si osservi che se il file era ordinato poi rispetto ad un altro campo, senza questo indice avremmo dovuto effettuare una ricerca sequenziale su tutto il file per trovare il record voluto. ■

Indici su campi con valori ripetuti Vediamo ora cosa succede se utilizziamo, per creare l'indice un campo che non è la chiave della tabella, ovvero all'interno del quale possono esistere valori ripetuti. Abbiamo due alternative. La prima è quando il file è ordinato sul campo dell'indice. In questo caso avremo un indice sparso e conterrà una voce $< k(i), p)(i) >$ per ogni blocco del file. Il valore $k(i)$, corrisponde al valore della chiave del primo record del blocco e $p(i)$ è il puntatore al blocco. Con una ricerca binaria sull'indice si trova il primo blocco che contiene il primo record che ha la data chiave e si scorrono sequenzialmente tutti i record che hanno il valore della chiave di ricerca.

La cancellazione, la modifica e l'inserimento di un record si effettuano con minime differenze rispetto ad un indice su di un campo con valori univoci.
Supponiamo ora che non sia possibile ordinare il file principale sul campo del-

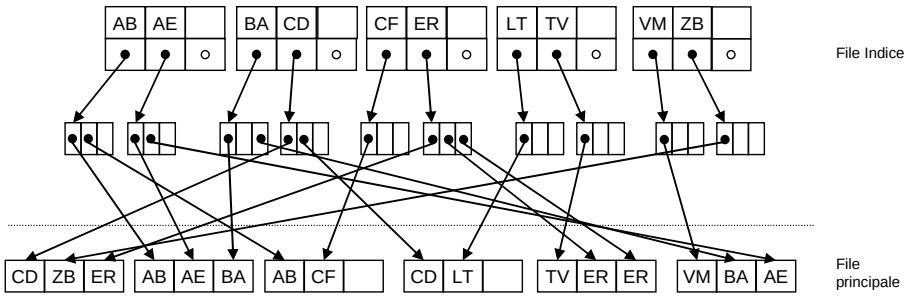


Figura 6.5: File con indice denso su un campo con valori ripetuti

l'indice. In questo caso l'indice conterrà tante voci quanti sono i valori distinti del campo scelto. Una strategia utilizzata in pratica è quella di associare ad ogni voce dell'indice una lista di blocchi che contengono i puntatori ai blocchi in cui è memorizzato un record con la data chiave, come indicato in Fig. 6.5. Per reperire quindi tutti i record che hanno come valore una data chiave, occorre cercare prima nell'indice con una ricerca binaria. Per mezzo del puntatore contenuto nella voce trovata, si legge la lista dei blocchi collegata. Quindi per mezzo dei puntatori nella lista, si reperiscono tutti i record contenenti il dato valore.

6.8 B$^+$-Alberi

Abbiamo visto nella precedente sezione che affiancando un indice ad un file ordinato, gli accessi richiesti per la ricerca di un file diminuiscono ulteriormente. Infatti il numero di accessi richiesto per trovare una dato record in un file ordinato è pari al logaritmo del numero di blocchi B del file, ovvero è $\lfloor log_2(B) \rfloor)+1$. Se

d è il numero di voci dell'indice che entrano in un blocco, allora l'accesso tramite il file indice, richiede la ricerca binaria sull'indice, più un accesso al file principale ovvero $\lfloor log_2(\lceil B/d \rceil) \rfloor + 2$. Se si costruisse un indice dell'indice, avremo che la ricerca binaria sull'indice di secondo livello richiederebbe $\lfloor log_2(\lceil B/d^2 \rceil) \rfloor + 1$. Per accedere al record dobbiamo effettuare altri due accessi, uno all'indice di primo livello e uno al file principale. Il costo complessivo di ricerca con due livelli di indice sarebbe $\lfloor log_2(\lceil B/d^2 \rceil) \rfloor + 3$. Possiamo continuare in questo modo a costruire indici degli indici fino a che nell'ultimo livello, tutte le voci dell'indice risiedano in un singolo blocco. Se h è il numero di indici che abbiamo creato allora deve essere che

$$B/d^h \leq 1$$

Il tempo pertanto necessario per accedere ad un record del file principale è pari al numero di indici che sono stati creati più uno ovvero è $h + 1$.

I *B^+-alberi* (B^+-tree) generalizzano l'idea di associare ad un file una gerarchia di indici. Introduciamo di seguito la notazione utilizzata per gli alberi in generale ed in particolare per i B^+-alberi.

In un albero esiste un particolare nodo detto *radice*. Ogni altro nodo diverso dalla radice ha un particolare nodo detto *padre* a cui è connesso in qualità di figlio. A sua volta un nodo può avere zero o più nodi figli. Un nodo che non è radice ed ha almeno un figlio viene detto *nodo interno*. Un nodo è detto *foglia* se è privo di figli. Dato un arbitrario nodo dell'albero, è possibile raggiungere la radice dell'albero visitando ricorsivamente prima il padre del nodo, poi il padre del padre e così via. Il *livello* di un nodo è pari al numero di nodi che occorre visitare, partendo da esso, per arrivare alla radice. La radice non avendo padre è al livello zero. La *profondità* di un albero è pari alla lunghezza del più lungo percorso tra una foglia e la radice. Un albero è *bilanciato* quando il livello di ogni foglia è pari alla profondità dell'albero; in altre parole, quando un albero è bilanciato allora tutte le foglie sono allo stesso livello.

Un albero T può essere definito ricorsivamente come segue. Un albero T è o un singolo nodo radice oppure è un nodo radice che ha r figli ognuno dei quali è la radice di un albero T_i. Un B^+-albero è un albero bilanciato.

Per semplicità di trattazione nel seguito supporremo che il file principale è ordinato su di un campo C e che i valori su questo campo siano univoci nel file ovvero che C sia una chiave della tabella.

In un B^+-albero ogni nodo contiene un sottoinsieme ordinato di chiavi del file principale. Inoltre, ogni nodo che non è la radice è vincolato ad avere un numero compreso tra d e $2d$ valori del campo chiave al suo interno. Il valore d viene chiamato *grado* del B^+-albero.

Ogni nodo che ha $q - 1$ chiavi e non è un nodo foglia contiene q puntatori ai nodi figli. Le foglie dell'albero contengono tutti i puntatori ai blocchi del file principale (questa distinzione diverrà superflua poiché vedremo che un nodo interno di un B^+-albero è costituito in genere da un singolo blocco). Dato un generico nodo interno del B^+-albero questo contiene, come detto, q puntatori $p_1, p_2, \ldots, p_q$ e $q - 1$ chiavi $k_1, k_2, \ldots, k_{q-1}$ che hanno le seguenti proprietà:

1. $k_1 < k_2 < \cdots < k_{q-1}$

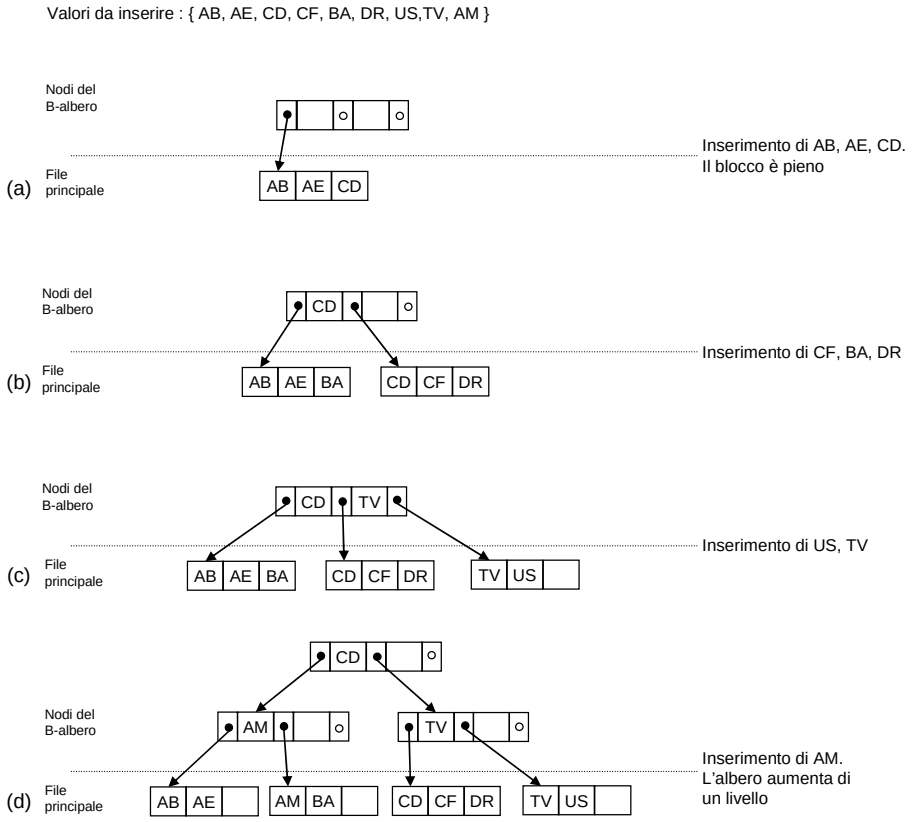


Figura 6.6: Un esempio di organizzazione con B$^+$-albero con $d = 1$

2. il puntatore p_1 punta ad un nodo che è la radice di un B$^+$-albero i cui nodi contengono chiavi x tutte minori di k_1

3. il puntatore p_i, con $1 < i < q$ punta ad un nodo che è la radice di un B$^+$-albero i cui nodi contengono chiavi x tali che $k_i \leq x < k_{i+1}$

4. il puntatore p_q punta ad un nodo che è la radice di un B$^+$-albero i cui nodi contengono chiavi x tutte maggiori od uguali a k_{q-1}

Se il nodo in questione è foglia allora i puntatori e le chiavi in esso contenuti hanno le seguenti proprietà

1. $k_1 < k_2 < \cdots < k_{q-1}$

2. il puntatore p_1 punta al blocco del file principale che contiene record con chiavi x tutte minori di k_1

3. il puntatore p_i, con $1 < i < q$ punta al blocco del file principale che contiene record con chiavi x tali che $k_i \leq x < k_{i+1}$

4. il puntatore p_q punta al blocco del file principale che contiene record con chiavi x tutte maggiori od uguali a k_{q-1}

Per effettuare la ricerca di un record con una data chiave v si ripete ricorsivamente il seguente procedimento iniziando dal nodo radice del B$^+$-albero: quando si analizza un generico nodo non foglia si confronta la chiave v con le chiavi $k_1, k_2, \ldots, k_{q-1}$ presenti nel nodo:

- se $v < k_1$ la ricerca procede nel nodo puntato da p_1
- se esiste una coppia di chiavi $k_i \leq v < k_{i+1}$ allora la ricerca prosegue nel nodo puntato da p_i
- se $k_{q-1} \leq v$ la ricerca procede nel nodo puntato da p_q

Se invece il nodo corrispondente è un nodo foglia allora si confronta la chiave v con le chiavi $k_1, k_2, \ldots, k_{q-1}$ presenti nel nodo:

- se $v < k_1$ allora si legge il blocco del file principale puntato da p_1.
- se esiste una coppia di chiavi $k_i \leq v < k_{i+1}$ allora si legge il blocco del file principale puntato da p_i.
- se $k_{q-1} \leq v$ allora si legge il blocco del file principale puntato da p_q.

