CAPITOLO 11

ORGANIZZAZIONE E GESTIONE DEI DATI

Nella maggior parte dei casi, le informazioni contenute in una base di dati sono registrate su supporti magnetici (dischi, cilindri, tamburi, …) della *memoria secondaria* (o "memoria di massa"). In questo capitolo riassumiamo per grandi linee le *tecniche di organizzazione fisica* comunemente usate per memorizzare il contenuto informativo (relazioni) di una base di dati, per rendere spedite le operazioni di aggiornamento e per ridurre quanto più possibile i tempi di attesa per le risposte alle interrogazioni dei suoi utenti. Dunque, nella progettazione fisica di una base di dati occorre tener conto di diversi fattori: la quantità di memoria occupata, il tipo e la frequenza degli aggiornamenti, il tipo di domande degli utenti che ricorrono più frequentemente, etc.

Cominciamo dalla memoria secondaria, di cui diamo una rappresentazione schematica, che è tutto quello che serve per illustrare le varie tecniche di organizzazione.

12.1 Memoria secondaria

Per *memoria secondaria* (o "memoria di massa") si intende il complesso di dispositivi magnetici (dischi, cilindri, tamburi, …) dove sono fisicamente registrate le relazioni di una base di dati. Possiamo rappresentare la memoria come una serie di *celle*, capaci ciascuna di contenere un'informazione di 1 *byte* (1 B = 8 *bit*); le celle sono numerate progressivamente da 0 ad M–1, dove M varia tipicamente tra 2^{24} byte e 2^{30} byte, e per ogni i, $0 \le i \le M$–1, diremo che la cella i-esima ha *indirizzo* i. Diremo anche che la *dimensione* della memoria secondaria è pari ad M byte.

La memoria secondaria è fisicamente suddivisa in *blocchi*. Un blocco è un insieme di celle fisicamente contigue e l'indirizzo della sua prima cella è anche il suo indirizzo. La dimensione D di un blocco varia tra 2^9 byte e 2^{12} byte. Pertanto, la memoria secondaria sarà composta da $b = M/D$ blocchi; così, se $M = 2^{24}$ byte e $D = 2^9$ byte, allora $b = 2^{15}$, mentre se $M = 2^{30}$ byte e $D = 2^{12}$ byte, allora $b = 2^{18}$. Si osservi che l'indirizzo di un blocco è sempre un multiplo di D (0, D, 2D, …) e un puntatore al blocco che ha indirizzo I è la codifica binaria di I. Tipicamente, al puntatore ad un blocco si riservano 4 byte che è quanto basta per coprire tutti i numeri da 0 a 2^{32}–1 e, quindi, per fornire l'indirizzo di uno dei b blocchi di cui si compone la memoria secondaria.

Le due tabelle che seguono riportano i multipli del byte come potenze di 10 e come potenze di 2.

Potenze del byte in base 10

1 *kilobyte*	1 kB	=	10^3 B
1 *megabyte*	1 MB	=	10^6 B
1 *gigabyte*	1 GB	=	10^9 B
1 *terabyte*	1 TB	=	10^{12} B
1 *petabyte*	1 PB	=	10^{15} B
1 *exabyte*	1 EB	=	10^{18} B
1 *zettabyte*	1 ZB	=	10^{21} B
1 *yottabyte*	1 YB	=	10^{24} B

Potenze del byte in base 2

1 *kibibyte*	1 KiB	=	2^{10} B	=	1,024 kB
1 *mebibyte*	1 MiB	=	2^{20} B	=	1,049 MB
1 *gibibyte*	1 GiB	=	2^{30} B	=	1,074 GB
1 *tebibyte*	1 TiB	=	2^{40} B	=	1,100 TB
1 *pebibyte*	1 PiB	=	2^{50} B	=	1,126 PB
1 *exbibyte*	1 EiB	=	2^{60} B	=	1,153 EB
1 *zebibyte*	1 ZiB	=	2^{70} B	=	1,181 ZB
1 *yobibyte*	1 YiB	=	2^{80} B	=	1,209 YB

Dunque, la dimensione M della memoria secondaria varia tra 2^{24} byte = 4 MiB (≈ 4 MB) e 2^{30} byte = 1 GiB (≈ 1 GB), mentre la dimensione D di un blocco varia tra 2^9 byte = 0,5 KiB (≈ 0,5 kB) e 2^{12} byte = 4 KiB (≈ 4 kB).

È compito del *sistema di controllo* del dispositivo di memoria tradurre l'indirizzo logico in un indirizzo fisico. Con 2^{18} blocchi abbiamo 2^{18} indirizzi; vale a dire che, per specificare l'indirizzo (logico) di un blocco servono almeno 18 bit; usualmente, l'indirizzo di un blocco occupa 4 celle, cioè ha una lunghezza di 4 byte e, per motivi di efficienza, l'indirizzo di ogni blocco è sempre specificato da un multiplo di 4.

L'*accesso* ai blocchi della memoria secondaria è regolato dal *sistema di controllo* della memoria secondaria: con un comando di *lettura* del blocco di indirizzo I, il

contenuto del blocco viene integralmente copiato in un'area riservata (*buffer*) di D celle contigue della memoria centrale (RAM), mentre con un comando di *scrittura* nel blocco di indirizzo I, il contenuto del buffer viene copiato nel blocco di indirizzo I. In entrambi i casi, l'operazione prende un tempo che è dell'ordine di decine o più di millisecondi dovuto al movimento di parti elettromeccaniche del dispositivo di memoria. D'altra parte, una tipica operazione di elaborazione nella memoria centrale (RAM) è di natura elettronica ed è dell'ordine di nanosecondi (dovuta alla velocità c di propagazione di segnali elettromagnetici).

