Capitolo 9

REALIZZAZIONE DEI DBMS

Si presentano l'architettura dei DBMS relazionali centralizzati e alcune delle tecniche utilizzate per realizzarne le funzionalità essenziali: la gestione dei dati, delle interrogazioni, della concorrenza e dell'affidabilità. Una conoscenza, sia pure elementare, di tali tecniche è indispensabile per utilizzare questi sistemi in maniera efficace.

9.1 Architettura dei sistemi relazionali

In Figura 9.1 sono mostrati i moduli principali di una possibile architettura di un sistema relazionale centralizzato.

Un DBMS è organizzato su due livelli, che chiameremo la *macchina logica* e la *macchina fisica*. La macchina logica comprende i moduli che trattano i comandi del linguaggio SQL e stabiliscono come eseguirli usando gli operatori forniti dalla macchina fisica, che gestisce la memoria permanente e le strutture per la memorizzazione e recupero dei dati.

Nei sistemi commerciali le funzionalità di questi moduli non sono nettamente separate come la figura potrebbe far pensare, ma questa schematizzazione consente di comprendere meglio gli scopi di ognuno.

Nelle prossime sezioni si esaminano brevemente i moduli dedicati alla gestione dei dati, delle interrogazioni, della concorrenza e dell'affidabilità. Di ogni modulo viene descritto il livello di astrazione fornito e le funzionalità che rendono disponibili agli altri moduli. Per approfondire gli argomenti si veda [Alb01].

9.2 Gestore della memoria permanente

Il *gestore della memoria permanente* offre una visione di una basi di dati come un insieme di file di blocchi di caratteri (*pagine fisiche*) di grandezza prefissata, compresa generalmente fra 1 e 8 Kbyte, che sono l'unità minima di trasferimento fra la memoria permanente e quella temporanea. Esso consente agli altri livelli di usare la memoria permanente astraendo dalle diverse modalità di gestione dei file dei sistemi operativi.

I dati memorizzati nella memoria permanente sono quelli descritti nello schema logico, le strutture ausiliarie per agevolare gli accessi alla base di dati (indici), e i dati di servizio necessari per il funzionamento del sistema.

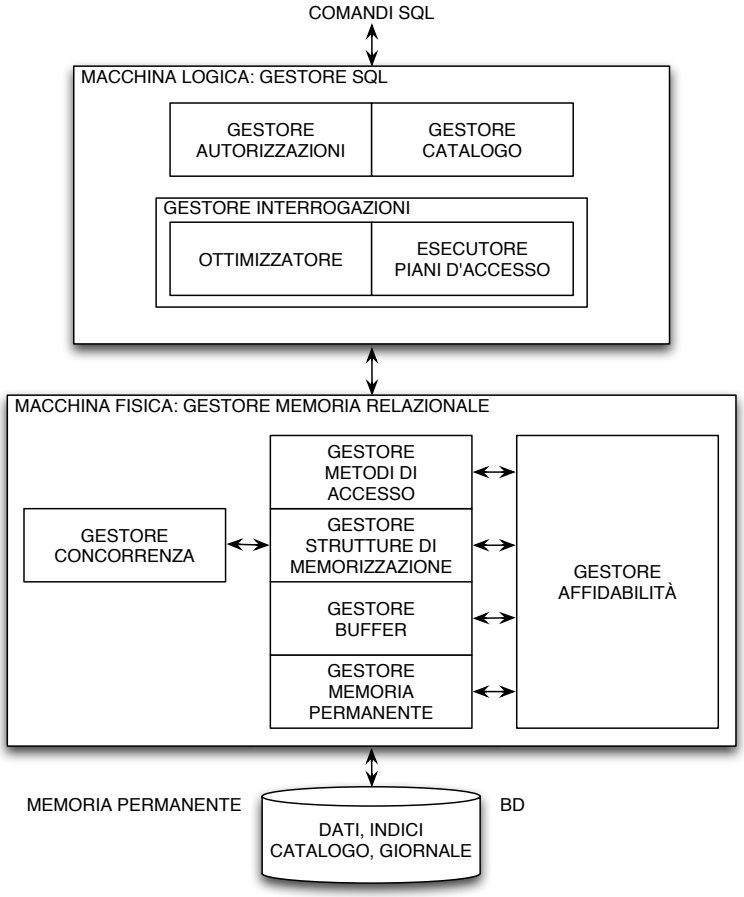


Figura 9.1: Architettura di un DBMS relazionale

9.3 Gestore del buffer

Il *gestore del buffer* gestisce uno spazio della memoria temporanea destinato a contenere un insieme di pagine fisiche trasferite dalla memoria permanente. Una pagina fisica è rappresentato in memoria temporanea come una struttura logica detta *pagina* e gli altri livelli del sistema hanno una visione della memoria permanente come un insieme di pagine utilizzabili in memoria temporanea astraendo da quando esse vengano trasferite fra i due tipi di memoria.

Poiché il buffer può contenere molte pagine, quando si opera su dati usati di recente esiste una certa probabilità che tali dati siano ancora disponibili nel buffer, evitandone così la rilettura dal disco. Similmente, gli aggiornamenti ai dati vengono in realtà effettuati all'interno del buffer, e i dati sono riportati sul disco solo quando è necessario liberare il buffer o quando il protocollo per la gestione dell'affidabilità lo richiede (si veda la Sezione 9.8).

I record all'interno delle pagine sono memorizzati come stringhe di caratteri con un prefisso contenente informazioni di servizio seguito dai valori dei campi. Il prefisso può contenere, ad esempio, una marca per la cancellazione logica, la lunghezza del record, il numero degli attributi, l'identificatore interno del record, unico all'interno

della base di dati e assegnato automaticamente dal sistema ad ogni nuovo record. Un record con una dimensione inferiore a quella di una pagina si memorizza tutto in una pagina.

Quando un record viene inserito in una relazione, il sistema gli assegna un identificatore, chiamato TID (*Tuple Identifier*), o RID (*Row Identifier*), che diventa il riferimento da usare nelle strutture dati. L'esatta natura del TID può variare da un sistema ad un altro, ma l'obiettivo comune a tutti è di garantire che un TID sia un'informazione che non cambia fintantoché il record esista nella base di dati. La soluzione più comune è la seguente: un TID è una coppia (P, j), dove P è il numero di pagina e j è la posizione relativa in un vettore memorizzato nella pagina, contenente il riferimento all'inizio del record (Figura 9.2). Se il record si sposta nella pagina, ad esempio in seguito a modifiche che ne cambiano la lunghezza, basta aggiornare il contenuto del vettore, senza cambiare il TID. Nel caso in cui la modifica del record ne comporti uno spostamento in un'altra pagina, il record viene sostituito con la coppia (P', j'), un altro TID, dove P' è l'indirizzo della nuova pagina e j' è la posizione relativa in P'. Anche in questo caso, il TID originariamente assegnato al record non cambia. Se successivamente, nell'accedere al record si scopre che nella pagina P esiste sufficiente spazio libero per contenerlo, allora questo può essere riportato nella pagina originaria.

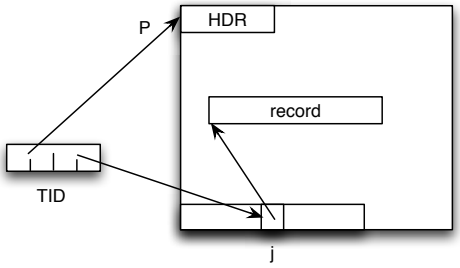


Figura 9.2: Riferimenti ai record

9.4 Gestore delle strutture di memorizzazione

Il *gestore delle strutture di memorizzazione* offre agli altri livelli del sistema una visione dei dati permanenti organizzati in collezioni di record e indici astraendo dalle strutture fisiche usate per la loro memorizzazione in memoria permanente.

Avendo stabilito come si può memorizzare in modo persistente una collezione di record dotati di un TID univoco, resta ancora da stabilire:

- come individuare la pagina in cui inserire un nuovo record quando questo viene aggiunto alla collezione;
- come gestire le situazioni in cui una sequenza di cancellazioni o di inserimenti rendono una pagina troppo vuota o troppo piena;
- quali strutture ausiliarie prevedere per facilitare l'esecuzione delle ricerche.

