

4. Memorizzazione di record ed organizzazione dei file

I database sono salvati in file memorizzati tipicamente su dischi magnetici. Esamineremo come vengono organizzati i DB fisicamente e descriveremo i metodi e le tecniche per rendere più efficienti le operazioni di manipolazione dei dati.

I dati in un computer sono memorizzati in forma binaria. Il più piccolo elemento di informazione è il bit, che può assumere valore 0 o 1. Otto bit formano un byte, necessari per descrivere un carattere. Per indicare grosse quantità di caratteri si usano i multipli del byte:

- KiloByte (Kb) = 1.024 (2^{10}) byte
- MegaByte (Mb) = 1.048.576 (2^{20}) byte
- GigaByte (Gb) = circa un miliardo di byte
- TeraByte (Tb) = circa mille miliardi di byte

La collezione di dati che costituisce una base di dati viene memorizzata su qualche supporto di memoria del calcolatore. Il DBMS recupera modifica ed elabora questi dati quando necessario. L'insieme delle memorie di un computer forma una gerarchia, divisa in due principali categorie:

- Memoria primaria, composta da tutte le memorie direttamente accessibili dalla CPU (central processing unit).
- Memoria secondaria, composta da dischi magnetici, dischi ottici e nastri.

Memoria primaria

Al primo livello della gerarchia, troviamo le memorie cache, formate da Static RAM, che operano in tempi confrontabili a quelli della CPU. Negli ultimi tempi, tali memorie sono inglobate direttamente nei chip delle CPU per fornire dati ad altissima velocità. Ovviamente hanno dimensioni molto limitate.

Al livello successivo troviamo la memoria *DRAM (dynamic RAM)*, la vera area di lavoro per la CPU: è chiamata anche *memoria centrale*. Memorizza i programmi ed i dati. Ha il vantaggio di essere relativamente economica ma ha il grande svantaggio di essere volatile, ovvero di perdere i dati contenuti in assenza di alimentazione elettrica. I grossi server possono arrivare a qualche decina di GigaByte di DRAM.

Le memorie secondarie sono di diversi ordini di grandezza più lente ed economiche rispetto alle memorie primarie. Le unità di misura dei tempi per accedere ai dati si misurano in millisecondi, contro i nanosecondi delle DRAM (10^6 più piccolo). I prezzi sono centinaia di volte inferiori rispetto alle DRAM: nel 2001 per i supporti ottici i prezzi sono sulle 5-6 £ a MegaByte. Sono dette anche **memorie di massa**. Hanno il vantaggio di essere non volatili.

Memoria Secondaria

Tipici esempi di memorie secondarie sono:

- Dischi ottici (CD-Rom, DVD)
- Dischi magnetici (Hard Disk) . Sono dispositivi **on-line** a cui si può accedere ogni momento.
- Nastri di backup : I dati memorizzati sul nastro sono **off-line**; è necessario l'intervento di un operatore o di un dispositivo automatico per caricare i nastri prima di rendere disponibili i dati.

A colmare il divario esistente tra le DRAM e le memorie di massa stanno sorgendo nuove

tecnologie. La più promettente sembra essere quella delle memorie Flash, che coniugano prestazioni vicine alle DRAM alla proprietà di non perdere dati in assenza di alimentazione, pur con qualche svantaggio (un intero blocco deve essere cancellato e scritto ogni volta).

I Cd-Rom sono un diffusissimo supporto di memorizzazione. Contengono fino a 700 Mb per disco. I dati, di tipo ottico, sono letti e scritti per mezzo di un laser. Spesso sono organizzati in batterie di decine di dischi (dette *Optical Juke-Box*), contenenti svariati Gb di dati. Hanno un tempo di accesso molto lento rispetto ai dischi magnetici. Stanno per essere sostituiti dai DVD, che possono contenere fino a 15-18 Gb per disco.

I nastri magnetici, per via del loro basso costo, sono usati principalmente per il backup di grosse quantità di dati. Forniscono un accesso sequenziale e non diretto ai dati. Sono molto lenti ed hanno un tempo di ricerca molto elevato. Batterie di nastri (dette *Tape Juke-Box*) permettono di archiviare diversi TeraByte di dati.

Memorizzazione di un database

In genere i database contengono grosse quantità di dati che devono essere persistenti per molto tempo. Sono quindi memorizzati in memorie secondarie (tipicamente dischi magnetici o nastri) poiché:

- Sono troppo grandi per essere memorizzati nella memoria principale.
- La memoria principale è volatile.
- I dischi sono molto meno costosi per unità di memorizzazione rispetto alla memoria centrale.

È compito del DBA decidere, tra le tecnologie proposte, quali permettono di ottenere il massimo delle prestazioni con la minima spesa. Il design del database fisico consiste nello scegliere tra le organizzazioni dei dati esistenti quella che meglio si adatta all'applicazione.

I dati memorizzati sul disco sono organizzati come file di record. I record dovrebbero essere memorizzati sul disco in modo da rendere efficiente l'individuazione della loro collocazione. Ci sono molte organizzazioni primarie dei file:

- Collocazione fisica sul disco e come ci si può accedere. Es:
 - File heap (file non ordinato).
 - File sequenziale (ordinato su un campo).
 - File hash (funzione hash applicata ad un campo particolare).
 - Btree (strutture ad albero).
- Un'organizzazione secondaria dei file consente un accesso efficiente ai record di un file basata sui campi alternativi rispetto a quelli usati per l'organizzazione primaria. La maggior parte di questi esistono come indici.

