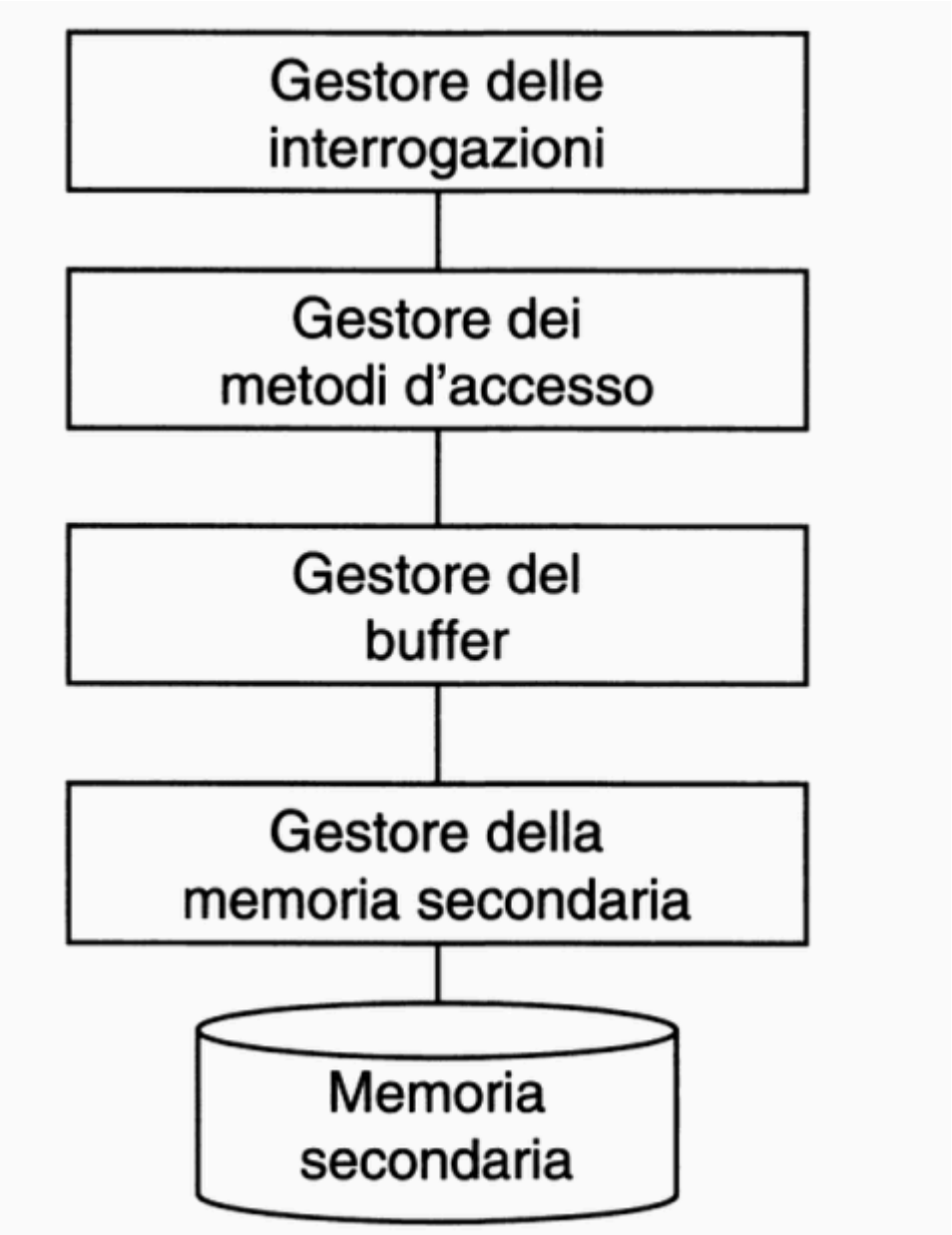
6 - Organizzazione fisica e gestione delle interrogazioni

Per illustrare i concetti principali dell'organizzazione fisica della basi di dati e gestione delle interrogazioni facciamo riferimento a questo schema, che illustra schematicamente le componenti di un DBMS coinvolte nel processo:



Nelle applicazioni di per basi di dati, le operazioni vengono affidate a un modulo detto **gestore delle interrogazioni**, che traduce le interrogazioni in una forma interna, le trasforma al fine di renderle più efficienti (tramite il modulo chiamato **ottimizzatore delle interrogazioni**) e le realizza in termini di operazioni a livello più basso (scansione, ordinamento, accesso diretto), che fanno riferimento alla struttura fisica dei file e sono eseguite da un modulo sottostante chiamato **gestore dei buffer**, che ha la responsabilità di mantenere temporaneamente porzioni della basi di dati in memoria centrale, per favorire l'efficienza garantendo allo stesso tempo affidabilità. Il gestore dei buffer invia poi al **gestore della memoria secondaria** le richieste effettive di lettura e scrittura fisica

Memoria principale, memoria secondaria e gestione dei buffer

Ci sono due motivi fondamentali per cui le basi di dati hanno necessità di gestire dati in memoria secondaria:

- La dimensione della memoria principale non risulta quasi mai sufficiente per contenere una base di dati
- Una delle caratteristiche fondamentali per le base di dati è la persistenza, cosa che la memoria principale non può offrire poiché limitata a contenere informazioni nel lasso di tempo di esecuzione dei programmi e non oltre (quindi spegnimento della macchina o guasto).

Gestione dei buffer

L'interazione fra memoria centrale e memoria secondaria è realizzata nei DBMS attraverso l'utilizzo di un'apposita grande zona di memoria detta **buffer**, gestita dal DBMS in modo condiviso per tutte le applicazioni.
Il buffer è organizzato in **pagine**, che hanno dimensioni pari a un numero intero di blocchi di memoria secondaria (assumeremo che ogni blocco corrisponda a esattamente un blocco di memoria secondaria).
Il gestore dei buffer si occupa del caricamento e del salvataggio delle pagine dalla memoria centrale alla memoria di massa.
L'interfaccia del gestore offerta dal gestore del buffer è relativamente più complessa di quanto accennato, poiché non può limitarsi alle richieste di lettura e scrittura ma ha bisogno di informazioni sul prevedibile riutilizzo delle pagine (per evitarne la lettura ripetuta) e sull'eventuale necessità di effettuare immediatamente gli aggiornamenti (per garantirne l'effettuazione ed evitare che vadano persi in modo irrecuperabile in caso di guasto).

