Organizzazione Fisica dei Dati (Parte I)

Angelo Montanari

Dipartimento di Matematica e Informatica
Università di Udine

Sommario - 1

Introduzione

Cenni ai dispositivi di memorizzazione: memoria primaria (memoria principale e memorie cache), secondaria (dischi magnetici) e terziaria (dischi ottici e nastri magnetici)

Tecniche di memorizzazione di file e record su disco

- Record di lunghezza fissa e variabile
- Organizzazione dei record in blocchi
- Allocazione dei blocchi su disco
- File header

Sommario - 2

Operazioni sui file

File di record non ordinati (heap file)

File di record ordinati (sorted file)

Tecniche di hashing (interno, esterni, estendibile)

Dispositivi di memorizzazione

L'insieme dei dati che costituiscono una base di dati (DB) è composto da **record** strutturati in uno o più campi. Su tale insieme di dati, il DBMS deve poter eseguire con facilità operazioni di interrogazione e di aggiornamento

Il DB è fisicamente memorizzato su un computer mediante opportuni **dispositivi di memorizzazione**:

1. *Primari.* Consentono un rapido accesso ai dati, ma sono di dimensioni ridotte. Possono essere utilizzati direttamente dalla CPU (memoria primaria, memoria cache, etc.).
2. *Secondari (e terziari).* Hanno elevate capacità di memorizzazione e bassi costi, ma tempi di accesso decisamente più elevati. Comprendono dischi magnetici, dischi ottici, nastri magnetici, etc.

File di record

I dati memorizzati su disco sono organizzati in **file di record**.

Ogni record è una collezione di dati interpretabile come un insieme di fatti relativi a entità, loro attributi e loro relazioni (ad esempio, una tupla può essere memorizzata come un record nel quale il valore di un attributo può essere memorizzato in un campo del record)

Esistono diverse tecniche di memorizzazione fisica dei record di un file su un disco magnetico:

1. **Heap file** (file non ordinato) posiziona i record sul disco in modo non ordinato
2. **Sorted file** (file sequenziale) ordina i record secondo il valore di un particolare campo
3. **Hashed file** usa una funzione di hash per determinare la posizione dei record sul disco

Memoria secondaria: capacità, robustezza e costo

Le basi di dati vengono abitualmente utilizzate per memorizzare grandi quantità di dati che devono essere mantenute a lungo in memoria. Per tale ragione esse vengono memorizzate in modo permanente su dispositivi di memorizzazione secondari, quali, ad esempio, i dischi magnetici.

In particolare,

1. la mole di dati da gestire è **troppo grande** per poter essere contenuta nella sua totalità in memoria primaria;
2. i dispositivi di memorizzazione secondari sono **più sicuri** (in quanto non volatili);
3. i **costi** (memoria per unità di dati) sono **più contenuti**.

Modalità di accesso alla base di dati

Molte applicazioni che utilizzano basi di dati richiedono di esaminare solo una piccola porzione di esse.

Ogni specifico insieme di dati richiesto da un'applicazione

1. deve essere **localizzato** all'interno della base di dati;
2. deve essere **copiato** in memoria principale, per poter essere elaborato;
3. se sono stati effettuati degli aggiornamenti, i dati aggiornati vanno copiati (**riscritti**) in memoria secondaria.

Tecniche di memorizzazione di file/record su disco

Tipi di record. Un record è una collezione di valori di dati (data value o item). Ogni valore è associato ad uno dei campi del record e consiste di uno o più byte.

I record possono essere utilizzati per descrivere le istanze di un dato tipo di entità (ad esempio, *IMPIEGATO*) del modello ER. I campi dei record corrisponderanno agli attributi del tipo di entità (ad esempio, *CF, NOME, STIPENDIO, ..*).

Il **tipo** di un **record** (o formato di un record) è una collezione di nomi dei campi, coi relativi tipi di dato. I **tipi** di **dato** possono essere numerici (integer, long integer, real, ..), di tipo carattere, booleani (dominio *0/1*, oppure *vero/falso*) o di tipo speciale (definiti dal programmatore).

Un semplice esempio

RECORD TYPE NAME	**FIELD NAME**	**DATA TYPES**
type IMPIEGATO = record	NOME	packed array [1..30] of character;
	CF	packed array [1..16] of character;
	STIPENDIO	integer;
	CODICE_LAVORO	integer;
	DIPARTIMENTO	packed array [1..20] of character;
end;		

Record di lunghezza fissa e variabile - 1

Un file è una sequenza di record, usualmente dello stesso tipo.

Se i record di un file hanno tutti la stessa dimensione (in byte), si dice che il file è composto da record di **lunghezza fissa**; altrimenti, si dice che è composto da record di **lunghezza variabile**.

Le ragioni per cui un file può avere record di lunghezza variabile possono essere varie:

(a) tutti i record del file sono dello stesso tipo, ma uno o più campi hanno una **dimensione variabile** (ad esempio, il campo *NOME* di *IMPIEGATO*);

(b) tutti i record del file sono dello stesso tipo, ma uno o più campi possono avere **valori multipli** per singoli record; tale campo viene chiamato *campo ripetuto* e un gruppo di valori per tale campo viene chiamato gruppo ripetuto;

Record di lunghezza fissa e variabile - 2

(c) tutti i record del file sono dello stesso tipo, ma uno o più campi sono opzionali, ossia possono avere un valore per alcuni record, ma non per altri (**file** con campi **opzionali**);

(d) il file contiene record di più tipi e di dimensioni variabili (**file mixed**). Ciò accade, ad esempio, quando record di tipo diverso sono collocati all'interno degli stessi blocchi (i record *ESAME* relativi ad un determinato studente possono essere posizionati subito dopo il record *STUDENTE* cui si riferiscono; ciò è quanto accadeva nel modello gerarchico dei dati).

Un esempio - 1

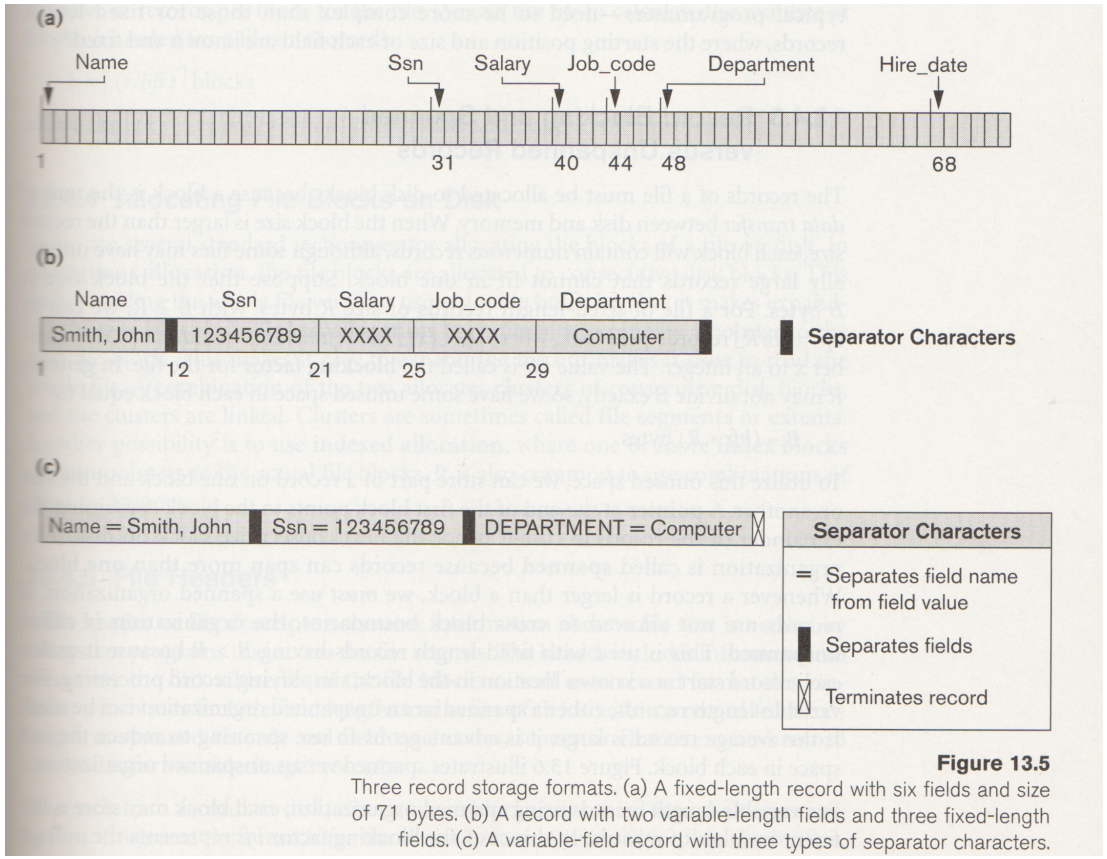


Figure 13.5
Three record storage formats. (a) A fixed-length record with six fields and size of 71 bytes. (b) A record with two variable-length fields and three fixed-length fields. (c) A variable-field record with three types of separator characters.