Una volta letto il blocco del file principale in memoria si effettua la ricerca nel blocco con la chiave v.

Esempio 6.4 Si consideri il B$^+$-albero di Fig. 6.6(d). Se si vuole ricercare la chiave BA si parte dalla radice. Poiché BA è minore di CD si procede la ricerca nel puntatore corrispondente. Si procede a confrontare BA con l'unica chiave AM presente nel blocco foglia. Poiché BA è maggiore di AM si cerca il record nel blocco del file principale puntato dal puntatore successivo ad AM. ■

Essendo i B$^+$-alberi degli alberi bilanciati, la ricerca di un record con un dato valore di chiave richiede un tempo che è pari all'altezza dell'albero più un accesso al file principale. Infatti l'operazione di ricerca parte dalla radice e arriva fino al nodo foglia che ricopre la chiave di ricerca. Dal nodo foglia si ottiene un puntatore al blocco del file principale.

L'altezza di un B$^+$-albero si ottiene nel seguente modo. I nodi foglia devono contenere un numero di puntatori ai blocchi pari a $\lceil R/e \rceil = B$ dove R è il numero di record del file ed e è il numero di record che entrano in un blocco (si ricordi che il file principale è ordinato sulla chiave). Se, nel caso peggiore, ogni nodo foglia è pieno per metà allora questo conterrà $d+1$ puntatori ed d chiavi. Allora avremmo bisogno per memorizzare le foglie del B$^+$-albero $\lceil B/d \rceil$ nodi. Se i nodi foglia si trovano al livello h e poiché come abbiamo visto ce ne sono B/d avremmo bisogno, livello $h-1$, di $\lceil (B/d)/d \rceil = \lceil B/d^2 \rceil$. L'ultimo livello, il nodo radice del B$^+$-albero, richiederà un singolo blocco per memorizzare i $\lceil B/d^{h-1} \rceil$ nodi del livello inferiore. Quindi avremo che

$$d^{h-1} < B < d^h$$

ovvero passando al logaritmo in base d, che,

$$h = \lceil log_d(B) \rceil$$

Quindi il tempo necessario per accedere ad un record con un organizzazione B$^+$-albero dipende dal $log_d(B)$. Più è grande d e più il livello del B$^+$-albero

diminuisce a parità di dimensioni del file principale. Il valore d è tanto più grande tanto più il numero di coppie $< k, p >$ che possono essere contenute in un blocco è grande. Poiché la grandezza di un puntatore al blocco è costante, allora il numero di coppie in un blocco aumenta tanto più è piccola la dimensione della chiave.

Esempio 6.5 Si considerino i dati dell'Esempio 6.1. Supponiamo, che ogni blocco del file principale e dell'indice sia pieno per metà, ponendoci così nel caso peggiore. Quindi un blocco conterrà $e = \lfloor 4.096/2/120 \rfloor = 17$ record. Allora avremo bisogno di $B = \lceil R/e \rceil = 88.236$ blocchi per memorizzare il file principale. Poiché i blocchi del file indice sono pieni per metà avremo, se p è la lunghezza di un puntatore e k quella della chiave e b la grandezza del blocco, che il numero d dei puntatori di un blocco soddisferà la seguente disequazione

$$k\,d + p(d+1) \leq b/2$$

e quindi abbiamo che

$$k\,d + p\,d + p = d(k+p) + p \leq b/2$$

perciò

$$d = \left\lfloor \frac{b - 2\,p}{2(k+p)} \right\rfloor$$

avendo che $p = 4$, $k = 8$ e che $b = 4.096$ otteniamo che $d = 170$. Avremmo quindi bisogno di $\lceil B/d \rceil = \lceil 88.236/170 \rceil = 520$ blocchi per memorizzare le foglie dell'indice. Quindi occorrono $\lceil B/d \rceil = \lceil 520/170 \rceil = 4$ blocchi per memorizzare il primo livello dell'indice. Infine ci vuole solo un blocco per memorizzare i puntatori ai quattro blocchi del primo livello dell'indice. Il B^+-albero avrà pertanto un altezza pari a 3. Occorreranno quindi 4 accessi per trovare un record con una data chiave[2]. ■

Inserimento L'inserimento di un record in un file a cui è associato un B^+-albero, richiede la ricerca del blocco in cui inserire il record. Se nel blocco c'è spazio a sufficienza il record viene inserito nella posizione corretta eventualmente spostando gli altri record presenti nel blocco. Se invece nel blocco non c'è spazio a sufficienza occorre allocare un nuovo blocco nel file principale dividendo i record presenti nel vecchio blocco uniformemente tra il nuovo e il vecchio blocco. Quindi occorre inserire nel nodo foglia che contiene il puntatore al vecchio nodo un nuovo puntatore ed una nuova chiave, corrispondente al valore della chiave del primo record nel nuovo blocco. Se il nodo foglia ha abbastanza spazio, l'operazione di inserimento termina. Altrimenti occorre allocare un nuovo

[2]Si osservi che per calcolare $h' = log_d(B)$ possiamo utilizzare il logaritmo naturale ln. Infatti

$$ln\,(d^{h'}) = ln\,(B)$$

e quindi

$$h'\,ln(d) = ln\,(B)$$

da cui otteniamo

$$h' = \frac{ln(B)}{ln(d)}$$

nel caso dell'esempio avremo che $h' = 11,38777/5,13579 = 2,21733$. Pertanto passando all'intero superiore, si conferma il calcolo effettuato dell'altezza dell'albero

nodo nel B$^+$-albero e distribuire le voci presenti a metà tra il vecchio ed il nuovo nodo. Inoltre occorre inserire nel padre del vecchio nodo un nuovo puntatore ed una nuova chiave. Questa procedura può ripetersi ricorsivamente in tutti i nodi nel percorso tra il nodo foglia e la radice. Nel caso che si verifichi uno sdoppiamento nel nodo radice allora l'albero aumenta di un livello complessivamente. In genere questo caso si verifica di rado poiché nel B$^+$-albero i nodi sono pieni per metà e l'eventualità di trovare un percorso tra una foglia fino alla radice di nodi tutti pieni è infrequente. In ogni caso l'operazione di inserimento richiede, nel caso peggiore, un numero di accessi proporzionale all'altezza dell'albero ed è pertanto $O(log_d(B))$ dove d e B sono, come sopra, il numero minimo di puntatori presenti nei blocchi e B il numero di blocchi del file principale.

Esempio 6.6 Si consideri il B$^+$-albero di Fig. 6.6(c). L'inserimento di AM deve essere fatto nel blocco del file principale più a sinistra. Poiché questo blocco è pieno si alloca un nuovo blocco e si ripartiscono i record tra il nuovo ed il vecchio blocco (Fig. 6.6(d)). L'aggiunta di un nuovo blocco comporta l'inserimento di un nuovo puntatore nel blocco padre. In questo caso anche il blocco padre è pieno. Ciò comporta un nuovo sdoppiamento con l'aggiunta di un nuovo nodo che diventa la radice dell'albero e conseguente crescita di un livello del B$^+$-albero. ■

Cancellazione La cancellazione di un record comporta l'individuazione del blocco nel file principale contenente quel record e la sua cancellazione. Se il blocco diventa pieno per meno di metà allora è possibile ripartire i record tra il blocco a destra o a sinistra del blocco in considerazione. Se entrambi questi blocchi sono pieni a metà, allora occorre fondere due blocchi tra di loro. La fusione di due blocchi comporta la cancellazione, nel nodo del B$^+$-albero, di un puntatore e della corrispondente chiave, essendoci ora un blocco in meno da puntare. Se nel nodo foglia, la cancellazione di un puntatore fa scendere il numero dei puntatori al di sotto della metà, anche qui si deve cercare di ripartire le voci dell'indice tra i nodi fratelli del nodo in questione, immediatamente alla destra o alla sinistra. Se entrambi questi nodi sono pieni per metà allora, di nuovo, occorre fondere i due nodi ed eliminare il puntatore corrispondente nel nodo padre. Se necessario, si ripete questo procedimento fino ad arrivare alla radice. In quel caso se la radice aveva solo due figli e questi vengono fusi tra di loro allora ciò comporta l'eliminazione della radice con conseguente accorciamento della altezza dell'albero. In ogni caso l'operazione di cancellazione richiede, nel caso peggiore, un numero di accessi proporzionale all'altezza dell'albero ed è pertanto $O(log_d(B))$ dove d e B sono, come prima, il numero di puntatori minimo presenti nei blocchi e B il numero di blocchi del file principale.