Periferiche di memorizzazione di massa

I dischi magnetici si dividono in due categorie:

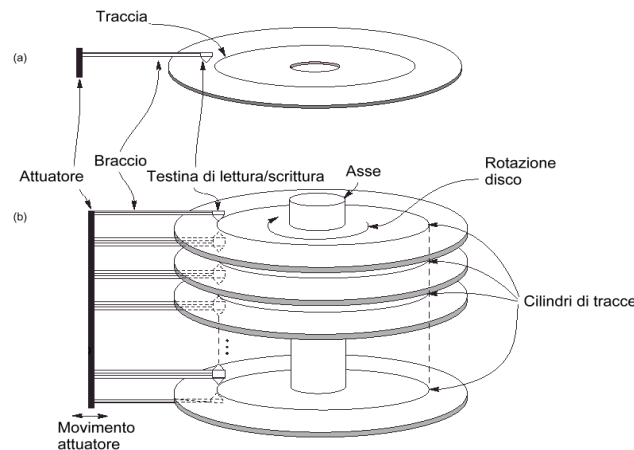
- Dischi fissi, con capacità nell'ordine di decine di GB.
- Dischi rimovibili, con capacità variabile da pochi Kb a centinaia di Mb.

Entrambi i tipi si basano sugli stessi concetti e su tecnologie molto simili.

Le informazioni su un disco sono organizzate in cerchi concentrici, chiamati *tracce*: in pack di dischi, l'insieme di tracce dello stesso diametro formano un *cilindro*. Una traccia è divisa in *settori*

(archi di circonferenza) o *blocchi*: Tale suddivisione viene fatta dal sistema operativo durante la *formattazione* del disco. La dimensione di un settore/blocco varia in genere da 512 bytes a 16 Kb.

I dischi sono periferiche ad accesso diretto (ogni settore è raggiungibile in tempo costante, a differenza dei nastri). La lettura/scrittura di informazioni viene effettuata da una testina mobile, che si posiziona su una particolare traccia. La rotazione del disco fa sì che i dati cercati passino sotto la testina.



Il tempo che impiega la testina per posizionarsi sulla traccia giusta è detto **seek time**. Il tempo trascorso finché l'inizio del blocco desiderato ruota nella posizione sotto la testina è detto **ritardo rotazionale** (o **latenza**).

Tempo totale di accesso ai dati = *seek time* + *latenza* + *tempo trasferimento blocchi* (**block transfer time**).

In genere il *seek time* e la *latenza* sono molto maggiori del *block transfer time*:

- Seek time medio: 8-10 millisecondi.
- Latenza media: 5-50 millisecondi.
- Trasferimento blocco: 1-2 millisecondi.

Sono tempi molto alti rispetto alla CPU o alla DRAM.

Per ottimizzare le prestazioni, si trasferiscono più blocchi consecutivi sulla stessa traccia o cilindro.

- Tempo totale lettura dati: ordine di 12 - 60 millisecondi.
- Localizzazione del primo blocco: da 12 a 60 millisecondi.
- Trasferimento di ogni blocco consecutivo: 1 o 2 millisecondi.

Visti i lunghi tempi richiesti per trovare dati (di diversi ordini di grandezza maggiori rispetto alla velocità di elaborazione della CPU), il principale *collo di bottiglia* nei database system è costituito dal localizzare i dati sul disco. Ciò fa capire quanto sia importante implementare strutture dati efficienti per il reperimento delle informazioni.

Bufferizzazione dei blocchi

In caso di lettura di più blocchi, si possono riservare ai dati in arrivo più aree di memoria (dette *buffer*).

Mentre un buffer è in lettura/scrittura, la CPU può processare i dati di un altro buffer:

- Ciò è possibile solo in casi di lettura DMA (Direct Memory Access), quando non è richiesto l'intervento della CPU per gestire i flussi di dati.
- Presenza di un processore indipendente per l'I/O del disco.

Se il tempo per processare un blocco in memoria è minore del tempo richiesto per leggere il blocco

successivo, si utilizza una tecnica detta **double buffering**:

- La CPU inizia a processare un blocco al termine del suo trasferimento verso la memoria.
- Contemporaneamente il disco sta inviando il blocco successivo in un altro buffer.

Si realizza così un flusso continuo di dati da e verso la memoria.

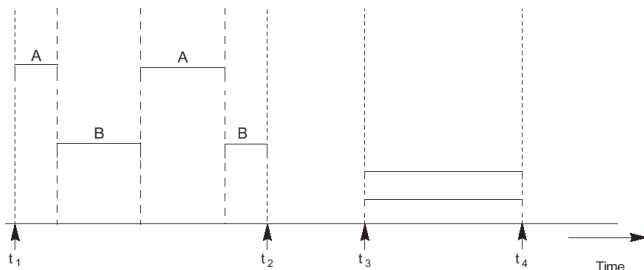


Illustrazione 2: I processi A e B eseguono in modo concorrente ma interleaved, mentre C e D sono concorrenti e paralleli. L'esecuzione parallela è possibile solo con più CPU.

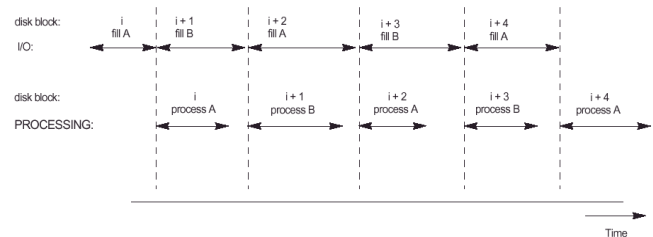


Illustrazione 1: Utilizzo di due buffer, A e B, nella lettura da disco. Vengono eliminati il seek time e il ritardo rotazionale per il trasferimento di tutti i blocchi, eccetto il primo.

Posizionamento dei file su disco

I dati sono di solito memorizzati sotto forma di record: ogni record è una collezione di valori di dati o item; ogni valore è formato da uno o più byte e corrisponde a un particolare campo del record. Nel modello relazionale, i record descrivono le entità ed i loro attributi.