Un'*organizzazione dei dati* è un insieme di algoritmi per gestire le operazioni su una collezione di record che risponde a queste tre domande. In questa sezione presentiamo brevemente le organizzazioni più importanti.

Organizzazione seriale e sequenziale L'organizzazione *seriale* (*heap*) è il modo più semplice di organizzare i record di una collezione perché essi si memorizzano consecutivamente nell'ordine in cui vengono inseriti. Le pagine possono essere contigue nella memoria permanente oppure no, e in quest'ultimo caso sono collegate a lista. Questa soluzione, per la sua semplicità, è usata da tutti i DBMS quando non ne vengono richieste altre. Le operazioni di inserzioni di nuovi record sono veloci, mentre le ricerche di un record per valore di una chiave o di piccoli sottoinsiemi di record sono lente se i dati sono molti. Per questa ragione si usa per piccole collezioni o quando interessa operare principalmente su grandi sottoinsiemi di record.

Quando i record di una collezione sono memorizzati consecutivamente nell'ordine dei valori di un insieme di attributi $A_1, \ldots, A_n$, si parla di organizzazione *sequenziale* su $A_1, \ldots, A_n$. Questa organizzazione permette di trovare velocemente i record che hanno un valore specificato degli attributi di ordinamento, ma è più complessa di quella seriale se deve garantire l'ordinamento dei dati in presenza di molte inserzioni di nuovi record. Per questa ragione, i DBMS che la prevedono richiedono di usarla per collezioni statiche e dopo aver caricato tutti i dati. Se poi vengono fatte inserzioni si perde l'ordinamento dei dati. Per avere invece i dati ordinati di collezioni dinamiche, i DBMS prevedono un'altra soluzione con i dati memorizzati con una struttura ad albero, descritta più avanti.

Per rendere efficienti le operazioni di ricerca di piccoli sottoinsiemi di collezioni dinamiche di record, in particolare di un record noto il valore di una chiave, si usa una delle seguenti organizzazioni: *procedurale*, *ad albero* o *con indice*.

Organizzazione procedurale. L'organizzazione *procedurale*, o *hash*, prevede l'esistenza di un opportuno algoritmo (*trasformazione della chiave*) che, applicato al valore della chiave, restituisce l'indirizzo della pagina in cui memorizzare, e successivamente cercare, il record; in caso di insuccesso, esiste un criterio per proseguire la ricerca in modo da completare l'operazione con pochi accessi supplementari. Se ben configurata, questa organizzazione permette in genere di ritrovare un record, a partire dal valore della chiave primaria, con un solo accesso alla memoria permanente.

Organizzazione ad albero. L'organizzazione ad *albero* di una collezione C sulla chiave A prevede l'utilizzo di una struttura dati in memoria permanente detta B^+-*albero* che generalizza l'albero di ricerca bilanciato, con le seguenti caratteristiche:

1. Permette di trovare il record con un valore della chiave A, se esiste, con pochi accessi alla memoria permanente.
2. Mantiene la collezione C ordinata sulla chiave A.

Questa organizzazione è la più complessa tra quelle illustrate ed è lievemente meno efficiente della precedente quando si deve rispondere ad un'interrogazione con condizione $A = k$. Tuttavia, la seconda caratteristica sopra elencata la rende molto adatta a rispondere ad interrogazioni con condizioni tipo $A \leq k$, $A \geq k$, o $k_1 \leq A \leq k_2$.

Indici. Un indice su un attributo A di una relazione R è una struttura dati con un ruolo analogo all'indice analitico di un libro, costituito da un insieme di pagine che contengono ciascuna un insieme di righe di testo. Un indice analitico è un insieme ordinato di termini con associato il numero della pagina dove vengono usati, di solito dove il termine viene introdotto per la prima volta. Per trovare la pagina dove appare un termine, invece di sfogliarle tutte una dopo l'altra, si consulta l'indice analitico e si passa alla pagina segnalata.

L'organizzazione dei dati con indice è simile all'organizzazione di un libro: i record nei DBMS sono memorizzati in pagine di un file, come le righe di un libro in pagine di testo, ma l'indice nei DBMS consente di trovare rapidamente i record di una pagina in base al valore di un attributo, mentre l'indice analitico di un libro non fornisce l'informazione sulla riga del testo che contiene il termine, ma solo l'informazione sulla pagina dove occorre. Più precisamente valgono le seguenti considerazioni.

Un indice su un attributo A di una relazione R, dal punto di vista logico, è una relazione con due attributi (A, TID), con gli elementi ordinati su A e valori (k_i, r_j), dove k_i è un valore di A presente in un record di R, ed r_j è un riferimento (TID) al record di R in cui A vale k_i. Se A non è una chiave, nell'indice sono presenti tanti record (k_i, r_j) con lo stesso valore k_i di A quanti sono i record di R in cui A vale k_i.

Di solito un indice è organizzato a B^+- albero per permettere di trovare con pochi accessi, a partire da un valore v, i record di R in cui il valore di A è in una relazione specificata con v. Se interessano solo ricerche per chiave del tipo $A = k$ l'indice può anche essere organizzato in modo procedurale.

In Figura 9.3 sono mostrati due esempi di indici su due attributi della relazione R, la chiave K e A, supponendo per semplicità che (a) i record di R siano di lunghezza uguale e memorizzati con l'organizzazione seriale e (b) i TID siano interi che rappresentano la posizione dei record nella relazione, traducibili in modo ovvio nella coppia (numero della pagina, posizione nella pagina), nota la capacità della pagina.

R

TID	**K**	**A**	**...**
1	k5	d	...
2	k3	b	...
3	k7	a	...
4	k6	c	...
5	k2	b	...
6	k4	g	...
7	k1	c	...
...	...	...	...

IdxK

K	**TID**
k1	7
k2	5
k3	2
k4	6
k5	1
k6	4
k7	7
...	...

IdxA

A	**TID**
a	3
b	2
b	5
c	4
c	7
d	1
g	6
...	...

Figura 9.3: Esempio di relazione con due tipi di indici

Un indice può anche essere definito su di un insieme $A_1, \ldots, A_n$ di attributi. In questo caso, l'indice contiene un record per ogni combinazione di valori assunti dagli attributi $A_1, \ldots, A_n$ nella relazione, e può essere utilizzato per rispondere in modo efficiente ad interrogazioni che specifichino un valore per ciascuno di questi attributi.

Organizzazioni statiche e dinamiche. In tutti i metodi descritti, non abbiamo specificato come si gestiscono le situazioni in cui una pagina diventa troppo vuota o troppo piena. A seconda di come si affronta questo problema, l'organizzazione dei dati che ne scaturisce può essere *statica* o *dinamica*. Un'organizzazione è detta *statica* se, una volta dimensionata per una certa quantità di dati, non si riconfigura automaticamente in seguito ad un aumento dei dati memorizzati, il quale comporta, quindi, un degrado delle prestazioni, che si elimina con una riorganizzazione. Un'organizzazione è detta invece *dinamica* se è in grado di adeguarsi alla quantità di dati memorizzati, preservando le prestazioni senza bisogno di riorganizzazioni. Tutte le organizzazioni sopra descritte ammettono una versione statica ed una dinamica.

Scelta dell'organizzazione. La scelta dell'organizzazione più opportuna per ogni relazione costituisce il nocciolo della progettazione fisica, ed è un compito arduo che necessita di esperienza e di strumenti di supporto. La scelta dell'organizzazione

non è mai definitiva, ma varia durante l'uso del sistema, e in particolare quando si osservano delle prestazioni poco soddisfacenti. La scelta dell'organizzazione per una relazione è in genere guidata dall'applicazione che la usa ed è la più importante da eseguire in modo efficiente. In estrema sintesi, se tale applicazione utilizza un'alta percentuale dei dati si sceglie un'organizzazione seriale, se l'applicazione seleziona un piccolo insieme di record in base al valore di un attributo A si sceglie un'organizzazione ad albero, mentre se seleziona un unico record sulla base del valore di una chiave si sceglie un'organizzazione procedurale. Per facilitare l'esecuzione di ogni altra applicazione è possibile aggiungere indici sugli attributi coinvolti, tenendo conto delle indicazioni date nel Capitolo 7.