Un esempio - 2

Nel **caso (a)**, il record *IMPIEGATO* di lunghezza fissa ha dimensione pari a 71 byte. Si compone di 6 campi, le cui dimensioni sono fisse. Ciò consente di identificare la posizione del byte iniziale di ciascun campo rispetto alla posizione del byte iniziale del record.

Un file di record di lunghezza variabile può essere rappresentato attraverso un file di record di lunghezza fissa, specificando (fissando) la lunghezza massima di ogni campo. Per indicare che un determinato record è privo di un valore per un dato campo opzionale può essere utilizzato un valore speciale (nullo).

Un esempio - 3

Nel **caso (b)**, il record *IMPIEGATO* ha tre campi di lunghezza fissa (SSN, STIPENDIO e CODICE_LAVORO) e due di lunghezza variabile (NOME e DIPARTIMENTO). Nel caso di un campo di lunghezza variabile non opzionale (ogni record ha un valore per tale campo, ma non è possibile conoscere a priori la lunghezza esatta del (valore di tale) campo) possono essere utilizzati degli opportuni caratteri separatori (ad esempio, ?, %, $), che segnalano la fine dei (valori di) campi di lunghezza variabile.

Nel **caso (c)**, più generale, in cui sono presenti campi opzionali e/o di lunghezza variabile, vengono utilizzati caratteri separatori distinti, che rispettivamente separano il nome di un campo dal suo valore, segnalano la fine di un campo, segnalano la fine di un record.

Organizzazione dei record in blocchi

I record di un file devono essere allocati in blocchi (**record blocking**) in quanto il blocco è l'unità di trasferimento dati tra disco e memoria principale.

Quando la dimensione del blocco è superiore a quella del singolo record, si ha che un blocco contiene più record.

Supponiamo che la dimensione del blocco sia pari a B **byte**. Assumendo che un dato file sia composto da record di dimensione fissa pari a R **byte**, con $B \geq R$, in ogni blocco possono essere inseriti bfr record (**blocking factor**):

$$bfr = \left\lfloor \frac{B}{R} \right\rfloor$$

Esempio: se $B = 120$ e $R = 15$, allora $bfr = 8$

Un problema

In generale, B può non essere divisibile per R. Se imponiamo che ogni record sia interamente contenuto in uno e un solo blocco, ogni blocco conterrà dello **spazio inutilizzato** pari a:

$$B - (bfr \cdot R) \text{ byte}$$

Esempio: se $B = 120$ e $R = 25$, allora $bfr = 4$ e $B - (bfr \cdot R) = 20$ (più del 15% dello spazio viene sprecato)

Se la dimensione dei record è comparabile a quella del blocco (esempio, $B = 120$ e $R = 70$), il vincolo proposto risulta inaccettabile.

Se la dimensione del record è maggiore di quella del blocco, ossia la condizione $B \geq R$ non risulta più soddisfatta (esempio, $B = 120$ e $R = 125$), diventa impossibile memorizzare i record.

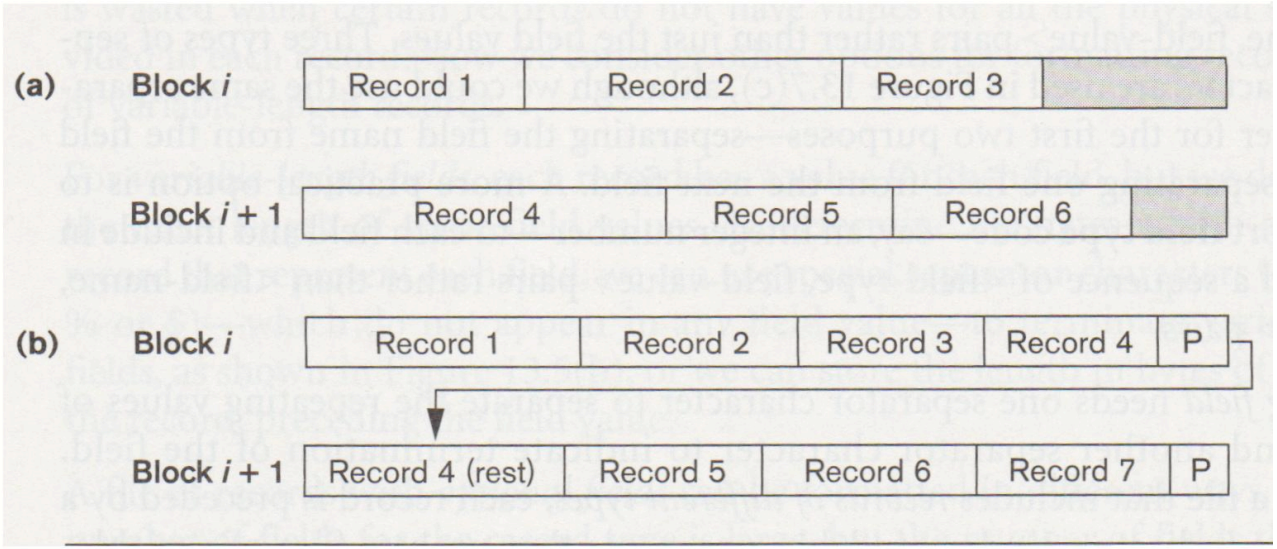
Record spanned e unspanned - 1

Una soluzione alternativa consiste nel consentire la memorizzazione di un record in due (o più) blocchi (**spanned record**). Nel caso in cui ogni record sia distribuito su al più due blocchi, può essere utilizzato un puntatore alla fine del primo blocco per individuare blocco che contiene la parte rimanente del record, a meno che i due blocchi non siano uno consecutivo all'altro.

Nel caso in cui la suddivisione di un record in più blocchi sia vietata (**unspanned record**), si parla di organizzazione di tipo uspanned.

Un esempio di record spanned e unspanned è riportato nel lucido successivo.

Record spanned e unspanned - 2



Organizzazione unspanned (a) e spanned (b) dei record

Il caso dei record di lunghezza variabile

Nel caso di record di lunghezza variabile, può essere utilizzata sia un'organizzazione di tipo spanned che di tipo unspanned.

Nel caso in cui la dimensione media del record sia elevata (rispetto alla dimensione del blocco), è conveniente utilizzare una organizzazione di tipo spanned per ridurre lo spazio sprecato in ogni blocco.

Ogni blocco può ovviamente memorizzare un numero differente di record. In tal caso, il blocking factor *bfr* rappresenta il numero medio di record per blocco contenuti nel file.

Dimensionamento dei file

Noti la dimensione del blocco, la dimensione del record e il numero di record contenuti nel file (tale numero può ovviamente variare nel tempo; occorre, quindi, disporre di una stima attendibile del numero medio/massimo di record del file), è possibile determinare il numero n_b di blocchi necessari per memorizzare il file.

Assumiamo un'organizzazione dei record di tipo *unspanned*. Siano r e bfr il numero di record del file e il blocking factor, rispettivamente. Il **numero di blocchi** necessari è pari a:

$$n_b = \left\lceil \frac{r}{bfr} \right\rceil$$

Allocazione dei blocchi di un file su disco

Esistono diverse tecniche consolidate per effettuare l'allocazione dei blocchi di un file su disco.

1. **Allocazione contigua**. I blocchi del file sono allocati in blocchi consecutivi sul disco. Vantaggio: lettura rapida dell'intero file con la tecnica del doppio buffering. Svantaggio: espansione difficoltosa.
2. **Allocazione con collegamenti**. Ogni blocco del file contiene un puntatore al blocco successivo. Vantaggio: espansione estremamente semplice. Svantaggio: scansione/lettura dell'intero file dispendiosa/ lenta.
3. **Combinazione** delle due precedenti soluzioni. Vengono definiti e allocati cluster (detti anche segmenti o extent) di blocchi consecutivi su disco, collegati fra loro tramite puntatori.
4. **Allocazione indicizzata**. Uno o più blocchi di indici contengono i puntatori agli effettivi blocchi del file.

