Capitolo 7

Memorizzazione dei dati ed elaborazione delle interrogazioni

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

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

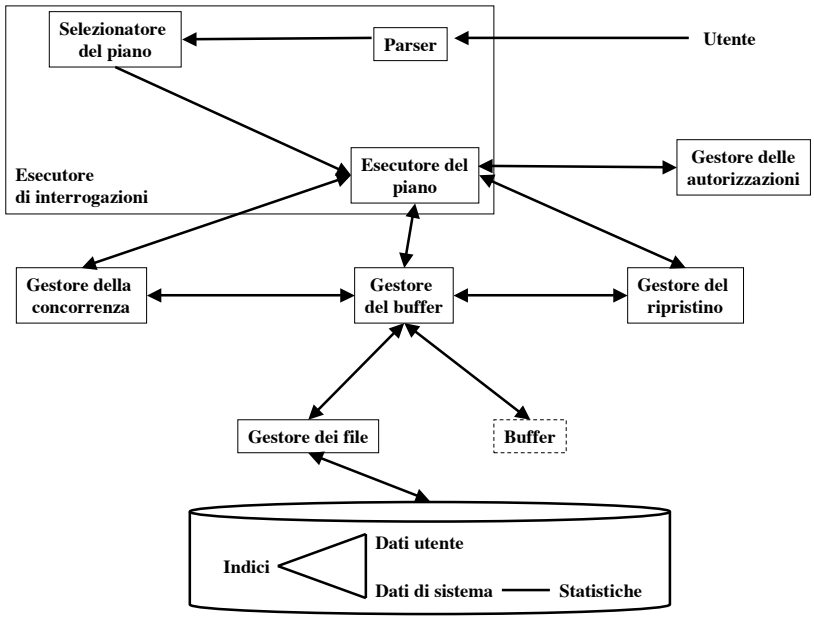


Figura 7.1: Architettura di un DBMS

7.1 Architettura di un DBMS

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

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

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

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

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

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

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

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

Figura 7.2: Catalogo relativo allo schema della videoteca

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

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

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

7.2 Memorizzazione dei dati e gestione del buffer

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

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

7.2.1 File, record e blocchi

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

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

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

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

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

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

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

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

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

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

7.2.2 Organizzazione in cluster

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

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

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

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

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

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

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

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

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

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

```
SELECT * FROM Cliente;
```

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

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

7.2.3 Gestione del buffer

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

Esempio 7.3 Consideriamo l'operazione di join:

$$\texttt{Noleggio} \bowtie \texttt{Cliente}$$

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

7.3 Strutture ausiliarie di accesso

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

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

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

[2] Ricordiamo nuovamente che questo concetto è diverso dal concetto di *chiave primaria* introdotto relativamente all'identificazione delle tuple nel Capitolo 2.