Il gestore del buffer gestisce, oltre al buffer appunto, un direttorio che per ogni pagina mantiene:

- File fisico e numero blocco corrispondete alla pagina
- Due variabili di stato: un contatore che indica quanti programmi utilizzano la pagina, un bit che indica se la pagina è “sporca”, cioè se è stata modificata

La conoscenza di questi dati è fondamentale nel momento in cui diventa necessario introdurre nuove pagine in un buffer saturo, per capire quali pagine andare a sostituire. Il buffer comunica con il sistema mediante delle operazioni primitive:

- fix: consiste nella richiesta di accesso ad una pagina, restituisce il riferimento alla pagina richiesta, richiede una lettura se la pagina non è nel buffer, incrementa il contatore per l’utilizzo della pagina
- $\mathrm{setDirty}$: comunica al buffer manager che la pagina è stata modificata

- unfix: indica che la transazione ha concluso l'utilizzo della pagina, quindi decrementa il contatore di utilizzo di pagina
- force: trasferisce in modo sincrono una pagina in memoria secondaria su richiesta del gestore dell'affidabilità.

Il buffer può richiedere scritture in modo sincrono, cioè esplicitamente richiesto con una primitiva force, oppure in modo asincrono, quindi le scritture delle pagine modificate e conservate nel buffer vengono scritte quando il buffer lo ritiene opportuno.

DBMS e file system

Il file system è un modulo messo a disposizione dal sistema operativo che gestisce la memoria secondaria, utilizzato dai DBMS oggi, seppur in maniera limitata, per:

- Creare ed eliminare file
- Leggere e scrivere blocchi o sequenze di blocchi contigui

In molti casi, il DBMS crea file di grandi dimensioni che utilizza per memorizzare diverse relazioni (o l'intera basi di dati), in altri invece vengono creati file in tempi successivi e può succedere che un file contenga i dati di più relazioni e che varie tuple di una relazione siano in file diversi. In sostanza, il DBMS gestisce i blocchi come se fossero un unico grande spazio di memoria secondaria e costruisce, in tale spazio, le strutture fisiche con cui implementa le relazioni

Strutture primarie per l'organizzazione di file

La struttura primaria di un file stabilisce il criterio secondo il quale sono disposte le tuple nell'ambito del file. Le strutture possono essere divise in tre categorie principali:

- Sequenziali;
- Ad accesso calcolato (hash);
- Ad albero;

Strutture sequenziali

Nelle strutture sequenziali, un file è costituito da vari blocchi di memoria "logicamente" consecutivi, e le tuple vengono inserite nei blocchi rispettando una sequenza:

- **Seriale**: sequenza delle tuple indotta dall'ordine di immissione (organizzazione disordinata).
  Di solito viene chiamata anche **heap**, ossia mucchio
- **Array**: le tuple sono disposte come in un array, e la loro posizione dipende dal valore assunto in ciascuna tupla da un campo di indice.
  Possibile soltanto quando le tuple di una tabella sono di dimensione fissa.
- **Ordinata**: la sequenza delle tuple dipende dal valore assunto in ciascuna tupla da un

campo (attributo) del file, ossia la chiave.

Strutture ad accesso calcolato (hash)

Una struttura con accesso ad **hash** garantisce un accesso associativo ai dati, ovvero un tipo di accesso in cui la locazione fisica dei dati dipende dal valore assunto da un campo chiave. Questo avviene tramite specifiche funzioni hash che consentono di trasformare un attributo chiave nell'indice di un array, e quindi associare ad ogni record una posizione specifica in una struttura sequenziale. Un problema di questa tecnica è dovuto al fatto che l'insieme delle chiavi è molto più grande dell'insieme dei possibili valori dell'indice, è quindi sempre possibile che si generino collisioni, cioè valori diversi della chiave che portano allo stesso valore dell'indice. Una buona funzione hash rende bassa la probabilità che le collisioni si verifichino.
Va sottolineato che per definizione stessa di funzione hash, questa tecnica non è efficiente per ricerche basate su intervalli o ordinamenti. La forza 'hash infatti risiede nel determinare con complessità costante, data una certa chiave, il corrispondente valore (accessi puntuali). Trovare invece tutti gli elementi successivi (o in un certo intervallo) all'elemento corrispondete alla chiave inserita rende l'hash inutile.

Strutture ad albero

Le strutture ad albero, denominate anche **indici**, favoriscono l'accesso in base al valore di uno o più campi, consentendo sia accessi puntuali che corrispondenti a valori con complessità logaritmica (sulla base della profondità dell'albero).
L'organizzazione ad albero può essere utilizzata per realizzare sia strutture primarie, cioè strutture per contenere i dati, sia strutture secondarie, che favoriscono gli accessi ai dati senza peraltro contenere i dati stessi.

Indici primari e secondari

Iniziamo col dire che, in prima approssimazione, dato un file di dati f con un campo chiave k, un **indice secondario** è essenzialmente un file separato. In esso, ogni record è composto logicamente da due campi: il valore della chiave k e l'indirizzo fisico dove si trova quel dato nel file principale f. Essendo ordinato in base alla chiave, questo indice consente di fare ricerche rapide. Intuitivamente, si può pensarlo come l'**indice analitico** di un libro: un elenco di termini ordinati alfabeticamente che ti dice esattamente a che pagina andare.

Quando l'indice contiene al suo interno i dati, oppure quando è realizzato su un file che è già fisicamente ordinato in base a quella stessa chiave, in questo caso si parla **indice primario**, perché non garantisce solo l'accesso, ma vincola proprio l'allocazione fisica dei record. Un file può avere un solo indice primario (i dati possono essere ordinati fisicamente in un unico modo), mentre può avere più indici secondari. Continuando il paragone con i libri, possiamo considerare l'indice primario come l'**indice generale** (o sommario), che riflette l'ordine sequenziale dei capitoli e dei paragrafi.

In generale, un file può avere al più un indice primario e un numero qualunque di indici secondari (su campi diversi).
Senza entrare in dettagli troppo tecnici sulla struttura, vale la pena notare che questi "indirizzi" contenuti nell'indice possono puntare genericamente a un blocco di dati o, più specificamente, al singolo record. Sebbene i puntatori ai singoli record occupino più spazio, offrono il vantaggio di poter risolvere interrogazioni complesse leggendo solo l'indice, senza dover accedere al file dei dati se non alla fine.