Esempio 6.7 Si consideri il B$^+$-albero di Fig. 6.7(a). La cancellazione del record con chiave BM comporta il fatto che il nodo diventa pieno per meno di metà. Poiché il nodo fratello alla sinistra è pieno per metà, allora si accorpano tutti i record in un unico blocco. Viene quindi cancellata la corrispondente coppia nel nodo foglia del B$^+$-albero. Anche qui, dopo la cancellazione il nodo è pieno per meno di metà. Un ulteriore accorpamento dei due nodi foglia fà diminuire il livello dell'albero. ■

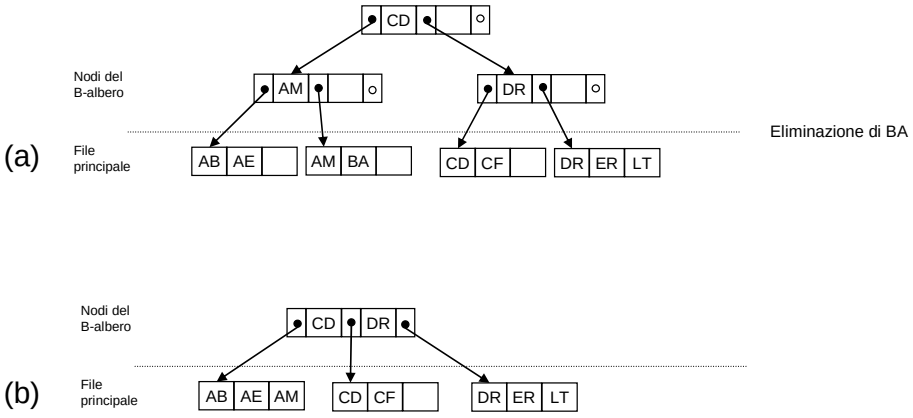


Figura 6.7: (a) Un B albero. (b) Dopo la cancellazione di BM

La trattazione fatta sopra dei B^+-alberi presupponeva che il file principale fosse ordinato sul valore della chiave. Possiamo facilmente applicare i B^+-alberi anche a file heap, avendo così un indice denso sul file. Infine il B^+-albero si presta, con poche e semplici variazioni ad essere utilizzato come indice di campi con valori ripetuti.

Grazie alla loro flessibilità ed efficienza, i B^+-alberi sono il tipo di struttura dati più frequentemente utilizzata per l'indicizzazione e l'organizzazione dei file nei sistemi commerciali di gestione delle basi di dati.

6.9 Esercizi

Esercizio 6.1 Un file heap contiene 230.000 record. Lo spazio utile di ciascun blocco è 4.000 byte. Ogni record, la cui lunghezza è 243 byte, è memorizzato in un singolo blocco e mediamente ogni blocco è pieno al 75%. Si calcoli il tempo medio di ricerca di un record.

Esercizio 6.2 Un file di 2.500.000 record viene ordinato lasciando il 25 % di spazio libero in ciascun blocco. Ogni record ha lunghezza 54 byte e un blocco ha spazio utile per la memorizzazione dei record pari a 8.000 byte. Calcolare il tempo di accesso ad un record con un data chiave. Calcolare lo stesso tempo di accesso per un record che possiede un particolare valore per un campo diverso dalla chiave.

Esercizio 6.3 Si vuole creare un file hash con 340.000 record dove ogni record ha dimensione 340 byte ed un blocco ha dimensione di 8.000 byte. Si trovi il numero di elementi per la tablella dei bucket.

Esercizio 6.4 Si crei da zero un indice B^+-albero denso (che ha tante chiavi nei nodi foglia quanti sono i record del file principale) per il file heap contenente le seguenti chiavi: {12, 34, 2, 7, 93, 13, 27, 83, 49, 71, 64, 53, 92, 6, 15, 17, 23, 39, 41, 50, 38}. Si inseriscano nel B^+-albero le voci nell'ordine indicato supponendo che il grado del B^+-albero sia $d = 2$.

Esercizio 6.5 Un file con 3.432.000 record viene organizzato con un B^+-albero. Il file principale viene ordinato in base al valore di una chiave di lunghezza di 8 byte e ciascun record è composto da 193 byte. Inoltre un puntatore ai blocchi richiede 4 byte. Se un blocco è grande 8.000 byte calcolare il costo nel caso peggiore, necessario per trovare un record che ha come chiave un dato valore. Calcolare lo spazio in MB complessivo necessario per memorizzare sia il file principale che i nodi del B^+-albero.

Esercizio 6.6 Un file con 456.700 record viene memorizzato come file heap. Si vuole costruire però un indice B^+-albero denso su di un campo chiave del file senza ordinare il file. Il campo chiave ha lunghezza 13 byte, un puntatore richiede 8 byte e ciascun blocco è grande 4.000 byte. Calcolare il tempo necessario per ricercare un record.

Capitolo 7

Gestione della concorrenza[1]

Un criterio per classificare i sistemi di gestione di basi di dati è il numero di utenti che possono usare il sistema concorrentemente. Un sistema è *singolo-utente* se può essere usato da al più un utente alla volta ed è *multiutente* se molti utenti possono usarlo contemporaneamente; la maggior parte dei sistemi di gestione di basi di dati è del secondo tipo.

Se il sistema di calcolo ha più CPU allora è possibile il simultaneo processamento di due programmi da parte di due diverse CPU; tuttavia la maggior parte della teoria del controllo della concorrenza nelle basi di dati è stata sviluppata per sistemi di calcolo con una sola CPU. In tali sistemi i programmi sono eseguiti concorrentemente in modo *interleaved* (interfogliato): la CPU può eseguire un solo programma alla volta, tuttavia il sistema operativo permette di eseguire alcune istruzioni di un programma, sospendere quel programma, eseguire istruzioni di un altro programma e quindi ritornare ad eseguire istruzioni del primo. In tal modo l'esecuzione concorrente dei programmi è interleaved; ciò consente di tenere la CPU occupata quando un programma deve effettuare operazioni di I/O.

In un sistema di gestione di basi di dati multiutente la principale risorsa a cui i vari programmi accedono concorrentemente è la base di dati. L'esecuzione di una parte di un programma che rappresenta un'unità logica di accesso o modifica del contenuto della base di dati è detta *transazione*. Ci sono sistemi (ad esempio le basi di dati statistici) in cui gli utenti effettuano solo interrogazioni ma non modifiche; in tali sistemi l'esecuzione concorrente di più transazioni non crea problemi. Al contrario nei sistemi in cui vengono effettuate da più utenti sia operazioni di lettura che di scrittura (un tipico esempio di sistemi di questo tipo è costituito dai sistemi per la prenotazione di posti sui voli) l'esecuzione concorrente di più transazioni può provocare problemi se non viene controllata in qualche modo.

Prima di esaminare alcuni dei problemi che possono sorgere quando l'esecuzione concorrente di più transazioni non è controllata, introduciamo il concetto di *schedule* (piano di esecuzione) di un insieme di transazioni. Dato un insieme $\mathcal{T}$ di transazioni uno schedule S di $\mathcal{T}$ è un ordinamento delle operazioni nelle

[1] Il materiale di questo capitolo è stato interamente tratto dalle dispense del corso di basi di dati della Prof. Marina Moscarini

transazioni in $\mathcal{T}$ tale che per ogni transazione T in $\mathcal{T}$ se o_1 e o_2 sono due operazioni in T tali che o_1 precede o_2 in T allora o_1 precede o_2 in S (in altre parole uno schedule deve conservare l'ordine che le operazioni hanno all'interno delle singole transazioni). Qualsiasi schedule ottenuto permutando le transazioni in $\mathcal{T}$ è detto *seriale*. Consideriamo le seguenti transazioni

T_1
$read(X)$
$X := X - N$
$write(X)$
$read(Y)$
$Y := Y + N$
$write(Y)$

T_2
$read(X)$
$X := X + M$
$write(X)$

e i seguenti schedule di $\{T_1, T_2\}$

T_1	T_2
$read(X)$	
$X := X - N$	
	$read(X)$
	$X := X + M$
$write(X)$	
$read(Y)$	
	$write(X)$
$Y := Y + N$	
$write(Y)$	

T_1	T_2
$read(X)$	
$X := X - N$	
$write(X)$	
	$read(X)$
	$X := X + M$
	$write(X)$
$read(Y)$	
T_1 fallisce	

Nel primo caso l'aggiornamento di X prodotto da T_1 viene perso in quanto T_2 legge il valore di X prima che l'aggiornamento prodotto da T_1 sia stato reso permanente. Nel secondo caso T_2 legge e aggiorna il valore di X dopo che l'aggiornamento prodotto da T_1 è stato reso permanente, ma prima che venga ripristinato il vecchio valore di X in conseguenza del fallimento di T_1 .

Consideriamo ora una transazione T_3 che somma i valori di X e di Y. Il seguente schedule di $\{T_1, T_3\}$ fa sì che la somma prodotta da T_3 sia la somma del valore di X dopo che X è stato aggiornato da T_1 e del valore di Y prima che sia stato aggiornato da T_1.

T_1	T_3
	$s := 0$
$read(X)$	
$X := X - N$	
$write(X)$	
	$read(X)$
	$s := s + X$
	$read(Y)$
	$s := s + Y$
$read(Y)$	
$Y := Y + N$	
$write(Y)$	

In tutti e tre i casi visti siamo portati a considerare gli schedule non corretti in quanto i valori prodotti non sono quelli che si avrebbero se le due transazioni fossero eseguite nel modo "naturale" cioè sequenzialmente.

In generale possiamo osservare che l'esecuzione naturale e, quindi, intuitivamente corretta di un insieme di transazioni è quella sequenziale; la possibilità di eseguire concorrentemente un insieme di transazioni, come si è detto, è introdotta nei sistemi per motivi di efficienza. Pertanto tutti gli schedule seriali sono corretti e uno schedule non seriale è corretto se è *serializzabile*, cioè se è equivalente ad uno schedule seriale. Sorge quindi la necessità di definire un concetto di equivalenza di schedule.

La più semplice definizione di equivalenza potrebbe essere basata sul confronto del risultato: due schedule sono equivalenti se producono lo stesso stato finale. Tale definizione non è però soddisfacente in quanto due schedule potrebbero produrre lo stesso stato finale solo per alcuni valori iniziali. Consideriamo ad esempio le due transazioni

T_1
$read(X)$
$X := X + 5$
$write(X)$

T_2
$read(X)$
$X := X * 1.5$
$write(X)$

e gli schedule

T_1	T_2
$read(X)$	
	$read(X)$
$X := X + 5$	
	$X := X * 1.5$
	$write(X)$
$write(X)$	

T_1	T_2
$read(X)$	
	$read(X)$
	$X := X * 1.5$
$X := X + 5$	
$write(X)$	
	$write(X)$

Tali schedule producono gli stessi valori solo se il valore iniziale di X è 10; ma producono valori diversi in tutti gli altri casi. Problemi di questo tipo potrebbero essere evitati sfruttando proprietà algebriche che garantiscano che il risultato è lo stesso indipendentemente dai valori iniziali delle variabili; tuttavia tale soluzione richiederebbe dei costi inaccettabili (che non sono giustificati dallo scopo che si vuole raggiungere). Pertanto si fa l'assunzione più restrittiva che *due valori sono uguali solo se sono prodotti da esattamente la stessa sequenza di operazioni.* Quindi, ad esempio, date le due transazioni

T_1
$read(X)$
$X := X + N$
$write(X)$

T_2
$read(X)$
$X := X - M$
$write(X)$

i due schedule

T_1	T_2
$read(X)$	
$X := X + N$	
$write(X)$	
	$read(X)$
	$X := X - M$
	$write(X)$

T_1	T_2
	$read(X)$
	$X := X - M$
	$write(X)$
$read(X)$	
$X := X + N$	
$write(X)$	

non sono considerati equivalenti.