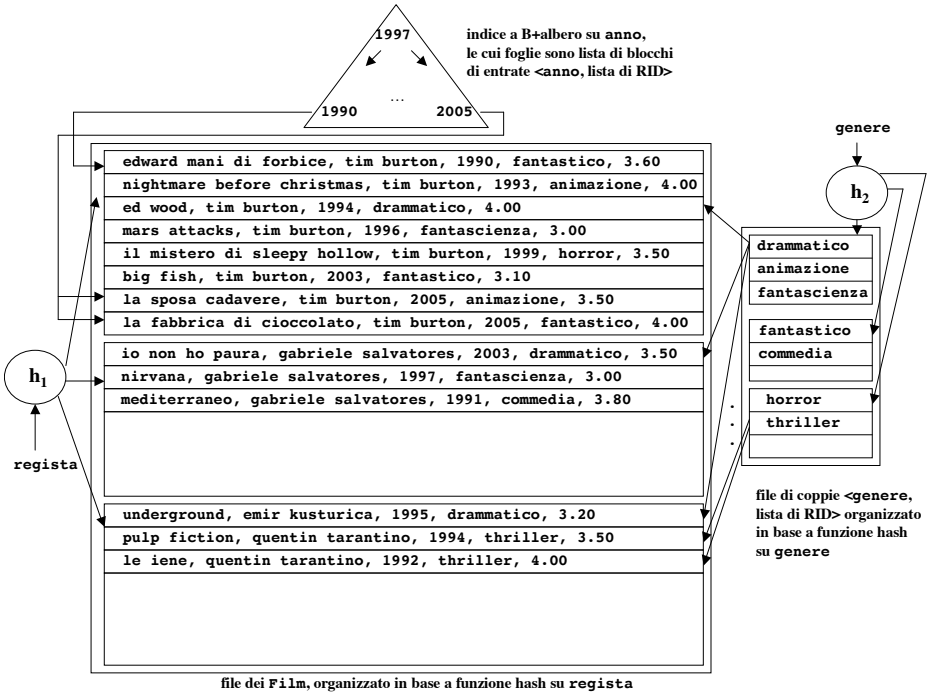


Figura 7.4: Organizzazioni primarie e secondarie

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

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

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

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

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

Tabella 7.1: Classificazione dei vari tipi di indice

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

7.3.1 Tipologie di indici

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

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

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

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

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

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

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

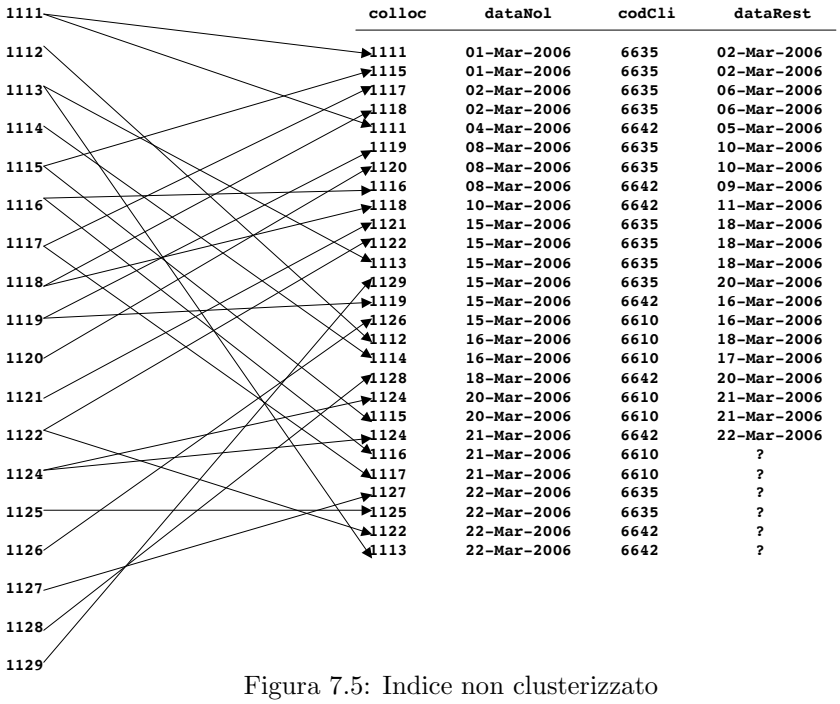
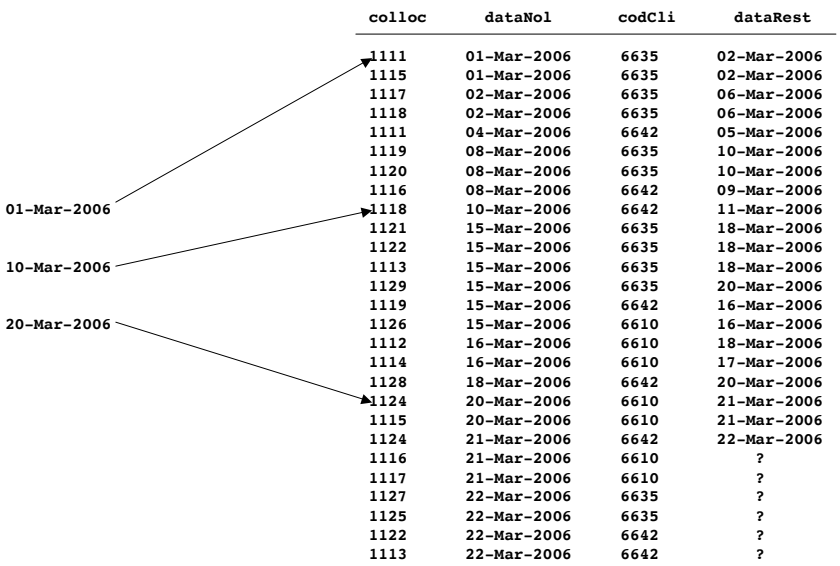