Passando agli aspetti gestionali, una caratteristica fondamentale dell'indice primario è che può essere "sparso". Grazie al fatto che i dati nel file principale sono già ordinati, non serve che l'indice punti a ogni singola riga; è sufficiente che "punti" a un solo record per ogni blocco di dati (ad esempio il primo o l'ultimo). Poiché i valori sono consecutivi, il sistema sa che i dati intermedi si trovano necessariamente in quel blocco. Questo rende l'indice primario molto compatto, o "sparso", a differenza degli indici secondari che devono per forza contenere riferimenti a tutti i valori (ed essere quindi "densi").

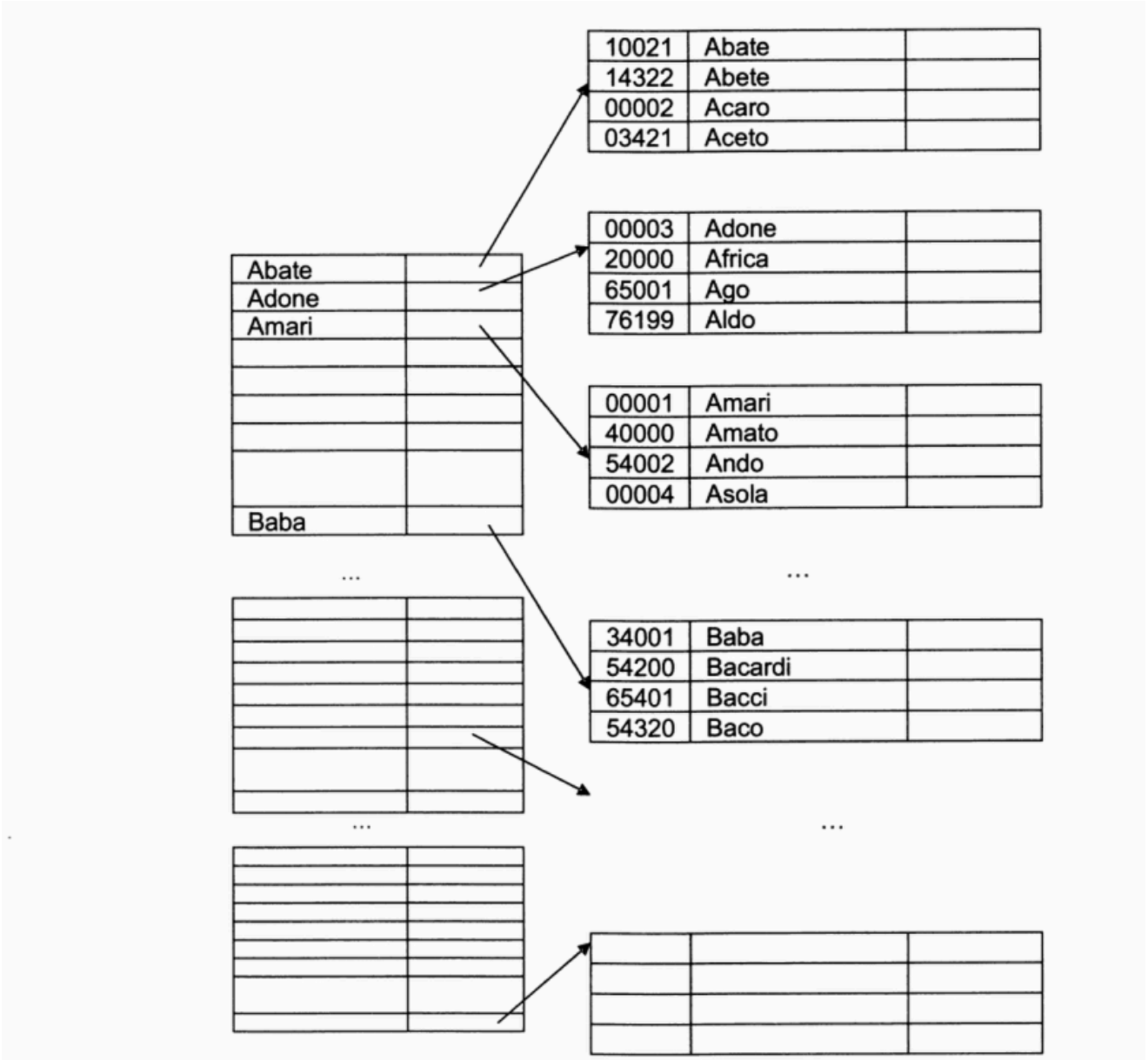


Figura 11.6 Un indice primario sparso.