File header

Un **file header** (o descrittore del file) contiene le informazioni necessarie per determinare gli indirizzi sul disco dei blocchi del file.

Descrizioni relative al formato dei record:

- record di lunghezza fissa (e unspanned): lunghezza dei campi e ordine dei campi nel record;
- record di lunghezza variabile: codici tipo dei campi, caratteri di separazione, codici tipo dei record.

Ricerca di record su disco

La ricerca di un record su disco prevede di copiare uno o più blocchi nel buffer della memoria principale. Specifici programmi cercano all'interno dei buffer il/i record desiderati, utilizzando l'informazione contenuta nel file header.

Se l'indirizzo del blocco che contiene il record voluto non è noto, i programmi di ricerca devono eseguire una **ricerca lineare** attraverso i blocchi dei file: ogni blocco del file è copiato nel buffer e la ricerca prosegue sino a quando o viene individuato il record desiderato o sono stati controllati tutti i record del file. Tale ricerca risulta estremamente dispendiosa per file di notevole lunghezza.

Operazioni sui file

Le operazioni sui file possono essere suddivise in due categorie:

- operazioni di recupero dei dati (ricerca)
- operazioni di aggiornamento

Le **operazioni di ricerca** non modificano alcun dato presente nel file, ma si limitano a localizzare alcuni record del file in modo da poter esaminare ed eventualmente elaborare il contenuto di alcuni dei loro campi.

Le **operazioni di aggiornamento** modificano il file, inserendo o cancellando record o modificando il valore di alcuni dei loro campi.

In entrambi i casi, è necessario selezionare uno o più record sulla base di un'opportuna **condizione di selezione**.

Un esempio

Consideriamo un file *IMPIEGATO* con campi *CF*, *STIPENDIO*, *CODICE_LAVORO* e *DIPARTIMENTO*.

Due esempi di semplici **condizioni di selezione**, che utilizzano gli operatori di confronto $=, >, <, \ldots$ applicati ai valori di alcuni campi dei record del file, sono:

$$CF = 'RSSNTL72S04F756L'$$

e

$$DIPARTIMENTO = 'ricerca' \text{ AND } STIPENDIO > 30000$$

Le operazioni di ricerca - 1

Le operazioni di ricerca su file sono generalmente basate su semplici condizioni di selezione.

Operazioni di selezione **complesse** vengono decomposte in operazioni di selezione semplici che possono essere usate per localizzare i record sul disco.

Esempio. Immaginiamo di dover selezionare tutti gli impiegati del dipartimento *Ricerca* che percepiscono uno stipendio maggiore di 30000 euro. Possiamo estrarre, e valutare preliminarmente, la condizione semplice $DIPARTIMENTO = 'ricerca'$ dalla condizione complessa $STIPENDIO > 30000$ AND $DIPARTIMENTO = 'ricerca'$. Successivamente, ogni record che soddisfa tale condizione verrà analizzato per verificare se soddisfa anche il primo congiunto.

Le operazioni di ricerca - 2

Quando più record del file soddisfano la condizione di ricerca, si individua solo il primo record (**record corrente**).

Per selezionare altri record che soddisfano la condizione, è necessario utilizzare operazioni addizionali.

Le operazioni di ricerca successive iniziano dal record corrente.

Consideriamo ora un insieme di operazioni per l'individuazione e l'accesso ai record dei file (esse variano da sistema a sistema).

Operazioni record-at-a-time

Le operazioni record-at-a-time sono operazioni applicate ad un **singolo record**.

Find (o **Locate**): ricerca il primo record che soddisfa una data condizione (record corrente) e trasferisce il blocco che contiene tale record in un buffer della memoria primaria.

Read (o **Get**): copia il record corrente dal buffer in una variabile del programma, o in un area di lavoro del programma, e fa avanzare il puntatore al record corrente al successivo record nel file.

FindNext: individua il record successivo che soddisfa la condizione di ricerca e trasferisce il blocco che lo contiene in un buffer della memoria primaria.

Delete: cancella il record corrente e (in taluni casi) aggiorna il file sul disco.

Modify: modifica i valori di alcuni campi del record corrente e (in taluni casi) aggiorna il file sul disco.

Insert: inserisce un nuovo record nel file individuando il blocco in cui il record deve essere inserito, trasferendo tale blocco in un buffer della memoria primaria, inserendo il nuovo record nel blocco contenuto nel buffer e (in tali casi) ricopiando il contenuto del blocco nel buffer sul disco.

Operazioni set-at-a-time

Le operazioni set-at-a-time sono operazioni applicate ad un **intero file**.

FindAll: individua tutti i record di un file che soddisfano la condizione di ricerca.

FindOrdered: recupera tutti i record presenti file in un ben preciso ordine.

Reorganize: esegue un processo di riorganizzazione dei record del file (ad esempio, riordina i record sulla base del valore che essi assumono su uno specifico campo).

Altre operazioni sono necessarie per preparare un file per l'accesso (**Open**) o per la chiusura (**Close**).

Organizzazione dei file e metodi di accesso

Organizzazione dei file: organizzazione dei dati contenuti in un file in record, blocchi e strutture di accesso (fanno parte dei criteri di organizzazione di un file le modalità secondo le quali i record e i blocchi sono memorizzati e collegati fra loro)

Metodi di accesso: insieme di programmi che permettono di effettuare operazioni sui file. Ogni metodo di accesso può essere applicato solo a file organizzati in modo opportuno. Ad esempio, un metodo di accesso basato su indice non può ovviamente essere applicato ad un file che non è organizzato attraverso indici.

Utilizzo delle condizioni di ricerca - 1

Consideriamo ancora il file IMPIEGATO.

In talune situazione occorre **inserire** nuovi record (ad esempio, viene assunto un nuovo impiegato),

in altre occorre **cancellare** dei record (ad esempio, quando un impiegato viene licenziato),

in altre ancora occorre **modificare** dei record (ad esempio, quando viene modificato lo stipendio di un impiegato).

In tutti questo casi, vanno applicate opportune **condizioni di selezione**.

Utilizzo delle condizioni di ricerca - 2

Se vuole consentire l'applicazione (efficiente) di una condizione di ricerca basata su CF, il progettista deve scegliere un'organizzazione del file che faciliti l'individuazione del record sulla base del valore CF: ordinare i record sulla base del CF o definire un indice.

Se vuole consentire l'esecuzione (efficiente) di operazioni sugli stipendi dei dipendenti dipartimento per dipartimento, il progettista deve scegliere un'organizzazione del file che raggruppi i record di IMPIEGATO che fanno riferimento ad uno stesso dipartimento. Questa organizzazione dei record contrasta, però, con l'organizzazione precedentemente definita.

Se possibile, il progettista deve scegliere un'organizzazione che permetta di svolgere entrambe le operazioni in modo efficiente: compromesso fra diverse tecniche (ordinamento, hashing, indicizzazione)

File di record non ordinati (heap file)

Nel caso più semplice di organizzazione, i record sono posizionati nel file nell'ordine in cui vengono inseriti: ogni nuovo record viene inserito alla fine del file.

Un file organizzato in questo modo è detto **heap file**.

L'inserimento di un nuovo record può essere effettuato in modo molto efficiente:

(a) l'ultimo blocco del file (su disco) è copiato in un buffer;

(b) il nuovo record viene aggiunto nel blocco contenuto nel buffer;

(c) il blocco aggiornato viene riscritto su disco.

L'indirizzo dell'ultimo blocco del file viene mantenuto nel file header.

Ricerca in un heap file

La ricerca di un record in un heap file richiede una **ricerca lineare** su file blocco per blocco (procedura computazionalmente molto costosa).

Se un solo record soddisfa la condizione di ricerca, mediamente dovrà essere controllato almeno il 50% dei blocchi del file (su disco).

Per un file di n_b blocchi, il programma dovrà controllare $n_b/2$ blocchi nel caso medio; se nessun blocco soddisfa la condizione di ricerca, il programma dovrà controllare n_b blocchi.

Cancellazione in un heap file

Per **cancellare** un record, il programma deve effettuare le seguenti operazioni: (i) cercare il record; (ii) copiare il blocco che lo contiene in un buffer; (iii) cancellare il record dal buffer; (iv) riscrivere il blocco sul disco.

In generale, ogni cancellazione crea dello spazio inutilizzato nel blocco su disco. La cancellazione di molti record causa la perdita di quantità significative di spazio (wasted space).