Oltre alla definizione di equivalenza, un altro elemento che ha influenza sulla complessità del problema di decidere se uno schedule è serializzabile (cioè se è equivalente ad uno schedule seriale) è costituito dal fatto che il valore calcolato da una transazione per ogni dato sia dipendente o meno dal vecchio valore di quel dato. Nel primo caso il problema della serializzabilità può essere risolto in tempo polinomiale con un semplice algoritmo su grafi; nel secondo caso il problema risulta essere NP-completo.

Nella pratica è difficile testare la serializzabilità di uno schedule. Infatti l'ordine di esecuzione delle operazioni delle diverse transazioni è determinato in base a diversi fattori: il carico del sistema, l'ordine temporale in cui le transazioni vengono sottomesse al sistema e le loro priorità. Pertanto è praticamente impossibile determinare in anticipo come le operazioni saranno interleaved, cioè in quale ordine verranno eseguite; d'altra parte, se prima si eseguono le operazioni e poi si testa la serializzabilità dello schedule, i suoi effetti devono essere annullati se lo schedule risulta non serializzabile. Inoltre quando le transazioni vengono sottomesse al sistema in modo continuo è difficile stabilire quando uno schedule comincia e quando finisce. Quindi l'approccio seguito nei sistemi è quello di determinare metodi che garantiscano la serializzabilità di uno schedule eliminando così la necessità di dover testare ogni volta la serializzabilità di uno schedule. Uno di tali metodi consiste nell'imporre dei *protocolli*, cioè delle regole, alle transazioni in modo da garantire la serializzabilità di ogni schedule. Questi protocolli usano tecniche di *locking* (cioè di controllo dell'accesso ai dati) per prevenire l'accesso concorrente ai dati. Altri metodi di controllo usano i *timestamp* delle transazioni, cioè degli identificatori delle transazioni che vengono generati dal sistema e in base ai quali le operazioni delle transazioni possono essere ordinate in modo da assicurare la serializzabilità.

7.1 Item

Tutte le tecniche per la gestione della concorrenza richiedono che la base di dati sia partizionata in *item*, cioè in unità a cui l'accesso è controllato. Le dimensioni degli item devono essere definite in base all'uso che viene fatto della base di dati in modo tale che in media una transazione acceda a pochi item. Ad esempio se la transazione tipica su una base di dati relazionale è la ricerca di una ennupla mediante un indice, è appropriato trattare le ennuple come item; se invece la transazione tipica consiste nell'effettuazione di un join di due relazioni, è opportuno considerare le relazioni come item. Le dimensioni degli item usate da un sistema sono dette la sua granularità. Una granularità grande

permette una gestione efficiente della concorrenza; una piccola granularità può invece sovraccaricare il sistema, ma consente l'esecuzione concorrente di molte transazioni.

7.2 Tecniche di locking per il controllo della concorrenza

Queste tecniche fanno uso del concetto di lock. Un *lock* è un privilegio di accesso ad un singolo item. In pratica è una variabile associata all'item il cui valore descrive lo stato dell'item rispetto alle operazioni che possono essere effettuate su di esso. Un lock viene richiesto da una transazione mediante un'operazione di *locking* e viene rilasciato mediante un'operazione di *unlocking*; fra l'esecuzione di un'operazione di locking su un certo item X e l'esecuzione di un'operazione di unlocking su X diciamo che la transazione mantiene un lock su X. Sono stati studiati diversi tipi di lock; in ogni caso si assume che una transazione debba effettuare un'operazione di locking ogni volta che deve leggere o scrivere un item e che l'operazione agisca come primitiva di sincronizzazione, cioè se una transazione richiede un lock su un item su cui un'altra transazione mantiene un lock, la transazione non può procedere finchè il lock non viene rilasciato dall'altra transazione. Inoltre si assume che ciascuna transazione rilascia ogni lock che ha ottenuto. Uno schedule è detto *legale* se obbedisce a queste regole.

7.2.1 Lock binario

Un lock binario può assumere solo due valori *locked* e *unlocked*. Le transazioni fanno uso di due operazioni $lock(X)$ e $unlock(X)$; la prima serve per richiedere l'accesso all'item X, la seconda per rilasciare l'item X consentendone l'accesso ad altre transazioni. Se una transazione richiede l'accesso ad un item X mediante un $lock(X)$ e il valore della variabile è *locked* la transazione viene messa in attesa, altrimenti viene consentito alla transazione l'accesso ad X e alla variabile associata ad X viene assegnato il valore *locked*. Se una transazione rilascia un item X mediante un $unlock(X)$, alla variabile associata ad X viene assegnato il valore *unlocked*; in tal modo se un'altra transazione è in attesa di accedere ad X l'accesso gli viene consentito.

Consideriamo di nuovo le due transazioni dell'esempio precedente e vediamo come l'uso dei lock può prevenire il problema dell'"aggiornamento perso". Le transazioni T_1 e T_2 risultano modificate nel modo seguente

T_1
$lock(X)$
$read(X)$
$X := X - N$
$write(X)$
$unlock(X)$
$lock(Y)$
$read(Y)$
$Y := Y + N$
$write(Y)$
$unlock(Y)$

T_2
$lock(X)$
$read(X)$
$X := X + M$
$write(X)$
$unlock(X)$

Uno schedule legale di T_1 e T_2 è il seguente

T_1	T_2
$lock(X)$	
$read(X)$	
$X := X - N$	
$write(X)$	
$unlock(X)$	
	$lock(X)$
	$read(X)$
	$X := X + M$
	$write(X)$
	$unlock(X)$
$lock(Y)$	
$read(Y)$	
$Y := Y + N$	
$write(Y)$	
$unlock(Y)$	

Il più semplice modello per le transazioni è quello che considera una transazione come una sequenza di operazioni di *lock* e *unlock*. Si assume che ogni operazione di *lock* su un item X implica la lettura di X e ogni operazione di *unlock* di un item X implica la scrittura di X. Il nuovo valore dell'item viene calcolato da una funzione che è associata in modo univoco ad ogni coppia *lock-unlock* ed ha per argomenti tutti gli item letti (locked) dalla transazione prima dell'operazione di *unlock*. I valori che un item assume durante l'esecuzione di una transazione sono formule costruite applicando le funzioni suddette ai valori iniziali degli item. Due schedule sono *equivalenti* se le formule che danno i valori finali per ciascun item sono le stesse per i due schedule.

Consideriamo ad esempio le due transazioni seguenti

T_1
$lock(X)$
$unlock(X)$ $f_1(X)$
$lock(Y)$
$unlock(Y)$ $f_2(X,Y)$

T_2
$lock(Y)$
$unlock(Y)$ $f_3(Y)$
$lock(X)$
$unlock(X)$ $f_4(X,Y)$

e il seguente schedule S di $\{T_1, T_2\}$

T_1	T_2
$lock(X)$	
$unlock(X)$	
	$lock(Y)$
	$unlock(Y)$
$lock(Y)$	
$unlock(Y)$	
	$lock(X)$
	$unlock(X)$

Se indichiamo con X_0 e Y_0 i valori iniziali di X e Y, abbiamo che il valore finale per X prodotto da S è dato dalla formula $f_4(f_1(X_0), Y_0)$. D'altra parte il valore finale per X prodotto dallo schedule seriale T_1, T_2 è dato dalla formula $f_4(f_1(X_0), f_2(X_0, Y_0))$, mentre il valore finale per X prodotto dallo schedule seriale T_2, T_1 è dato dalla formula $f_1(f_4(X_0, Y_0))$; pertanto S non è serializzabile in quanto non è equivalente a nessuno schedule seriale di $\{T_1, T_2\}$.

Consideriamo ora le due transazioni seguenti

T_1
$lock(X)$
$unlock(X)$ $f_1(X)$
$lock(Y)$
$unlock(Y)$ $f_2(X,Y)$

T_2
$lock(X)$
$unlock(X)$ $f_3(X)$
$lock(Y)$
$unlock(Y)$ $f_4(X,Y)$

e il seguente schedule S di $\{T_1, T_2\}$

T_1	T_2
$lock(X)$	
$unlock(X)$	
	$lock(X)$
	$unlock(X)$
$lock(Y)$	
$unlock(Y)$	
	$lock(Y)$
	$unlock(Y)$

Se di nuovo indichiamo con X_0 e Y_0 i valori iniziali di X e Y, abbiamo che il valore finale per X e Y prodotti da S sono dati rispettivamente dalle formule $f_3(f_1(X_0))$ e $f_4(f_1(X_0), f_2(X_0, Y_0))$. Poiché tali formule coincidono con quelle prodotte dallo schedule seriale T_1 ,T_2 lo schedule S è serializzabile.

La serializzabilità di uno schedule in questo semplice modello, può essere testata mediante un semplice algoritmo su grafi diretti. L'idea su cui si basa tale algoritmo è la seguente. Dato uno schedule, per ogni item si esamina l'ordine in cui le varie transazioni fanno un lock su quell'item; se lo schedule è serializzabile questo ordine deve essere consistente con quello di uno schedule seriale; pertanto se l'ordine imposto sulle transazioni da un certo item è diverso da quello imposto da un altro item, lo schedule non è serializzabile.

L'algoritmo, dato uno schedule S, lavora nel modo seguente.

Algoritmo 1

- crea un grafo diretto G (*grafo di serializzazione*) i cui nodi corrispondono alle transazioni; in tale grafo c'è un arco diretto da T_i a T_j se in S abbiamo che T_i esegue un $unlock(X)$ e T_j esegue il successivo $lock(X)$ (il significato

intuitivo dell'esistenza di un arco da T_i a T_j è che T_j legge il valore per X scritto da T_i e quindi se esiste uno schedule seriale equivalente ad S in tale schedule T_i deve precedere T_j)

- verifica se G ha un ciclo. Se G ha un ciclo allora S non è serializzabile; altrimenti esiste un ordinamento S' delle transazioni tale che T_i precede T_j se c'è in G un arco diretto da T_i a T_j; tale ordinamento può essere ottenuto applicando a G il procedimento noto come *ordinamento topologico* consistente nell'eliminare ricorsivamente da un grafo diretto i nodi che non hanno archi entranti (l'ordine di eliminazione dei nodi fornisce lo schedule seriale S')

Il seguente teorema prova la correttezza dell'algoritmo.