Sottomultipli del *secondo* (s)

1 *millisecondo*	1 *ms*	$= 10^{-3}$ s
1 *microsecondo*	1 μs	$= 10^{-6}$ s
1 *nanosecondo*	1 *ns*	$= 10^{-9}$ s
1 *picosecondo*	1 *ps*	$= 10^{-12}$ s

Pertanto, l'*accesso* ad un blocco (scrittura o lettura) richiede molto ma molto più tempo di un'operazione di elaborazione nell'unità centrale. Per dare un'idea, è come se, per fare una breve colazione al banco del bar (che mettiamo richieda qualche minuto), si dovesse fare un viaggio della durata di qualche anno (!). Per questo enorme divario tra i tempi di accesso alla memoria secondaria e i tempi di elaborazione nell'unità centrale, da una parte si tenta di ridurre al minimo gli accessi ai blocchi e dall'altra si utilizzano due o più buffer cosicché, mentre si copia il contenuto di un blocco in un buffer, l'unità centrale può elaborare le informazioni contenute in un altro blocco il cui contenuto sia già stato copiato in un altro buffer.

12.2 *File*

Le relazioni contenute in (uno stato di) una base di dati vengono memorizzate in strutture di dati (*file*) residenti nella memoria secondaria.

Un *file* è un insieme di *record* che hanno tutti lo stesso *formato* e questo è definito da una sequenza di *campi*, ciascuno con un proprio *tipo di dato* (stringa, reale, intero, booleano, puntatore, …). Supponiamo che il formato dei record del file sia definito dalla sequenza $(C_1, \ldots, C_k)$ di campi. Così un record sarà formato da una sequenza $(c_1, \ldots, c_k)$, dove è un valore del campo compatibile con il tipo di dato di C_h, $1 \leq h \leq k$. Per esempio, se C_h è di tipo intero, c_h sarà un numero intero (nonnegativo) ed occuperà $l_h = 4$ byte; se C_h è di tipo puntatore, c_h di nuovo occuperà $l_h = 4$ byte; se invece C_h è di tipo stringa, c_h sarà una stringa ed occuperà un numero di byte l_h minore o uguale al massimo consentito dal tipo di C_h. Chiamiamo *lunghezza* del

record (c_1, …, c_k), la somma $L = l_1 + … + l_k$. Ne viene che due distinti record (c_1, …, c_k) e (c'_1, …, c'_k), possono avere lunghezze diverse, nel qual caso i record si dicono a *lunghezza variabile*. D'ora in poi, per semplicità, assumiamo che i record del file abbiano tutti la stessa lunghezza.

Come per una relazione contenuta in uno stato della base di dati, possiamo aggiornare un file

aggiungendo un record,

cancellando o modificando uno o più record

oppure cercare dei particolari record di un file.

La selezione dei record da cancellare, da modificare o da cercare si basa su una qualche condizione su un dato insieme X di campi; per esempio, la condizione che X assuma un dato valore oppure un valore appartenente ad un dato insieme. L'insieme X è chiamata la *chiave della ricerca*.

Nel valutare il costo computazionale di ognuna delle operazioni terremo conto soltanto delle operazioni di trasferimento del contenuto di uno o più blocchi nel buffer della memoria centrale e viceversa, cioè del numero di accessi alla memoria secondaria.

12.4 *Configurazione dei blocchi*

Il file è memorizzato in un certo numero di blocchi della memoria secondaria ed è compito del *file system* riservarne un numero sufficiente e individuarli tra quelli non impegnati da altri file. Supponiamo che il file sia memorizzati in n blocchi, B_1, …, B_n. Un'apposita struttura dati (ad esempio una lista) tiene traccia degli indirizzi dei blocchi B_1, …, B_n (block directory del file), un'altra contiene la descrizione del file (data dictionary); entrambe, se non occupano troppo spazio, sono tenute permanentemente nella memoria centrale per tutto il tempo che il file viene usato. Inoltre, usualmente i blocchi B_1, …, B_n sono *concatenati* nel senso che in ogni blocco B_i c'è una zona di celle (LINK) che contiene l'indirizzo del blocco B_{i+1} se $i < n$, altrimenti un valore convenzionale (*valore nullo*) che sta a segnalare che B_n è l'ultimo blocco.

Supponiamo di voler memorizzare un certo numero di record in uno dei blocchi che il *file system* abbia assegnato al file. In linea teorica, il blocco potrà ospitare non più di $P = \lfloor D/l \rfloor$ record, dove D è la dimensione del blocco ed L è la lunghezza di ciascun

record. Chiameremo P la *capacità* del blocco in termini di record del file. Così, se $D = 2^{12}$ byte e $L = 20$ byte, allora $P = 204$.

A tale scopo, i blocchi che il *file system* ha riservato al file vengono tutti suddivisi in zone (sub-blocks), che per comodità chiamiamo *pagine*, tutte della stessa lunghezza l dei record del file. La *posizione di una pagina* all'interno di un blocco è data dal numero delle pagine di quel blocco che la precedono. Pertanto, una pagina che occupa la posizione p, $0 \leq p \leq P-1$, in un blocco che ha indirizzo I si estende dalla cella di indirizzo $I+pL$ alla cella di indirizzo $I+(p+1)L-1$. Inoltre, ogni pagina è suddivisa in segmenti, che per comodità chiamiamo *linee*, della stessa dimensione dei campi dei record del file. La *posizione di una linea* di una certa pagina all'interno di un blocco è data da $pL + l$ dove p è il numero delle pagine di quel blocco che precedono quella pagina e l è la somma delle lunghezze delle linee che precedono quella linea è data dal numero delle linee di quella pagina che la precedono. Infine, una zona (LINK) del blocco è riservata all'indirizzo dell'eventuale blocco successivo; in sua assenza, LINK conterrà un valore nullo.