Un'altra tecnica per cancellare record utilizza un extra byte o bit (deletion marker) associato ad ogni record. Il record è cancellato (logicamente) assegnando al delation marker un certo valore. Un valore diverso dal deletion mark indica che il record è valido. Tale tecnica richiede, però, una riorganizzazione periodica del file per recuperare dello spazio di memoria.

Lettura ordinata in un heap file

Lettura ordinata. Per leggere tutti i record secondo l'ordine definito dai valori di uno o più campi, viene creata una copia ordinata (sorted) del file.

L'ordinamento del file è computazionalmente oneroso per file di grandi dimensioni:

(a) i record contenuti in ciascun blocco vengono ordinati;

(b) coppie di blocchi ordinati vengono unite per creare gruppi (run) di record ordinati (di dimensione pari a 2 blocchi);

(c) run di due blocchi vengono a loro volta uniti per formare run di 4 blocchi; tale operazione viene ripetuta fino a quando si ottiene il run finale che coincide con il file ordinato.

Un esempio

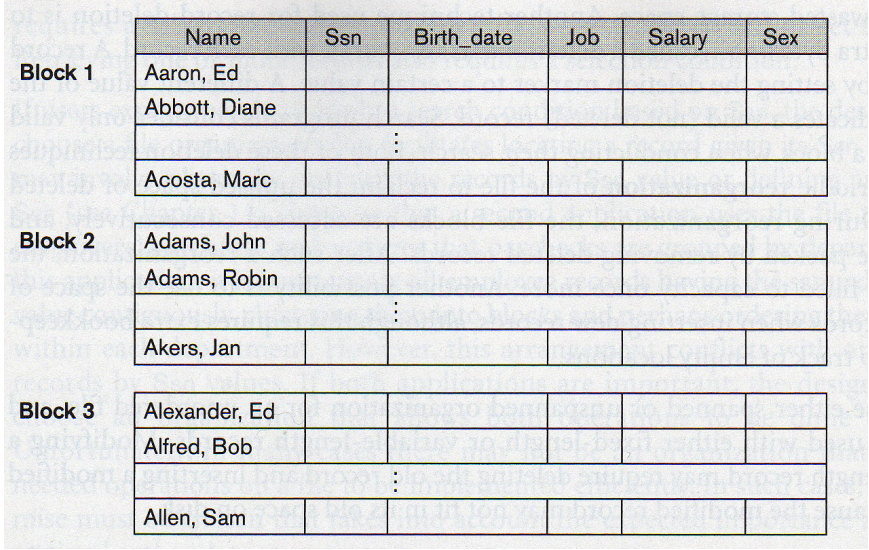
In un file di record non ordinati di lunghezza fissa, con un'organizzazione di tipo unspanned e un'allocazione contigua dei blocchi, l'**accesso ad un record** dipende dalla sua posizione all'interno del file.

Se immaginiamo che gli r record del file siano numerati in modo crescente $0, 1, 2, \ldots, r-1$ e che bfr sia il valore del blocking factor, l'i-esimo record del file risulta posizionato nel blocco $\left\lfloor \frac{i}{bfr} \right\rfloor$ ed è l'(i mod bfr)-esimo record del blocco.

Tale file è detto anche relative file in quanto i record possono essere facilmente raggiunti facendo riferimento alla loro posizione relativa.

File di record ordinati (file ordinati)

E' possibile ordinare fisicamente i record di un file su disco sulla base dei valori di uno (o più) dei loro campi (**ordinamento dei campi**).



	Name	Ssn	Birth_date	Job	Salary	Sex
Block 1	Aaron, Ed					
	Abbott, Diane					
	⋮					
	Acosta, Marc					
Block 2	Adams, John					
	Adams, Robin					
	⋮					
	Akers, Jan					
Block 3	Alexander, Ed					
	Alfred, Bob					
	⋮					
	Allen, Sam					

Vantaggi dei file ordinati

I file ordinati presentano degli ovvi vantaggi:

(a) la lettura di tutti i record secondo l'ordine definito dal campo ordinante risulta molto efficiente (non è richiesto alcun ordinamento);

(b) la ricerca del record successivo al record corrente è efficiente perché non richiede accessi addizionali a blocco (il record successivo e il record corrente si trovano nello stesso blocco, a meno che il record corrente non sia l'ultimo record del blocco);

(c) una condizione di ricerca basata sul valore del campo (chiave) ordinante può essere verificata in modo efficiente perché può essere utilizzata la **ricerca binaria** (anziché la ricerca lineare).

File ordinati e ricerca binaria

La ricerca binaria viene effettuata a livello di blocchi, non di record.

Supponiamo di avere un file con n_b blocchi, numerati da 1 a n_b, i cui record siano ordinati in modo crescente sulla base del valore che assumono sul campo (chiave) ordinante (ordering key field), e di voler individuare un record con valore del campo (chiave) ordinante uguale a k.

Una ricerca binaria usualmente accede a $log_2 n_b$ blocchi, indipendentemente dal fatto che il record sia presente o meno, contro $n_b/2$ accessi (nel caso medio), qualora il record sia presente, e n_b accessi, qualora il record sia assente, nel caso della ricerca lineare

L'algoritmo

$l := 1; u := n_b$
while $u \geq l$ do
    begin $i := (l + u)$ DIV 2
    read block i of the file into the buffer
    if $k <$ ordering key field value of the first record in the block
    then $u := i - 1$
    else if $k >$ ordering key field value of the last record in the block
        then $l := i + 1$
        else if the record with ordering key field value $= k$ is in the buffer
            then goto found
            else goto notfound
    end
goto notfound

Ricerca, inserimento e modifica

Una condizione di ricerca che coinvolge gli operatori di confronto $>, <, \geq$, e $\leq$ su un campo (chiave) ordinante risulta molto efficiente in quanto tutti i record che soddisfano la condizione sono fra loro contigui nel file.

Esempio. La condizione di ricerca $NAME < 'F'$, dove $<$ denota la relazione di ordinamento alfabetico, seleziona tutti i record compresi fra l'inizio del file e il primo record con valore del campo $NAME$ che inizia con la lettera F.

Le operazioni di **inserimento** e **modifica** di record sono operazioni costose in quanto occorre preservare l'ordinamento (fisico) del file.

Inserimento di un record

L'**inserimento** di un record richiede di:

(a) cercare la posizione in cui il nuovo record va inserito;

(b) creare lo spazio necessario per il posizionamento del nuovo record (occorre mediamente spostare metà dei record del file, ossia leggere e riscrivere metà dei blocchi del file).

Possibili soluzioni:

(i) mantenere in ogni blocco dello spazio libero per gli eventuali inserimenti di nuovi record;

(ii) creare un file temporaneo non ordinato (file di overflow o transaction file). I nuovi record vengono inseriti alla fine del file di overflow. Tale file viene periodicamente unito al file principale attraverso un'operazione di riorganizzazione.

Modifica di un record

Le modalità di esecuzione della **modifica** del valore di un campo di un record dipendono da due fattori:

(i) la condizione di ricerca del record da modificare;

(ii) il campo da modificare.

Se la condizione di ricerca utilizza il campo (chiave) ordinante, è possibile localizzare il record mediante una ricerca binaria; altrimenti, va effettuata una ricerca lineare.

Considerazione conclusiva. I file ordinati sono raramente utilizzati nell'ambito delle basi di dati per applicazioni reali, a meno che non sia disponibile una struttura di indicizzazione (indice primario) che consenta di definire un opportuno cammino per un accesso efficiente ai dati.

Tecniche di hashing

Un altro tipo di organizzazione primaria dei file è quello basato sulle tecniche di **hashing** che consente un accesso molto veloce ai record rispetto a determinate condizioni di ricerca (*hash* o *direct file*).

La **condizione di ricerca** è tipicamente una condizione di uguaglianza su un singolo campo (*hash field del file*); tale campo è spesso anche un campo chiave del file (*hash key*).

Funzionamento (**hashing esterno**): viene utilizzata una funzione h (*hash o randomizing function*) che, applicata al valore del campo hash di un record, restituisce l'indirizzo del blocco su disco in cui il record è contenuto.

Le tecniche di hashing si possono utilizzare anche per manipolare file di record temporanei, di piccole dimensioni, all'interno di un programma (*hashing interno*).

Hashing interno

Per i file interni, la tecnica di hashing è implementata per mezzo di array di record.

Funzionamento (**hashing interno**): supponiamo che l'indice dell'array vari da 0 a $m-1$; si hanno m slot il cui indirizzo corrisponde all'indice dell'array.