Teorema 1 *L'Algoritmo 1 determina correttamente se uno schedule è serializzabile.*

Dim: Cominciamo con il dimostrare che se il grafo di serializzazione G contiene un ciclo allora S non è serializzabile. Supponiamo, per assurdo, che G contenga un ciclo $T_1, T_2, \ldots, T_k, T_1$ e che esista uno schedule seriale S' equivalente ad S. Sia T_i, $1 \leq i \leq k$, la transazione del ciclo che compare per prima in S' e sia X l'item che causa la presenza in G dell'arco da T_{i-1} a T_i. Il valore finale per X prodotto da S' è dato da una formula in cui compare una funzione f associata ad una coppia *lock-unlock* in T_{i-1} e almeno uno degli argomenti di f è una formula in cui compare una funzione g associata ad una coppia *lock-unlock* in T_i. D'altra parte in S l'effetto di T_{i-1} su X precede quello di T_i su X; quindi nella formula che dà il valore finale per X prodotto da S la funzione f compare più internamente di g (f è applicata prima di g). Pertanto S' ed S non sono equivalenti (contraddizione).

Mostriamo ora che se G non ha cicli allora S è serializzabile. A tal fine definiamo *profondità* di una transazione T la lunghezza del più lungo cammino in G da un qualsiasi nodo a T.

Sia S' lo schedule seriale costruito dall'algoritmo; mostreremo per induzione sulla profondità delle transazioni che ogni transazione T per ogni item su cui effettua un'operazione di *lock* legge in S lo stesso valore che legge in S' (e quindi per ogni item su cui effettua un'operazione di *lock* produce in S lo stesso valore che produce in S').

Base dell'induzione. Se per una transazione T la profondità è 0 vuol dire che in G non ci sono archi entranti in T; pertanto in S la transazione T legge solo valori iniziali. D'altra parte in S' la transazione T viene prima di qualsiasi transazione che effettua un'operazione di *lock* su un item su cui T effettua un'operazione di *lock*. Infatti, se X è un item su cui T effettua un'operazione di lock e $T_{i_1}, T_{i_2}, \ldots, T_{i_k}$ è la sequenza in S delle transazioni che effettuano un'operazione di lock su X, $T_{i_1}, T_{i_2}, \ldots, T_{i_k}$ è un cammino in G; poiché T deve comparire in tale cammino e in G non ci sono archi entranti in T, si deve avere $T = T_{i_1}$ e quindi nessun altro nodo del cammino può essere eliminato dal procedimento di ordinamento topologico prima di T.

Induzione. Cominciamo con il mostrare che ogni transazione per ogni item su cui effettua un'operazione di *lock* legge sia in S che in S' il valore prodotto da una stessa transazione. Supponiamo, per assurdo, che esistano una transazione T e un item X tali che T legge in S il valore di X prodotto da una transazione

T' e in S' il valore prodotto da un'altra transazione T''. Sia $T_{i_1}, T_{i_2}, \ldots, T_{i_k}$ la sequenza in S delle transazioni che effettuano un'operazione di lock su X; $T_{i_1}, T_{i_2}, \ldots, T_{i_k}$ è un cammino in G in cui T' compare immediatamente prima di T. Anche T'' compare in tale cammino, e quindi deve comparire prima di T'. Poiché in S', T'' deve seguire T', S' non può essere ottenuto da G mediante il procedimento di ordinamento topologico (contraddizione). Sia T una transazione che ha profondità d, $d > 0$, in G; per quanto visto T, per ogni item su cui effettua un'operazione di lock, legge sia in S che in S' il valore prodotto da una stessa transazione T'. Poiché T' ha profondità minore di d per l'ipotesi induttiva T', per ogni item su cui effettua un'operazione di lock, legge sia in S che in S' lo stesso valore e, quindi, produce sia in S che in S' lo stesso valore. Pertanto T, per ogni item su cui effettua un'operazione di lock, legge sia in S che in S' lo stesso valore. □

Diciamo che una transazione obbedisce al protocollo di locking a due fasi, o più semplicemente che *è a due fasi*, se prima effettua tutte le operazioni di *lock* (*fase di locking*) e poi tutte le operazioni di *unlock* (*fase di unlocking*). Mostreremo che il protocollo di locking a due fasi garantisce la serializzabilità; più precisamente, si ha che:

Teorema 2 *Sia $\mathcal{T}$ un insieme di transazioni. Se ogni transazione in $\mathcal{T}$ è a due fasi allora ogni schedule di $\mathcal{T}$ è serializzabile.*

Dim: Sia S uno schedule di $\mathcal{T}$. Supponiamo, per assurdo, che S non sia serializzabile. Per il Teorema 1, il grafo di serializzazione G di S contiene un ciclo $T_1, T_2, \ldots, T_k, T_1$; ciò significa che T_{i+1}, $i = 1, \ldots, k-1$, effettua un'operazione di lock, su un item su cui T_i ha effettuato un'operazione di unlock e che T_1 effettua un'operazione di lock, su un item su cui T_k ha effettuato un'operazione di unlock. Pertanto in S si ha che T_1 effettua un'operazione di lock dopo aver effettuato un'operazione di unlock e quindi T_1 non è a due fasi (contraddizione). □

Mostreremo ora che se una transazione T_1 non è a due fasi, esiste sempre una transazione T_2 tale che esiste uno schedule di $\{T_1, T_2\}$ che non è serializzabile. Infatti, se T_1 non è a due fasi effettua un'operazione di lock dopo aver effettuato un'operazione di unlock:

T_1
$\vdots$
$unlock(X)$
$\vdots$
$lock(Y)$
$\vdots$

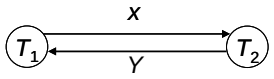
Sia T_2 la seguente transazione

T_2
$lock(X)$
$lock(Y)$
$unlock(X)$
$unlock(Y)$

Il seguente schedule di $\{T_1, T_2\}$ non è serializzabile

T_1	T_2
$\vdots$	
$unlock(X)$	
$\vdots$	
	$lock(X)$
	$lock(Y)$
	$unlock(X)$
	$unlock(Y)$
$\vdots$	
$lock(Y)$	
$\vdots$	

Infatti il suo grafo di serializzazione



contiene un ciclo. Naturalmente possono esistere schedule di transazioni che non sono a due fasi che sono serializzabili. Tuttavia poiché è normale non conoscere l'insieme di transazioni con cui una transazione può essere eseguita concorrentemente siamo costretti a richiedere che tutte le transazioni siano a due fasi perché sia garantita la serializzabilità di ogni schedule.

7.2.2 Lock a tre valori

Consideriamo ora un modello per le transazioni che consente un maggior grado di concorrenza. In tale modello si fa l'assunzione realistica che una transazione possa accedere ad un item solo per leggerlo, senza modificarlo. Se una transazione desidera solo leggere un item X effettua un'operazione $rlock(X)$ che impedisce a qualsiasi altra transazione di modificare il valore di X; tuttavia un qualsiasi numero di transazioni può ottenere contemporaneamente un lock di lettura su X. Se, invece, una transazione desidera modificare il valore di X effettua un'operazione $wlock(X)$; in tal caso nessuna altra transazione può ottenere un lock di scrittura o di lettura su X. Entrambi i lock sono rilasciati mediante un'operazione di $unlock(X)$. Pertanto si fa uso di un lock a tre valori: *rlocked*, *wlocked*, *unlocked*.

Quanto detto per gli schedule legali vale anche in questo modello con l'unica differenza che una transazione che mantiene un lock di lettura su un certo item può richiedere un lock in scrittura su quello stesso item. Analogamente a quanto fatto per il modello precedente assumiamo che quando una transazione effettua un *wlock* su un item il nuovo valore dell'item viene calcolato da una funzione, univocamente associata a quell'operazione di *wlock*, che ha come argomenti tutti gli item su cui la transazione ha già effettuato un lock (di lettura o di scrittura) prima dell'*unlock* associato al *wlock*. Quindi, analogamente a quanto accadeva nel modello precedente, l'effetto di uno schedule sulla base di dati può essere

espresso dalle formule che calcolano (a partire dai valori iniziali degli item e applicando le funzioni associate ai *wlock*) i valori degli item che sono modificati da almeno una transazione. Tuttavia, poiché in questo modello si assume che una transazione possa leggere un item senza modificarlo, la definizione di equivalenza di schedule deve essere modificata per tener conto di tale eventualità.

Due schedule sono *equivalenti* se:
1. producono lo stesso valore per ogni item su cui viene effettuato un *wlock*
2. ogni operazione $rlock(X)$ legge lo stesso valore di X nei due schedule.

Vediamo ora quali condizioni di precedenza tra transazioni è possibile inferire dalla semantica delle transazioni appena vista.

Supponiamo che in uno schedule S una transazione T_1 effettui un'operazione *wlock* su un item X e che una transazione T_2 effettui un'operazione *rlock* su X prima che una terza transazione T_3 esegua la successiva operazione di *wlock* su X (in altre parole, T_1 modifica il valore di X e T_2 legge il valore di X prodotto da T_1 prima che X venga nuovamente modificato da T_3). Allora in qualsiasi schedule seriale equivalente ad S, T_1 deve precedere T_2 e T_2 deve precedere T_3. D'altra parte, se due transazioni T_1 e T_2 leggono entrambe il valore di un item X prodotto da una transazione non è lecito stabilire nessuna precedenza tra T_1 e T_2 . Per rappresentare le precedenze tra le transazioni è possibile usare, come per il modello precedente, un grafo diretto che ha per nodi le transazioni e ha un arco da una transazione T_i a una transazione T_j se la semantica delle transazioni impone che T_i debba precedere T_j. Analogamente a quanto accade per il modello precedente, un semplice algoritmo su tale grafo permette di decidere se uno schedule è serializzabile e in caso affermativo di ottenere uno schedule seriale equivalente ad esso. L'algoritmo, dato uno schedule S, lavora nel modo seguente.

