SISTEMI INFORMATIVI 2 – CAPITOLO 1

1.1 Memoria principale, secondaria e gestione dei buffer

Le basi di dati utilizzano la memoria secondaria per due motivi:

- Le dimensioni della memoria principale (RAM) seppur grande, non è sufficiente per un base di dati
- La memoria principale è volatile, mentre le basi di dati devono essere conservate anche con sistemi spenti

La memoria secondaria non è direttamente utilizzabile dai programmi. I dati, per poter essere usati, devono essere trasferiti nella memoria principale. Sui dischi e sugli altri dispositivi secondari i dati sono organizzati in *blocchi* di dimensione fissa che varia da supporto a supporto, compresa tra alcuni KB ad alcune decine di KB. L'unica operazione consentita è la lettura/scrittura di un unico blocco. L'accesso ad un unico bit è quindi uguale all'accesso ad un intero blocco, ed è comunque molto maggiore del tempo di accesso alla memoria principale. Il tempo di accesso ad un blocco dipende da 3 fattori:

- tempo di posizionamento della testina
- tempo di latenza (tempo tra la richiesta e la partenza della testina)
- tempo di trasferimento

Il trasferimento di grandi parti di dati salvate su blocchi contigui è quindi proporzionalmente molto più veloce dell'accesso ad un unico blocco.

L'interazione tra memori principale e secondaria è realizzata nei DBMS, con l'utilizzo di una zona della memoria centrale setta *buffer*. La gestione del buffer è fondamentale per l'accesso ai dati. Il buffer è organizzato in *pagine*, spazio di disco di dimensione pari ad alcuni blocchi. Il gestore del buffer carica/scarica pagine dalla RAM al disco. Il gestore è come un modulo che riceve richieste dai programmi e le esegue.

Nel caso della **lettura**, se la pagine è già presente nel buffer, non viene eseguita una nuova lettura fisica. Nel caso di **scrittura**, se il sistema è affidabile e la scrittura già presente, il sistema stesso può decidere di differirla. Il buffer viene gestito con il *principio della località dei dati*, cioè i dati recenti hanno più probabilità di essere riutilizzati. Il gestore del buffer deve contenere alcune informazioni:

- un *direttorio*, che descrive il contenuto indicando i file e il numero del blocco
- delle *variabili di stato*, dei contatori per sapere quanti programmi utilizzano la pagina, se la pagina è stata modificata (*bit di stato*) e se contiene dati validi

La *fix* (primitiva) viene usata per acceder alle pagine f funziona in questo modo:

1. cerca se la pagina è già presente in memoria. Se sì l'operazione viene conclusa. Altrimenti passa al punto 2;
2. viene scelta una pagina con contatore=0. Se la pagina è stata modificata viene salvata nella memoria di massa.
3. Se non ci sono pagine libere, può comportarsi in due modi:
    A. **steal**: viene sottratta una pagina ad un transazione e viene salvata sulla memoria di massa (*flush*)
    B. **no-steal**: non viene sottratta la pagina ad alcuna transizione; l'operazione viene messa in attesa di una pagina libera
4. viene incrementato il contatore

- La primitiva *set dirty* indica se la pagina è stata modificata
- Unifix indica che il modulo chiamante ha finito di usare la pagina (viene decrementato il contatore)
- Force trascrive in memoria di massa una pagina del buffer (*modo sincrono*)
La scrittura può anche avvenire in *modo asincrono* a seguito di una decisione del gestore del buffer.

Il **file system** è un modulo del SO e i DBMS usano la sua funzionalità. La relazione tra queste due funzioni è complicata. Attualmente i DBMS usano il file system ma si creano una propria astrazione dei file, per garantire efficienza. I DBMS usano il file system per la creazione e l'eliminazione dei file, per leggere e scrivere blocchi. Il DBMS crea un file di grosse dimensioni per memorizzare le relazioni. Gestisce i blocchi come se fossero un unico grande spazio.