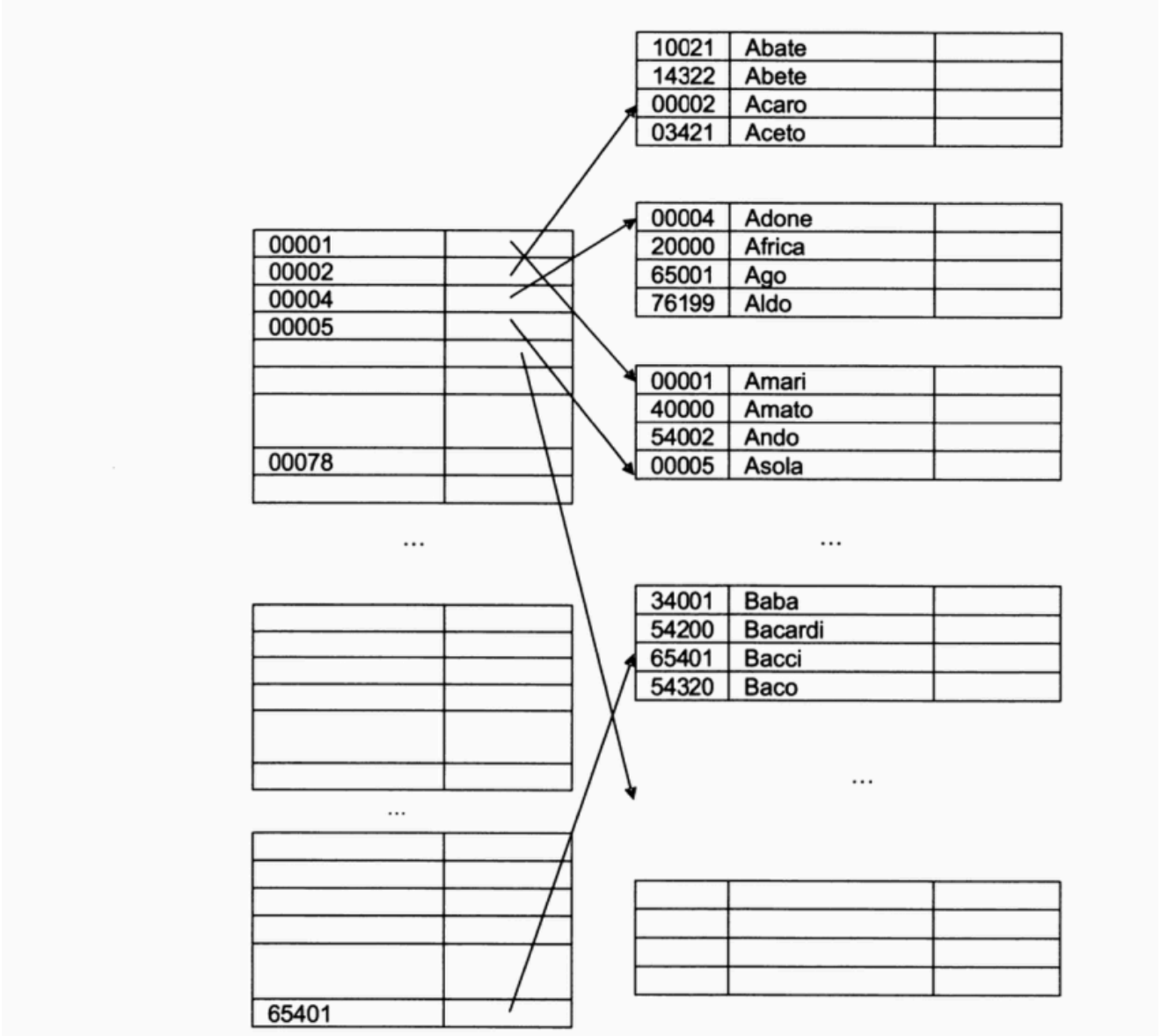


Figura 11.7 Un indice secondario denso.

L'efficienza è la ragion d'essere di queste strutture. Essendo file molto piccoli rispetto all'archivio dati completo, gli indici possono spesso essere caricati interamente nella memoria veloce (buffer), permettendo ricerche rapidissime. Mentre le strutture hash sono imbattibili per accessi puntuali (trovare un singolo dato specifico), gli indici si dimostrano superiori quando dobbiamo cercare intervalli di valori o scorrere i dati in ordine, operazioni in cui le hash risultano inefficienti.

Strutture ad albero dinamiche

Le strutture viste fin’ora sono basate su strutture ordinate e quindi poco flessibili in presenza di elevata dinamicità. Gli indici utilizzati dai DBMS sono più sofisticati in quanto utilizzano strutture ad albero dinamiche multi-livello, efficienti anche in caso di aggiornamenti.

Ogni albero è caratterizzato da un nodo radice, vari nodi intermedi e vari nodi foglia; ogni nodo coincide con una pagina o blocco a livello di file system e di gestore del buffer. I legami tra nodi vengono stabiliti da puntatori che collegano fra loro le pagine; in genere, ogni nodo ha un numero di discendenti abbastanza grande, che dipende dall'ampiezza della pagina

(non è raro il caso di alberi in cui ogni nodo ha decine o addirittura centinaia di successori); questo consente di costruire alberi con un numero limitato di livelli, nei quali la maggioranza delle pagine è occupata da nodi foglia. Un altro requisito importante per il buon funzionamento di queste strutture dati è che gli alberi siano **bilanciati** (il B-Tree classico), cioè che la lunghezza di un cammino che collega il nodo radice a un qualunque nodo foglia sia costante; in tal caso, il tempo di accesso alle informazioni contenute nell'albero è lo stesso per tutte le foglie ed è pari alla profondità dell'albero.