Un tipo di *record* (o formato di record) è una collezione di nomi di campi e dei corrispondenti tipi di dati. Il *tipo di dato*, associato a ciascun campo, specifica il tipo dei valori che un campo può assumere. Tipi di dati possibili:

- *Numerici* (integer, long integer o floating point)
- *Stringhe di caratteri* (a lunghezza fissa o variabile)
- *Booleani* (0/1 o TRUE/FALSE)
- *Data/Ora* (date e time).

A ciascun tipo di dato corrisponde un'occupazione in byte, dipendente dal sistema usato.

- Integer → 4 byte
- Long integer → 8 byte
- Data → 10 byte (YYYY-MM-DD)
- Stringa → k byte (k numero di caratteri)

Nelle nuove applicazioni di db è sorta la necessità di nuovi tipi di dati per gestire informazioni multimediali (non strutturate). Per tali scopi si usa il tipo **BLOB** (Binary Large Objects). Il dato di tipo BLOB è memorizzato altrove e nel record è incluso un puntatore ad esso.

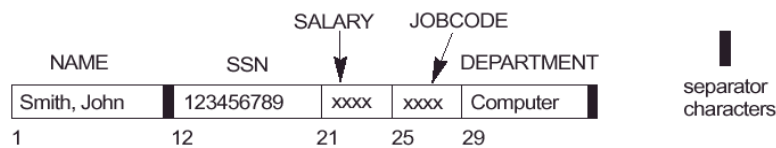
Un **file** è una sequenza di record. Quasi sempre un file contiene un unico tipo di record. Se ogni record ha esattamente la stessa dimensione (in byte), il file è detto essere costituito da *record a lunghezza fissa*. Se record diversi hanno dimensioni diverse, il file è detto essere costituito da *record a lunghezza variabile*.

Un file può essere costituito da record a lunghezza variabile per diversi motivi:

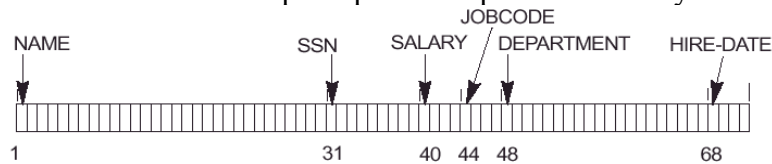
1. I record nel file sono dello stesso tipo, ma alcuni campi sono di *lunghezza variabile*.

2. I record nel file sono dello stesso tipo, ma uno o più campi possono avere valori multipli per record individuali. Un tale campo è detto *repeating field* e un gruppo di valori per il campo è spesso detto *repeating group*.
3. I record nel file sono dello stesso tipo, ma uno o più campi sono *facoltativi*. In corrispondenza di un campo facoltativo ci potrebbero essere valori per alcuni ma non per tutti i record del file.
4. Il file contiene *tipi di record diversi*, con dimensione diversa (detto file misto). Ciò accade se record correlati che hanno tipi diversi sono posti insieme in un blocco del disco; **Es:** i record VOTAZIONE di uno studente possono essere posti subito dopo il record dello studente corrispondente.

Per determinare i byte all'interno di un particolare record che rappresentano ciascun campo, si possono usare speciali caratteri separatori, che non compaiono in nessun valore di un campo (Es: ?, % o §) per indicare il termine di un campo a lunghezza variabile; oppure si può memorizzare la lunghezza del campo nel record.



Il vantaggio dei file con record di lunghezza fissa è nella maggior facilità con cui si riesce ad individuare i valori dei campi. Poiché ogni record ha la stessa lunghezza, è possibile identificare la posizione del byte iniziale di ciascun campo rispetto alla posizione del byte iniziale del record.



Un file di record con campi facoltativi può essere formattato in modi diversi. Se il numero di campi per il tipo di record è elevato, ma il numero di campi che realmente compaiono in un record tipico è piccolo, si può includere in ciascun record una sequenza di coppie **<nome-campo, valore-campo>** piuttosto che solo i valori del campo.

Un'alternativa più pratica sarebbe di assegnare un codice “*tipo di campo*” a ciascun campo, ed includere in ciascun record una sequenza di coppie **<Tipo-campo><Valore-campo>**. Un file che contiene record di tipi diversi richiede per ciascun record un indicatore di tipo.

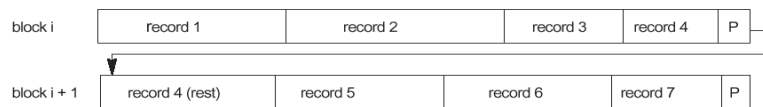
Un campo repeating ha bisogno di un carattere speciale per separare i valori repeating del campo e di un altro per indicare il termine del campo.

I record di un file devono essere allocati a blocchi di disco, poiché un blocco è la minima unità di dati trasferita tra disco e memoria. Se la dimensione di un blocco è maggiore della dimensione di un record, ogni blocco conterrà più record.

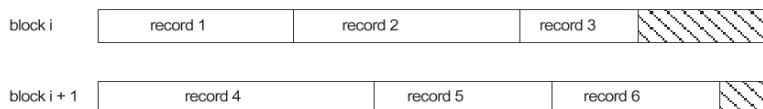
Supponiamo che la dimensione di un blocco sia B byte:

- per un file di record a lunghezza fissa di dimensione R byte, con $B \geq R$, possiamo far entrare $\mathbf{bfr} = \lceil \mathbf{(B/R)} \rceil$ record in ciascun blocco. Il valore **bfr** è detto *blocking factor* per il file.
- L'eventuale spazio inutilizzato (SI) in ciascun blocco è di $\mathbf{B - (bfr * R)}$ byte.
- Es:
 $B = 512$ byte $R = 45$ byte $bfr = 11$ (record per blocco)
 $SI = 512 - (11 * 45) = 512 - 495 = 17$ byte

Per non sprecare lo spazio alla fine di ogni blocco, si può spezzare un record su più blocchi. Un puntatore alla fine del primo blocco indica quale blocco contiene il resto del record. Tale organizzazione è detta *spanned*, ed è obbligatoria se $R > B$.