1.2 Gestione delle tuple nelle pagine

In ogni pagina ci sono sia informazioni utili che di controllo (che servono per accedere ai dati veri e propri). Una possibile organizzazione delle pagine può essere la seguente:

- ogni pagina ha un **block header** (parte iniziale) e un **block trailer** (finale) con le informazioni di controllo per il file system
- ogni pagina gestita dal DBMS ha la **page header** (inizio) e la **page trailer** (fine) con le informazioni di controllo per il metodo di accesso
- ogni pagina ha il dizionario di pagina con i puntatori agli oggetti contenuti nella pagina, e i dati utili. Solitamente crescono entrambi come *stack* contrapposti
- ogni pagine contiene il *bit di parità* che indica la validità delle informazioni della pagina

Molti gestori di pagine non consentono la divisione di una tupa in più pagine. Mentre altri si.
Alcuni gestori creano tuple di ugual dimensione. Si facilizza la gestione, ma si occupa spazio inutile.
Le primitive del gestore di pagine sono le seguenti:

- **Inserimento e aggiornamento:** se i byte che devono essere inseriti hanno spazio sufficiente per essere scritti, non avviene alcuna riorganizzazione della pagina, altrimenti ne segue una ristrutturazione affinché i nuovi byte inseriti siano contigui
- **Cancellazione**: è sempre possibile e non ne segue alcune riorganizzazione
- **Accesso a una tupla**: è identificata dal valore della chiave o dell'offset
- **Accesso a un campo**: tramite valore della chiave o dell'offset

1.3 Strutture primarie per l'organizzazione di file

Le strutture sequenziali sono costituite da blocchi logicamente contigui e le tuple vengono inserite con una particolare sequenza:

- **heap** (organizzazione seriale) ordine di immissione
- **array** : disposte secondi un array in base al valore dell'indice
- **sequenziale ordinata**: in base al valore di un campo della tupla detto *chiave*

Struttura seriale
Le operazioni di caricamento e scrittura avvengono sequenzialmente e in blocchi adiacenti, quindi basta avere un puntatore all'ultimo blocco per poter eseguire l'operazione. Questa struttura è molto efficiente. Il problema di questo metodo è la *modifica* di una tupla. Se la modifica va ad aumentare le dimensioni del dato può causare problemi perché non è possibile cancellare il dato successivo al blocco perché è già occupato da altri dati. Per accedere a dei dati viene eseguita una scansione sequenziale di tutti i blocchi. Quindi per la ricerca di un singolo dato questa struttura non è molto

efficiente. Queste strutture sono molto diffuse nelle basi di dati relazionali perché facilmente gestibili e personalizzabili con indici per evitare continue scansioni sequenziali.

Struttura sequenziale ad array
Questa struttura è possibile nel caso che i dati della tabella abbiano dimensioni fisse ed uguali per ogni record. In questo caso al file viene associata una parte di memoria di n blocchi adiacenti, ed ogni blocco è dotato di m posizioni. Conoscendo il valore di n, moltiplicandolo per m sarà possibile trovare la posizione del dato di indice n. Le primitive di qusta struttura sono *read-ind* (lettura di una tupla di indice n), *insert-at, insert-near* e *insert-at-end* (inserimento di tuple in una determinata posizione) e le analoghe *update-ind e dolete-ind.*
Queste strutture, sebbene molto funzionali, non sono quasi mai utilizzate perché è raro che i dati siano di lunghezza fissa.

Struttura sequenziale ordinata
Questa struttura ordina le tuple in base al valore di un *campo* chiave. Questa struttura è molto funzionale grazie alla facilità di accesso ai dati conoscendo la chiave ma non è molto utilizzata a causa dei grandi costi di gestione. Principalmente questa struttura veniva utilizzata per salvare dei dati in un file principale sequenzialmente all'interno di un nastro. Le modifiche venivano salvate sempre sequenzialmente in file secondari, mentre in background avveniva periodicamente un aggiornamento (merge) del file principale. Il merge periodico con le tecniche attuali è inaccettabile. Scartata questa ipotesi di aggiornamento, le possibili soluzioni per il problema della modifica dei dati sono:
prevedere a priori delle zone libere per consentire solo un riordinamento locale dei blocchi
inserire dei blocchi a seguito delle saturazioni, evitando la perdita dell'ordine ma rinunciando alla contiguità dei blocchi
integrare il file ordinato con un file di overflow, che gestisce le tuple con dimensioni troppo elevate. I blocchi dell'overflow sono tra loro legati con una catena di overflow; ciascuna catena parte da un blocco del file ordinato. Le ricerche, in questo modo, devono anche interpellare il file di overflow