Esempio 9.1 Il linguaggio per la definizione dello schema della base di dati prevede comandi per scegliere le strutture per memorizzare i dati. Vediamo le soluzioni adottate da alcuni sistemi commerciali.

Tutti i DBMS memorizzano una relazione $R(A_1 : T_1, ..., A_n : T_n)$ con un'organizzazione seriale, in assenza di altre specifiche. Per agevolare le operazioni, si utilizzano indici su alcuni attributi o su combinazioni di attributi. Gli indici sono memorizzati ad albero dinamico e si definiscono con il comando:

CREATE [UNIQUE] INDEX Nome ON $R(A_i)$

L'opzione UNIQUE si usa per indici su attributi chiave.

Nel sistema INGRES, una volta caricati i dati, è possibile trasformare l'organizzazione di una relazione in sequenziale (*HEAPSORT*), *hash* statica, oppure ad albero statico (*ISAM*), con uno dei seguenti comandi:

MODIFY R TO HEAPSORT ON A_i ASC
MODIFY R TO HASH ON A_i
MODIFY R TO ISAM ON A_i

Le dichiarazioni delle organizzazioni prevedono inoltre la possibilità di imporre che i valori della chiave siano unici (ad esempio, HASH UNIQUE ON A_i) oppure che i valori siano memorizzati compressi, eliminando i caratteri bianchi (ad esempio, CHEAPSORT). È possibile poi aggiungere indici con il comando

CREATE INDEX Nome ON $R(A_i)$

che vengono trattati dal sistema come relazioni binarie con attributi (A_i, TID), dove i valori di TID sono gli identificatori interni delle ennuple. Un attributo può essere sostituito da una combinazione di più attributi, fino ad un massimo di sei. Questi indici possono a loro volta essere organizzati in modo sequenziale, *hash* o ISAM, come ogni altra relazione.

In Oracle l'organizzazione dinamica ad albero di una relazione, detta IOT (*index organized table*), si dichiara al momento della creazione di una tabella, con la chiave primaria e la specifica ORGANIZED INDEX

```
CREATE TABLE R(Pk Tipo PRIMARY KEY, ...) ORGANIZED INDEX;
```

In SQL Server l'organizzazione dinamica ad albero di una relazione si ottiene definendo sulla chiave primaria un CLUSTERED INDEX.

```
CREATE TABLE R(Pk Tipo PRIMARY KEY, ...)
CREATE CLUSTERED INDEX RAlbero ON R(Pk)
```

9.5 Gestore dei metodi di accesso

Il *gestore dei metodi di accesso* offre operatori per costruire, o eliminare, collezioni di record o indici e per recuperare i record uno dopo l'altro nell'ordine in cui sono memorizzati, oppure attraverso indici, astraendo dalla loro organizzazione fisica.

Gli accessi alle relazioni e agli indici avvengono con modalità simili a quelle previste per i cursori descritte nel Capitolo 8, quali apertura relazione, avanzamento del cursore, posizionamento del cursore sulla base del valore di un attributo, verifica di fine relazione, chiusura.

9.6 Gestore delle interrogazioni

La macchina logica offre una visione dei dati permanenti come un insieme di tabelle relazionali sulle quali si opera con gli operatori dell'SQL. Essa prevede i seguenti moduli:

- Il *gestore delle autorizzazioni* per controllare che solo gli utenti autorizzati facciano uso dei dati con le modalità consentite.
- Il *gestore del catalogo* per trattare i metadati, ovvero le informazioni sulle caratteristiche logiche e fisiche dei dati presenti.
- Il *gestore delle interrogazioni* per controllare la correttezza delle interrogazioni e stabilire la strategia migliore per eseguirle con un opportuno algoritmo detto *piano di accesso*.

Nel seguito ci soffermiamo sulla gestione delle interrogazioni, compito fra i più importanti di un DBMS svolto con le seguenti fasi:

1. Controllo lessicale, sintattico e semantico dell'interrogazione e sua rappresentazione in forma di albero logico.
2. Riscrittura algebrica dell'albero logico.
3. Ottimizzazione fisica e generazione del piano di accesso.
4. Esecuzione del piano di accesso.

Una volta controllata la correttezza dell'interrogazione, essa viene rappresentata con un albero logico, usando gli operatori algebrici, sul quale si opera nelle fasi successive come segue.

9.6.1 Riscrittura algebrica

Durante la riscrittura algebrica si applicano tecniche di trasformazione dell'interrogazione, basate sulle proprietà algebriche degli operatori relazionali, per trovarne una forma equivalente eseguibile con costi inferiori. Un possibile algoritmo di riscrittura algebrica è riportato nel Capitolo 4.

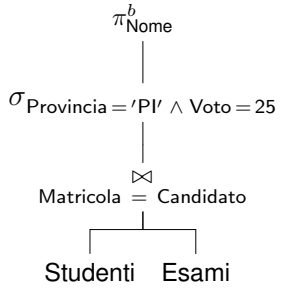
Vediamo alcuni esempi di riscritture algebriche, con riferimento alle seguenti relazioni, con Candidato chiave esterna di Esami per Studenti:

Studenti(Matricola, Nome, Provincia, AnnoNascita)
Esami(Codice, Materia, Candidato, Voto, Lode)

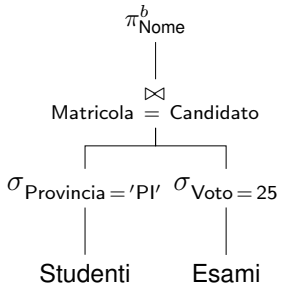
Esempio 9.2 Si consideri la seguente interrogazione:

```
SELECT  Nome
FROM    Studenti, Esami
WHERE   Matricola = Candidato AND Provincia = 'PI' AND Voto > 25;
```

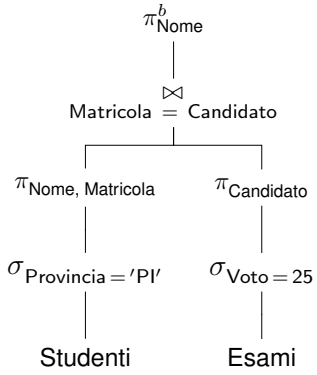
rappresentata con l'albero logico iniziale:[1]



Un esempio tipico di riscrittura algebrica è l'anticipazione delle restrizioni e delle proiezioni rispetto alle giunzioni, allo scopo di ridurre la dimensione degli argomenti di questi ultimi. Con l'anticipazione delle restrizioni l'albero logico iniziale diventa:



Applicando anche l'anticipazione della proiezione, il precedente albero logico diventa:



1. π^b è la proiezione senza l'eliminazione dei duplicati

Un'altra trasformazione, che di solito si prende in considerazione, è la normalizzazione delle condizioni nella forma normale prodotti di somme e i NOT di predicati vengono eliminati negando l'operatore di confronto (per es. NOT A>10 diventa A $\leq$ 10).

Le trasformazioni precedenti sono relativamente semplici, mentre quelle più complesse si hanno quando nell'interrogazione sono presenti *sottoselect* o viste, che in questa fase si tenta di eliminare, anche se non è sempre possibile farlo. In generale l'eliminazione delle *sottoselect* o delle viste aumenta poi le possibilità di generare piani di accesso migliori. Per esempio:

```
SELECT   Matricola, Nome
FROM     Studenti
WHERE    Matricola IN (SELECT   Candidato
                       FROM     Esami
                       WHERE    Materia = 'BD');
```

viene trasformata nell'interrogazione equivalente:

```
SELECT   DISTINCT Matricola, Nome
FROM     Studenti, Esami
WHERE    Matricola = Candidato AND Materia = 'BD';
```

Supponendo che esistano le viste:

```
CREATE VIEW VistaStudentiPisani
     (SELECT  *
      FROM    Studenti
      WHERE   Provincia ='PI' );

CREATE VIEW VistaEsamiBD
     (SELECT  *
      FROM    Esami
      WHERE   Materia = 'BD' );
```

l'interrogazione:

```
SELECT   Matricola, Nome
FROM     VistaStudentiPisani, VistaEsamiBD
WHERE    Matricola = Candidato;
```

viene trasformata in:

```
SELECT   Matricola, Nome
FROM     Studenti, Esami
WHERE    Matricola = Candidato AND Materia = 'BD' AND Provincia = 'PI';
```

9.6.2 Ottimizzazione fisica

Durante l'ottimizzazione fisica, si stabilisce come eseguire nel modo "migliore" l'albero logico di un'interrogazione considerando i parametri fisici in gioco, quali la dimensione delle relazioni, l'organizzazione dei dati e la presenza di indici. Il problema è particolarmente difficile perché, come si vedrà più avanti, ogni operazione dell'algebra relazionale può essere realizzata in modi diversi ed occorre utilizzare opportune strategie per stimare i costi delle possibili alternative e scegliere quella con costo inferiore.