Algoritmo 2

- crea un grafo diretto G (*grafo di serializzazione*) i cui nodi corrispondono alle transazioni; in tale grafo c'è un arco diretto da T_i a T_j se
  - in S la transazione T_i esegue una $rlock(X)$ o una $wlock(X)$ e T_j è la transazione che esegue la successiva $wlock(X)$
  - in S la transazione T_i esegue una $wlock(X)$ e T_j esegue una $rlock(X)$ dopo che T_i ha eseguito la $wlock(X)$ e prima che un'altra transazione esegua una $wlock(X)$.
- verifica se G ha un ciclo. Se G ha un ciclo allora S non è serializzabile; altrimenti esiste un ordinamento S' delle transazioni tale che T_i precede T_j se c'è in G un arco diretto da T_i a T_j; tale ordinamento può essere ottenuto applicando a G il procedimento di ordinamento topologico.

Il seguente teorema, che può essere dimostrato con la stessa tecnica usata per il Teorema 1, prova la correttezza dell'algoritmo.

Teorema 3 *L'Algoritmo 2 determina correttamente se uno schedule è serializzabile.*

Consideriamo le seguenti transazioni

T_1
$rlock(X)$
$unlock(X)$
$wlock(Y)$
$unlock(Y)$

T_2
$wlock(X)$
$unlock(X)$
$rlock(Z)$
$unlock(Z)$

T_3
$rlock(Y)$
$unlock(Y)$
$wlock(Z)$
$unlock(Z)$

Il seguente schedule di $\{T_1, T_2, T_3\}$ è serializzabile

T_1	T_2	T_3
$rlock(X)$		
$unlock(X)$		
	$wlock(X)$	
	$unlock(X)$	
$wlock(Y)$		
$unlock(Y)$		
		$rlock(Y)$
		$unlock(Y)$
		$wlock(Z)$
		$unlock(Z)$
	$rlock(Z)$	
	$unlock(Z)$	

infatti il suo grafo di serializzazione riportato in Fig. 7.1(a)

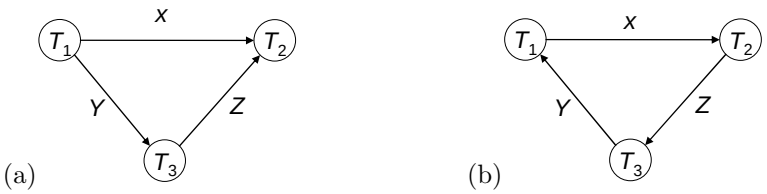


Figura 7.1: Grafi di serializzazione

non contiene cicli; lo schedule seriale equivalente fornito dall'algoritmo è T_1 T_3 T_2 . Al contrario il seguente schedule di $\{T_1, T_2, T_3\}$ non è serializzabile

T_1	T_2	T_3
$rlock(X)$		
$unlock(X)$		
	$wlock(X)$	
	$unlock(X)$	
		$rlock(Y)$
		$unlock(Y)$
	$rlock(Z)$	
	$unlock(Z)$	
		$wlock(Z)$
		$unlock(Z)$
$wlock(Y)$		
$unlock(Y)$		

infatti il suo grafo di serializzazione di Fig. 7.1(b) contiene un ciclo.

Analogamente a quanto mostrato per il modello precedente anche per questo modello è possibile dimostrare che qualsiasi schedule di un insieme di transazioni a due fasi (cioè transazioni in cui nessuna operazione di lock può seguire una operazione di *unlock*) è serializzabile e che per qualsiasi transazione T a due fasi esiste uno schedule di transazioni contenente T e almeno una transazione non a due fasi che non è serializzabile.

7.2.3 Read-only, write-only

Nei due modelli di transazioni esaminati precedentemente viene fatta l'assunzione che ogni volta che una transazione modifica un item X legge il vecchio valore di X e il nuovo valore di X dipende dal vecchio; in particolare nel primo modello l'insieme degli item letti e quello degli item scritti da una transazione coincidono, mentre nel secondo modello l'insieme degli item scritti da una transazione è contenuto nell'insieme degli item letti dalla stessa transazione. Un modello più realistico dovrebbe prevedere che non ci sia necessariamente una relazione di contenimento tra l'insieme degli item letti (*read set*) e quello degli item scritti (*write set*) da una transazione. Pertanto considereremo ora un modello in cui, analogamente al modello precedente, una transazione può effettuare operazioni di *rlock* e *wlock*, ma in cui:
a) non si assume che quando una transazione scrive un item debba averlo letto
b) non si assume che il valore di un item letto da una transazione sia significativo indipendentemente dal fatto che abbia influenza sul valore finale di qualche item prodotto dalla transazione.
In conseguenza di b), dovremmo considerare equivalenti due schedule se producono lo stesso valore per ogni item su cui effettuano operazioni di scrittura. Una definizione di serializzabilità basata su tale nozione di equivalenza non è però soddisfacente, come mostrato dall'esempio seguente. Consideriamo lo schedule

T_1	T_2	T_3
$wlock(A)$		
$unlock(A)$		
	$wlock(C)$	
	$unlock(C)$	
	$rlock(A)$	
	$wlock(B)$	
	$unlock(A)$	
	$unlock(B)$	
$rlock(C)$		
$wlock(D)$		
$unlock(C)$		
$unlock(D)$		
		$wlock(B)$
		$wlock(D)$
		$unlock(B)$
		$unlock(D)$

In base alla definizione di serializzabilità (detta *view serializability*) data sopra, tale schedule è serializzabile; infatti i valori finali prodotti dallo schedule per

gli item A, B, C e D sono i valori per tali item calcolati rispettivamente dalle transazioni T_1, T_3, T_2 e T_3 e sono gli stessi valori prodotti dallo schedule seriale T_1, T_2, T_3. Tuttavia, se la transazione T_3 non viene eseguita (ad esempio per la caduta del sistema) i valori finali prodotti per gli item B e D non coincidono né con i valori prodotti dallo schedule seriale T_1, T_2, né con i valori prodotti dallo schedule seriale T_2, T_1. Inoltre è stato dimostrato che se si adotta tale definizione di serializzabilità, il problema di decidere se uno schedule è serializzabile risulta essere NP-completo.

Prenderemo pertanto in esame un'altra definizione di serializzabilità, più restrittiva della precedente, detta *conflict serializabilty*, basata su alcuni vincoli di precedenza imposti dalla semantica delle transazioni; in base a tale definizione lo schedule appena visto risulta non serializzabile.

Nel modello esaminato precedentemente se in uno schedule S una transazione T_1 effettua un $wlock(X)$ e T_2 è un'altra transazione che effettua il successivo $wlock(X)$ allora T_2 legge il valore di X scritto da T_1 e in base a tale valore calcola il nuovo valore di X; pertanto in qualsiasi schedule seriale T_1 equivalente ad S deve precedere T_2. Nel modello attualmente in esame, se T_2 non legge X non c'è alcun motivo per cui debba seguire T_1 in uno schedule seriale equivalente ad S; in altre parole due scritture successive di uno stesso item da parte di due distinte transazioni non impongono nessun vincolo di precedenza. Vediamo quindi quali sono i vincoli che vanno effettivamente imposti per uno schedule seriale equivalente ad uno schedule S:
1. se in S una transazione T_2 legge il valore di un item X scritto da una transazione T_1 allora T_1 deve precedere T_2 e,
2. se T_3 è una terza transazione che scrive X allora T_3 deve precedere T_1 o seguire T_2.
A tali vincoli devono essere aggiunti i seguenti:
3. se una transazione legge il valore iniziale di un item X allora deve precedere qualsiasi transazione che scriva X
4. se una transazione scrive il valore finale di un item X allora deve seguire qualsiasi transazione che scriva X.
Se postuliamo l'esistenza all'inizio dello schedule di una transazione *iniziale* T_0 che scrive i valori iniziali di tutti gli item (senza leggerne nessuno) e alla fine dello schedule di una transazione *finale* T_f che legge i valori finali di tutti gli item (senza scriverne alcuno), gli ultimi due vincoli sono riassorbiti dal vincolo 1. Nel seguito dato uno schedule S indichiamo con $\bar{S}$ lo schedule ottenuto da S aggiungendo all'inizio la transazione iniziale e alla fine quella finale. Diciamo allora che uno schedule S è serializzabile se esiste uno schedule seriale che rispetta tutti i vincoli di tipo 1 e 2 generati da S.

Analogamente a quanto visto per i modelli precedenti anche in questo caso possiamo rappresentare i vincoli di precedenza imposti dalla semantica delle transazioni mediante gli archi in un particolare tipo di grafo diretto. In particolare il vincolo 1 può essere rappresentato da un arco diretto $T_1 \rightarrow T_2$ e il vincolo 2 dalla coppia di archi alternativi $T_3 \rightarrow T_1$ e $T_2 \rightarrow T_3$. Una collezione di nodi, archi e coppie di archi alternativi viene detta *poligrafo*. Un poligrafo è *aciclico* se almeno uno dei grafi che possono essere ottenuti prendendo un arco da ogni coppia di grafi alternativi è aciclico. Analogamente a quanto visto per i modelli precedenti, anche in questo caso la serializzabilità di uno schedule può

essere testata da un algoritmo che costruisce un poligrafo e quindi verifica se è aciclico; inoltre, se è aciclico, fornisce (applicando l'ordinamento topologico ad un grafo aciclico ottenuto dal poligrafo nel modo detto sopra) uno schedule seriale che rispetta i vincoli di precedenza tra transazioni imposti dallo schedule dato. Prima di esaminare l'algoritmo osserviamo che poiché in questo modello non si assume che il valore di un item letto da una transazione sia significativo indipendentemente dal fatto che abbia influenza sul valore finale di qualche item prodotto dalla transazione, una transazione che non ha alcun effetto sui valori finali prodotti da uno schedule è considerata *inutile* in quello schedule. Come vedremo le transazioni inutili vengono eliminate dall'algoritmo in quanto sono irrilevanti nel determinare la serializzabilità di uno schedule (in altre parole una transazione inutile in uno schedule S può leggere in S e in uno schedule seriale S', che rispetta i vincoli di precedenza tra transazioni imposti da S, valori diversi).