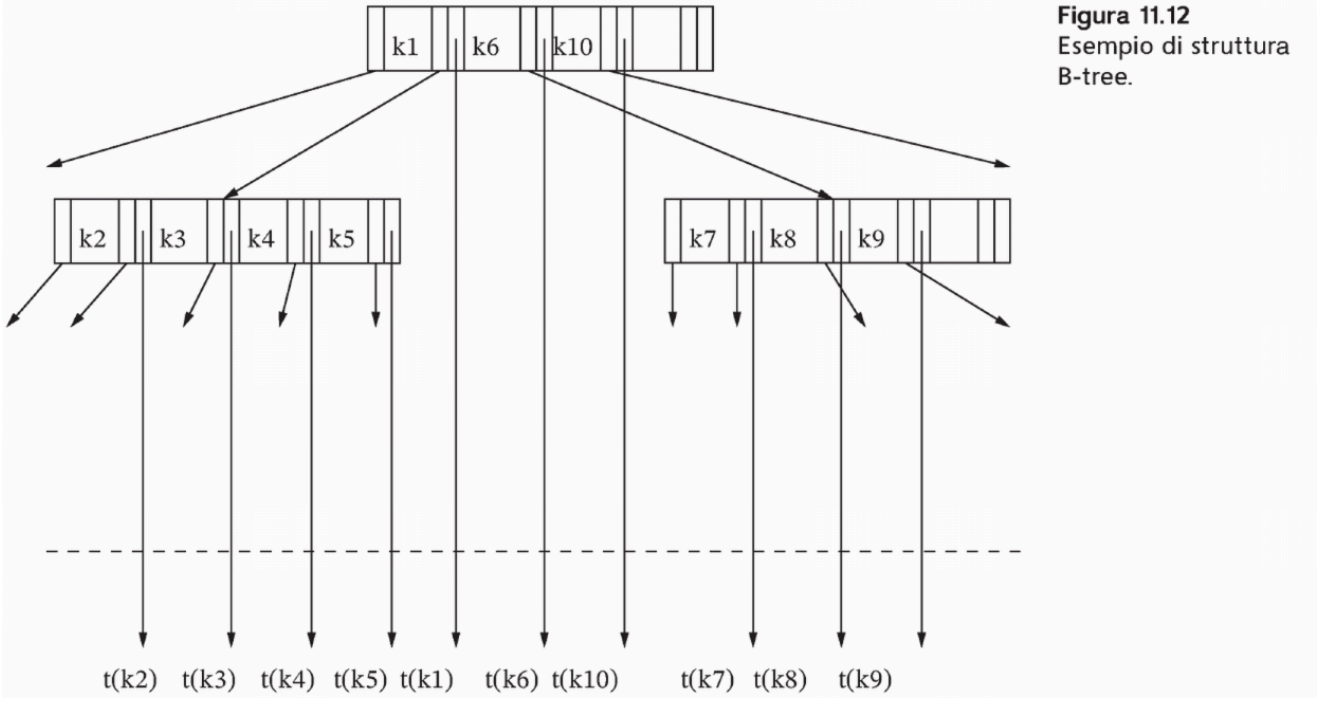
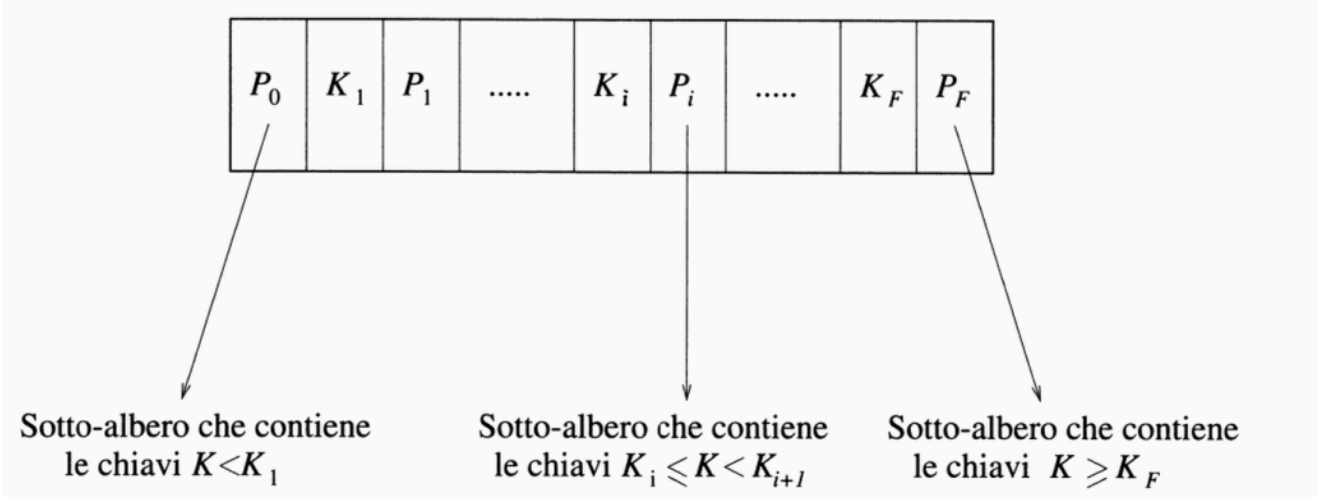


Figura 11.12
Esempio di struttura B-tree.

Contenuti dei nodi e tecnica di ricerca

Per capire come funziona un albero n-ario, è sufficiente analizzare la struttura di un suo qualsiasi nodo non foglia. Come si può vedere in figura, ogni nodo presenta una sequenza di F valori ordinati di chiave.



Ogni chiave K_i, con $1 \leq i \leq F$, è seguita da un puntatore P_i, mentre K_1 è preceduta da un puntatore P_0. Ciascun puntatore indirizza un sottoalbero così caratterizzato:

- il puntatore P_0 indirizza al sottoalbero che permette di accedere ai record con chiavi minori di K

- il puntatore P_f indirizza al sottoalbero che permette di accedere ai record con chiavi maggiori o uguali a K_f
- ciascun puntatore intermedio P_i, $0 < i < F$, indirizza un sottoalbero che contiene chiavi comprese nell'intervallo $[K_i, K_{i+1})$.

Rispetto all'albero binario, invece di un'etichetta e due puntatori, si hanno F etichette (valori di chiave) ed $F + 1$ puntatori.
A questo punto, dato un valore V, la *ricerca* viene effettuata seguendo i puntatori partendo dalla radice. Ad ogni nodo intermedio:

- Se $V < K_1$ si segue il puntatore P_0
- Se $V \geq K_f$ si segue il puntatore P_f
- Altrimenti si segue il puntatore P_j t.c. $K_j \leq V < K_{j+1}$
- La ricerca prosegue fino ai nodi foglia dell'albero.

Le operazioni di inserimento ed eliminazione di tuple provocano anche aggiornamenti degli indici, che devono riflettere la situazione generata da una variazione dei valori del campo chiave.
Un inserimento non provoca problemi quando è possibile inserire il nuovo valore della chiave in una foglia dell’albero. Quando invece la pagina della foglia non ha spazio disponibile, si rende necessaria un’operazione di split, che suddivide l’informazione già presente nella foglia e la nuova informazione in due, allocando due foglie al posto di una. Questo significa aggiungere un nuovo puntatore nel nodo padre del nodo foglia splittato. Ma, se il nodo padre non ha spazio sufficiente, lo split si propaga, fino, in casi estremi, alla radice dell’albero. Per evitare i casi estremi, quindi per ottenere degli aggiustamenti locali, si applica il criterio del riempimento parziale, in base al quale si cerca di riempire al massimo il 70% dell’intero albero.
Una cancellazione può essere sempre fatta localmente, marcando lo spazio precedentemente allocato a una tupla come invalido. In questo caso si potrebbe verificare il problema inverso all'inserimento. Infatti, dopo la cancellazione, il nodo foglia potrebbe presentare una quantità di spazi vuoti tale da richiedere un’operazione di merge, ovvero l’unione di due foglie in una unica. Analogamente all'operazione di split, anche quella di

merge potrebbe riverberarsi sui nodi precedenti, sino al raggiungimento della radice.