L'ottimizzazione fisica delle interrogazioni non verrà considerata nel seguito perché esula dai fini di questo testo e per approfondimenti si rinvia ai testi citati nelle note bibliografiche. L'attenzione sarà invece posta sul formalismo usato nei DBMS per

mostrare il risultato dell'ottimizzazione fisica con una rappresentazione ad albero dell'interrogazione detta *albero fisico* o *piano di accesso*, in cui i nodi rappresentano degli *operatori fisici* che realizzano un algoritmo per eseguire un'operazione dell'algebra relazionale, o una sua parte.

La conoscenza del formalismo dei piani di accesso è utile perché i DBMS, di solito, su richiesta di chi formula un'interrogazione, mostrano il piano di accesso per eseguirla in modo che si possa (a) capire eventualmente come mai il sistema impieghi più tempo del previsto a produrre la risposta e (b) decidere poi opportune azioni correttive riguardanti l'organizzazione fisica dei dati per consentire al sistema di generare piani di accesso migliori.

Ogni DBMS ha i propri operatori fisici e per comodità si useranno quelli del sistema JRS,[2] che sono più semplici di quelli dei sistemi commerciali e, essendo il JRS disponibile gratuitamente, si possono facilmente fare delle prove per prendere familiarità con l'ottimizzazione fisica delle interrogazioni.

Descriviamo ora gli operatori fisici del sistema JRS che prenderemo in considerazione per realizzare le seguenti operazioni su relazioni della base di dati o risultato di un'altra operazione:

- Scansione di relazioni memorizzate con l'organizzazione seriale.
- Proiezione dei record di una relazione.
- Restrizione dei record di una relazione a quelli che soddisfano una condizione.
- Ordinamento dei record di una relazione.
- Giunzione dei record di due relazioni.
- Raggruppamento dei record di una relazione.

Gli operatori fisici, come quelli dell'algebra relazionale, ritornano collezioni di record con una struttura che dipende dal tipo di operatore.

Operatori per la scansione di relazioni (R)

- **TableScan**(R): ritorna la collezione dei record di R.
- **SortScan**(R, $\{A_i\}$): ritorna la collezione dei record di R ordinati sui valori degli $\{A_i\}$ in modo crescente (in realtà si può ordinare anche in modo decrescente, ma per semplicità si omette questa possibilità).

Si noti che l'argomento di questi operatori è R, il nome di una relazione, quindi devono necessariamente stare su una foglia di un albero fisico.

Operatori per la proiezione (π^b, π)

Per eseguire la proiezione di una relazione senza l'eliminazione dei duplicati, l'algoritmo è banale. Per eliminare invece i duplicati, un modo per procedere è di ordinare prima la relazione. Altri modi di procedere sono possibili per eseguire questa operazione che non richiedono l'ordinamento dei dati, ma per semplicità non si prendono in considerazione.

Vediamo gli operatori fisici che realizzano questi algoritmi, che prevedono come argomento la collezione dei record O restituiti da un altro operatore fisico:

- **Project**(O, $\{A_i\}$): ritorna la proiezione dei record di O sugli attributi $\{A_i\}$, senza l'eliminazione dei duplicati (realizza quindi il π^b).

2. Il JRS (*Java Relational System*) è stato sviluppato in Java per scopi didattici presso il Dipartimento di Informatica dell'Università di Pisa.

– **Distinct**(O): ritorna la collezione dei record *diversi* di O, che devono essere *ordinati*, senza fare proiezioni (realizza quindi solo in parte il π).

Operatore per l'ordinamento $\tau_{\{A_i\}}$

Per ordinare i dati ritornati da un operatore fisico O si usa il seguente operatore fisico:

– **Sort**(O, $\{A_i\}$): ritorna la collezione dei record di O ordinati sugli $\{A_i\}$.

A differenza del **SortScan**, che prima ordina i record di una relazione in memoria permanente e poi li ritorna, il **Sort** prima memorizza i record di O in una relazione temporanea, poi li ordina e infine li ritorna. Se non c'è spazio disponibile nel buffer, la relazione temporanea viene memorizzata in memoria permanente.

Per ordinare una relazione in memoria permanente si usa l'algoritmo di *ordinamento per fusione* (*merge sort*)

Esempio 9.3 In questi esempi, e in quelli che vedremo più avanti, l'obiettivo è solo di mostrare l'uso del formalismo dei piani di accesso, senza pretendere che il piano sia quello migliore per eseguire un'interrogazione, avendo evitato, per ragioni di spazio, di approfondire il problema dell'ottimizzazione fisica basata sui costi di esecuzione delle operazioni dell'algebra relazionale. Di ogni interrogazione SQL si presenta prima l'albero logico, con l'eventuale anticipazione delle restrizioni, e poi un possibile albero fisico, con riferimento alle seguenti relazioni:

R(A, B)
S(C, D)

Si consideri l'interrogazione:

```
SELECT   DISTINCT A
FROM     R;
```

Si lascia al lettore, come esercizio, definire un altro possibile albero fisico equivalente al precedente sostituendo il **SortScan** con un **TableScan**.

Operatori per la restrizione (σ_ψ)

L'operazione $\sigma_\phi(R)$ può essere realizzata in modi diversi. In assenza di indici si esaminano tutti i record di R e si controlla quali di essi soddisfano la condizione ϕ. I casi più interessanti si presentano quando la condizione riguarda attributi sui quali sono definiti degli indici.

Se la condizione è del tipo $A\,\theta\,c$ (*condizione semplice*), con c una costante e θ un predicato di confronto (=, $<$, $\leq$, $>$, $\geq$), i record che soddisfano la condizione si trovano usando l'indice su A (si veda la Sezione 9.2).

Se la condizione ϕ è il prodotto logico di condizioni semplici $\phi = \phi_1 \wedge \phi_2$ su attributi con indice, si può procedere in due modi. Il primo prevede l'uso di un solo

indice e si basa sul fatto che $\sigma_{\phi}(R) = \sigma_{\phi_1}(\sigma_{\phi_2}(R))$, pertanto si usa l'indice per risolvere la restrizione $\sigma_{\phi_2}(R)$ e si controlla sul risultato la condizione ϕ_1. Il secondo metodo usa entrambi gli indici disponibili calcolando gli insiemi dei riferimenti S_1 e S_2 ai record che soddisfano le condizioni ϕ_1 e ϕ_2 per recuperare poi i record con identificatori in $S_1 \cap S_2$.

Analogamente, se la condizione ϕ è la somma logica di condizioni semplici $\phi = \phi_1 \vee \phi_2$ su attributi con indice, si possono calcolare gli insiemi dei riferimenti S_1 e S_2 dei record che soddisfano le condizioni ϕ_1 e ϕ_2 per recuperare poi i record con identificatori in $S_1 \cup S_2$.

Vediamo gli operatori fisici che realizzano questi algoritmi, supponendo di usare al più un solo indice:

– **Filter**(O, ψ): ritorna la collezione dei record di O che soddisfano la condizione ψ.
– **IndexFilter**(R, Idx, ψ): ritorna la collezione dei record di R che soddisfano la condizione ψ, con l'uso dell'indice Idx definito su attributi di R. La condizione ψ è un prodotto logico di predicati che interessano solo i valori degli attributi sui quali è definito l'indice. Il risultato è ordinato sugli attributi di R sui quali è definito l'indice.