La suddivisione di record migliora l'utilizzazione dello spazio ma peggiora le prestazioni ed aumenta la complessità dei programmi che accedono ai record. L'organizzazione dei file che non permette di suddividere i record è detta *unspanned*, ed è usata con record a lunghezza fissa.



Allocazione dei blocchi .

Allocazione contigua: i blocchi del file sono allocati in blocchi consecutivi del disco.

- Lettura veloce di tutto il file con il doppio buffering.
- Espansione difficile del file.

Allocazione linked: Ciascun blocco del file contiene un puntatore al blocco successivo.

- Espansione del file più facile.
- Maggiore lentezza nella lettura di tutto il file.

Allocazione di cluster:

- Una combinazione delle prime due prevede l'allocazione di cluster di blocchi consecutivi del disco, dove i cluster sono legati gli uni agli altri.
- I cluster sono anche detti segmenti o extents.

Allocazione indicizzata:

- Uno o più blocchi di indici contengono puntatori ai blocchi reali del file.

Queste tecniche spesso sono usate in maniera combinata.

Un file header o descrittore di un file contiene informazioni sul file necessarie ai programmi che vi accedono, quali:

- informazioni per determinare gli indirizzi su disco dei blocchi del file.
- informazioni per registrare le descrizioni dei formati (lunghezze di campi e ordine dei campi per record a lunghezza fissa oppure codici di tipi di campo, caratteri separatori e codici di tipi di record per record a lunghezza variabile).

Per cercare un record su disco, uno o più blocchi vengono copiati in un buffer, utilizzando le informazioni dell'header. Se non è noto l'indirizzo del blocco che contiene il record desiderato, si deve effettuare una ricerca lineare attraverso i blocchi del file:

- ogni blocco del file viene copiato in un buffer.
- la ricerca continua finché non viene individuato il record nel buffer o finché tutti i blocchi del file sono stati scanditi senza successo.
- Tale tecnica è molto dispendiosa.

Operazioni sui File

Sono suddivise in operazioni di **retrieval** (o ritrovamento) e di **update** (aggiornamento):

Retrieval:

- Non cambiano i dati nel file.
- Localizzano i record per esaminarne/elaborarne i valori dei campi.

Update:

- Modificano i dati del file tramite:
 - inserimento record.
 - cancellazione record.
 - modifica di valori di campi.

Qualsiasi operazione è sempre preceduta dalla selezione di uno o più record, basata su una *condizione di selezione* che il o i record devono soddisfare. Se più di un record soddisfa un criterio di ricerca, viene localizzato il primo rispetto alla sequenza fisica di record. Il record più recentemente individuato è detto *record corrente*.

Le seguenti operazioni su file sono dette *record-at-a-time* (un record alla volta), poiché operano su un singolo record:

- **Find** (o **Locate**) – cerca il primo record che soddisfa il criterio di ricerca.
- **Read** (o **Get**) – copia il record corrente dal buffer in una variabile di programma del programma utente.
- **FindNext** – ricerca il record successivo.
- **Delete** – cancella il record corrente.
- **Modify** – modifica il valore dei campi del record corrente.
- **Insert** – inserisce un nuovo record prima nel buffer e poi sul disco.

Nel database system è possibile trovare delle operazioni dette *set-at-a-time* (un insieme per volta), che operano sull'intero file:

- **FindAll** – localizza tutti i record che soddisfano una condizione di ricerca.
- **FindOrdered** – recupera secondo un ordine specificato.
- **Reorganize** – comincia il processo di riorganizzazione.

Tra le altre operazioni per gestire un database troviamo:

- **Open** - prepara il file per l'accesso, ritrovando il file header e preparando il buffer di memoria per successive operazioni sul file.
- **Close** - termina l'accesso al file, rilasciando il buffer ed effettuando tutte le necessarie operazioni di clean-up.
- **Reset** - imposta il file pointer di un file aperto all'inizio del file.
- **Scan** – se sono state eseguite Open o Reset, restituisce il primo record altrimenti quello successivo. Snellisce le operazioni: Find, FindNext e Read.

File di record non ordinati

Gli **heap file** (o file sequenziali) sono la più semplice organizzazione di dati: i record sono posti nel file nell'ordine in cui sono inseriti, ed i nuovi record sono inseriti alla fine del file:

- Inserimento efficiente.

- Ricerca costosa: si applica una ricerca lineare nel file blocco per blocco;
- in media per un file di b blocchi si ricerca in $b/2$ blocchi.

Esistono due strategie per la cancellazione di un record:

- Fisicamente si cancellano i dati dal file, lasciando un “**buco**” tra i record.
- Si usa un *deletion marker*, ovvero un flag indicante che quel record è stato eliminato. I dati non sono cancellati fisicamente. Entrambe le tecniche di cancellazione richiedono una *riorganizzazione periodica* del file per riutilizzare lo spazio perso.

In un file di record a lunghezza fissa che usa blocchi unspanned e allocazione contigua, è possibile accedere ad un record tramite la sua posizione: se i record nel file sono numerati $0, 1, 2, \dots, r-1$ ed i record di ogni blocco $0, 1, 2, \dots, bfr-1$, allora l' i -mo record del file è localizzato al blocco $\lfloor i/bfr \rfloor$ ed è l' $(i \bmod bfr)$ -mo record nel blocco. Tale tipo di file è spesso detto relativo o diretto.

File di record ordinati

È possibile ordinare fisicamente i record di un file, in base ai valori di uno dei campi (*ordering field*): ne risulta un file ordinato o sequenziale. Se l'*ordering field* è un campo chiave, viene detto *ordering key* del file.