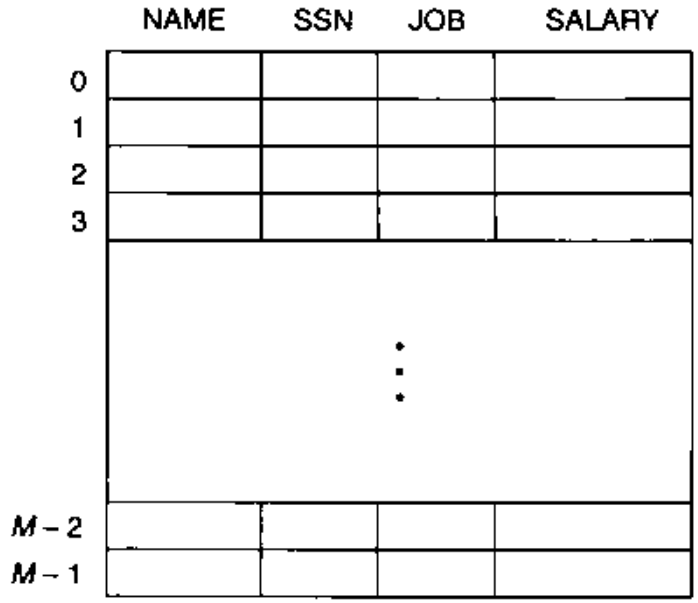
Si usa una funzione di hash che trasforma il valore del campo hash in un numero intero compreso tra 0 e $m-1$. Ad esempio, se il campo hash è di tipo intero,

$$h(k) = k \text{ MOD } m$$

restituisce il valore intero, compreso tra 0 e $m-1$, del resto della divisione k/m (*indirizzo del record*).

Un esempio di hashing interno - 1

Si consideri una tabella *IMPIEGATO* con campi NAME, *SSN*, *JOB* e *SALARY*.



Un esempio di hashing interno - 2

Array di m posizioni per internal hashing

$$h(SSN) = SSN \text{ MOD } m$$

Valori del campo hash non interi possono essere trasformati in interi prima di applicare loro la funzione MOD.

Nel caso di stringhe di caratteri, ad esempio, la trasformazione può far uso del codice numerico associato ai singolo caratteri (si noti come, nell'esempio del lucido successivo, parole che sono l'una una permutazione dell'altra generino lo stesso indirizzo di hash).

Un esempio di hashing interno - 3

Esempio. Dato un campo hash k di tipo stringa di lunghezza predefinita (packed array [1..20] of character), per calcolare l'indirizzo di hash si può utilizzare il seguente algoritmo (la funzione *code* ritorna il codice numerico di un carattere)

temp := 1;
for i := 1 to 20 do temp := temp $\cdot$ $code(k[i])$;
hash_address := temp MOD m;

Altre funzioni di hashing

Altre funzioni di hashing comunemente utilizzate sono:

- tecnica di folding che utilizza una funzione aritmetica (+) o una funzione logica (*exclusive_or*) a differenti parti del valore del campo hash per calcolare l'indirizzo;
- un'altra tecnica utilizza alcune cifre del valore (numerico) del campo hash (ad esempio, la terza, la quinta e l'ottava) per formare l'indirizzo di hash.

Problema

La maggior parte delle funzioni di hashing non garantisce che a valori distinti del campo hash corrispondano indirizzi distinti, poiché l'insieme dei possibili valori del campo hash è molto più grande dello spazio degli indirizzi.

Quando l'indirizzo determinato dalla funzione di hashing in corrispondenza del valore del campo hash di un nuovo record coincide con quello già assegnato ad un altro record si genera una **collisione**.

In tal caso, bisogna determinare un'altra posizione nella quale inserire il nuovo record; il processo di ricerca di tale posizione è chiamato **risoluzione delle collisioni** (collision resolution).

Tecniche di risoluzione delle collisioni - 1

Open addressing. Il programma, partendo dalla posizione (occupata) determinata dalla funzione di hashing, controlla le posizioni successive finché non trova una posizione libera:

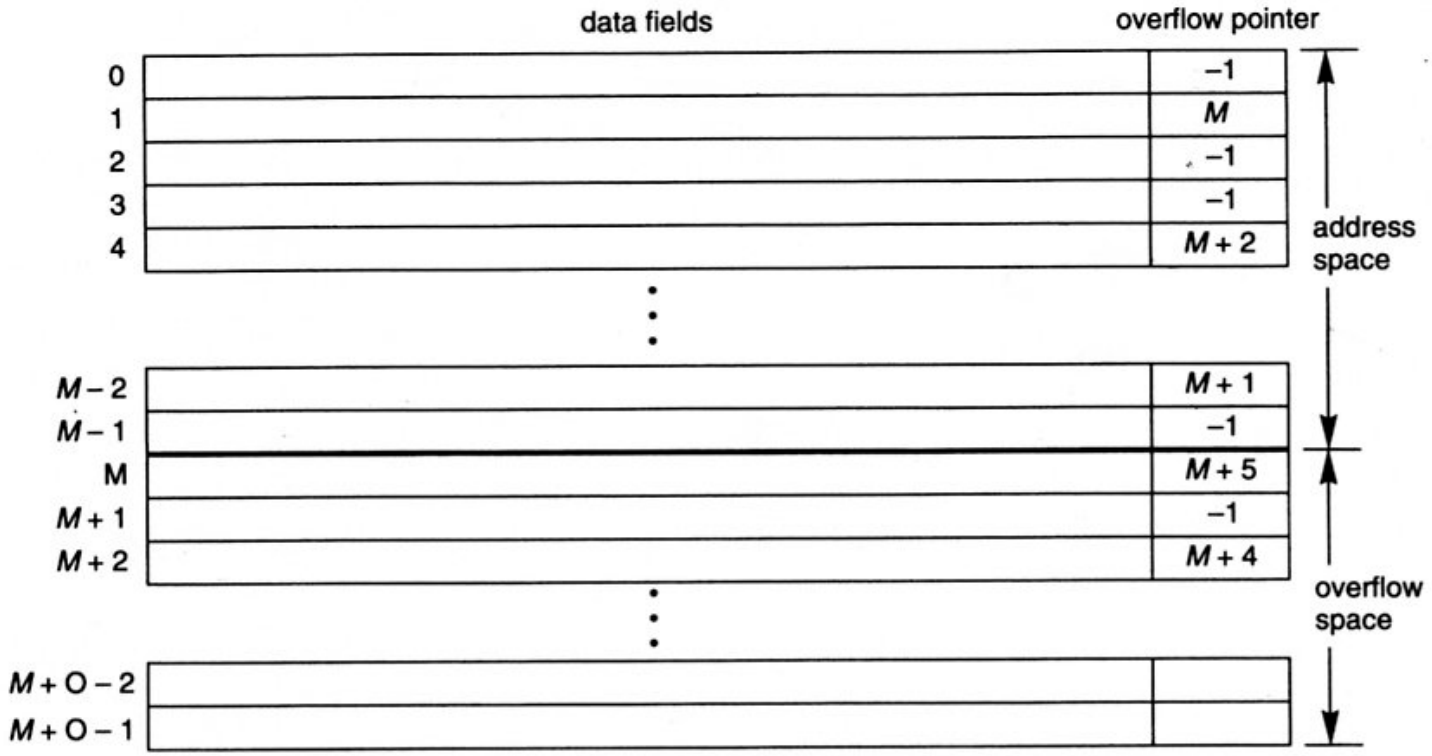
$i :=$ hash_address;
if location i is occupied
    then begin $i := (i + 1)$ MOD m;
        while ($i <>$ hash_address) **and** location i is occupied
            do $i := (i + 1)$ MOD m;
        if ($i =$ hash_address) **then** all positions are occupied
            else new_hash_address $:= i$;
        end;

Tecniche di risoluzione delle collisioni - 2

Multiple hashing. Questo metodo utilizza una seconda funzione di hash se l'applicazione della prima ha prodotto una collisione. Se si verifica una seconda collisione, o si utilizza il metodo di open addressing o si applica una terza funzione di hash, e così via.

Chaining. Questo metodo considera diverse locazioni di overflow, estendendo l'array con un certo numero di posizioni. Inoltre, esso aggiunge un campo puntatore ad ogni posizione dell'array. Ogni collisione è risolta posizionando il nuovo record in una locazione di overflow non utilizzata e settando il puntatore dell'ultima posizione occupata incontrata all'indirizzo della locazione di overflow (vedi la figura alla pagina successiva).

Tecniche di risoluzione delle collisioni - 3



- null pointer = −1.
- overflow pointer refers to position of next record in linked list.