Figura 7.5: Indice non clusterizzato

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

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



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

Figura 7.6: Indice sparso (clusterizzato)

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

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

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

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

7.3.2 Indici ad albero

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

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

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

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

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

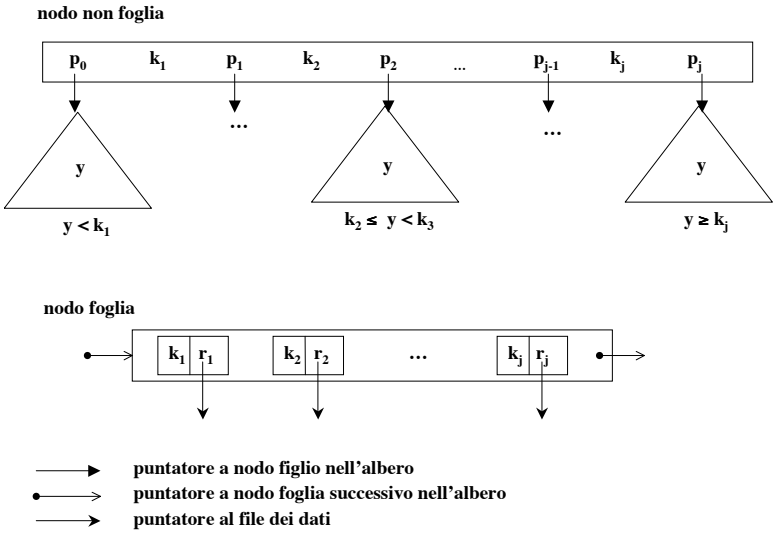


Figura 7.7: Formato di un nodo di un B^+-albero

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

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

[5] È possibile anche adottare la convenzione che i valori uguali siano contenuti nel sotto-albero sinistro.

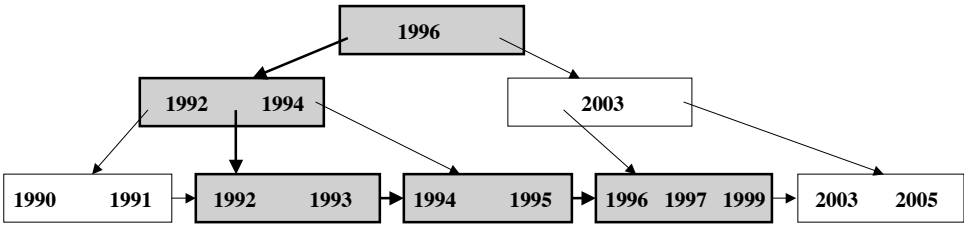


Figura 7.8: B^+-albero sull'attributo `anno` della relazione `Film`

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

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

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

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

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

7.3.3 Indici hash

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

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

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

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

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

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

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

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

$\texttt{SELECT A}_1\texttt{,A}_2\texttt{,...,A}_n\texttt{ FROM R WHERE A}_i\texttt{=C;}$

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

Le strutture ad albero sono invece preferibili se le interrogazioni che coinvolgono la chiave $\texttt{A}_i$ usano condizioni di intervallo come:

$\texttt{SELECT A}_1\texttt{,A}_2\texttt{,..., A}_n\texttt{ FROM R WHERE A}_i\texttt{ BETWEEN C}_1\texttt{ AND C}_2\texttt{;}$

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

7.3.4 Definizione di cluster ed indici in SQL

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

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

7.3.4.1 Definizione di cluster

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

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

dove:

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

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

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

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

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

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

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

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

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

7.3.4.2 Definizione di indici

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

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

dove:

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

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

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

CREATE INDEX idxNol ON CLUSTER NolCli;
```

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

7.3.4.3 Inserimento di relazioni in un cluster

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

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

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

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

□

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