	NAME	SSN	BIRTHDATE	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					
		⋮				
block n-1	Wong, James					
	Wood, Donald					
			⋮			
	Woods, Manny					
block n	Wright, Pam					
	Wyatt, Charles					
			⋮			
	Zimmer, Byron					

Illustrazione 3: Un file ordinato sul campo
NAME

Vantaggi:

- Non è necessaria un'operazione di sorting prima di leggere i record nell'ordine del loro campo di ordinamento.
- Il ritrovamento del record successivo a quello corrente, basato sull'ordering field non richiede ulteriori accessi a blocchi.
- L'utilizzo di una condizione di ricerca basata sul valore di un ordering key field porta ad un accesso più veloce quando si usa una tecnica di ricerca binaria ...

Se la ricerca è su un campo ordering, si accede a soli $\log_2(b)$ blocchi. È un notevole miglioramento rispetto alla ricerca lineare. Non offre nessun vantaggio nell'accesso casuale né in quello ordinato se

la ricerca pone condizioni su campi non ordering. Offre maggiore efficienza su condizioni di ricerca quali $>$, $<$, \leq e \geq poste sull'ordering field perché tutti i record che soddisfano la condizione sono contigui fisicamente.

Una ricerca binaria per file su disco può essere fatta sui blocchi piuttosto che sui record. Cerchiamo un record che ha valore k di ordering key field, posto che il file ha b blocchi numerati $1, 2, \dots, b$, ed i record sono ordinati per valore crescente del loro ordering key field ...

```

l = 1;
u = b;      /* b è il numero di blocchi del file*/
while (u ≥ l) do
  begin
    i = (l + u) div 2;
    leggi l'i-mo blocco del file nel buffer;
    if K < (valore del campo ordering key del primo record nel blocco i)
      then u = i - 1
    else if K > (valore del campo ordering key dell'ultimo record nel blocco i)
      then l = i + 1
      else if (il record con valore del campo ordering key K è nel buffer)
        then goto found
        else goto notfound;
  end;
goto notfound;

```

Svantaggi: inserimenti e cancellazioni costose. Per inserire un nuovo record:

- si trova la sua posizione corretta nel file, basandosi sul valore dell'ordering field.
- si fa spazio nel file per inserirlo in quella posizione. In media la metà dei record devono essere spostati → metà dei blocchi devono essere letti e riscritti dopo che i record sono spostati.

Per rendere l'operazione di inserimento più efficiente si mantengono due file:

- Il main o master file: il file ordinato reale.
- Il file di overflow o transaction file: i nuovi record sono inseriti alla fine di esso.

Periodicamente l'overflow file, dopo essere stato ordinato, viene fuso con il main file.

Maggiore complessità della ricerca: se la ricerca nel file ordinato fallisce, si procede a una ricerca lineare nel file di overflow.

La modifica di un valore in un campo dipende da due fattori:

- Condizione di ricerca per trovare il record.
- Campo da modificare.

Se la condizione è su un campo ordering, si usa la ricerca binaria, altrimenti si usa la ricerca lineare. La modifica di un ordered field può richiedere uno spostamento del record per riordinare il file (cancellazione seguita da inserimento per rispettare l'ordinamento).

Tempi medi di accesso

Tipo di organizzazione	Metodo di accesso/ricerca	Tempo medio di accesso a un record specifico
Heap (non ordinato)	Scansione sequenziale (ricerca lineare)	$B/2$
Ordinata	Scansione sequenziale	$B/2$
Ordinata	Ricerca binaria	$\text{Log}_2 B$

$B = \text{numero di blocchi}$

Tecniche di hashing

Un'altra tecnica di organizzazione per i file è quella basata sull'*hashing*, che fornisce un accesso ai record molto veloce. La condizione di ricerca deve essere un'*uguaglianza*, posta su un *singolo* campo, detto *hash field*: Se il campo è chiave, si parla di *hash key*. L'idea è di avere una funzione **h**, detta *funzione hash* che applicata all'hash field di un record, restituisce l'indirizzo del disk block su cui è memorizzato il record.

Per la maggior parte dei record è sufficiente accedere a un singolo blocco per ritrovare il record.

- Negli heap file si analizzano in media $n/2$ blocchi per ritrovare il record.
- Nei file ordinati, con la ricerca binaria si analizzano $\log_2 n$ blocchi per ritrovare il record.

La tecnica hashing è usata a due livelli:

- Per organizzare i record in un file (**hashing interno**).
- Per organizzare i file sul disco (**hashing esterno**).

Hashing interno

L'hashing è implementato con una *tabella hash*, per mezzo di un *array di record*. Dato un array con range di indici da 0 a $M-1$:

- Ci sono M *slots*, i cui indirizzi corrispondono all'indice dell'array.
- La funzione hash trasforma il valore dell'hash field in un intero tra 0 e $M-1$.

	NAME	SSN	JOB	SALARY
0				
1				
2				
3				
	⋮			
$M-2$				
$M-1$				

Illustrazione 4: Tabella Hash con M slot, rappresentata da un array di record, di dimensione M.

Una tipica funzione hash è

$$h(K) = K \bmod M$$

con K valore del campo.

La funzione “*mod*” (il resto della divisione per M) restituisce un intero compreso tra 0 ed M-1. Valori di campi hash non interi possono essere trasformati in interi prima di applicare la funzione mod. Per stringhe di caratteri si usano i codici numerici (ASCII) associati ai caratteri.

La funzione hash mappa l'*hash field space* (numero di possibili valori che un campo hash può assumere) nell'*address space* (numero di indirizzi disponibili per i record). Poiché in genere l'*hash field space* è molto maggiore dell'*address space*, non è detto che a valori distinti corrispondano indirizzi distinti: se due record hanno lo stesso valore di hash, si verifica una *collisione*.

Una prima metodologia per evitare collisioni è scegliere una buona funzione di hashing. Ad esempio, effettuare l'hashing sulla prima lettera di una parola non è una buona scelta: *tutte le parole che iniziano con la stessa lettera andrebbero in collisione!*

Scelte migliori:

- Tecnica di folding: applicare una funzione aritmetica (es: addizione) o una logica (es: xor) a porzioni diverse del valore campo hash.
- Scegliere alcune cifre del valore del campo hash (es: la terza, la quinta e l'ottava) per formare l'indirizzo hash.