Soluzione ottimale

L'obiettivo di una funzione di hashing è quello di distribuire i record uniformemente nello spazio degli indirizzi, in modo da minimizzare le collisioni e non lasciare troppe locazioni inutilizzate.

Simulazioni e studi hanno dimostrato che la **soluzione ottimale** è quella di avere una hash table completa dal 70% al 90%, in modo che il numero di collisioni rimanga basso e, al tempo stesso, non si sprechi troppo spazio.

Esempio. Se immaginiamo di dover memorizzare r record nella tabella, dobbiamo utilizzare uno spazio degli indirizzi comprendente m locazioni in modo che il rapporto r/m sia compreso tra 0.7 e 0.9.

Alcune funzioni di hash richiedono che m sia una potenza di 2.

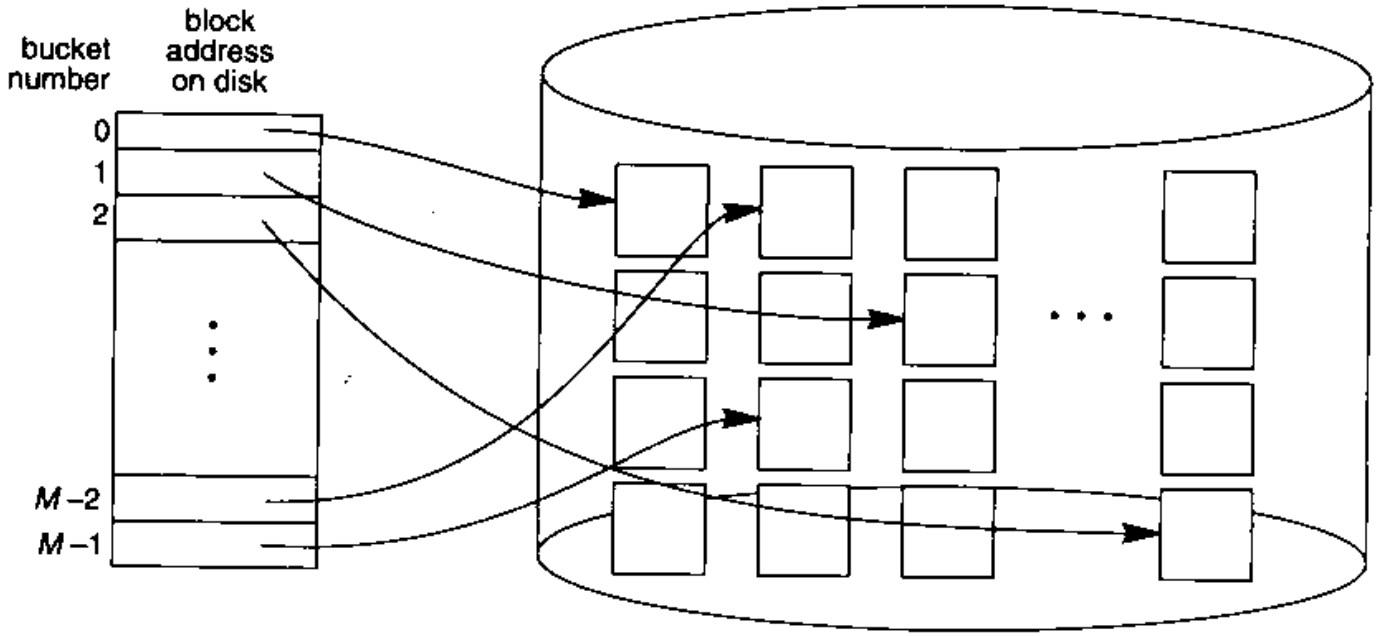
External hashing - 1

La tecnica di hashing per file che risiedono su disco (disk file) è chiamata **external hashing**.

Date le caratteristiche della memorizzazione su disco, lo spazio degli indirizzi è composto di celle (bucket), ognuna delle quali contiene più record. Una cella può essere costituita sia da un singolo blocco che da un insieme di blocchi contigui.

Una tabella contenuta nel file header converte il numero di cella nel corrispondente indirizzo del blocco sul disco (vedi figura).

External hashing - 2



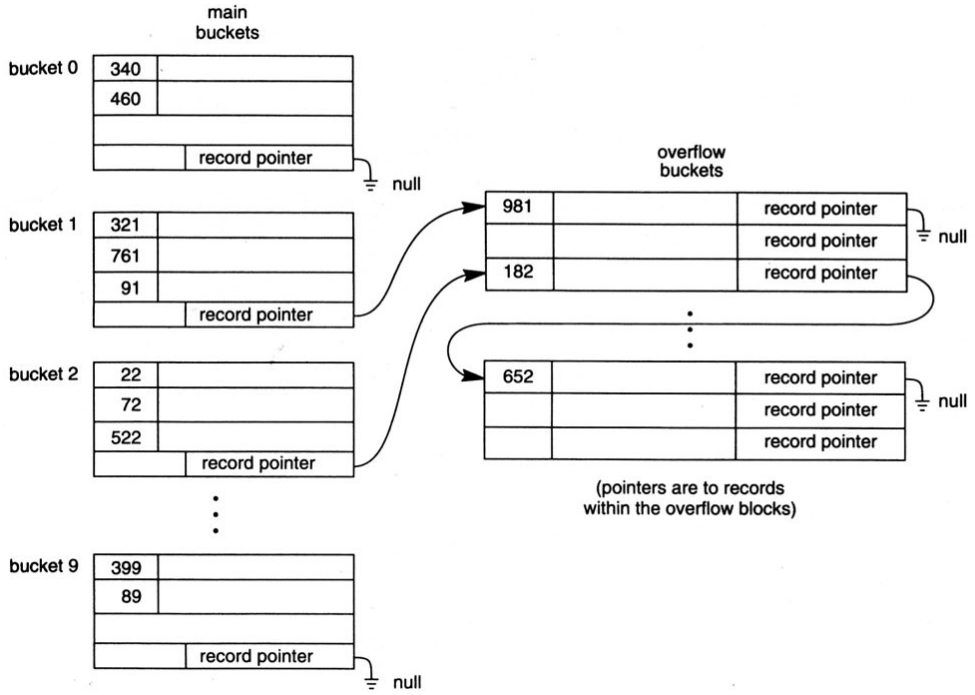
External hashing - 3

Il problema delle collisioni è meno grave rispetto all'internal hashing, poiché molti record possono essere contenuti nella stessa cella.

Insorgono dei problemi solo quando una cella è piena e un nuovo record deve essere inserito in quella cella.

In tal caso, si usa una variante del metodo di chaining visto in precedenza, in cui in ogni cella si mantiene un puntatore ad una lista di record di overflow per quella cella (vedi figura).

External hashing - 4



N.B. Un record pointer include sia l'indirizzo di un blocco che la posizione relativa del record all'interno del blocco.

Un limite significativo

Un limite rilevante delle tecniche di hashing è che lo spazio allocato per un file è fisso.

Supponiamo di voler allocare M celle per lo spazio degli indirizzi e sia m il numero massimo di record che possono essere contenuti in una cella; ne segue che, al massimo, potranno essere allocati $m \cdot M$ record.

Se il numero di record risulta $<< m \cdot M$ si avrà una grossa quantità di spazio inutilizzato, mentre se sarà $>> m \cdot M$ si avranno numerose collisioni (e le operazioni di retrieval saranno notevolmente rallentate).

Soluzione. Cambiare il numero di blocchi allocati e, quindi, utilizzare una diversa funzione di hashing per ridistribuire i record nelle varie celle.

Conseguenza: il numero di celle deve variare dinamicamente.

Vengono usate tecniche di hashing con **espansione dinamica** del file.

Espansione dinamica del file

Esistono (almeno) **tre tecniche di hashing** diverse che permettono di espandere dinamicamente i file:

- dynamic hashing;
- extendible hashing;
- linear hashing.