Strutture con accesso calcolato
Una struttura di questo tipo garantisce un accesso associativo ai dati, in cui cioè la locazione dipende dal *valore* assunto dalla chiave. La struttura viene organizzata associando ad ogni blocco un numero B di blocchi, tale che la tupla non riempia totalmente i blocchi assegnati. Il principio di questa struttura è l'estensione delle strutture ad array anche se l'array non è direttamente applicabile. Per esempio, se si vuole creare un array per un azienda con 10000 dipendenti con numero di matricola da 1 a 10000, ciò è efficiente utilizzando come chiave un intero a 5 cifre. Se invece gli operai fossero solo qualche decina, questa struttura non sarebbe ottimale perché verrebbe sprecato molto spazio. Se si volesse creare comunque un array, si potrebbe convertire la chiave in un indice dell'array attraverso una funzione di *hash* (in questo caso, per esempio, utilizzando il resto della divisione della matricola per il numero di operai). Questo però potrebbe portare a delle collisioni perché il resto della divisione di due diverse matricole potrebbe restituire lo stesso resto. Per questo motivo la funzione di hash è fondamentale per il corretto funzionamento di una struttura di questo tipo. Viene utilizzato un fattore di riempimento che indica la percentuale del blocco dedicato ad una tupla realmente occupato.

1.4 Strutture ad albero

Le strutture ad albero, denominate indici, consentono l'accesso ai dati tramite la chiave. Questa struttura permette di avere strutture primarie, cioè conteneti dati, e secondarie, ovvero senza; quest'ultima è eliminabile dinamicamente.

Indici primari e secondari

Dato un file f con campo chiave k, un indice *secondario* è un altro file contenente due campi; il primo uguale alla chiave k del file f, il secondo l'indirizzo fisico dei record di f con quella chiave k. Questo indice è ordinato secondo il valore della chiave k ed è utilizzato per un accesso più veloce ai dati. Quando invece questo indice contiene al suo interno dei dati anziché degli indirizzi, viene detto primario. Ogni file può avere un solo indice primario ed n indici secondari.

Per esempio l'indice primario è come l'indice generale di un libro, l'indice secondario è un sottoindice per chiave (esempio un indice di ristoranti o alberghi in una guida turistica).

L'indice contiene gli *indirizzi fisici* dei dati. Esso può essere relativo solo al blocco o allo specifico offset. Strutturalmente è identico, praticamente il puntatore al singolo record è più efficiente perché permette di accedere ai soli dati da recuperare.

Un indice primario può essere realizzato puntando a un solo record per ciascun blocco, in quanto gli altri record sono adiacenti al record puntato; si può scegliere nell'indice se puntare agli elementi minimi o massimi dei blocchi. L'indice viene chiamato sparso perchè non contiene tutti i valori delle chiavi presenti nel blocco. Gli indici secondari invece devono contenere tutti i riferimenti ai record perché record successivi nell'indice secondario non necessariamente saranno in blocchi adiacenti. Per questo vengono detti *indici densi*.

Gli indici sono molto utili perché consentono ricerche molto veloci e, contenendo solo i dati necessari all'individuazione del record, sono di dimensioni molto ridotte. Essendo ordinati permettono anche ricerche basate su intervalli, su cui le strutture ad *hash* invece sono inefficienti.