L'obiettivo di una buona funzione hash è di distribuire i record in modo uniforme nell'*address space*, minimizzando il numero di collisioni. I molti studi svolti sulle funzioni hash hanno portato alla formulazione di due linee guida:

1. mantenere una tavola hash piena al 70-90%. Esempio: per memorizzare r record si devono allocare M locazioni nella tabella per l'*address space* in modo che (r/M) sia compreso tra 0.7 e 0.9.
2. scegliere come M un numero primo o una potenza di 2: si ottiene una distribuzione migliore degli indirizzi hash nell'*address space* applicando la mod hashing function.

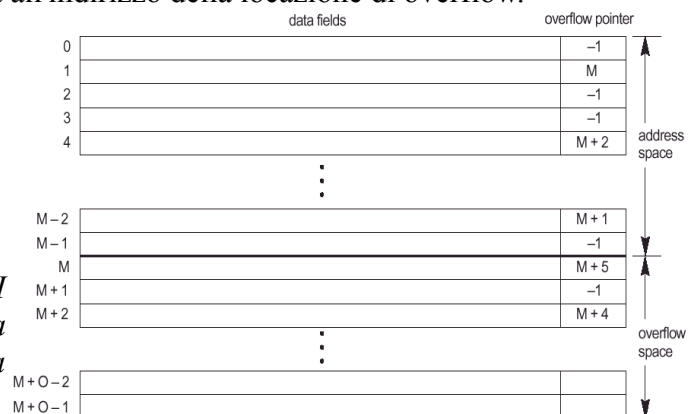
È possibile limitare ma non evitare completamente le collisioni: se nell'inserimento di un nuovo record, l'indirizzo hash risultante contiene già un altro record, si deve cercare una nuova posizione.

Metodi di risoluzione di collisioni:

Open addressing: A partire dalla posizione occupata si controllano gli slot successivi finché non se ne trova uno non utilizzato.

Chaining: Si allocano varie locazioni di overflow, estendendo l'array. Ogni slot contiene anche un campo puntatore. Una collisione si risolve inserendo il nuovo record in una locazione di overflow e settando il puntatore della locazione occupata all'indirizzo della locazione di overflow.

Esempio di chaining:



Il valore -1 sta per “null pointer”. I puntatori di overflow riferiscono alla posizione del record successivo nella lista linkata.

Multiple hashing: viene applicata una nuova funzione hash se la prima dà luogo ad una collisione; se avviene un'altra collisione, si usa l'open addressing oppure si applica una terza funzione hash e poi, se necessario, l'open addressing.

Hashing esterno

L'hashing esterno è utilizzato per i file su disco. L'address space è costituito da buckets: un bucket è un singolo blocco oppure un cluster di blocchi contigui; ogni bucket contiene più record. La funzione hash mappa una chiave in un numero di bucket relativo, piuttosto che assegnare al bucket un indirizzo di blocco assoluto.

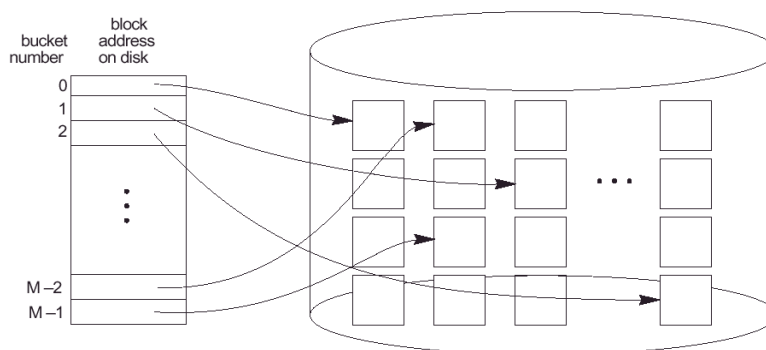


Illustrazione 5: Corrispondenza tra i numeri di bucket e gli indirizzi dei blocchi del disco.

Il problema delle collisioni è meno sentito: una funzione hash può associare diversi record allo stesso bucket fino a riempirlo. Al riempimento del bucket, si crea un puntatore a una lista linkata di record di overflow per tale bucket. I puntatori nella lista linkata sono dei record pointer, contenenti un indirizzo di blocco e una posizione relativa del record nel blocco.

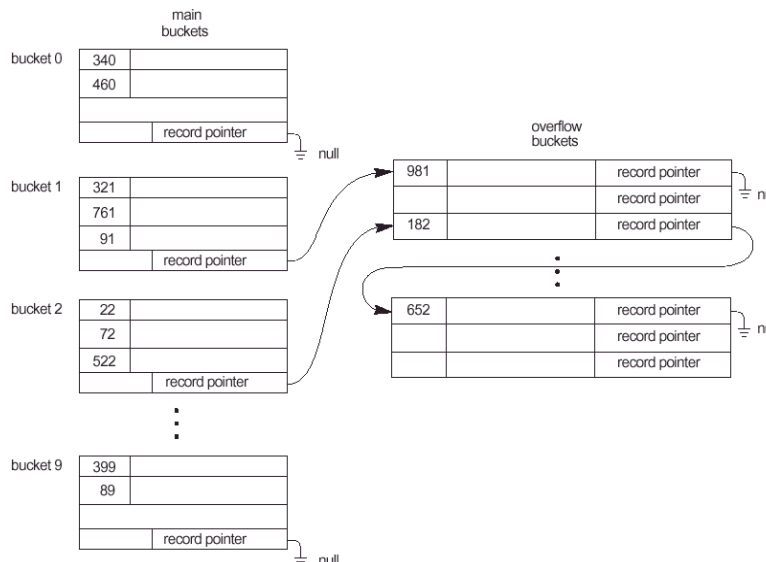


Illustrazione 6: Gestione dell'overflow nei buckets.