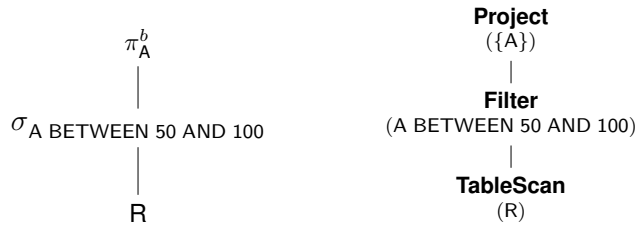
  L'operatore esegue due operazioni: utilizza l'indice per trovare rapidamente i riferimenti ai record che soddisfano la condizione e poi recupera i record di R. Queste due operazioni potrebbero essere realizzate con due operatori fisici che di solito nei DBMS commerciali sostituiscono l'operatore **IndexFilter**(R, Idx, ψ), ma in questo testo non si considerano per semplificare i piani di accesso delle interrogazioni: **TIDIndexFilter**(Idx, ψ), che ritorna un insieme di TID, e **TableAccess**(O, R), che ritorna la collezione dei record di R con i TID in O.

L'**IndexFilter** ha come argomento la relazione R sulla quale è definito l'indice (e non un altro operatore), pertanto in un piano di accesso l'**IndexFilter** può essere solo una foglia e non un nodo interno dell'albero.

Esempio 9.4 Si mostrano alcuni esempi d'uso degli operatori fisici visti in precedenza per definire possibili piani di accesso.

1. Interrogazione SELECT-FROM-WHERE (SFW)

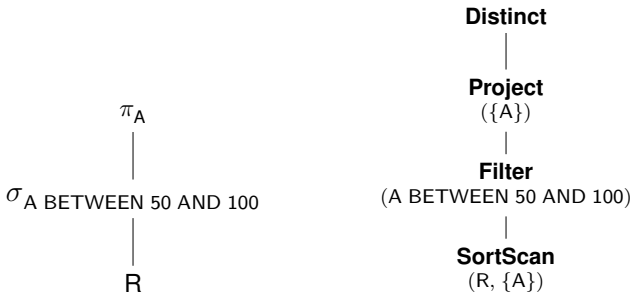
```
SELECT  A
FROM    R
WHERE   A BETWEEN 50 AND 100;
```



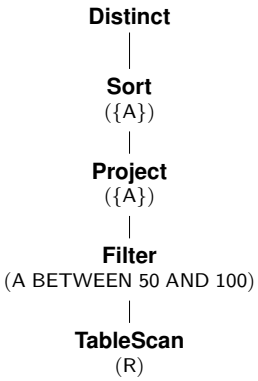
   Quando l'interrogazione è semplice c'è una corrispondenza uno a uno tra gli operatori logici e fisici dei due alberi.
   Un altro possibile albero fisico equivalente al precedente si ottiene scambiando il **Project** con il **Filter**.

2. Interrogazione SFW con DISTINCT

```
SELECT  DISTINCT A
FROM    R
WHERE   A BETWEEN 50 AND 100;
```



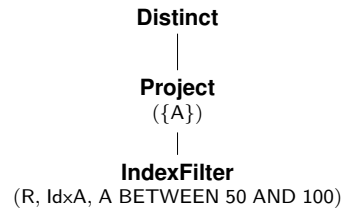
In questo esempio, con **SortScan** si ottiene la collezione ordinata dei record di R, che vengono filtrati dal **Filter**, proiettati dal **Project** e si eliminano i duplicati con **Distinct**.
Se A fosse una chiave, il **Distinct** e l'ordinamento di R sarebbero inutili.
Un altro possibile albero fisico equivalente al precedente si ottiene ordinando il risultato del **Project** e non R:



3. Interrogazione SFW con indice

```
SELECT   DISTINCT A
FROM     R
WHERE    A BETWEEN 50 AND 100;
```

con l'ipotesi che esista un indice su A.
L'albero logico è identico al caso precedente.

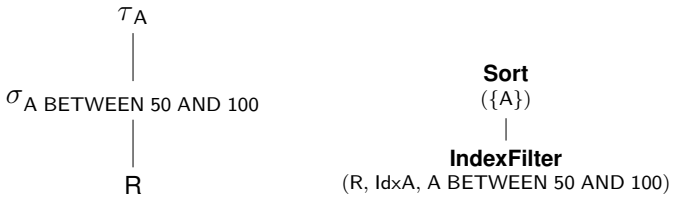


L'indice su A consente di usare l'operatore **IndexFilter** che, restituendo la collezione dei record di R che soddisfano la condizione, ordinati su A, rende inutile il **Sort**. Dopo aver generato un possibile piano di accesso, è bene controllare se alcuni operatori possono essere eliminati.

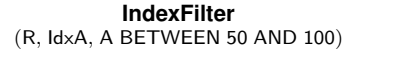
4. Interrogazione SFW con ORDER BY e indice

```
SELECT    *
FROM      R
WHERE     A BETWEEN 50 AND 100
ORDER BY  A;
```

con l'ipotesi che esista un indice su A.



Il **Sort** è usato per trattare l'ORDER BY, ma in questo caso è inutile perché l'**IndexFilter** ritorna la collezione dei record di R, che soddisfano la condizione, ordinati su A. L'albero fisico finale diventa



Operatori per la giunzione ($\bowtie_{\psi_J}$)

La giunzione è l'operazione più complessa dell'algebra relazionale e può essere eseguita in più modi. Vediamo i metodi principali supponendo di dover eseguire la giunzione $R \underset{B=D}{\bowtie} S$ delle relazioni $R(A, \underline{B})$ e $S(\underline{C}, D)$.

L'algoritmo più ovvio per valutare la giunzione è il *nested loop* che ha la seguente struttura:

for each $r \in R$ **do**
  for each $s \in S$ **do**
    if $r[B] = s[D]$ **then** aggiungi $< r, s >$ al risultato;

Se esiste un indice sull'attributo di giunzione D della relazione S (detta *relazione interna*), un algoritmo più efficiente è l'*index nested loop* che ha la seguente struttura:

for each $r \in R$ **do** {
  usa l'indice su D per trovare tutti i record $s \in S$ *tali che* $s[D] = r[B]$;
  aggiungi $< r, s >$ al risultato };

Quando la condizione di giunzione è fra la chiave primaria di R e la chiave esterna di S, e le due relazioni sono ordinate sugli attributi di giunzione, un altro algoritmo interessante è il *merge join* che ha la seguente struttura:

r = primo record di R; // r = *null* se R è vuota
s = primo record di S; // s = *null* se S è vuota
// sia succ(w) il record successivo a w
// o il valore *null* se w è l'ultimo;
while not r == *null* **and not** s == *null* **do**
  if $r[B]$ = $s[D]$
  then{
    while not s == *null* **and** $r[B]$ = $s[D]$ **do**
      {aggiungi $< r, s >$ al risultato;
       s = succ(s) };

```
        r = succ(r) }
  else if r[B] < s[D]
        then r = succ(r)
        else s = succ(s);
```

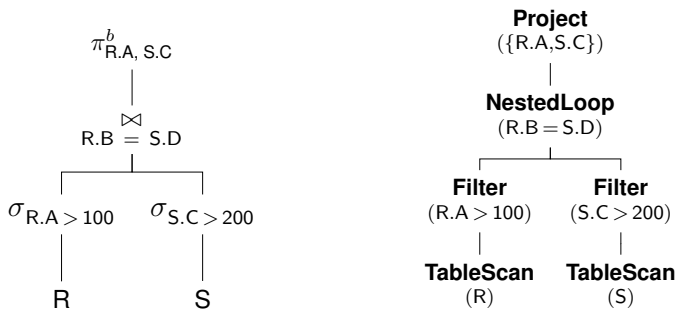
Vediamo gli operatori fisici che realizzano questi algoritmi:

- **NestedLoop**(O_E, O_I, ψ_J): ritorna la giunzione dei record di O_E e O_I che soddisfano la condizione di giunzione ψ_J.
- **IndexNestedLoop**(O_E, O_I, ψ_J): ritorna la giunzione dei record di O_E e O_I che soddisfano la condizione di giunzione ψ_J, supponendo che esista un indice Idx sugli attributi di giunzione della relazione interna.
  Supponiamo per semplicità che la condizione di giunzione sia $O_E.A_i = O_I.A_j$ e che Idx sia un indice su $S.A_j$. L'operatore interno O_I può essere:
  - un **IndexFilter**(S, Idx, ψ_J): per ogni record r di O_E si usa l'indice Idx su $S.A_j$ per trovare i record s di O_I che soddisfano la condizione di giunzione $S.A_j = \underline{O_E.A_i}$, con $\underline{O_E.A_i} = r.A_i$;
  - un **Filter**(O, ψ') con O un **IndexFilter**(S, Idx, ψ_J): per ogni record r di O_E si usa l'indice Idx su $S.A_j$ per trovare i record s di O_I che soddisfano la condizione di giunzione, come nel caso precedente, ma di questi si ritornano solo quelli che soddisfano anche la condizione ψ' del filtro.
- **MergeJoin**(O_E, O_I, ψ_J): ritorna la giunzione dei record di O_E e O_I che soddisfano la condizione di giunzione ψ_J, supponendo che (a) i record di O_E e O_I siano ordinati sui relativi attributi di giunzione e (b) la condizione di giunzione sia fra una chiave di O_E e una chiave esterna di O_I.