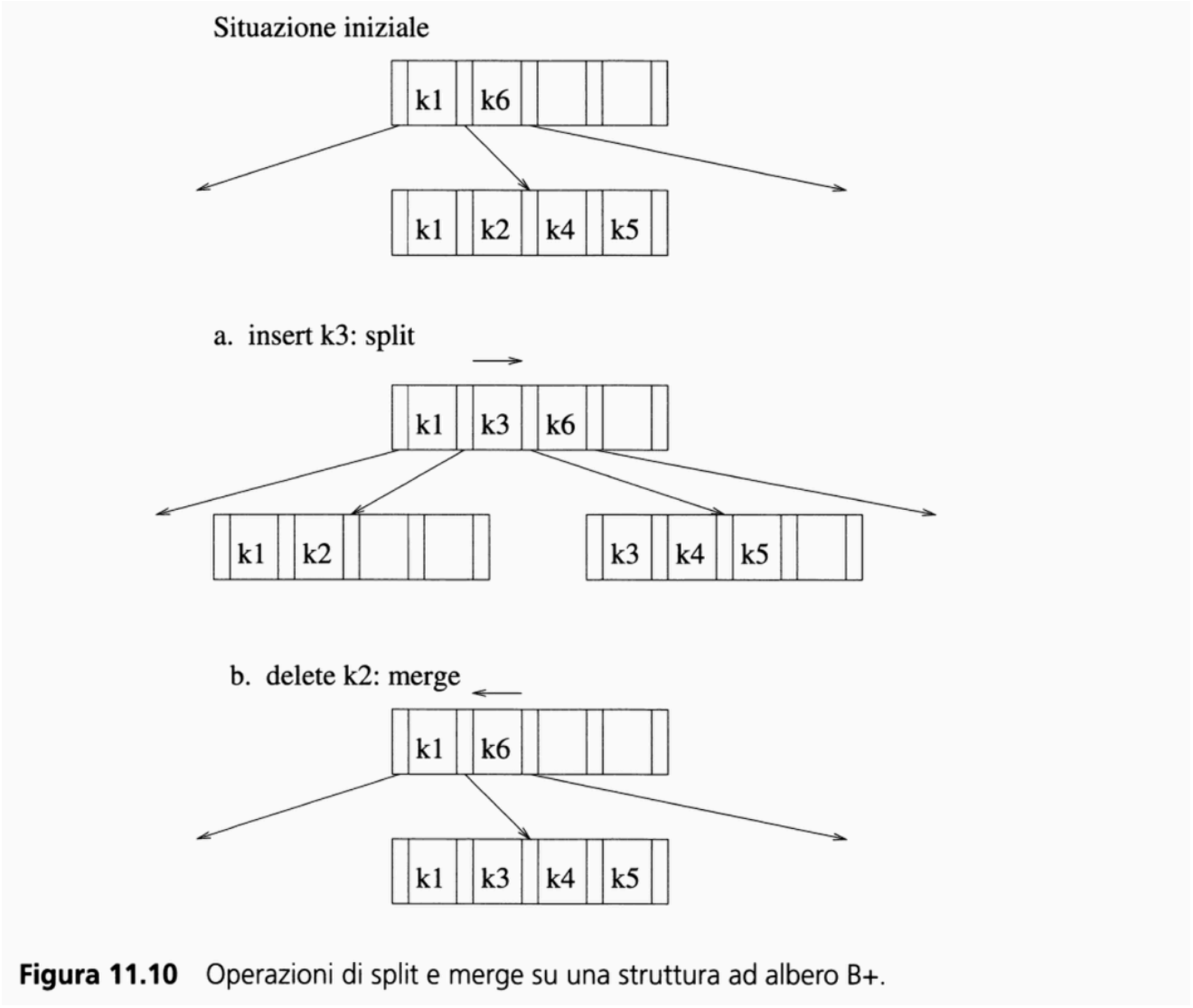
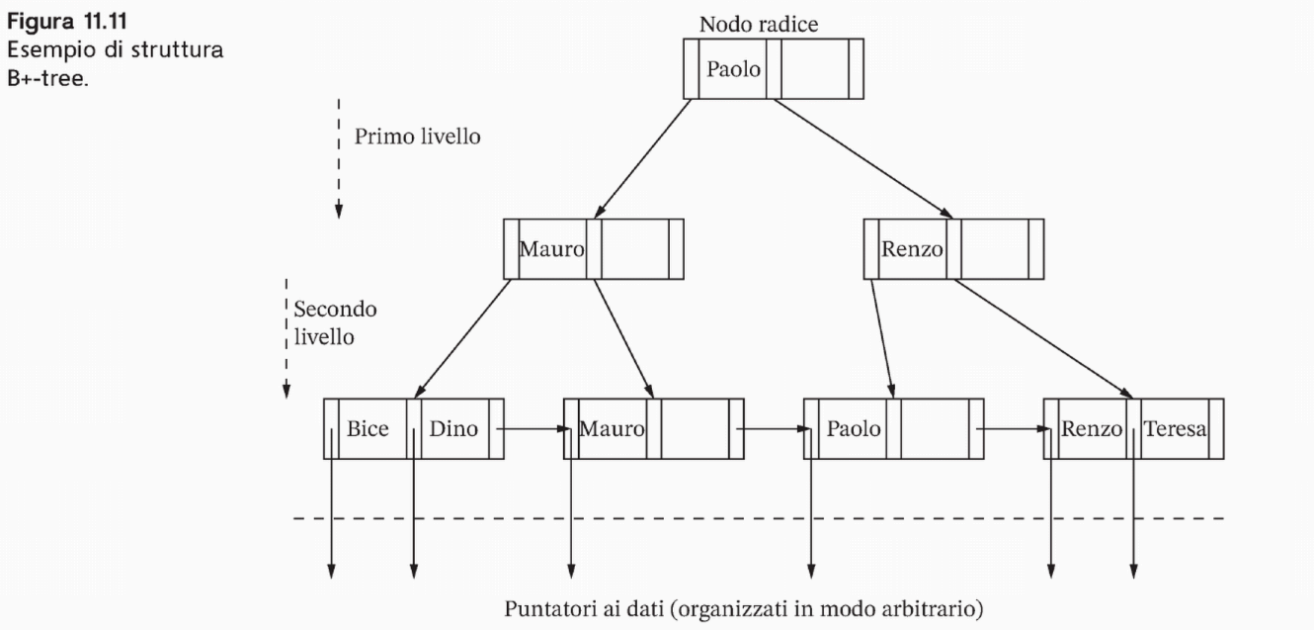


Figura 11.10 Operazioni di split e merge su una struttura ad albero B+.