Strutture ad albero dinamiche

Queste strutture sono studiate in modo da essere efficienti in presenza di aggiornamenti. Sono strutturate da un nodo radice, seguito da vari nodi intermedi, terminati da nodi foglia. Il percorso per raggiungere ogni nodo foglia deve avere la medesima lunghezza (profondità). I DB costruiti con questa struttura hanno quindi un numero limitato di livelli e la maggioranza delle pagine e occupata da nodi foglia.

La ricerca all'interno di questa struttura di una chiave nota funziona in questo modo:

Si cerca nel nodo radice la chiave k (es. 1587) passata. Nel nodo radice, ad ogni chiave corrisponde un puntatore al sottonodo successivo.

Trovato l'intervallo in cui è presente la chiave che dobbiamo cercare, ovvero se la chiave è compresa tra il valore corrente e quello del record successivo (es. record corrente 1500, record successivo 1600), prendiamo il puntatore del record corretto e analizziamo il sottonodo.

La ricerca procede sempre ad intervalli, fino ad arrivare al nodo foglia in cui troveremo il valore esatto della chiave.

Esempio: Chiave da ricercare = 1063

Nodo radice

UNIVERSITA'	
Bergamo	1000
Milano	2000
Lecce	3000
…	4000

Nodo intermedio 1

FACOLTA'	
Ingegneria	1000
Economia	1100
Lettere	1200
…	1300

Nodo intermedio 2

INDIRIZZO	
Meccanica	1000
Tessile	1030
Informatica	1060
…	1090

Nodo foglia

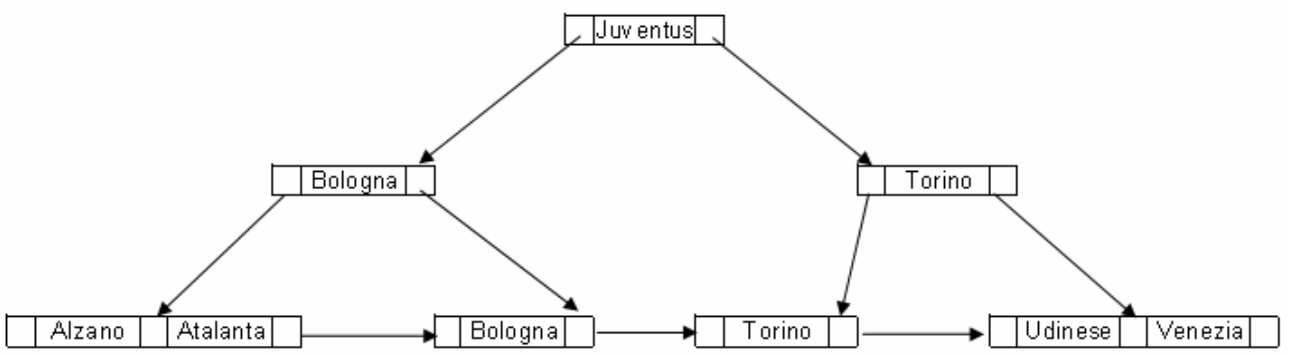
STUDENTE	
Bonfanti	1061
Pistocchi	1062
Mazzoleni	1063
…	1064

Viene cercata la chiave nel nodo radice. Fa parte dell'intervallo dell'università di Bergamo, quindi viene preso il puntatore del record Bergamo che punta al sottonodo facoltà. Qui la chiave risulterà essere compresa nell'intervallo ingegneria. Preso il puntatore di ingegneria che punta all'indirizzo,

si analizzerà il nodo intermedio 2. Verrà preso il puntatore di informatica che punta agli studenti (nodo foglia). La ricerca per chiave restituirà lo studente *Mazzoleni.*