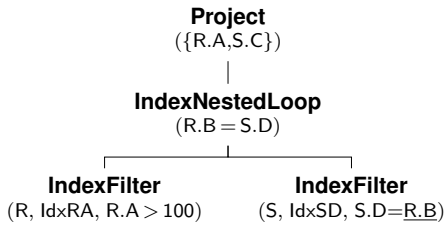
Tutti gli operatori fisici che realizzano la giunzione ritornano collezioni di record che sono la concatenazione di un record r di O_E e di un record s di O_I, ovvero hanno sia gli attributi di r che quelli di s, e preservano l'ordine dei record di O_E.

Esempio 9.5 Si consideri l'interrogazione

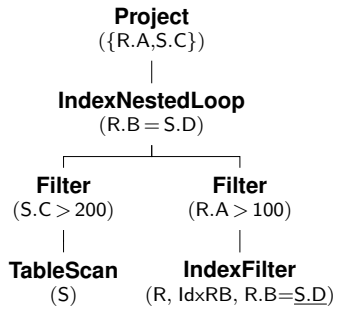
```
SELECT  R.A, S.C
FROM    R, S
WHERE   R.B = S.D  AND  R.A > 100  AND  S.C > 200;
```



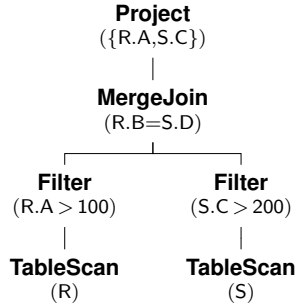
Supponiamo di avere due indici su R.A e su S.D e di usare l'**IndexNestedLoop** per la giunzione; l'albero fisico diventa:



Se ci fosse anche un indice su R.B, si potrebbe usare l'**IndexNestedLoop** con R come relazione interna, ottenendo l'albero fisico:



Infine, se le relazioni R e S fossero ordinate sugli sugli attributi di giunzione R.B e S.D, si potrebbe usare il **MergeJoin** con R come relazione esterna, ottenendo l'albero fisico:



Operatore per il raggruppamento ($_{\{A_i\}}\gamma_{\{f_i\}}$)

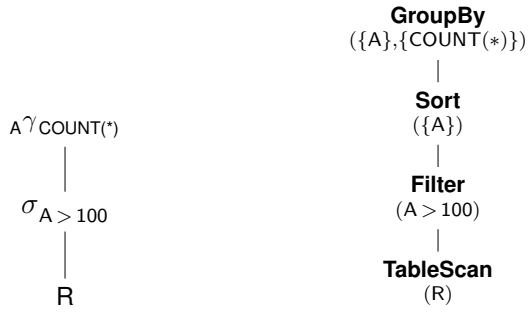
– **GroupBy**(O, $\{A_i\}$, $\{f_i\}$): ritorna una collezione di record, uno per ogni gruppo di record di O, con attributi quelli di raggruppamento $\{A_i\}$ e le funzioni di aggregazione $\{f_i\}$, supponendo che i record di O siano ordinati sugli attributi di raggruppamento. Il risultato è ordinato sugli attributi di raggruppamento.[3]

3. Nei sistemi commerciali è previsto anche un altro operatore **HashGroupBy** che raggruppa i record di O non ordinati usando una tecnica *hash*, ma per semplicità non si prende in considerazione.

Esempio 9.6 Vediamo possibili piani di alcune interrogazioni con raggruppamento:

1. Interrogazione SFW con GROUP BY

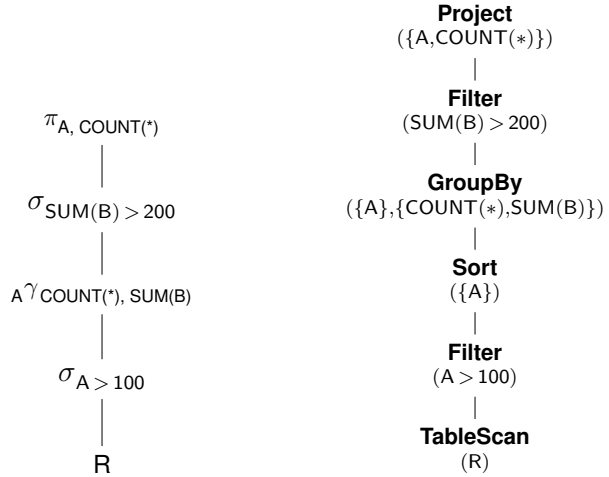
```
SELECT    A, COUNT(*)
FROM      R
WHERE     A > 100
GROUP BY  A;
```



Per usare il **GroupBy** occorre ordinare i dati sui quali opera. Il **Project** è necessario solo se gli attributi della SELECT sono un sottoinsieme di quelli dei record prodotti dal **GroupBy**.

2. Interrogazione SFW con GROUP BY e HAVING

```
SELECT    A, COUNT(*)
FROM      R
WHERE     A > 100
GROUP BY  A
HAVING    SUM(B) > 200;
```



Si noti che (a) le funzioni di aggregazione di γ e **GroupBy** sono quelle diverse della SELECT e dell'HAVING, (b) la condizione dell'HAVING diventa un **Filter** sul **GroupBy** e (c) per produrre il risultato occorre un **Project**.

Unione, differenza e intersezione

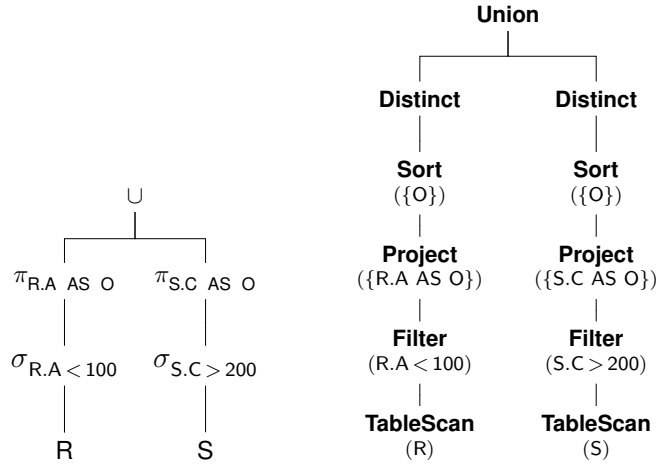
L'operazione di unione è semplice da realizzare se sono ammessi record duplicati nel risultato (operatore UNION ALL in SQL). Se i duplicati vanno eliminati (operatore insiemistico UNION in SQL), l'operazione è ancora semplice da realizzare nell'ipotesi che i record delle due relazioni siano ordinati e privi di duplicati. In modo analogo si procede per la differenza e l'intersezione insiemistica di due relazioni. Altri modi di procedere sono possibili per eseguire queste operazioni che non richiedono l'ordinamento dei dati e l'assenza di duplicati, ma per semplicità non si prendono in considerazione.

Vediamo gli operatori fisici che realizzano questi algoritmi:

– **Union**(O_E, O_I), **Except** (O_E, O_I), **Intersect** (O_E, O_I): ritornano la collezione dei record delle operazioni insiemistiche, supponendo che i record degli operandi siano dello stesso tipo, ordinati e privi di duplicati.
– **UnionAll** (O_E, O_I): ritorna l'unione dei record degli operandi senza l'eliminazione dei duplicati.