12.4 *Formattazione delle pagine*

Ogni pagina è suddivisa in segmenti, uno per ogni campo del record. Per semplicità, anche questi segmenti li chiamiamo *campi*.

Consideriamo ora un'operazione di cancellazione. Supponiamo di voler cancellare un record r memorizzato in una certa pagina. La maniera più ovvia sarebbe quella di avere un ulteriore campo info (1 byte) che riporta lo *stato* di quella pagina, cioè se va considerata *libera* o *occupata*. Così per cancellare r basterebbe modificare lo stato della pagina da occupata a libera, così che essa possa essere riutilizzata per registrarvi un nuovo record quando se ne presenti l'occasione. Ma cosa avviene se un altro record (dello stesso file o di un altro file) contiene un puntatore che rimanda proprio al record r. In tal caso, se si andasse a registrare un nuovo record r' del file nella stessa pagina che ospitava il record r, tutti i puntatori ad r verrebbero allora erroneamente imputati ad r'. Per ovviare a questo inconveniente, si considera il record r "appuntato" (pinned) alla pagina, cosicché la pagina resta impegnata dal record r anche dopo che questo sia stato cancellato ed allora tutti i puntatori ad r restano per così dire "in sospeso" (dangling). Vanno dunque distinte le pagine occupate da record appuntati cancellati da quelle occupate da record del file "in uso". A questo scopo, il campo info deve contenere anche un *bit di cancellazione* con valore "1" se il record va considerato "in uso" (cioè facente parte del file) e "0" se va considerato "non in uso". Allora per cancellare il record r, basterà modificare il bit di cancellazione nel campo info da 1 a 0. Poi, si modifica lo stato della pagina da occupata a libera solo se non esistono puntatori al record r.

Infine, il campo info contiene anche un bit “1” oppure “0” a seconda che il blocco contenga o meno nelle pagine successive altri record del file.

Supponiamo di voler memorizzare una relazione che contiene i primi *N* numeri di Fibonacci

(1 1 2 3 5 8 13 21 34 55 …)

Assumiamo che la relazione abbia schema {progressivo, numero e cifre}, e che progressivo sia la chiave primaria. Ovviamente, il file, che per comodità chiamiamo Fibonacci, sarà fatto di record il cui formato contiene i tre “campi/attributi” progressivo, numero e cifre:

progressivo	4 B	il numero progressivo del numero di Fibonacci
numero	4 B	un numero di Fibonacci
cifre	4 B	numero di cifre decimali

con l’aggiunta del campo info (1 byte). Così, fino a questo punto, la lunghezza di un record è di 12 byte. Comunque, per motivi di efficienza nella gestione del file, si fa in modo che

- le pagine siano “allineate” nel senso che la posizione della prima linea di ogni pagina sia un multiplo di 4,
- i campi che contengono dati numerici o puntatori siano “allineati” nel senso che la posizione di ogni campo sia un multiplo di 4.

A questo scopo, non solo va scelta una conveniente sequenza dei campi ma va anche previsto di aggiungere *campi non significativi* (padding). Un possibile formato dei record del file Fibonacci (e, quindi, di una pagina di un blocco riservato al file) è dunque il seguente

Formato di un record del file Fibonacci

campo	*lunghezza*
info	1 B
spazio (campo non significativo)	3 B
numero	4 B
progressivo	4 B
cifre	4 B

cosicché la lunghezza del record è $L = 16$ byte.

12.5 *Configurazione dei blocchi*

Supponiamo che un blocco di dimensione *D* sia riservato ad un file i cui record hanno lunghezza *L*. Come si è detto, ogni blocco sarà suddiviso in $P = \lfloor D/L \rfloor$ pagine, a cui seguirà la zona LINK (di quattro byte).

Nel caso del file `Fibonacci`, se $D = 2^9$ (= 512) byte, allora ogni blocco sarà suddiviso in $P = 32$ pagine e la sua configurazione sarà la seguente

Configurazione di un blocco

PRIMA PAGINA

campo	*lunghezza*	*posizione*
info	1 B	0
spazio	3 B	1
numero	4 B	4
progressivo	4 B	8
cifre	4 B	12

SECONDA PAGINA

campo	*lunghezza*	*posizione*
info	1 B	16
spazio	3 B	17
numero	4 B	20
progressivo	4 B	24
cifre	4 B	28

…

ULTIMA PAGINA

campo	*lunghezza*	*posizione*
info	1 B	496
spazio	3 B	497
numero	4 B	500
progressivo	4 B	504
cifre	4 B	508

LINK	4 B	512

Si osservi che la posizione della prima linea di ogni pagina, così come le posizioni dei campi `numero`, `progressivo` e `cifre` sono sempre multipli di 4, e lo stesso dicasi per la posizione di LINK.
Infine, il numero n di blocchi necessari a memorizzare gli N record del file `Fibonacci` non è inferiore ad $n = \lceil N/P \rceil$ dove P è il numero di pagine di cui si compone un blocco. Così per $N = 10000$ e $P = 32$, abbiamo $n = 313$.
Questa configurazione non è molto efficiente se si vuole individuare una pagina libera dove memorizzare un nuovo record perché occorre scorrere una alla volta le pagine fino a trovarne una in cui il campo `info` contenga lo stato della pagina uguale ad "1". Si può fare di meglio riservando all'inizio del blocco una zona di uno o più byte che contenga un array dove siano riportati gli stati delle 32 pagine del blocco.