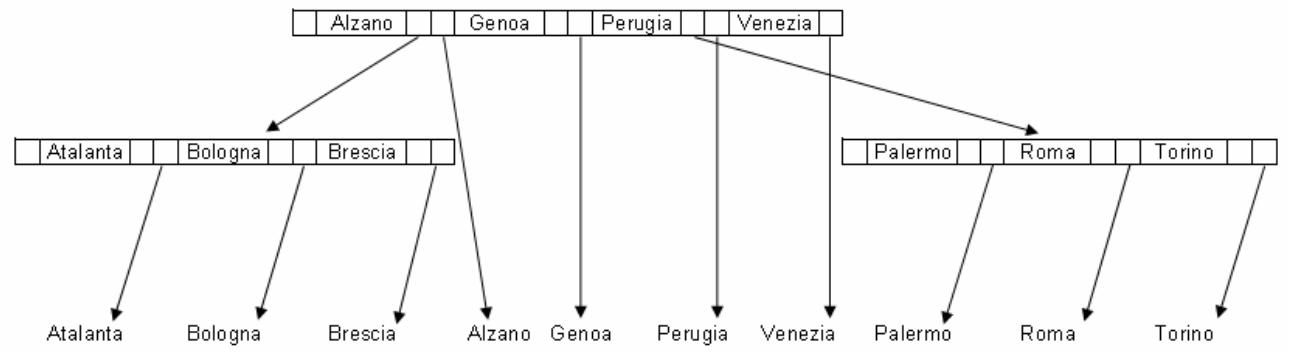
Esistono due diverse strutture ad albero dinamiche per la gestione della base dati. La prima contiene i dati direttamente nei nodi foglia (*key-sequenced)* e permette di organizzare un file ordinato rispetto al proprio indice primario; la seconda, nel nodo foglia, contiene il puntatore ai blocchi contenente i dati. In questo caso la struttura è detta indiretta.
Nel caso di un inserimento non esiste alcun tipo di problema se ci sono ancora pagine libere. Infatti l'indice viene aggiornato col nuovo valore della chiave, che viene recuperata dall'algoritmo di ricerca in caso di necessità. Nel caso non ci siano pagine libere, i dati vengono divisi in due parti e assegnati a due pagine. Verranno quindi cambiati i valori dei puntatori, perché dovranno puntare a più pagine anziché ad una solamente. Questa operazione viene chiamata **split.**
Nel caso di una cancellazione, basta che lo spazio occupato dalla tupla da eliminare venga marchiato come invalido. Sarebbe utile però andare a prendere il valore del record successivo e sostituirlo al dato cancellato, per ottimizzare lo spazio. Se due pagine contigue (sempre al livello) vengono lasciate in parte vuote, in modo tale che sia possibile ricomporre i dati di entrambe le pagine su una soltanto, viene operato un **merge** (inverso dello split).
Un buon utilizzo dei merge e degli split permette di occupare circa il 70% dello spazio effettivamente dedicato alla base dati. Ogni dato può essere presente più di una volta (per ottimizzare la ricerca può essere presente in indici e sottoindici).

I quadratini ai lati del dato sono i puntatori; quello a sinistra del dato punta al dato del sottoalbero con valore della chiave minore, quello a destra punta a un dato con chiave maggiore.



Alberi B e B+
B e B+ sono due versioni della struttura sopra descritta. La differenza sta nel fatto che nella struttura B+ i nodi foglia sono legati da una catene che li connette in base al valore all'ordine imposto dalla chiave. In questo caso è semplice effettuare ricerche anche per intervalli, trovando il primo risultato corrispondente al criterio della ricerca e scandendo ordinatamente i nodi foglia.
Nella struttura B-tree, i dati vengono organizzati secondo una struttura a sotto alberi. Nel nodo principale sono presenti solo alcuni valori, accompagnati da un puntatore che punta al dato stesso e al sotto albero con chiave inferiore. Ogni dato è contenuto una sola volta nel database, e per questo vengono risparmiate pagine nell'allocazione di dati.

I quadratini ai lati del dato sono i puntatori; il primo a destro del dato punta direttamente al sottoalbero con valore della chiave maggiore, quello più a destra punta al direttamente al dato.