Esempio 9.7 Si consideri l'interrogazione:

```
SELECT  R.A AS O
FROM    R
WHERE   R.A < 100
    UNION
SELECT  S.C AS O
FROM    S
WHERE   S.C > 200;
```



9.6.3 Esecuzione di un piano di accesso

Ogni operatore fisico di un piano di accesso è un *iteratore*, cioè è realizzato con un oggetto con quattro metodi:

- **open**(): inizializza lo stato dell'operatore ed esegue il metodo **open** degli operatori fisici dei suoi argomenti.

- **next**(): ritorna il record successivo del risultato interagendo con gli operatori fisici dei suoi argomenti.
- **isDone**(): ritorna *true* se non ci sono altri record da ritornare, *false* altrimenti.
- **close**(): termina le operazioni dell'operatore fisico e dei suoi argomenti.

Supponendo che AlberoFisico sia il nome di un piano di accesso, il risultato dell'interrogazione si ottiene eseguendo il seguente programma:

```
AlberoFisico.open();                    //inizia le operazioni
while !AlberoFisico.isDone()            //finché ci sono elementi
    print(AlberoFisico.next());         //stampa prossimo record
AlberoFisico.close();                   //termina le operazioni
```

Quando si esegue il metodo **open** della radice dell'albero, esso inizializza lo stato dell'operatore ed esegue l'**open** del figlio e così via fino ad arrivare alle foglie. Terminata la fase di inizializzazione, inizia la fase di generazione del risultato.

Quando si esegue il metodo **next** della radice dell'albero, esso esegue il **next** del figlio per ottenere il prossimo record del risultato, il figlio fa la stessa operazione sul figlio e così via fino ad arrivare alla foglia: i record ritornati dalla foglia faranno la strada inversa. Ogni operatore quindi è pronto a restituire un record alla volta, risultato dell'elaborazione del record ottenuto dal figlio (o dai figli, nel caso di operatori binari).

Il Sort è l'unico operatore che si discosta da questo schema: il metodo che compie gran parte del lavoro è **open**, che richiede tutti i record al nodo figlio, li memorizza ordinati in una relazione temporanea per poi restituirli uno alla volta su richiesta del nodo padre.

9.7 Gestore della concorrenza

La tecnica utilizzata più comunemente per realizzare il controllo della concorrenza nei sistemi centralizzati è la tecnica del *blocco a due fasi* (*Two Phase Lock, 2PL*) .

Utilizzando questa tecnica, si associa ad ogni dato usato da un'operazione un blocco (*lock*) in lettura o in scrittura. Ogni transazione, prima di eseguire un'azione su di un dato, richiede il blocco corrispondente di lettura o scrittura. Due transazioni non possono avere blocchi *incompatibili* sullo stesso dato, ovvero blocchi con almeno uno dei due di scrittura. Pertanto in ogni momento, per ogni dato possono essere stati assegnati o più blocchi in lettura o, alternativamente, un solo blocco in scrittura. La transazione che richiede un blocco incompatibile su un dato viene messa in attesa, e risvegliata solo quando il dato diventa disponibile. Per garantire la serializzabilità e l'isolamento, ogni transazione viene di solito eseguita con la tecnica del blocco a due fasi stretto, che prevede che i blocchi di una transazione vengano rilasciati tutti assieme, solo dopo che la transazione sia terminata.

La tecnica del blocco prevede che una transazione venga posta in attesa quando richiede un blocco che non può ottenere. Può accadere che questa attesa diventi infinita: supponiamo che T_1 ottenga un blocco in scrittura su X_1 e T_2 ottenga un blocco in scrittura su X_2, e che successivamente T_1 richieda anch'essa un blocco in scrittura su X_2: in questo caso T_1 viene messa in attesa del fatto che T_2 rilasci il proprio blocco. Se a questo punto T_2 richiede un blocco su X_1, anche T_2 va in attesa del fatto che T_1 rilasci il proprio blocco, e si crea una situazione di attesa "circolare", ovvero di *stallo* (*deadlock*).

I metodi comunemente usati per sbloccare una situazione di stallo sono i seguenti.

– Rilevazione tramite grafo delle attese: si costruisce un grafo avente come nodi le transazioni, aggiungendo un arco da T_1 a T_2 ogni volta che T_1 va in attesa del rilascio di un blocco da parte di T_2. Ogni volta che si crea un ciclo nel grafo, una delle transazioni coinvolta viene fatta abortire (tipicamente la più giovane, quella che ha meno risorse o quella il cui aborto ha il minor costo).
– Rilevazione per *time-out*: ogni volta che un'attesa si prolunga oltre un certo limite, la transazione in attesa viene abortita, presupponendo l'esistenza di uno stallo.

Il blocco a due fasi garantisce la serializzabilità, ma limita la concorrenza possibile tra diverse transazioni. Ad esempio, un'applicazione lunga che legge una grande quantità di dati potrebbe non riuscire mai ad acquisire tutti i blocchi necessari, oppure, quando li avesse acquisiti tutti, potrebbe impedire ad ogni altra transazione che voglia effettuare modifiche di partire. Per questo motivo, molti sistemi permettono al programmatore di limitare la quantità di blocchi richiesti dalle applicazioni, anche se questo comporta la perdita della serializzabilità (si veda il Capitolo 8).

9.8 Gestore dell'affidabilità

Compito del gestore dell'affidabilità è di eseguire le operazioni delle transazioni e la loro terminazione garantendo che la base di dati contenga solo gli effetti delle transazioni terminate normalmente e sia protetta da fallimenti di transazione, di sistema e disastri.

Le operazioni delle transazioni possono essere eseguite con algoritmi diversi. Supponiamo che si adotti l'algoritmo *disfare-rifare*, come accade nei sistemi DB2 e Oracle:

– Una modifica di un dato può essere riportata sulla base di dati prima che la transazione termini. Nel caso di fallimento di transazione o di sistema occorre annullare le modifiche fatte dalla transazione sulla base di dati (disfare).
– Una transazione T è considerata terminata normalmente, e viene scritto nel *giornale* (descritto più avanti) il record (T, *commit*), senza che le sue modifiche vengano preventivamente riportate nella base di dati; questo compito viene svolto dal gestore del buffer quando lo ritiene opportuno. Nel caso di fallimento di sistema occorre rifare le modifiche fatte dalle transazioni terminate normalmente perché non si è certi che i loro effetti siano stati riportati sulla base di dati.

La struttura dati che viene utilizzata in maniera cruciale tanto per disfare che per rifare gli effetti delle transazioni è il *giornale delle modifiche* (*log*). Questo è un archivio gestito in maniera sicura (cioè mantenuto in due copie su dispositivi con fallimento indipendente) che contiene, per ogni operazione effettuata da una transazione sui dati, le seguenti informazioni:

– L'identificatore della transazione che ha effettuato l'operazione.
– L'operazione eseguita (inserzione, aggiornamento, cancellazione, inizio transazione, *commit*, *abort*).
– L'identificatore del record modificato.
– Il vecchio ed il nuovo valore del record.

Poiché un malfunzionamento può anche intercorrere tra il momento in cui un'operazione viene eseguita ed il momento in cui tale operazione viene registrata nel giornale, è essenziale registrare tutte le operazioni nel giornale prima che esse vengano eseguite. Più precisamente, è necessario seguire le due regole seguenti:

1. *Regola per disfare* (*Write ahead log*): prima di eseguire una modifica, occorre salvare il vecchio valore nel giornale, in modo che non vada perduto in caso di malfunzionamento.
2. *Regola per rifare* (*Commit Rule*): prima di considerare terminata una transazione, occorre salvare nel giornale i nuovi valori dei dati modificati, in modo da poter rieseguire la transazione in caso di fallimento di sistema o di disastro.