Per riferirsi ad un record dovremo specificare l'indirizzo I del blocco che contiene la pagina dove è memorizzato nonché il valore x che in quel record ha la chiave primaria del file. In alternativa, possiamo riferirci al record specificando $I + p$ dove p è la posizione della prima linea della pagina che contiene il record.

12.6 Organizzazione non-ordinata (heap)

Con questa tecnica i record vengono ammassati l'uno dopo l'altro. Supponiamo che $B_1, \ldots, B_n$ sia la catena dei blocchi impegnati dal file. Il blocco di testa B_0 del file conterrà sia l'indirizzo del primo blocco (B_1) che quello dell'ultimo (B_n). In "condizioni normali" per ogni $i < n$ il blocco B_i è pieno nel senso che non contiene pagine libere.

Aggiunta. Il record da aggiungere viene memorizzato nella prima pagina libera dell'ultimo blocco B_n, ammesso che ve ne sia una. Se questo è il caso, sono necessari solo 2 accessi alla memoria secondaria: uno per trasferire il contenuto di B_n in memoria centrale, l'altro per copiare il contenuto aggiornato di B_n nella memoria secondaria. Nel caso però che tutte le pagine di B_n siano occupate, viene richiesto al *file system* un nuovo blocco B e il record viene memorizzato nella prima pagina di B. Va da sé che, se tutte le pagine di B_n erano occupate, allora nel blocco di testa B_0 l'indirizzo del blocco B prenderà il posto dell'indirizzo del blocco B_n e la zona LINK di B_n conterrà anche l'indirizzo del blocco B.

Ricerca. Consideriamo una selezione di record con chiave di ricerca X. Per selezionare i record, viene effettuata una *ricerca lineare* dei blocchi: si esamina il primo blocco B_1 del file (il cui indirizzo è contenuto in B_0) e poi, via via, i blocchi successivi $B_2, \ldots, B_n$ tenendo a mente che l'indirizzo del blocco B_i è contenuto nella zona LINK del blocco B_{i-1} per $i > 1$. Noto l'indirizzo di un blocco, si procede alla maniera seguente. Inizialmente, l'intero contenuto del blocco viene copiato nel buffer della memoria centrale e una variabile intera i viene inizialmente posta uguale ad 1. Il contenuto della copia della pagina i.esima del blocco, cioè il contenuto del buffer dalla posizione $(i-1)l$ alla posizione $il-1$, viene copiato in una variabile v di tipo record (in un'area di lavoro della memoria centrale). Quindi, prima si controlla se la pagina è libera e, se così, se il record sia in uso (esaminando il bit di cancellazione del campo info); solo in tal caso, si procede a verificare se il contenuto dei campi in X della variabile v soddisfa o meno il criterio di selezione. Se sì, solo allora il record viene "selezionato". Quindi, si esamina il contenuto del campo info della variabile v

per sapere se il blocco contenga o meno altri record del file nelle pagine successive. Se non ve ne sono, allora la ricerca in quel blocco termina; altrimenti, si incrementa il valore di i di un'unità e si passa ad esaminare la pagina successiva. Per quanto detto, siccome i record che soddisfano il criterio di selezione possono essere in uno qualsiasi degli n blocchi, la ricerca richiede

n accessi

alla memoria secondaria.

A titolo d'esempio, consideriamo il file `Fibonacci` con N = 10000 record e supponiamo che i record siano stati memorizzati senza nessun ordine in 313 blocchi. Allora, la ricerca dei record che contengono i numeri di Fibonacci dal p.esimo al q.esimo, cioè dei record che soddisfano la condizione

$$p \leq \texttt{progressivo} \leq q$$

richiede sempre 313 accessi. Analogamente, la ricerca dei record che contengono i numeri di Fibonacci compresi in un dato intervallo [a, b], cioè che soddisfano la condizione

$$a \leq \texttt{numero} \leq b$$

richiede ancora 313 accessi.

Un caso a sé si ha quando la chiave di ricerca X è una chiave identificativa del file e il criterio di selezione è della forma $X = x$ perché, allora, la ricerca si può considerare conclusa non appena si sia trovato un record che soddisfi il criterio di selezione. Nel *caso peggiore*, il record si trova nell'ultimo blocco B_n oppure il file non contiene nessun record che soddisfa il criterio di selezione; in tal caso, il numero degli accessi alla memoria secondaria è pari ad n. Per valutare il numero di accessi nel *caso medio*, si ragiona come segue. Se il record si trovasse nell'i.esimo blocco B_i ($1 \leq i \leq n$), allora si dovrebbero esaminare i blocchi B_1, …, B_i e, quindi, sarebbero necessari i accessi. Sia π_i la probabilità che il record sia nel blocco B_i; allora, il numero medio a di accessi è dato da

$$a = \sum_{i=1,\ldots,n} i\, \pi_i \,.$$

Ora, se N è il numero dei record (in uso) del file ed N_i è il numero di record (in uso) che sono memorizzati nel blocco B_i, allora $\pi_i = N_i/N$ cosicché il numero medio di accessi alla memoria secondaria è

$$\sum_{i=1,\ldots,n} i\, (N_i/N) \,.$$

In condizioni normali, i blocchi $B_1, \ldots, B_{n-1}$ sono tutti pieni (non hanno pagine libere) cosicché $N_1 = \ldots = N_{n-1}$. Supponiamo ora, per puro esercizio, che anche il blocco B_n sia pieno e, quindi, che i record del file siano distribuiti in modo uniforme tra gli n blocchi; dunque, $N_1 = \ldots = N_n = N/n$. Allora, si ha che $\pi_i = N_i/N = 1/n$ e, quindi, il numero medio di accessi alla memoria secondaria è

$$a = (1/n) \sum_{i=1,\ldots,n} i = (n+1) / 2 \approx n/2.$$

Consideriamo, di nuovo, il file `Fibonacci` e supponiamo di voler cercare il p.esimo numero di Fibonacci, cioè quello che soddisfa la condizione