Caratteristiche della tecnica di hashing esterno :

- Massima velocità possibile di accesso per il retrieval di un record arbitrario, dato il valore del suo campo hash.
- Ricerche molto costose (lineari nel numero di bucket) se non effettuate sul campo hash.
- La cancellazione può essere implementata rimuovendo il record dal suo bucket. Se il bucket ha una lista di overflow, si sposta nel bucket uno dei record della lista.

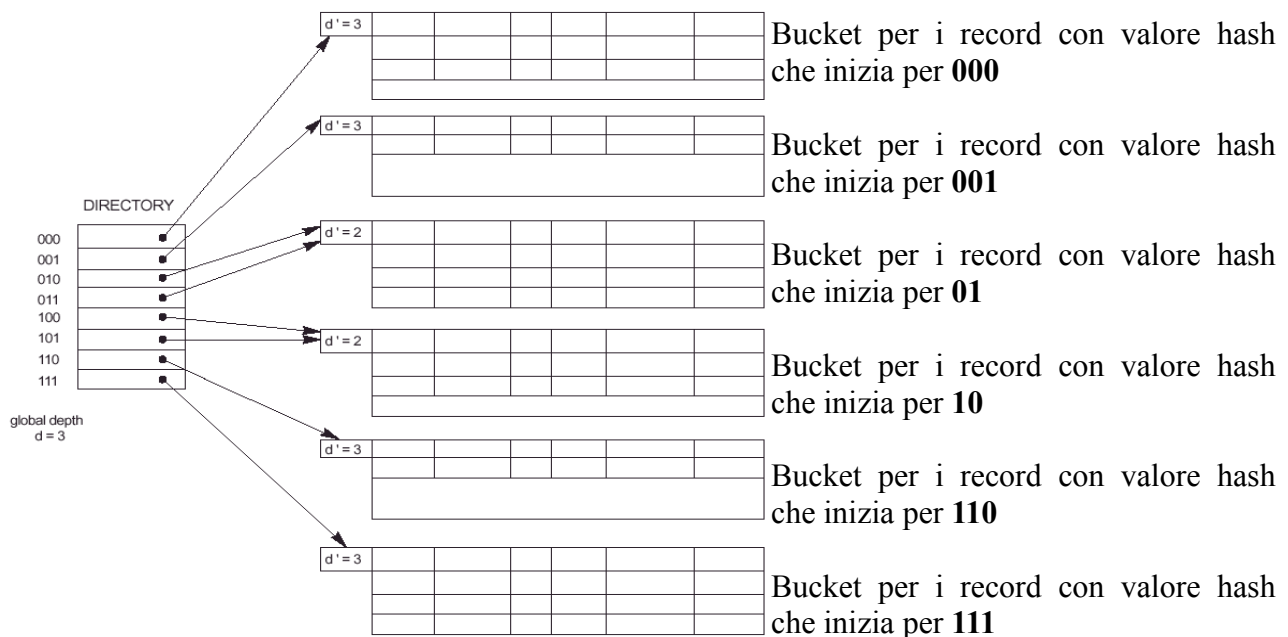
La tecnica vista è detta anche di *hashing statico*, poiché la quantità di spazio allocata al file è fissa:

- Sia M il numero di bucket allocati per l'address space ed m il numero massimo di record inseribili in un bucket. Allora $m \cdot M$ è il massimo numero di record che entrano nello spazio allocato.
- Se il numero di record è molto inferiore a $m \cdot M$ si ha molto spazio inutilizzato.
- Se il numero di record è molto superiore a $m \cdot M$ si hanno molte collisioni e un retrieval più lento dovuto a liste di record di overflow lunghe.

Esistono varie tecniche per permettere l'espansione dinamica dei file: **l'Hashing estendibile** e **l'Hashing lineare**. Tutte si avvantaggiano del fatto che il risultato di una funzione hash è un intero, e quindi è rappresentabile in notazione binaria.

Hashing estendibile

Nell'hashing estendibile si mantiene una struttura dati, detta directory, costituita da un array di 2^d indirizzi di bucket: d è detta profondità globale della directory. L'intero corrispondente ai d bit più significativi del valore hash costituisce l'indice per determinare la entry della directory. L'indirizzo contenuto nell'entry determina il bucket in cui sono memorizzati i record corrispondenti.



Non è necessario avere un bucket distinto per ognuna delle 2^d locazioni della directory: d' , **profondità locale** di un bucket specifica il numero di bit su cui si basano i contenuti del bucket.

Il valore di d può essere incrementato o decrementato di 1, raddoppiando o dimezzando il numero di entrate nell'array della directory: Si aumenta se per un bucket, $d' = d$ e si tenta di inserire un record, portando ad un overflow. Si diminuisce se, dopo qualche cancellazione, $d > d'$ per tutti i bucket.

La maggior parte dei record retrieval richiede due accessi a blocco: uno alla directory ed uno al bucket.

Esempio. Inserendo un record nel bucket contenente i record con valore hash che inizia con 01 (il 3° nell'ultima figura), si causa un overflow: Il bucket viene scisso ed i record verranno distribuiti su due bucket con profondità locale $d' = 3$ incrementata di 1

- 1° bucket: record il cui valore hash inizia con 010
- 2° bucket: record il cui valore hash inizia con 011

le locazioni 010 e 011 della directory che prima puntavano al singolo bucket ora puntano ai due nuovi bucket.

Se il bucket che va in overflow ha profondità locale $d' = d$, la dimensione della directory deve essere raddoppiata per poter usare un ulteriore bit con cui distinguere i due nuovi buckets: ciascuna locazione della directory originaria è scissa in due locazioni che puntano entrambe allo stesso bucket di quella originale.