1.5 Gestore delle interrogazioni

Questo gestore riceve in ingresso una richiesta in SQL e, dopo aver controllato la sua validità sintattica e lessicale, la trasforma in forma algebrica. L'ottimizzazione della richiesta è composta da tre fasi:

- Avviene una ottimizzazione algebrica, indipendentemente dal modello dei costi assunto dal sistema
- Viene svolta una ottimizzazione che dipende dal tipo di dati del livello sottostante e dal modello dei costi del sistema
- Infine viene generato il codice, cioè un programma di accesso in formato "oggetto" o "interno" che richiede l'uso delle strutture dati fornite dal sistema (indici, tabelle, ecc…)

Se una interrogazione viene compilata una sola volta per una esecuzione molteplice (compile-and-store) viene anch'essa salvata nella base dati con una indicazione delle *dipendenze* del codice dalle particolari tabelle o dagli indici. In questo modo, se la struttura della base dati cambia radicalmente e l'interrogazione appena salvata non può essere più eseguita, verrà ricompilata. Talvolta una interrogazione viene compilata e subito eseguita (approccio compile-and-go).

Profili delle relazioni

Ciascun DBMS commerciale contiene informazioni sulle tabelle, organizzate in strutture dette *profili delle relazioni*. I profili contengono le seguenti informazioni:

- La cardinalità, CARD(T) numero di tuple di ciascuna tabella T
- La dimensione in byte, SIZE (T) di ciascuna tupla di T
- La dimensione in byte, SIZE (A,T) di ciascun attributo A di T
- Il numero di valori distinti, VAL (A, T) di ciascun attributo A di T
- Il valore minimo, MIN (A, T) e quello massimo, MAX (A ,T) di ciascun attriubuto A di T

La primitiva update statistics calcola questi profili sui dati realmente contenuti nelle tabelle. Questi profili servono per una analisi statistica. Questa analisi è molto limitata perché non restituisce valori precisi, ma comunque sufficienti per calcolare approssimativamente le pagine che occuperà la base di dati e per svolgere una corretta ottimazione.

Rappresentazione interna delle interrogazioni

L'ottimizzazione tiene conto della struttura fisica utilizzata per l'implementazione della tabella. Consiste come prima cosa nel cambiare nodi foglia con nodi che tengono conto delle strutture fisiche, successivamente i nodi intermedi vengono trasformati in operazioni di accesso ai dati che sono supportate dalle strutture fisiche.

Operazioni di scansione
Un'operazione di scansione (scan) opera operazione algebriche e non algebriche:

- proiezione su una lista di attributi
- selezione su un predicato
- inserimenti, modifiche e cancellazioni delle tuple quando vi si fa accesso durante la scansione

Durante una scansione viene mantenuto un puntatore alla tupla corrente. Le primitive utilizzate sono:

- open, inizia la scansione
- next, procede la scansione avanzando il puntatore al record successivo
- read, legge la tupla corrente
- modify e delete, modificano o cancellano la tupla corrente
- insert, inserisce una tupla nella posizione corrente
- clos, chiude la scansione

Ordinamenti
Gli algoritmi per l'ordinamento di tuple nella base dati va studiato in base al tipo di supporto fisico da cui è contenuto il database e in base alle caratteristiche della struttura stessa

Accesso diretto
Il termine accesso diretto è utilizzato per descrivere il metodi di accesso ad un dato senza il bisogno di compiere una scansione totale dei dati, ma conoscendo esattamente la posizione del dato steso nella base dati. Quindi questo metodo è utilizzabile solo in strutture ad hash o ad indice che lo permettano. Le strutture ad indice consentono ricerche per attributi semplici (A=B) o per intervalli(A<B<C). Si dice che il predicato è *valutabile* tramite l'indice. Nelle strutture ad hash, invece, non è permessa la ricerca per intervalli.
Se una interrogazione per più predicati presenta una *congiunzione* dei predicati stessi, il DBMS deciderà di eseguire una scansione alla ricerca del predicato più significativo e, dopo aver caricato nel buffer le pagine che soddisfano la ricerca, eseguirà la scansione per l'altro predicato. Nel caso invece che i predicati siano *disgiunti* ma valutabili tramite indice o hash, verranno utilizzati ancora gli indici, mentre nel caso almeno uno dei due non sia valutabile, verrà effettuata ad ogni modo la scansione completa.