`progressivo` = p

Siccome `progressivo` è la chiave primaria del file, il numero medio di accessi è pari a circa $n/2 = 156$.

Cancellazione. La cancellazione dei record che soddisfano una condizione selettiva, viene effettuata con una ricerca lineare. Una volta individuati in un blocco i record da cancellare, basterà modificarne il bit di cancellazione e copiare in memoria secondaria il contenuto aggiornato del blocco. Pertanto, nel caso peggiore l'operazione richiede $2n$ accessi alla memoria secondaria.
Si osservi che se, dopo la cancellazione, un blocco si trova a contenere solo pagine libere, allora il blocco viene restituito al *file system* con la conseguente modifica della catena dei blocchi (modifica del campo LINK nel blocco precedente). Inoltre, periodicamente, il file viene compattato: in ogni blocco, fatta eccezione dell'ultimo, che contenga pagine libere (ma non tutte), si copiano altrettanti record presenti nell'ultimo blocco. Va da sé che, se l'ultimo blocco viene svuotato, allora il blocco viene restituito al *file system*. Dunque, come si diceva all'inizio, in *condizioni normali* tutti i blocchi della catena, ad eccezione dell'ultimo, sono pieni.

Qualora la chiave di ricerca X sia una chiave identificativa del file e il criterio di selezione sia della forma $X = x$ allora, come si è visto, la ricerca del record da cancellare richiede nel caso peggiore un numero degli accessi alla memoria secondaria pari ad n, e nel caso medio ne serviranno $\approx (n/2)$. Dunque, la cancellazione richiede nel caso peggiore un numero degli accessi alla memoria secondaria pari ad $n+1$, e nel caso medio $\approx (n/2) +1$.

Modifica. Come per la cancellazione, il costo dell'operazione in termini di numero medio di accessi alla memoria secondaria è pari a $2n$. Se poi la chiave di ricerca X è una chiave identificativa del file e il criterio di selezione è della forma $X = x$, allora la modifica richiede nel caso peggiore un numero degli accessi alla memoria secondaria pari ad n+1, e nel caso medio $\approx (n/2) + 1$.

12.4 Organizzazione modulare (hash)

Con questo tipo di organizzazione i record del file sono ripartiti in parti o *moduli* (bucket) in base al valore di un insieme di campi X, che chiamiamo *base della ripartizione*. A tale scopo provvede una *funzione di ripartizione* H (hashing function) che associa ad ogni possibile valore x di X un numero $h = H(x)$ compreso tra 0 ed m–1, dove m è un intero prefissato. Così, il modulo h.esimo conterrà tutti e solo i record del file (ammesso che ve ne siano) in cui X assume il valore

$$x = H^{-1}(h).$$

Ogni modulo viene fisicamente memorizzato come fosse un file non-ordinato e un *indice dei moduli* (bucket directory) riporta per ogni h ($0 \le h \le m$–1), gli indirizzi (eventualmente lo stesso valore nullo dei campi LINK se il modulo h.esimo è vuoto) del primo e dell'ultimo della catena dei blocchi impegnati dal modulo h.esimo. Ora se m non è eccessivamente grande, l'intero indice dei moduli (che ha dimensioni proporzionali ad m) può risiedere permanentemente in memoria centrale durante l'utilizzo del file.

Se indichiamo con N_h (≥ 0) il numero dei record nel modulo h.esimo, $0 \le h \le m$–1, allora serviranno $n_h = \lceil N_h/P \rceil$ blocchi, dove P è il numero delle pagine di un blocco, per un totale di $n = n_0 + \ldots + n_{m-1}$ blocchi. Particolare attenzione si pone alla scelta della funzione di ripartizione f, che nel caso ottimale distribuisce gli N record del file negli m moduli in maniera quanto più possibile uniforme. Nel caso ottimale $N_0 = \ldots = N_{m-1} = N/m$ cosicché per ogni modulo serviranno $\lceil N/mP \rceil$ blocchi.

Per ottenere una equi-distribuzione, si usa spesso come funzione di ripartizione il resto della divisione (della codifica binaria) del valore della base della ripartizione X per un numero primo.

A titolo di esempio, consideriamo il file `Fibonacci` con $N = 10000$ record e supponiamo di prendere come base della ripartizione il campo `numero`. Prendiamo poi tre possibili funzioni di ripartizione $H(x)$:

$H_2(x)$ = resto della divisione di x per 2

$H_3(x)$ = resto della divisione di x per 3

$H_5(x)$ = resto della divisione di x per 5.