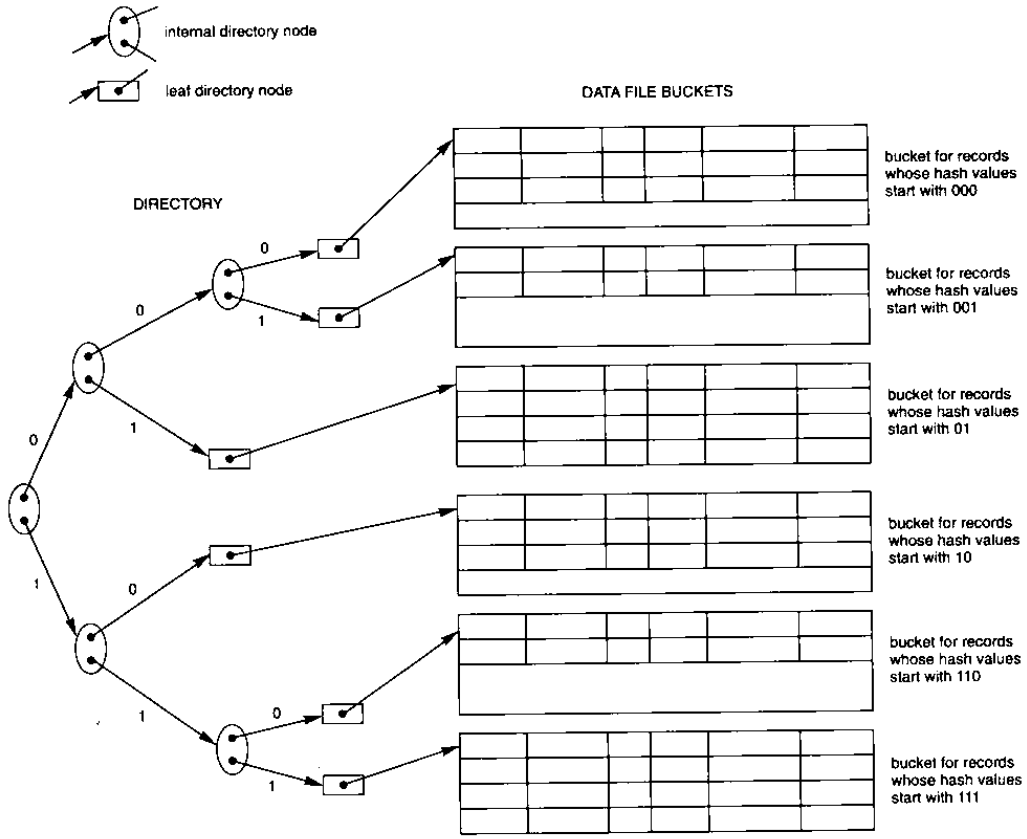
Dynamic hashing - 1

Il numero di celle non è fisso (come nel caso dell'hashing statico), ma può crescere o diminuire a seconda delle necessità.

Splitting. Il file inizia con una sola cella; quando tale cella è piena e un nuovo record deve essere inserito, la cella viene divisa in due. I record sono distribuiti tra le due celle sulla base del valore del primo bit (bit più significativo) dei loro valori di hash.

Memorizzazione. I record i cui valori di hash hanno il primo bit di valore 0 sono memorizzati in una cella, mentre quelli il cui hash value inizia con un bit di valore 1 sono memorizzati nell'altra cella. Tale processo viene iterato qualora una delle due celle risulti piena, utilizzando il secondo bit più significativo, e così via.

Dynamic hashing - 2



Dynamic hashing - 3

I nodi interni (ellissi) guidano la ricerca:

- puntatore sinistro corrisponde al bit 0;
- puntatore destro corrisponde al bit 1.

I nodi foglia (rettangoli) mantengono un puntatore ad una cella.

Implementazione del dynamic hashing

Organizzazione dei dati. Viene costruita una struttura ad albero binario (chiamata *directory* o *index*) caratterizzata da due tipi di nodi:

- Nodi interni (*internal node*) per guidare la ricerca. Ognuno ha un puntatore di sinistra, corrispondente al bit 0, ed uno di destra, corrispondente al bit 1.
- Nodi foglia (*leaf node*) contenenti un puntatore ad una cella (indirizzo di una cella).

Procedura di ricerca. La ricerca di un record viene effettuata mediante l'algoritmo riportato nel lucido successivo.

Algoritmo di ricerca per il dynamic hashing

$h :=$ hash value of record;
$t :=$ root node of directory;
$i := 1$;
while t is an internal node of the directory **do**
    begin
    if the i-th bit of h is a 0
        then $t :=$ left son of t
        else $t :=$ right son of t;
    $i := i + 1$
    end;
search the bucket whose address is in node t;

Gestione della directory

La **directory** viene mantenuta nella memoria principale finché non diventa troppo grande. Se occupa più di un blocco, viene distribuita su due o più livelli. Ogni nodo interno ha uno speciale bit (tag bit) che specifica il tipo di nodo, i puntatori left e right e un parent pointer. Ogni nodo foglia contiene l'indirizzo di una cella.

Se una cella diventa completamente occupata (overflow), viene divisa in due ed i record distribuiti sulla base del bit più significativo, non ancora utilizzato, del loro valore hash.

Esempio

Se un nuovo record è inserito nella cella A, contenente record il cui valore di hash inizia con 10, e causa overflow, la cella A è divisa in due celle $B1$ e $B2$. Tutti i record il cui hash value inizia con 100 sono posti nella cella $B1$, mentre quelli il cui hash value inizia con 101 sono posti in $B2$.

La directory è espansa con un nuovo nodo interno, che tiene traccia della sostituzione di A con $B1$ e $B2$ attraverso i puntatori ai due nodi foglia che indirizzano le due celle $B1$ e $B2$. I livelli dell'albero binario possono quindi variare dinamicamente (il numero di livelli non può comunque superare il numero di bit del valore di hash).

Ricompattamento dinamico

Se la funzione di hash distribuisce i record uniformemente, il directory tree risulterà bilanciato.

Le celle possono essere ricompattate dinamicamente se una diventa vuota o se il numero totale di record in due celle vicine può essere contenuto in una singola cella.

Extensible hashing - 1

E' basato su un tipo di *directory* differente: un array di 2^d celle di indirizzi di celle di memoria.

L'esponente d è detto *global depth* della directory.

Il valore intero corrispondente ai primi d bit (più significativi) del valore di hash è utilizzato come indice per l'accesso all'array/directory; nell'elemento dell'array così determinato è contenuto l'indirizzo della cella ove è memorizzato il record.

Extensible hashing - 2

In generale, ad ognuna delle 2^d locazioni della directory non corrisponde una cella distinta: diverse locazioni della directory che coincidono sui primi d' (con $d' < d$) bit del loro hash value possono contenere lo stesso indirizzo; in tal caso, tutti i record che fanno riferimento a tali locazioni appartengono alla stessa cella.

Una **local depth** d' (con $d' \leq d$), associata ad ogni cella, specifica il numero di bit su cui si basa il contenuto della cella.

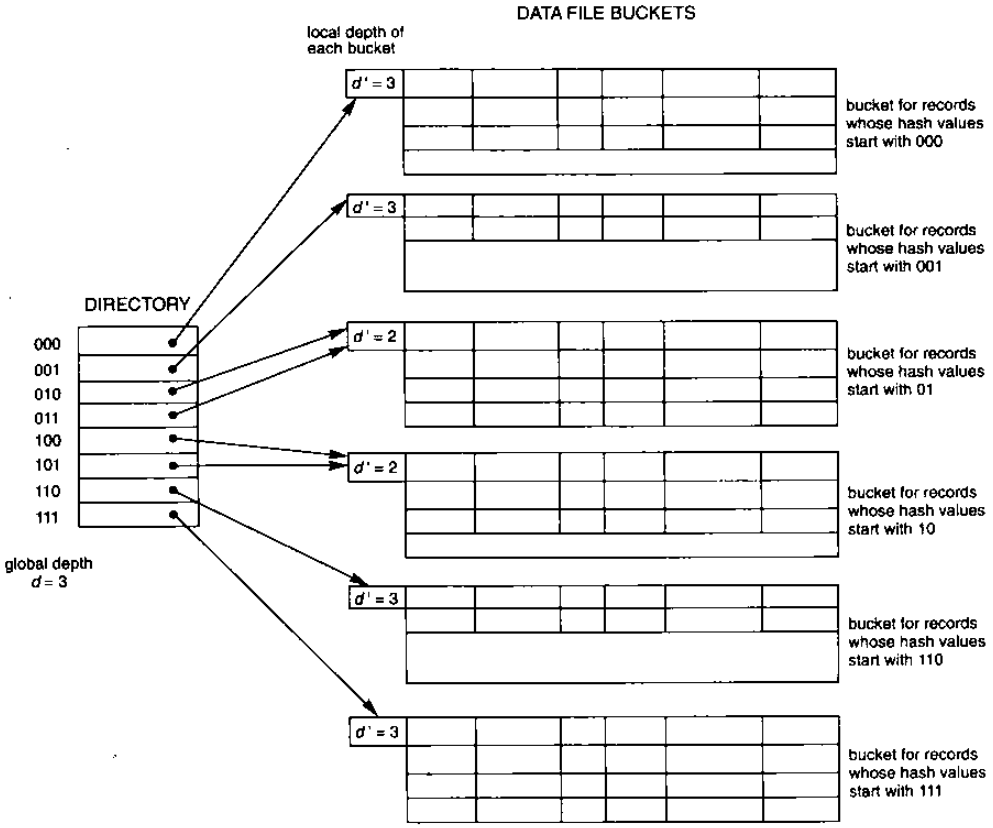
Extensible hashing - 3

La **global depth** d può aumentare o diminuire, raddoppiando o dimezzando il numero di elementi dell'array/directory.