Dato uno schedule S i nodi del poligrafo P sono le transazioni di $\bar{S}$. La costruzione degli archi di P avviene attraverso i seguenti passi:

1. vengono creati gli archi in accordo al vincolo 1; quindi se una transazione T_2 legge il valore di un item X scritto da una transazione T_1 viene aggiunto l'arco diretto $T_1 \rightarrow T_2$

2. vengono eliminati tutti gli archi entranti in transazioni inutili (una transazione inutile T può essere individuata facilmente perché non c'è nessun cammino in P da T a T_f)

3. vengono creati gli archi in accordo al vincolo 2; quindi per ogni arco $T_1 \rightarrow T_2$ se T_3 è transazione distinta da quella iniziale che scrive un item che ha imposto l'esistenza dell'arco $T_1 \rightarrow T_2$, si aggiunge:

   - l'arco $T_2 \rightarrow T_3$ se $T_1 = T_0$ e $T_2 \neq T_f$

   - l'arco $T_3 \rightarrow T_1$ se $T_2 = T_f$ e $T_1 \neq T_0$

   - la coppia di archi alternativi $T_3 \rightarrow T_1$ e $T_2 \rightarrow T_3$ se $T_1 \neq T_0$ e $T_2 \neq T_f$.

Si consideri il seguente schedule

T_1	T_2	T_3	T_4
	$rlock(A)$		
$rlock(A)$			
$wlock(C)$			
$unlock(C)$			
		$rlock(C)$	
$wlock(B)$			
$unlock(B)$			
			$rlock(B)$
$unlock(A)$			
	$unlock(A)$		
		$wlock(A)$	
			$rlock(C)$
	$wlock(D)$		
			$unlock(B)$
		$unlock(C)$	
	$rlock(B)$		
		$unlock(A)$	
			$wlock(A)$
	$unlock(B)$		
			$wlock(B)$
			$unlock(B)$
	$unlock(D)$		
			$unlock(C)$
			$unlock(A)$

In Fig 7.2(a), (b) e (c) è mostrato il poligrafo che viene costruito ai passi 1, 2 e 3 dell'algoritmo.

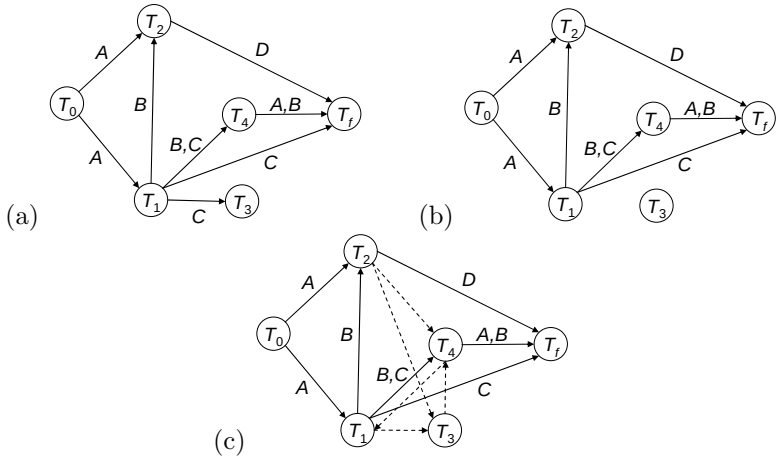


Figura 7.2: Poligrafi

Poiché il poligrafo risultante è aciclico lo schedule è serializzabile e lo schedule seriale (che rispetta i vincoli di precedenza tra transazioni imposti da esso) fornito dall'algoritmo è T_1, T_2, T_3, T_4. Viceversa il poligrafo mostrato nella

Fig. 7.3, è costruito dall'algoritmo per lo schedule visto in precedenza; poiché contiene un ciclo, tale schedule non è serializzabile.

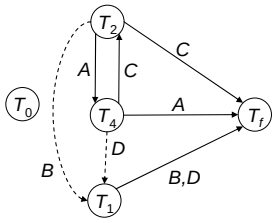


Figura 7.3: Poligrafo

7.3 Deadlock e livelock

In un sistema che utilizza i lock per il controllo della concorrenza si possono verificare delle situazioni anomale note con il nome di deadlock e livelock.

Un *livelock* si verifica quando una transazione aspetta indefinitamente che gli venga garantito un lock su un certo item. Ad esempio, consideriamo il caso di una transazione T in attesa di effettuare un lock su un item X su cui un'altra transazione T_1 mantiene un lock; se quando T_1 rilascia X un'altra transazione T_2 ottiene un lock su X, T deve rimanere in attesa; poiché questa situazione può ripetersi un numero indefinito di volte, T può rimanere indefinitamente in attesa di ottenere un lock su X.

Un *deadlock* si verifica quando ogni transazione in un insieme $\mathcal{T}$ è in attesa di ottenere un lock su un item sul quale qualche altra transazione in $\mathcal{T}$ mantiene un lock.

Entrambi i problemi si possono presentare in un qualsiasi sistema in cui i processi possono essere eseguiti concorrentemente e quindi sono stati largamente studiati nell'ambito dei sistemi operativi. Il primo problema può essere risolto con una strategia *first came- first served*; in altre parole il sistema mantiene memoria delle successive richieste di lock in modo che, quando un item X viene rilasciato, viene garantito un lock su X alla transazione che per prima ha richiesto un lock su X. Un'altra possibile strategia può essere quella di eseguire le transazioni in base alle loro priorità e di aumentare la priorità di una transazione all'aumentare del tempo in cui rimane in attesa in modo che possa ad un certo punto assumere la priorità più alta ed essere eseguita. Per risolvere il secondo problema possono essere seguiti due approcci. Un approccio è preventivo in quanto cerca di evitare il verificarsi di situazioni di stallo adottando opportuni protocolli; l'altro invece si preoccupa di risolvere le situazioni di stallo quando si verificano. Il sussistere di una situazione di stallo può essere rilevato mantenendo un *grafo di attesa*, cioè un grafo i cui nodi sono le transazioni e in cui c'è un arco $T_1 \rightarrow T_2$ se la transazione T_1 è in attesa di ottenere un lock su un item sul quale T_2 mantiene un lock; se in tale grafo c'è un ciclo si sta verificando una situazione di stallo che coinvolge le transazioni nel ciclo; per risolverla occorre che almeno una transazione nel ciclo sia *rolled-back* (cioè la transazione viene abortita, i suoi effetti sulla base di dati vengono annullati ripristinando i valori

dei dati precedenti l'inizio della sua esecuzione e, infine, tutti i lock mantenuti dalla transazione vengono rilasciati) e quindi venga fatta ripartire.

7.4 Protocollo di locking a due fasi stretto

Oltre al verificarsi di un deadlock, ci sono altri motivi per cui una transazione deve essere abortita, ad esempio: perché ha cercato di effettuare un accesso non autorizzato oppure perché ha cercato di eseguire una divisione per 0. Il punto in cui una transazione non può più essere abortita per uno dei suddetti motivi, cioè il punto in cui ha ottenuto tutti i lock che gli sono necessari e ha effettuato tutti i calcoli nell'area di lavoro, viene detto *punto di commit* della transazione; una transazione che ha raggiunto il suo punto di commit viene detta *committed* altrimenti viene detta *attiva*. Quando una transazione raggiunge il suo punto di commit effettua un'*operazione di commit* (nei sistemi reali tale operazione prevede lo svolgimento di diverse azioni, ma per i nostri scopi serve solo a marcare il punto di commit della transazione). I dati scritti da una transazione sulla base di dati prima che abbia raggiunto il punto di commit vengono detti *dati sporchi*. Il fatto che un dato sporco possa essere letto da qualche altra transazione può causare un effetto di *roll-back a cascata*. Consideriamo il seguente schedule

T_1	T_2
$wlock(X)$	
$read(X)$	
$X := X - N$	
$write(X)$	
$unlock(X)$	
	$rlock(X)$
	$wlock(Z)$
	$read(X)$
	$read(Z)$
	$Z := Z + X$
	$write(Z)$
	$commit$
	$unlock(X)$
	$unlock(Z)$
$wlock(Y)$	
$read(Y)$	
$Y := Y + N$	
$write(Y)$	
$commit$	
$unlock(Y)$	

Se la transazione T_1 viene abortita dopo che ha letto Y, il valore di X scritto da T_1 è un dato sporco; infatti, quando T_1 viene abortita è necessario annullare gli effetti di T_1 sulla base di dati ripristinando il vecchio valore di X (quello letto da T_1). Ma allora è necessario annullare anche gli effetti di T_2 sulla base di dati in quanto il valore di Z prodotto da T_2 è calcolato a partire dal valore di X scritto T_1. Pertanto sia T_1 che T_2 devono essere rolled-back.

Per evitare questo fenomeno detto *rollback a cascata* occorre impedire alle transazioni di leggere dati sporchi. Ciò può essere ottenuto adottando un protocollo di locking a due fasi stretto.
Una transazione soddisfa il *protocollo a due fasi stretto* se:

1. non scrive sulla base di dati fino a quando non ha raggiunto il suo punto di commit

2. non rilascia un lock finchè non ha finito di scrivere sulla base di dati

La condizione 1 garantisce che se una transazione è abortita allora non ha modificato nessun item nella base di dati; la condizione 2 garantisce che quando una transazione legge un item scritto da un'altra transazione quest'ultima non può essere abortita (quindi nessuna transazione legge dati sporchi).

Perché la transazione T_1 nell'esempio precedente soddisfi il protocollo di locking a due fasi stretto deve essere modificata nel modo seguente.

T_1
$wlock(X)$
$read(X)$
$X := X - N$
$wlock(Y)$
$read(Y)$
$Y := Y + N$
$commit$
$write(X)$
$write(Y)$
$unlock(X)$
$unlock(Y)$

I protocolli di locking a due fasi stretti possono essere classificati in *protocolli conservativi* e *protocolli aggressivi*; i primi cercano di evitare il verificarsi di situazioni di stallo, i secondi cercano di processare le transazioni il più rapidamente possibile anche se ciò può portare a situazioni di stallo e, quindi, alla necessità di abortire qualche transazione.