Con $H_2(x)$ avremo due moduli: il modulo 0 per i numeri di Fibonacci pari, e il modulo 1 per i numeri di Fibonacci dispari. È un utile esercizio calcolare la ripartizione dei record tra i due moduli. Se prendiamo la successione dei resti modulo 2 dei numeri di Fibonacci otteniamo cicli di lunghezza 3, tutti del tipo

$$(1, 1, 0).$$

Dunque, il modulo 0 conterrà $N_0 = N/3$ record del file e impegnerà $\lceil N_0/P \rceil = 105$ blocchi, mentre il modulo 1 ne conterrà $N_1 = 2N/3$ record del file e impegnerà $\lceil N_1/P \rceil = 210$ blocchi.

Con $H_3(x)$ avremo tre moduli. Ora nella successione dei resti modulo 3 otteniamo cicli di lunghezza 8, tutti del tipo

$$(1, 1, 2, 0, 2, 2, 1, 0).$$

Dunque il modulo 0 conterrà $N_0 = N/4$ record del file e impegnerà $\lceil N_0/P \rceil = 79$ blocchi, mentre i moduli 1 e 2 ne conterranno ciascuno $3N/8$ e impegneranno ciascuno $\lceil N/4P \rceil = 118$ blocchi.

Con $H_5(x)$ avremo cinque moduli. Ora nella successione dei resti modulo 5 otteniamo cicli di lunghezza 20 tutti del tipo

$$(1, 1, 2, 3, 0, 3, 3, 1, 4, 0, 4, 4, 3, 2, 0, 2, 2, 4, 1, 0).$$

Dunque, ciascuno dei cinque moduli conterrà $N/5$ record del file (equipartizione) e impegnerà $\lceil N/5P \rceil = 63$ blocchi.

Per finire, è chiaro che se prendiamo come base della ripartizione il campo `progressivo` (che è la chiave del file `Fibonacci`), allora avremo sempre una equipartizione cosicché ogni modulo impegnerà $\lceil N/mP \rceil$ blocchi.

Aggiunta. Per aggiungere un record che abbia il valore x della base della ripartizione X, si calcola prima $h = H(x)$. A questo punto, l'indice dei moduli fornisce l'indirizzo dell'ultimo della catena dei blocchi impegnati dal modulo h.esimo. Finalmente, il record viene memorizzato nella prima pagina libera di quel blocco (ammesso che ve ne sia una). Dunque, solitamente sono necessari 2 accessi alla memoria secondaria come nell'organizzazione non-ordinata.

Ricerca. Vanno distinti due casi a seconda che la chiave di ricerca coincida o meno con la base della ripartizione. Se sono diverse, allora la ricerca è lineare su tutti gli $n = N/P$ blocchi impegnati del file. Se, invece, la chiave di ricerca coincide con la base della ripartizione e la condizione selettiva coinvolge k degli m moduli, allora basta una ricerca lineare sulla catena dei blocchi dei k moduli interessati dalla ricerca, per un totale di $\lceil kN/mP \rceil$ blocchi.

Supponiamo che il file `Fibonacci` sia organizzato in maniera modulare con base di ripartizione `progressivo` e funzione di ripartizione $H_5(x)$. Se si vogliono i record soddisfano la condizione

$$1000 \leq \texttt{numero} \leq 2000$$

oppure la condizione

$$1000 \leq \texttt{progressivo} \leq 2000$$

vanno esaminati i blocchi di tutti e cinque i moduli per un totale di $\lceil N/P \rceil = 313$ blocchi. Se invece si vuole il millesimo numero di Fibonacci, allora basterà esaminare gli $\lceil N/5P \rceil = 63$ blocchi del modulo 0.

Cancellazione. Rispetto alla ricerca, la cancellazione dei record che soddisfano una condizione selettiva richiede un accesso in più per ogni blocco contenente un record selezionato.

Modifica. La modifica ha lo stesso costo della cancellazione quando nessun campo da modificare appartiene alla base della ripartizione; altrimenti, i record modificati vanno ricollocati dopo aver calcolato i nuovi moduli a cui appartengono.

12.5 Organizzazione ordinata (ISAM)

Si è visto che l'organizzazione modulare su base X è il meglio che si può ottenere pensando ad una ricerca con condizione selettiva $X = x$. Comunque, per le operazioni di ricerca con diverse condizioni selettive, per esempio $a \leq X \leq b$ oppure $Y = y$ con $Y \neq X$, l'organizzazione modulare non ha molto più da offrire rispetto all'organizzazione non-ordinata. La tecnica di organizzazione che ora presentiamo mira a rendere efficiente l'esecuzione di quelle operazioni di ricerca in cui la condizione selettiva sia nella forma

$$a \leq X \leq b$$

dove X è la chiave primaria del file. A questo scopo, i record del file vengono ordinati (in maniera lessicografica) in base ai valori di X. Esplicitamente, sia $(B_1, \ldots, B_n)$ la catena dei blocchi impegnati dal file. In ogni blocco i record sono memorizzati per valori crescenti di X; inoltre, se u_i e v_i sono i valori di X nel primo record e nell'ultimo record del blocco B_i, allora

$$v_1 < u_2 \qquad v_2 < u_3 \qquad \ldots \qquad v_{n-1} < u_n$$

Infine, in ogni blocco si lascia volutamente un certo numero di pagine libere (tipicamente il 20 %) per far posto all'inserimento di nuovi record del file. Oltre a questo, che viene chiamato *file principale*, viene creato un file ausiliario con n record (tanti quanti i blocchi occupati dal file principale), chiamato l'*indice* del file principale, anch'esso ordinato come il file principale sulla base di X. I record dell'indice, che per comodità chiamiamo *voci*, sono le coppie

$$(u_1, I_1) \qquad (u_2, I_2) \qquad \ldots \qquad (u_n, I_n),$$

dove I_i è l'indirizzo del blocco B_i. In realtà, il valore u_1 nella prima voce dell'indice è posto uguale ad un valore, convenzionalmente indicato con $-\infty$, che è più piccolo di ogni possibile valore di X. Ovviamente, X è anche una chiave identificativa dell'indice, cosicché ogni valore x di X determina al più una voce dell'indice.