B+-Tree

Per la struttura ad albero appena descritta esistono due versioni, denominate B e B+. L'unica differenza consiste nel fatto che negli alberi B+, i nodi foglia sono collegati da una catena che li connette in base all'ordine imposto dalla chiave. Tale catena consente di svolgere in modo efficiente anche interrogazioni il cui predicato di selezione definisce un intervallo di valori ammissibili. Per ottimizzare ulteriormente questo tipo di struttura si possono integrare dei puntatori ulteriori che collegano determinati nodi intermedi alle foglie

corrispondenti.



Definizione degli indici in SQL

In SQL, la sintassi del comando per la creazione di un indice è:

$$\text{CREATE [UNIQUE] INDEX } Nometabella \text{ on } Nometabella\ (ListaAttributi)$$

Con questo comando si crea un indice di nome $NomeIndice$ sulla tabella $NomeTabella$, operante sugli attributi elencati in $ListaAttributi$. UNIQUE è usato per indicare che $ListaAttributi$ è superchiave.
Per eliminare un indice è sufficiente eseguire il comando:

$$\text{DROP INDEX } Nometabella$$

Gestore delle interrogazioni: esecuzione e ottimizzazione

Il gestore delle interrogazioni è un modulo cruciale dell’architettura di un DBMS, in quanto responsabile dell’esecuzione efficiente di operazioni che sono specificate a livello molto alto. Esso riceve in ingresso un’interrogazione scritta in SQL, controlla che non vi siano errori lessicali, sintattici o semantici, una volta accettata, l’interrogazione viene tradotta in una forma interna di tipo algebrico. A questo punto, l’ottimizzazione vera e propria ha inizio, dividendosi in:

1. **Ottimizzazione algebrica**: effettua trasformazioni sulle operazioni (come l'anticipazione di selezioni e proiezioni verso le foglie dell'albero) che sono sempre convenienti indipendentemente dai costi fisici
2. **Ottimizzazione basata sul modello dei costi**: Sceglie la strategia di esecuzione (es. quale indice usare, quale algoritmo di join tra nested loop, merge scan o hash join) basandosi su un modello di costo che stima il numero di accessi in memoria secondaria e l'uso della CPU, sfruttando i profili statistici delle tabelle memorizzati nel catalogo (come spiegato nei profili delle relazioni);
3. Generazione del codice.

Profili delle relazioni

Ciascun DBMS commerciale possiede informazioni quantitative relative alle caratteristiche delle tabelle, organizzate in strutture dati, dette **profili delle relazioni**, che vengono memorizzate nel dizionario dei dati. I profili contengono alcune delle seguenti informazioni:

- La cardinalità di ciascuna tabella;
- Dimensioni di ciascuna tupla;
- Dimensioni di ciascun attributo;
- Il numero dei valori distinti di ciascun attributo;
- Il valore minimo e massimo di ciascun attributo.

I profili vengono calcolati in base ai dati effettivamente memorizzati nelle tabelle, utilizzando opportune primitive di sistema (es. update statistics) e sono utilizzati nella fase finale dell'ottimizzazione, per stimare le dimensioni dei risultati intermedi.

Ottimizzazione delle query

Sappiamo che una query può essere scritta in diversi modi, ma una volta ottimizzata, la query è l'unica possibile. Ora, però, l'esecuzione della stessa query ottimizzata può essere eseguita in diversi modi. Questo deriva dal fatto che, per quanto sia ottimizzata, la query è rappresentata in linguaggio di alto livello e di conseguenza utilizza operatori di alto livello che possono essere il risultato dell'unione di più operatori di basso livello. Ad esempio, per eseguire un'operazione join, si esegue un'operazione di ordinamento a sua volta composta da operatori di accesso diretto o di scansione (in base alla struttura fisica utilizzata per memorizzare i dati) di livello ancora più basso

Accesso diretto

Si usa il termine accesso diretto quando è possibile leggere o scrivere un record senza dover necessariamente esaminare il file in modo sequenziale, ma è possibile ottenere a partire dal valore di un campo l'indirizzo del blocco in cui il record si trova. Per eseguire l'accesso diretto è necessaria la presenza di una struttura hash o indice che lo permetta. Come sappiamo, gli indici favoriscono le interrogazioni che richiedono accessi puntuali oppure a intervallo. Si dice in tal caso che un predicato dell'interrogazione è **valutabile** tramite l'indice.
In generale:

- Se l'interrogazione presenta un solo predicato valutabile, allora c'è convenienza a usare l'indice o la struttura hash, consapevoli del fatto che per per un accesso puntuale è preferibile l'hash, su un intervallo è preferibile l'indice;
- Se l'interrogazione presenta una congiunzione di predicati valutabili tramite indice o funzione hash, il DBMS valuta i profili delle relazioni coinvolte e sceglie il metodo di accesso più selettivo;

- Se l’interrogazione presenta una disgiunzione di predicati è necessario che siano definiti indici o funzioni hash su tutti gli attributi affinché la query sia più ottimizzata possibile.