La versione più conservativa di un protocollo di locking a due fasi stretto è quella che impone ad una transazione di richiedere all'inizio tutti gli item di cui può avere bisogno. Lo scheduler permette alla transazione di procedere solo se tutti i lock che ha richiesto sono disponibili, altrimenti la mette in una coda di attesa. In tal modo non possono verificarsi deadlock, ma possono ancora verificarsi livelock. Per evitare anche il verificarsi di livelock si può impedire ad una transazione T di procedere (anche se tutti i lock da essa richiesti sono disponibili) se c'è un'altra transazione che precede T nella coda che è in attesa di ottenere un lock richiesto da T. È facile vedere che se si adotta un protocollo in cui tutti i lock sono richiesti all'inizio e uno scheduler che garantisce ad una transazione T tutti i lock richiesti se e solo se:

1. tutti i lock sono disponibili

2. nessuna transazione che precede T nella coda è in attesa di un lock richiesto da T

allora non si possono verificare né dealdock né livelock. Infatti per la condizione 1 nessuna transazione che mantiene un lock può essere in attesa di un lock mantenuto da un'altra transazione; inoltre, la condizione 2 garantisce che dopo un tempo finito ogni transazione raggiunge la testa della coda e quindi viene eseguita.
Il protocollo conservativo esaminato presenta due inconvenienti. Infatti l'esecuzione di una transazione può essere ritardata dal fatto che non può ottenere un lock anche se molti passi della transazione potrebbero essere eseguiti senza quel lock; inoltre una transazione è costretta a richiedere un lock su ogni item che potrebbe essergli necessario anche se poi di fatto non l'utlizza. Ad esempio, se una transazione deve effettuare una ricerca su un file con indice sparso e gli item sono i blocchi, la transazione deve effettuare un lock su ogni blocco del file principale e del file indice anche se poi accederà soltanto ad alcuni blocchi del file indice e ad un solo blocco del file principale.

La versione più aggressiva di un protocollo di locking a due fasi stretto è quella che impone ad una transazione di richiedere un lock su un item immediatamente prima di leggerlo o scriverlo. Se le transazioni soddisfano tale protocollo possono verificarsi situazioni di stallo; un modo per evitare ciò è quello di definire un ordinamento sugli item e di imporre alle transazioni di richiedere i lock in accordo a tale ordinamento. In tal modo non possono verificarsi deadlock. Infatti, sia $\mathcal{T}$ un insieme di transazioni (che richiedono i lock in accordo all'ordinamento fissato sull'insieme degli item) tale che ogni transazione T_k è in attesa di un item A_k e mantiene un lock su almeno un item (si osservi che se una transazione T in $\mathcal{T}$ non soddisfa la seconda condizione allora l'insieme $\mathcal{T} - T$ è ancora un insieme di transazioni in stallo) e sia A_i il primo item nell'ordinamento fissato sull'insieme degli item; allora la transazione T_i che è in attesa di A_i non può mantenere un lock su alcun item (contraddizione). Tuttavia il metodo di ordinare gli item non è molto praticabile perché non sempre una transazione può scegliere l'ordine in cui richiedere i lock.

Un protocollo aggressivo è adatto a situazioni in cui la probabilità che due transazioni richiedano un lock su uno stesso item è bassa (e quindi è bassa la probabilità che si verifichi una situazione di stallo), in quanto evita al sistema il sovraccarico dovuto alla gestione dei lock (decidere se garantire un lock su un dato item ad una data transazione, gestire la tavola dei lock, mettere le transazioni in una coda o prelevarle da essa). Un protocollo conservativo è invece adatto a situazioni opposte in quanto evita al sistema il sovraccarico dovuto alla gestione dei deadlock (rilevare e risolvere situazioni di stallo, eseguire parzialmente transazioni che poi vengono abortite, rilascio dei lock mantenuti da transazioni abortite).

7.5 Controllo della concorrenza basato sui timestamp

I metodi per il controllo della concorrenza basati sui timestamp ordinano le transazioni in base ai loro timestamp. Il timestamp è assegnato ad una transazione dallo scheduler quando la transazione ha inizio. A tale scopo lo scheduler può o gestire un contatore (ogni volta che ha inizio una transazione tale contatore viene incrementato e il suo valore è il timestamp della transazione) oppure usare l'orologio interno della macchina (in tal caso il timestamp è l'ora di inizio della transazione).

In tale approccio al controllo della concorrenza, uno schedule è serializzabile se è equivalente allo schedule seriale in cui le transazioni compaiono ordinate in base al loro timestamp.

Dato uno schedule occorre verificare che, per ciascun item acceduto da più di una transazione, l'ordine con cui le transazioni accedono all'item non viola la serializzabilità dello schedule. A tale scopo vengono associati a ciascun item X due timestamp:
- il *read timestamp* di X, denotato con *read_TS(X)*, che è il più grande fra tutti i timestamp di transazioni che hanno letto con successo X;
- il *write timestamp* di X, denotato con *write_TS(X)*, che è il più grande fra tutti i timestamp di transazioni che hanno scritto con successo X.

Ogni volta che una transazione T cerca di eseguire un *read(X)* o un *write(X)*, occorre confrontare il timestamp $TS(T)$ di T con il read timestamp e il write timestamp di X per assicurarsi che l'ordine basato sui timestamp non è violato. L'algoritmo per il controllo della concorrenza opera nel modo seguente:

1. ogni volta che una transazione T cerca di eseguire l'operazione $write(X)$:
   (a) se $read\_TS(X) > TS(T)$, T viene rolled back; infatti in tal caso qualche transazione con un timestamp maggiore di $TS(T)$ (cioè una transazione che segue T nell'ordinamento basato sui timestamp) ha già letto il valore di X prima che T abbia potuto scriverlo, violando in tal modo l'ordinamento basato sui timestamp;
   (b) se $write\_TS(X) > TS(T)$, l'operazione di scrittura non viene effettuata; infatti in tal caso qualche transazione T' con un timestamp maggiore di $TS(T)$ (cioè una transazione che segue T nell'ordinamento basato sui timestamp) ha già scritto il valore di X e quindi l'esecuzione dell'operazione di scrittura provocherebbe la perdita di tale valore; si noti che nessuna transazione T'' con $TS(T') > TS(T'') > TS(T)$, può aver letto X altrimenti al passo precedente T sarebbe stata rolled back;
   (c) se nessuna delle condizioni precedenti è soddisfatta allora l'operazione di scrittura è eseguita e $TS(T)$ diventa il nuovo valore di $write\_TS(X)$
2. ogni volta che una transazione T cerca di eseguire l'operazione $read(X)$:
   (a) se $write\_TS(X) > TS(T)$, T viene rolled back; infatti in tal caso qualche transazione con un timestamp maggiore di $TS(T)$ (cioè una transazione che segue T nell'ordinamento basato sui timestamp) ha

già scritto il valore di X prima che T abbia potuto leggerlo, violando in tal modo l'ordinamento basato sui timestamp;

(b) se $write\_TS(X) \leq TS(T)$, allora l'operazione di lettura è eseguita e se $read\_TS(X) < TS(T)$, $TS(T)$ diventa il nuovo valore di $read\_TS(X)$.

Consideriamo ad esempio le due transazioni seguenti

T_1
$read(X)$
$X := X + 10$
$write(X)$

T_2
$read(X)$
$X := X + 5$
$write(X)$

con i seguenti timestamp: $TS(T_1) = 110$, $TS(T_2) = 100$. Consideriamo il seguente schedule di $\{T_1, T_2\}$

T_1	T_2
	$read(X)$
$read(X)$	
	$X := X + 5$
$X := X + 10$	
$write(X)$	
	$(*)write(X)$

Quando T_2 cerca di seguire (*) viene abortita. Consideriamo ora le due transazioni seguenti

T_1
$read(Y)$
$X := Y + 10$
$write(X)$

T_2
$read(Y)$
$X := Y + 5$
$write(X)$

con i seguenti timestamp: $TS(T_1) = 110$, $TS(T_2) = 100$. Consideriamo il seguente schedule di $\{T_1, T_2\}$

T_1	T_2
	$read(Y)$
$read(Y)$	
	$X := Y + 5$
$X := Y + 10$	
$write(X)$	
	$(*)write(X)$

In questo caso l'operazione (*) non viene eseguita.

Note bibliografiche

I manuali che trattano in maniera approfondita ed esauriente gli aspetti legati alle basi di dati, dalla teoria ai modelli, dai linguaggi all'organizzazione fisica, da consultare per ulteriori approfondimenti sono [ACPT06] di P. Atzeni, S. Ceri, S. Paraboschi e R. Torlone, [AB07] di R. Elmasri e S. Navathe e, sebbene più datato, [Ull89] di J. Ullman.

Il modello Entity-Relationship del Capitolo 1, è stato introdotto da P.Chen [Che76]. Il modello relazionale è invece stato proposto da Codd [Cod70]. Molti risultati teorici relativi al modello relazionale e alle forme normali possono essere trovati in D. Maier [Mai83] ed J. Ullman [Ull89]. Il linguaggio SQL è stato proposto in [CB74] da D. D. Chamberlin e R. F. Boyce. La progettazione concettuale è trattata in P. Atzeni, S. Ceri, S. Paraboschi e R. Torlone in [ACPT06], R. Elmasri e S. Navathe [AB07] ed J. Ullman [Ull89]. Il DBMS PostgreSQl é stato sviluppato da M.Stonebraker [SRH90] ed è disponibile per un download gratuito in http://www.postgresql.org. L'organizzazione fisica dei dati è trattata ad esempio in in [AB07] di R. Elmasri e S. Navathe ed in [Ull89] di J. Ullman. Un articolo di D. Comer sui B-tree è [Com79]. Il controllo della concorrenza è discusso anche da J. Ullman [Ull89].

Bibliografia

[AB07] Elmasri Ramez A. and Navathe Shamkant B. *Sistemi di basi di dati. Fondamenti*. Pearson Education Italia, 2007.

[ACPT06] Paolo Atzeni, Stefano Ceri, Stefano Paraboschi, and Riccardo Torlone. *Basi di dati Modelli e linguaggi di interrogazione 2/ed*. Mcgraw-hill, 2006.

[CB74] Donald D. Chamberlin and Raymond F. Boyce. Sequel: A structured english query language. In *FIDET '74: Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control*, pages 249–264, New York, NY, USA, 1974. ACM.

[Che76] Peter P. Chen. The entity-relationship model - toward a unified view of data. *ACM Trans. Database Syst.*, 1(1):9–36, 1976.

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970.

[Com79] Douglas Comer. The ubiquitous b-tree. *ACM Comput. Surv.*, 11(2):121–137, 1979.

[Mai83] David Maier. *The Theory of Relational Databases*. Computer Science Press, 1983.

[SRH90] Michael Stonebraker, Lawrence A. Rowe, and Michael Hirohama. The implementation of postgres. In *IEEE Transactions on Knowledge and Data Engineering*, pages 340–355, 1990.

[Ull89] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base System*. Computer Science Press, 1989.