Indichiamo con $(b_1, \ldots, b_m)$ la catena dei blocchi necessari a memorizzare l'indice del file principale, e con $(k_1, \ldots, k_m)$ la *block directory* dell'indice, cioè la lista degli indirizzi dei blocchi $b_1, \ldots, b_m$. Ovviamente, $m < n$. Inoltre, siccome la lunghezza di una voce dell'indice è pari alla lunghezza di X più 4 byte (necessari per memorizzare l'indirizzo di un blocco), un blocco dell'indice può ospitare un numero di voci maggiore del numero di record che può ospitare un blocco del file principale cosicché $m << n$. Questo in molti casi fa sì che la *block directory* dell'indice possa essere mantenuta in memoria centrale durante l'intero utilizzo del file principale. D'ora in poi supponiamo che questo sia il caso.

file principale

blocco	*indirizzo*	*contenuto*
B_1	I_1	$(u_1, \ldots), \ldots, (v_1, \ldots)$
B_2	I_2	$(u_2, \ldots), \ldots, (v_2, \ldots)$
…	…	……
B_n	I_n	$(u_n, \ldots), \ldots, (v_n, \ldots)$

indice del file principale

blocco	*indirizzo*	*contenuto*
b_1	k_1	$(u_1, I_1), (u_2, I_2), \ldots$
…	…	……
b_m	k_m	$\ldots, (u_n, I_n)$

Ricerca sulla chiave primaria. Consideriamo il problema di localizzare un record che abbia $X = x$. Indipendentemente dall'esistenza di tale record nel file principale, si andrà a cercare la voce (v, J) dell'indice che "copre" x nel senso che

— $v \leq x$ e

— se (v, J) non è l'ultima voce dell'indice (cioè $v \neq u_n$) e (w, K) è la voce successiva a (v, J), allora $x < w$.

È chiaro che, una volta trovata la voce (v, J) che copre x, la localizzazione del record cercato è nel blocco che è all'indirizzo J. Si osservi che se $v = x$ allora il record appartiene al file principale ed è proprio il primo record nel blocco che è all'indirizzo J. Dunque, ora la questione è come trovare la voce che copre x e questo è un problema di ricerca sull'indice che può essere risolto con una *ricerca binaria*.

(1) Porre $i := 1; f := m$.

(2) Fintantoché $i \neq f$ ripetere:

(2.1) Porre $h := \lceil (i+f)/2 \rceil$.

(2.2) Trovare l'indirizzo h.esimo nella lista degli indirizzi (k_1, …, k_m) presente nella *block directory* dell'indice.

(2.3) Caricare in memoria centrale il blocco (b_h) che è all'indirizzo k_h.

(2.4) Sia (u, I) la prima voce nel blocco b_h.

Caso 1: $x = u$. In tal caso, la voce cercata è (v, I) e la ricerca termina.

Caso 2: $x < u$. Porre $f := h-1$ e tornare all'istruzione (2.1).

Caso 3: $x > u$. Esaminare le voci nel blocco b_h alla ricerca di una voce (v, J) che non sia l'ultima in b_h e sia tale che, detta (w, K) la voce successiva a (v, J), si abbia $v \leq x < w$. Se la si trova, allora la voce cercata è (v, J) e la ricerca termina. Se invece la ricerca dà esito negativo allora, porre $i := h$ e tornare all'istruzione (2.1).

(3) Se $i = f$ allora esaminare le voci nel blocco b_i alla ricerca di una voce (v, J) che non sia l'ultima in b_i e sia tale che, detta (w, K) la voce successiva a (v, J), si abbia $v \leq x < w$. Se la si trova, allora la voce cercata è (v, J). Se invece la ricerca dà esito negativo allora la voce cercata è l'ultima voce di b_i.

Si osservi che, poiché ad ogni iterazione il numero dei blocchi dell'indice viene dimezzato, dopo al più $\lceil \log_2 m \rceil + 1$ passi la ricerca binaria termina.
Finalmente, una volta trovata la voce (v, J) che copre x, basterà caricare il blocco che è all'indirizzo J per localizzare il record del file principale con $X = x$.

Dato un file (principale) con N = 30.000 record tutti di lunghezza L = 100 byte. Assumiamo che la dimensione di un blocco sia D = 1024 byte, cosicché un blocco può contenere fino a $P = \lfloor D/L \rfloor$ = 10 record. Occorrono dunque $n = \lceil N/P \rceil$ = 3000 blocchi per memorizzare il file principale. Supponiamo che il file principale sia ordinato in base alla sua chiave X che abbia lunghezza 11 B. Dunque, la lunghezza di una voce dell'indice del file principale è 15 B, cosicché un blocco può contenere fino a 68 voci. Pertanto, l'indice impegnerà m = 45 blocchi. Una ricerca binaria sull'indice richiederà al più $\lceil \log_2 m \rceil$ = 6 accessi alla memoria secondaria e, dunque, la localizzazione di un record richiede 7 accessi alla memoria secondaria. Vediamolo con un esempio. Supponiamo di cercare il record con $X = x$ e che la voce che copre x è l'ultima del 35.esimo blocco (b_{35}) dell'indice. Dunque, se (u, I) è la prima voce nel blocco b_i, allora $u < x$ per $i \leq 35$ mentre $x < u$ per $i > 35$.