Il **raddoppio** si rende necessario se in una cella la cui local depth d' è uguale alla global depth d si verifica un overflow ($d = d'$).

Il **dimezzamento** è possibile quando, dopo aver effettuato alcune operazioni di cancellazione, si ha che $d > d'$ per tutte le celle.

Extensible hashing - 4



$d =$ **global depth** $= 3$; $d' =$ **local depth**= numero di bit sui quali si basa il contenuto di una cella (in figura, $d' = 2$ o $d' = 3$)

Splitting di celle: un esempio - 1

Supponiamo che l'inserimento di un nuovo record causi un overflow nella cella il cui hash value inizia con 01.

I record saranno distribuiti in due celle: la prima conterrà tutti i record il cui hash value inizia con 010, mentre la seconda conterrà tutti i record il cui hash value inizia con 011.

Le due directory location 010 e 011 punteranno a due celle distinte (mentre prima dello split puntavano alla stessa cella).

La local depth d' delle due nuove celle è 3 (1 più della local depth della vecchia cella).

Splitting di celle: un esempio - 2

Se la cella contenente record il cui hash value inizia con 111 va in overflow, occorre estendere la directory, passando ad una global depth $d = 4$. Le due nuove celle saranno etichettate con 1110 e 1111 ed avranno local depth $d' = 4$.

La dimensione della directory risulta raddoppiata ed ognuna delle altre locazioni della directory originaria è splittata in due, mantenendo in entrambe le nuove locazioni lo stesso puntatore della locazione originale.

La massima dimensione della directory è 2^k, con k uguale al numero di bit dell'hash value.

Linear hashing - 1

L'idea alla base del linear hashing è quella di permettere ad un file hash di aumentare e/o diminuire dinamicamente il numero di celle senza l'ausilio di alcuna directory.

Supponiamo che il file contenga inizialmente m celle, numerate da 0 a $m-1$, ed utilizzi la funzione di hash $h(k) = k$ MOD m (initial hash function h_0).

Gli overflow causati dalle collisioni sono inizialmente gestiti in modo standard, associando a ciascuna cella una catena di overflow. Quando, però, una nuova collisione genera una situazione in cui vi è (almeno) un record di overflow per ogni cella del file, la prima cella del file (cella 0) viene suddivisa in due celle: la cella originale 0 e una nuova cella m alla fine del file. I record originariamente contenuti nella cella 0 vengono distribuiti tra le due celle sulla base della (nuova) funzione di hashing:

$$h_1(k) = k \text{ MOD } 2m$$

Linear hashing - 2

Una proprietà delle due funzioni di hashing h_0 e h_1 è che qualsiasi record che viene assegnato alla cella 0 dalla funzione h_0 è assegnato alla cella 0 o alla cella m dalla funzione h_1.

Se collisioni successive creano nuovi record di overflow, si procede alla suddivisione di altre celle, seguendo un ordine lineare (prima la cella 1, poi la cella 2, poi la cella 3, e così via).

I record di overflow vengono così ridistribuiti attraverso tale delayed split utilizzando la funzione h_1. Nessuna directory è necessaria; è sufficiente tener traccia del numero n di celle che sono state divise in due.

Linear hashing - 3

Per recuperare un record con hash key value k, si procede nel modo seguente:

- viene applicata la funzione h_0 a k;
- se $h_0(k) < n$, occorre applicare a k la funzione h_1 in quanto la cella è stata divisa in due.

Inizialmente, $n = 0$ e la funzione h_0 si applica a tutte le celle. Il valore di n cresce linearmente, in base al progressivo splitting delle celle.

Linear hashing - 4

Quando tutte le celle originali sono state divise in due ($n = m$), la funzione di hash h_1 va applicata direttamente a tutti i record del file.

A questo punto, la variabile n viene posta nuovamente a 0 e ad ogni nuova collisione che causa un overflow viene ripetuto il processo descritto in precedenza, utilizzando questa volta la funzione di base h_1 ed una nuova funzione di hashing:

$$h_2(k) = k \text{ MOD } 4m$$

Linear hashing - 5

In generale, viene usata una sequenza di funzioni di hashing del tipo:

$$h_j(k) = k \text{ MOD } (2^j m)$$

con $j = 0, 1, 2, \ldots$

Una nuova funzione di hashing h_{j+1} (e la ri-inizializzazione a 0 di n) si rende necessaria quando tutte le celle $0, 1, \ldots, (2^j m) - 1$ sono state splittate.

Algoritmo di ricerca per il linear hashing

if $n = 0$

then $m := h_j(K)$ # m is the hash value of record with hash key K

else begin

$m := h_j(K)$

if $m < n$ **then** $m := h_{j+1}(K)$

end

search the bucket whose hash value is m (and its overflow, if any)

Ricompattamento dinamico

Le celle che sono state spezzare in due possono essere ricombinate qualore il carico del file (file load factor) cada al di sotto di una certa soglia.

Si definisce **file load factor** il seguente valore l:

$$l = r/(bfr \cdot n)$$

dove r è il numero corrente di record presenti nel file, bfr è il numero massimo di record che possono essere contenuti in una cella e n è il numero corrente di celle del file.

Il file load factor può essere usato per mantenere *ottimale* il numero di celle del file, eseguendo divisioni o combinazioni di celle ogni qualvolta il fattore di carico oltrepassa opportune soglie prefissate (indicativamente, combinazioni se $l < 0.7$ e divisioni se $l > 0.9$).

Altre tecniche di organizzazione dei file

File di record misti. Le modalità di organizzazione dei file viste finora assumono che tutti i record di un particolare file siano dello stesso tipo.

Ad esempio, i record possono appartenere alle relazioni *IMPIEGATO*, *PROGETTO*, *STUDENTE* e *DIPARTIMENTO*, ma ogni file deve contenere solo record dello stesso tipo.

In molte applicazioni, esistono situazioni in cui diversi tipi di entità sono connessi in vari modi.

In tali situazioni, può essere utile rappresentare le relazioni tra record di file diversi attraverso **connecting field**.

Un esempio

Un record *STUDENTE* può avere un *connecting field* DIPMAGGIORE il cui valore fornisce il nome del DIPARTIMENTO in cui lo studente è majoring.

Il campo DIPMAGGIORE fa riferimento ad un'entità di tipo DIPARTIMENTO, rappresentata da un record del file DIPARTIMENTO.

Se si vogliono recuperare i valori di campi appartenenti a due record (di file diversi) tra loro collegati, occorre prima recuperare uno dei due record e poi usare il valore del suo *connecting field* per accedere al secondo record contenuto nell'altro file.

In tal modo, le relazioni sono implementate attraverso *logical field reference* tra record di file distinti.

L'uso delle physical relationship

L'organizzazione dei file nelle basi di dati gerarchiche e reticolari implementa le relazioni fra record attraverso *physical relationship*, che o sfruttano la contiguità fisica dei record coinvolti o utilizzano puntatori.

Questi modi di organizzare i file assegnano un'area del disco per memorizzare record di tipo diverso fra loro collegati in modo che possano essere fisicamente messi in relazione.

Esempio. Per gestire efficientemente interrogazioni che richiedano l'accesso ad un record di DIPARTIMENTO e a tutti i record di STUDENTI majoring in quel dipartimento, è utile memorizzare ogni record di DIPARTIMENTO ed il relativo insieme di record di STUDENTE in modo contiguo sul disco in un mixed file.

L'uso dei record type field

Per distinguere i record di un mixed file, ogni record ha associato, oltre al suo valore, un *record type field* che specifica il tipo di record.

Il record type field è generalmente il primo campo di ogni record ed è utilizzato dal sistema per determinare il tipo di record da elaborare.

Altre strutture dati utilizzate per l'organizzazione dei file nel caso in cui sia la dimensione dei record che il numero dei record siano limitati sono i B-alberi (verranno analizzate nel seguito).