Avendo a disposizione il giornale, ed una vecchia copia della base di dati, la procedura di ripristino in caso di malfunzionamento è la seguente:

– In caso di fallimento di transazione, si disfano gli effetti di tutte le operazioni della transazione, utilizzando le informazioni sul giornale, ed infine si memorizza una marca di *abort* sul giornale stesso.
– In caso di fallimento di sistema, prima di rendere operativa la base di dati, si esegue il comando *restart* che scandisce il giornale per disfare gli effetti delle transazioni attive al momento del fallimento e per rifare gli effetti delle transazioni terminate normalmente. Per limitare poi la porzione di giornale da scandire, i DBMS effettuano ad intervalli brevi e regolari un'operazione di allineamento (*checkpoint*) che consiste nel riportare in memoria permanente tutti gli aggiornamenti effettuati nei buffer, e nel memorizzare poi sul giornale una marca di *checkpoint*. In questo modo, in caso di fallimento di sistema, la procedura di ripristino può partire dallo stato del giornale e della base di dati in linea, avendo la certezza che non c'è bisogno di rifare nessuna delle operazioni eseguite prima del *checkpoint*. Si osservi tuttavia che è necessario disfare le operazioni eseguite prima e dopo il *checkpoint* da transazioni non terminate normalmente.
– In caso di disastro, si porta in linea la vecchia copia stabile dello stato della base di dati e si riapplicano ad essa tutte le operazioni registrate sul giornale da parte di transazioni che abbiano effettuato il *commit*. Se la vecchia copia era stata presa in un momento di attività del sistema, è anche necessario disfare gli effetti di tutte le transazioni che erano in corso al momento della copia e che non avevano ancora effettuato il *commit* al momento del malfunzionamento.

Si osservi che la regola per disfare garantisce che il vecchio valore di un record modificato non vada mai perduto, ma implica che, in seguito ad un malfunzionamento avvenuto poco dopo la scrittura del record nel giornale, non sia possibile sapere se l'operazione fosse stata realmente effettuata sui dati. Si osservi inoltre che un malfunzionamento può avere luogo anche durante il ripristino. Per ambedue questi motivi, è importante che le operazioni di "disfacimento" e "rifacimento" che si effettuano durante il ripristino siano effettuate in maniera "idempotente", ovvero in modo da ottenere l'effetto voluto sia che si stia disfacendo un'operazione realmente effettuata, sia che si stia disfacendo un'operazione già disfatta.

9.9 Conclusioni

Sono state presentate le principali tecniche utilizzate per realizzare le funzionalità fondamentali di un DBMS centralizzato: la gestione dei dati, delle interrogazioni, della concorrenza e dell'affidabilità. Per mostrare come venga eseguita un'interrogazione SQL è stato introdotto il formalismo dei piani di accesso, una versione semplificata di quello usato dai sistemi commerciali per visualizzare il risultato dell'ottimizzazione fisica di un'interrogazione. Questa informazione è molto utile all'amministratore della base di dati per la messa a punto delle strutture di memorizzazione dei dati al fine di un'esecuzione efficiente delle interrogazioni.

Per fare pratica con i piani di accesso, dal sito del libro si può scaricare il sistema JRS, sviluppato in Java per scopi didattici presso il Dipartimento di Informatica dell'Università di Pisa, che funziona con ogni sistema operativo dotato di una macchina virtuale Java, supporta un'ampia versione dell'SQL, l'ottimizzatore produce piani di accesso delle interrogazioni con gli operatori descritti in questo capitolo, e prevede due interfacce grafiche per definire ed eseguire alberi logici o fisici. È interessante confrontare il piano di accesso immaginato per un'interrogazione con quello prodotto da un sistema con un buon ottimizzatore.

Esercizi

1. Dire quali delle seguenti affermazioni è vera o falsa e giustificate la risposta:
   (a) I seguenti piani di accesso per l'interrogazione
   SELECT A FROM R ORDER BY A
   non sono equivalenti:

   (b) Operatore logico e operatore fisico sono sinonimi.
   (c) Ad ogni interrogazione SQL corrispondono più alberi logici.
   (d) Ad ogni albero logico corrispondono più alberi fisici.
   (e) La gestione della concorrenza con il blocco dei dati non crea condizioni di stallo.
   (f) Con il metodo disfare–rifare, nessun dato modificato da una T può essere riportato nella BD prima che il corrispondente record del giornale sia scritto nella memoria permanente.
   (g) Con il metodo rifare, tutte le modifiche di una T devono essere riportate nella BD prima che il record di commit sia scritto nel giornale.
2. Si considerino due relazioni unarie $R(A)$ e $S(B)$ con i seguenti record:
   $R = \{(A := 7), (A := 2), (A := 8), (A := 3), (A := 1), (A := 3), (A := 6)\}$
   $S = \{(B := 4), (B := 2), (B := 1), (B := 3), (B := 2), (B := 7), (B := 3)\}$

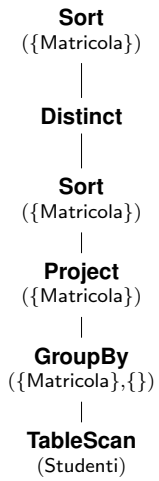
   Mostrare (a) il contenuto di un indice sull'attributo A di R e (b) il risultato prodotto dalla giunzione $R \underset{A=B}{\bowtie} S$ usando il NestedLoop.
3. Si consideri la relazione R(A, B, C), con chiave primaria A, e l'interrogazione:

```
SELECT    A, COUNT(*)
FROM      R
WHERE     A > 100
GROUP BY  A;
```

   Dare l'albero logico iniziale dell'interrogazione e un possibile piano di accesso. Come cambia la soluzione se nella SELECT ci fosse DISTINCT A, COUNT(*)?
4. Si consideri la relazione Studenti(Matricola, Nome, AnnoNascita), ordinata sulla chiave primaria Matricola, e l'interrogazione:

```
SELECT    DISTINCT Matricola, COUNT(*)
FROM      Studenti
WHERE     AnnoNascita = 1974
GROUP BY  Matricola
ORDER BY  Matricola;
```

Dare l'albero logico iniziale dell'interrogazione e si dica se il seguente piano d'accesso produce il risultato cercato. Se non va bene, lo si modifichi in tre modi: (a) aggiungendo prima solo le parti mancanti (operatori e parametri), (b) semplificando poi il piano eliminando operatori inutili e (c) modificando infine il piano supponendo che esita un indice su AnnoNascita:



5. Si consideri il seguente schema relazionale:

   Aule(CodiceA, Edificio, Capienza)
   Lezioni(CodiceA, CodiceC, Ora, GiornoSett, Semestre)
   Corsi(CodiceC, NomeC, Docente)

   e l'interrogazione:

   ```
   SELECT    A.Edificio, COUNT(*)
   FROM      Aule A, Lezioni L
   WHERE     A.CodiceA = L.CodiceA AND Semestre = 1
   GROUP BY  A.Edificio
   HAVING    COUNT(*) > 2;
   ```

   Si dia l'albero logico iniziale dell'interrogazione e un piano di accesso che utilizzi indici (a) nel caso di giunzione con l'operatore NestedLoop e (b) nel caso di giunzione con l'operatore IndexNestedLoop

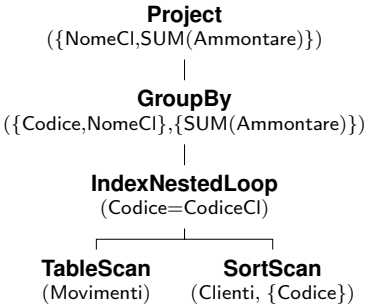
6. Si consideri la base di dati:

   Clienti(Codice, NomeCl, AnnoNascita), con chiave primaria Codice
   Movimenti(CodiceCl, Ammontare, Tipo), con chiave esterna CodiceCl

   e l'interrogazione:

   ```
   SELECT    NomeCl, SUM(Ammontare)
   FROM      Clienti, Movimenti
   WHERE     Codice = CodiceCl AND AnnoNascita = 1974
   GROUP BY  Codice, NomeCl
   HAVING    COUNT(*) > 5 ;
   ```

Dare l'albero logico iniziale dell'interrogazione e si dica (a) se il seguente piano d'accesso è corretto, (b) se produce il risultato cercato. Se non va bene, lo si modifichi aggiungendo le parti mancanti (operatori e parametri), e si dia il tipo del resultato.



Note bibliografiche

Per approfondire gli argomenti di questo capitolo si rinvia al libro [Alb01], dedicato alle strutture e algoritmi per la realizzazione di DBMS. Il sistema JRS con la relativa documentazione è disponibile sul sito Web del libro alla pagina
http://www.di.unipi.it/∼albano/costruireDBMS.html.