Prima Iterazione. Con $i = 1$ ed $f = 45$ si ha $h = \lceil (i+f)/2 \rceil = 23$. Siccome il valore di X nella prima voce nel blocco b_{23} è minore di x, si esamina l'intero contenuto di b_{23} ma senza successo. A questo punto si pone $i := 23$.

Seconda Iterazione. Con $i = 23$ ed $f = 45$ si ha $h = \lceil (i+f)/2 \rceil = 34$. Siccome il valore di X nella prima voce nel blocco b_{34} è minore di x, si esamina l'intero contenuto di b_{34} anche questa volta senza successo. A questo punto si pone $i := 34$.

Terza Iterazione. Con $i = 34$ ed $f = 45$ si ha $h = \lceil (i+f)/2 \rceil = 40$. Siccome il valore di X nella prima voce del blocco b_{40} è maggiore di x, si pone $f := 39$.

Quarta Iterazione. Con $i = 34$ ed $f = 39$ si ha $h = \lceil (i+f)/2 \rceil = 37$. Siccome il valore di X nella prima voce del blocco b_{37} è maggiore di x, si pone $f := 36$.

Quinta Iterazione. Con $i = 34$ ed $f = 36$ si ha $h = \lceil (i+f)/2 \rceil = 35$. Siccome il valore di X nella prima voce del blocco b_{35} è minore di x, si esamina l'intero contenuto di b_{35} di nuovo senza successo. A questo punto si pone $i := 35$.

Sesta Iterazione. Con $i = 35$ ed $f = 36$ si ha $h = \lceil (i+f)/2 \rceil = 36$. Siccome il valore di X nella prima voce del blocco b_{36} è maggiore di x, si pone $f := 35$. Essendo $i = f$, si riesamina l'intero contenuto del blocco b_{35} senza successo; allora, si conclude che la voce che copre x è l'ultima voce in b_{35}.

Nel discutere ora le operazioni di aggiornamento e di ricerca, assumeremo inizialmente che i record del file principale non siano appuntati alle pagine del blocco

Aggiunta. Per aggiungere un nuovo record, occorre individuare il blocco del file principale e la posizione dove va inserito. Se x è il valore della chiave del record, viene prima determinata la voce (v, J) dell'indice che copre x con una ricerca binaria sull'indice; quindi, si esamina il contenuto del blocco B che ha indirizzo J. Se B contiene pagine libere, allora il record viene inserito nella sua giusta posizione traslando i record i cui valori di X seguono x nell'ordinamento. Altrimenti (cioè se B non contiene pagine libere), si danno due casi. Siano u e v i valori di X nel primo e ultimo record presente in B.

Caso 1: il blocco che precede B ha pagine libere. In tal caso, si traferisce il primo record in B (con $X = u$) nel blocco che precede B (in ultima posizione); quindi, si memorizza il record da inserire in B e si modifica il valore di X nella voce dell'indice che si riferisce al blocco B.

Caso 2: il blocco che precede B non ha pagine libere oppure B è il primo blocco.

Caso 2*a*: il blocco che segue *B* ha pagine libere. In tal caso, se $x < v$ allora si traferisce l'ultimo record in *B* (con $X = v$) nel blocco che segue *B* (in prima posizione) e si memorizza il record da inserire in *B*; mentre, se $x > v$ allora si memorizza il record da inserire nel blocco che segue *B* (in prima posizione). In entrambi i casi, si modifica il valore di *X* nella voce dell'indice che si riferisce al blocco che segue *B*.

Caso 2*b*: il blocco che segue *B* non ha pagine libere oppure *B* è l'ultimo blocco. In tal caso, viene richiesto al *file system* un nuovo blocco *B′* (che andrà a seguire *B*) e si ripartiscono i record in *B* e quello da inserire tra i blocchi *B* e *B′*. Infine, nell'indice viene inserita una nuova voce corrispondente al blocco *B′* dopo la voce corrispondente al blocco *B*.

Cancellazione. Per l'eliminazione di record che soddisfano una condizione selettiva che faccia uso della chiave primaria (ad es., $a \leq X \leq b$) si procede con una ricerca binaria per selezionare il primo record da cancellare ($X = a$) il cui bit di cancellazione viene posto a 0; quindi, si procede a cancellare tutti i record che seguono il primo fintantoché $X \leq b$.
Per l'eliminazione di record che soddisfano una condizione selettiva con chiave di ricerca diversa dalla chiave primaria, si procede con una ricerca lineare.
In entrambi i casi, può presentarsi la necessità di modificare una o più voci dell'indice. Infine, se dopo la cancellazione un blocco si trova ad avere solo pagine libere, questo verrà restituito al *file system*.

Modifica. Per la modifica di record che soddisfano una condizione selettiva che faccia uso della chiave primaria (ad es., $a \leq X \leq b$) si procede con una ricerca binaria per selezionare il primo record da cancellare ($X = a$) che viene aggiornato ed eventualmente ricollocato qualora la modifica interessa qualche campo in *X*. Quindi, si procede ad aggiornare tutti i record che seguono il primo fintantoché $X \leq b$.
Per l'eliminazione di record che soddisfano una condizione selettiva con chiave di ricerca diversa dalla chiave primaria, si procede con una ricerca lineare.
In entrambi i casi, può presentarsi la necessità di modificare una o più voci dell'indice.