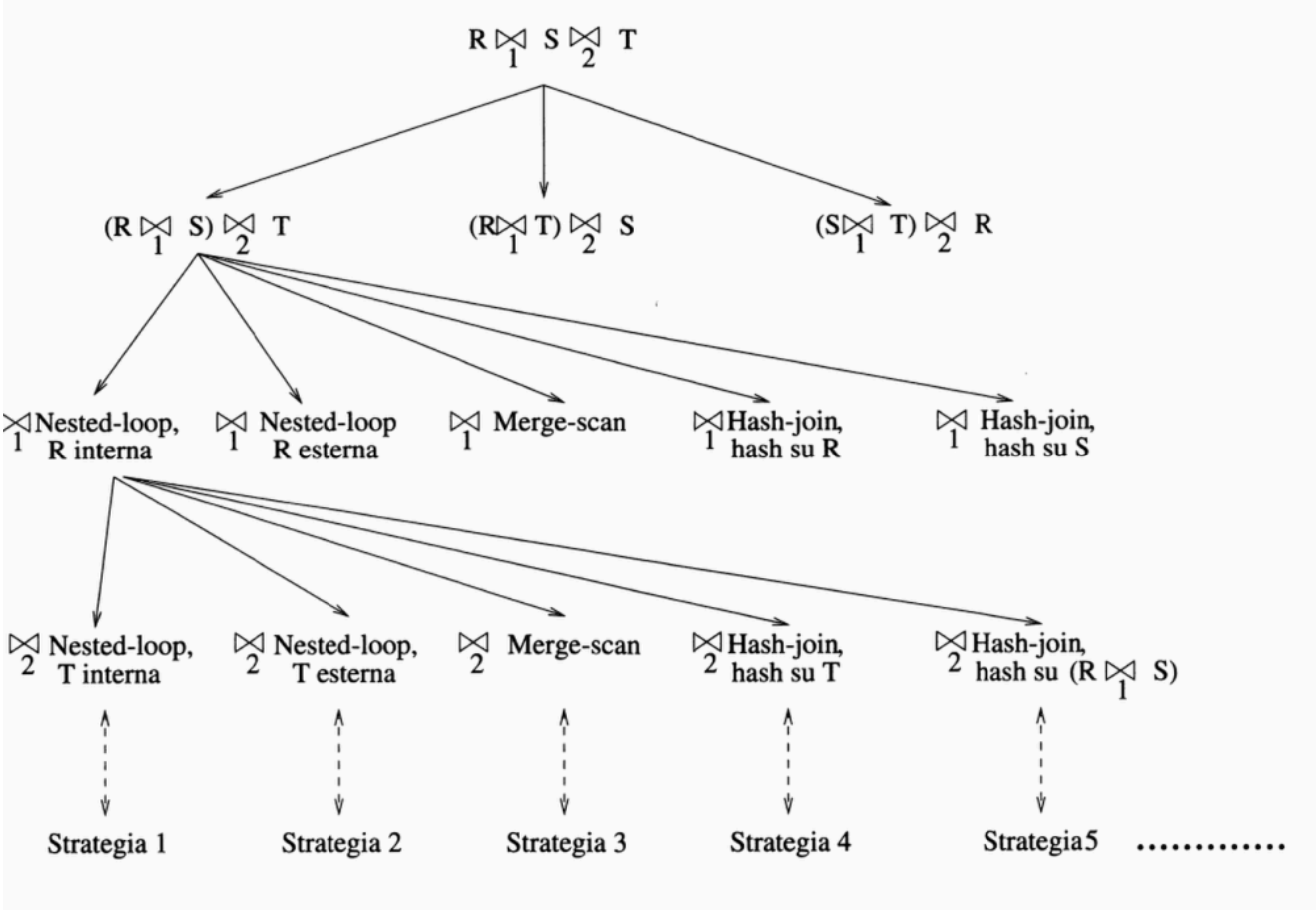
Da questo capiamo come, la medesima operazione di join sulle stesse identiche relazioni, sugli stessi identici attributi, a basso livello può essere eseguita in tre diverse modalità equivalenti: **bested-loop, merge-scan, hash-based**.

Ottimizzazione basata sui costi

Il problema di ottimizzazione basato sui costi è assai difficile, in quanto possono presentarsi varie dimensioni di ottimizzazione, con scelte relative a:

- Quale operazione di accesso ai dati svolgere (scansione o accesso diretto);
- Ordine delle operazioni da eseguire (in presenza di più join, va determinato in quale ordine eseguirli);
- Quando il sistema offre varie alternative per la realizzazione di un’operazione, occorre scegliere l’alternativa più adatta (quale metodo di join eseguire).

Difronte ad un problema tanto complesso, il DBMS costruisce un **albero di decisione** (o albero delle alternative), in cui ogni nodo intermedio corrisponde ad una particolare scelta di un particolare sotto-problema e ad ogni nodo foglia corrisponde una specifica strategia di esecuzione dell’interrogazione, descritta dalle scelte che si trovano percorrendo il cammino che va dalla radice al nodo foglia. Quindi, il problema di ottimizzazione è riformulato nella ricerca del nodo-foglia cui corrisponde il costo minore.



Progettazione fisica

La fase finale nel processo di progettazione di una base di dati è quella della progettazione fisica, che, ricevendo in ingresso lo schema logico della base dei dati, le caratteristiche del sistema scelto e le previsioni sul carico applicativo, produce in uscita lo schema fisico della base di dati, costituito da effettive definizioni delle relazioni (le istruzioni CREATE TABLE in SQL) e soprattutto delle strutture fisiche utilizzate con i relativi parametri.
La maggior parte delle scelte da effettuare nel corso della progettazione fisica dipende dal specifico DBMS utilizzato, quindi risulta difficile fornire una panoramica completa e di validità generale, ma esistono delle linee generali per delle basi di dati non enormi o con carichi non particolarmente complessi.
Le scelte fondamentali nella progettazione sono da ricondurre a due:

- Scelta della **struttura primaria** per ciascuna relazione, fra quelle disponibili dal DBMS
- Definizione di eventuali **indici secondari**

Per orientarci nelle scelte, è opportuno ricordare che le operazioni più delicate in una base di dati relazionale sono quelle di selezione (che corrisponde all'accesso a uno o più record sulla base dei valori di uno o più attributi) e di join (che richiede di combinare tuple di relazioni diverse sulla base dei valori di uno o più attributi di ognuna di tali relazioni).
Ciascuna delle due operazioni può essere eseguita in modo molto più efficiente se sui campi interessati è definito un indice (primario o secondario) o una struttura hash, rendendo così possibile un accesso diretto.
La definizione degli indici comporta, però, alcuni svantaggi. Infatti se è vero che accelerano le operazioni di ricerca, è anche vero che occupano memoria e soprattutto rallentano le operazioni di aggiornamento. Per questo motivo, per la definizione degli indici, è necessario seguire alcuni suggerimenti:

1. Non creare indici su tabelle piccole (meno di sei pagine)
2. Non creare indici su attributi poco selettivi, ovvero che hanno pochi valori diversi (sesso, stato civile etc.)
3. Evitare indici su attributi modificati di frequente
4. Prevedere più di quattro indici per relazione solo se le operazioni di modifica sono rare
5. Creare indici sulle chiavi esterne per agevolare l’esecuzione delle operazioni di giunzione.

Inoltre, se sull'applicazione ci sono frequenti query che ricorrono alle clausole ORDER BY, DISTINCT e GROUP BY, può avere senso definire indici rispetto agli attributi interessati dalle clausole stesse. Infatti l’esecuzione di interrogazioni con queste opzioni, un assenza di indici, comporta la creazione di tabelle temporanee da ordinare.