Vantaggi:

- le performance del file non degradano se aumenta di dimensioni.
- I bucket addizionali sono allocati dinamicamente secondo le necessità.
- Riorganizzare il file significa solo spezzare un bucket e distribuire i record fra due bucket.
- Lo spazio richiesto per la directory è trascurabile.

Svantaggi:

- È necessario effettuare due accessi al blocco per reperire un record (uno per accedere alla directory + uno per accedere al bucket).

Hash lineare

Idea: espandere e ridurre dinamicamente il suo numero di bucket senza aver bisogno di una directory.

Supponiamo $0,1,\dots, M-1$ bucket, e una funzione di hash $h(K)=K \bmod M$, e overflow gestito ancora con catene di overflow. Quando una collisione porta a un record di overflow per ogni bucket del file, il primo bucket nel file (**bucket 0**) viene sdoppiato in 2 bucket (**bucket 0** e **bucket M**) e i record vengono distribuiti tra questi 2 bucket sulla base di un'altra funzione di hash, $h_{i+1}(K)=K \bmod 2M$ (che spedisce i record o al bucket 0 o a quello M). Ulteriori collisioni portano ad ulteriori sdoppiamenti nell'ordine lineare 1,2,3, ...

Se si verificano abbastanza overflow da avere $2M$ bucket allora tutti i bucket utilizzeranno la stessa funzione di hash h_{i+1} .

Tecnologia RAID

Il divario di prestazioni tra dischi e CPU è in continuo aumento: Le CPU raddoppiano la potenza di calcolo ogni diciotto mesi (legge di Moore) (aumento del 133% all'anno), la capacità delle memorie

aumenta del 150-200% all'anno, il tempo di accesso dei dischi migliora di meno del 10% all'anno, il transfer rate migliora di circa il 20% all'anno, la capacità dei dischi aumenta del 50% all'anno.

Utilizzando una batteria di n dischi, l'affidabilità diminuisce di n volte. Ad esempio, se un disco ha un *tempo medio perché si verifichi un guasto (Mean Time To Failure)* di 200.000 ore (22,8 anni), una batteria che ne impiega cento avrà dei malfunzionamenti ai dischi ogni 2.000 ore (83 giorni). Tenendo una singola copia dei dati in tale batteria porterebbe ad un'affidabilità bassissima.

La tecnologia **RAID**, acronimo di *Redundant Array of Inexpensive (o Independent) Disks*, batteria ridondante di dischi economici (o indipendenti), si pone come possibile rimedio ai problemi visti. Ha diverse implementazioni, identificate con 7 livelli dallo 0 al 6.

L'idea del RAID è di vedere a livello logico tanti piccoli dischi indipendenti come un unico grosso disco ad alte prestazioni (o ad alta affidabilità).

Il concetto principale è quello della **suddivisione dei dati** (data striping): un file viene suddiviso su più dischi, che possono leggerne i dati **in parallelo**, offrendo prestazioni molto superiori a quelle di un singolo disco con lettura sequenziale.



Illustrazione 7: Il file A è suddiviso in quattro parti, salvate su quattro dischi diversi che possono effettuare letture in parallelo.

È possibile aumentare l'affidabilità introducendo **ridondanza**: il RAID utilizza il concetto di **mirroring** (o **shadowing**) per tali scopi. I dati sono scritti su due dischi uguali, visti dal sistema come una singola periferica. Ciò migliora anche le prestazioni, poiché i dischi possono leggere in parallelo. Se un disco fallisce, viene usato l'altro disco.

Un altro approccio consiste nel salvare informazioni addizionali, che permettono di ricostruire le informazioni in caso di problemi (*codici a correzione di errori*). Le informazioni ridondanti o sono distribuite fra i dischi o sono memorizzate su dischi appositi.

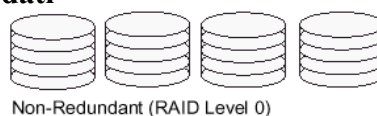
La principale tecnica per migliorare le prestazioni su disco è di suddividere i dati su più dischi per effettuare letture parallele. Lo striping si può avere a più livelli di granularità:

- *A livello di bit*. Si suddivide un byte in modo da scrivere il bit j sul j -mo disco.
- *A livello di blocchi*. Si scrive ogni blocco che compone un file su un disco diverso.

Lo striping abbassa l'affidabilità di un fattore pari al numero dei dischi. È necessario usare mirroring e codici a correzione di errori per accrescere l'affidabilità.

In base alle possibili combinazioni di granularità di striping e approcci per gestire la ridondanza, si hanno sette livelli di RAID:

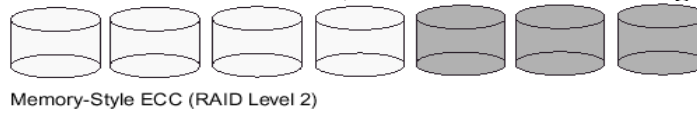
RAID 0: Nessuna ridondanza di dati



RAID 1: Dischi Mirrored



RAID 2: Ridondanza con correzione di errore, usando codici Hamming



RAID 3: Singolo disco di parità



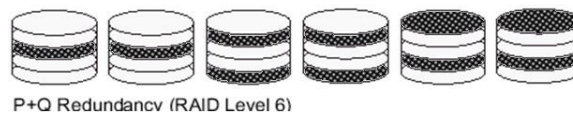
RAID 4: Suddivisione a livello di blocco e disco di parità



RAID 5: Blocchi e informazioni di parità suddivise su più dischi



RAID 6: Utilizza i codici Reed-Soloman per la ridondanza, per gestire il failure contemporaneo di due dischi con soli due dischi in più



Il livello 1 permette un'immediata ricostruzione dei dati in caso di crash, ed è utilizzato per applicazioni critiche (Es: log di transizioni) . I livelli 3 e 5 sono utilizzati per gestire grosse quantità di dati. Il livello 3 dà le prestazioni maggiori.