

BUFFER MANAGER

Il Buffer Manager è essenziale per migliorare l'efficienza e le prestazioni di un DBMS, garantendo un accesso rapido ai dati più utilizzati e mantenendo la coerenza tra la memoria volatile e quella permanente.

La sua funzione principale è gestire l'area di memoria temporanea, chiamata "**buffer pool**", dove vengono conservati i dati utilizzati di frequente.

Questo approccio consente al DBMS di accedere rapidamente ai dati senza doverli recuperare ogni volta dalla memoria secondaria, come un disco rigido, migliorando così l'efficienza complessiva del sistema.

Immagina un'applicazione che esegue una query SQL per cercare informazioni su un cliente:

1. Il **Buffer Manager** controlla se i dati sono già in memoria.
2. Se non lo sono, l'**Access File Manager** trova il file giusto e chiede al **Disk Manager** di recuperarlo dal disco.
3. Il **Disk Manager** legge il file e lo passa all'**Access File Manager**, che lo trasmette al **Buffer Manager**.
4. Il **Buffer Manager** salva i dati in RAM per futuri utilizzi e li restituisce all'utente.

Grazie a questo processo, se la stessa query viene eseguita più volte, il **DBMS** non dovrà ogni volta accedere al disco, rendendo il sistema molto più veloce.

Flusso dettagliato della richiesta di dati

1 . L'applicazione invia una query

- Il **Buffer Manager** controlla se i dati richiesti sono già nella **RAM (buffer pool)**.
- Se i dati sono presenti (**cache hit**), vengono subito restituiti all'applicazione.
- Se i dati **non sono presenti (cache miss)**, il Buffer Manager chiede all'**Access File Manager** dove trovare i dati.

2. L'Access File Manager individua il file

- Determina **quale file e quale posizione** contengono i dati.
- Passa l'informazione al **Buffer Manager**, che a sua volta chiede al **Disk Manager** di recuperare i blocchi di dati dal disco.

3. Il Disk Manager legge i dati dal disco

- Recupera i dati dalla **memoria secondaria**.

- **Restituisce direttamente i dati al Buffer Manager**, che li memorizza nel **buffer pool**.
4. **Il Buffer Manager restituisce i dati all'applicazione**
- Ora i dati sono disponibili in memoria principale, pronti per essere utilizzati.
 - Se l'applicazione richiede gli stessi dati in futuro, il **DBMS non dovrà accedere di nuovo al disco**, migliorando le prestazioni.

L'Access File Manager organizza i file, mentre il Disk Manager si occupa della lettura/scrittura fisica.

BUFFER POOL.

Il **buffer pool** è un'area della memoria principale (RAM) gestita dal **Buffer Manager**, dove vengono temporaneamente memorizzate le pagine di dati lette dal disco. Questo evita accessi frequenti alla memoria secondaria (disco), rendendo il DBMS molto più veloce.

Come è organizzato il Buffer Pool?

Il buffer pool è suddiviso in **frame**.

Ogni **frame** è una **pagina** della memoria principale, che corrisponde tipicamente alla dimensione di un **blocco** di dati sul disco.

La **dimensione del blocco** è l'unità di trasferimento tra la RAM e la memoria secondaria (tipicamente tra **2 e 64 KB**).

Come viene gestito il Buffer Pool?

Il buffer pool segue le stesse strategie delle **memorie cache** per massimizzare l'efficienza:

1. **Principio di località**

- Se un dato è stato letto di recente, è probabile che venga richiesto di nuovo a breve.
- Viene quindi mantenuto nel buffer il più a lungo possibile.

2. **Strategia di rimpiazzo a blocco**

- Se la memoria è piena e devono essere caricati nuovi dati, il DBMS utilizza un algoritmo di sostituzione per decidere **quale pagina eliminare**.
- Algoritmi comuni:
 - **LRU (Least Recently Used)** → rimuove la pagina usata meno di recente.
 - **FIFO (First-In-First-Out)** → rimuove la pagina più vecchia.

- **Clock Policy** → una variante ottimizzata di LRU.
-

3.

Euristiche di accesso

- Il principio "80-20": **l'80% delle richieste di accesso riguarda solo il 20% delle pagine del database.**
- Il buffer pool tiene quindi in memoria soprattutto le pagine più richieste.

Funzioni principali del Gestore del Buffer (Buffer Manager).

Il gestore del buffer fornisce una serie di operazioni primitive che consentono di gestire le pagine di dati nel buffer pool:



Fix (caricamento di una pagina)

- Carica una pagina dal disco nella memoria principale se non è già presente nel buffer pool.
- Se la pagina è già nel buffer, ne incrementa il contatore di utilizzo.

Per ciascun frame, il gestore del buffer mantiene:

- L'informazione su quale pagina esso contiene;
- **pin counter:** un contatore che indica quante transazioni attualmente attive utilizzano la pagina contenuta nel frame (inizialmente zero);
- **dirty bit:** un bit il cui valore indica se la pagina contenuta nel frame è stata modificata o meno (inizialmente impostato a falso);

Cosa sono il Pin Counter e il Dirty Bit?

- **Pin Counter** 
 - Tiene traccia di quante operazioni stanno usando una pagina nel buffer.
 - Se **Pin Counter > 0**, la pagina **non può essere rimpiazzata.**
 - Quando un'operazione termina, chiama **unfix** per ridurre il Pin Counter.
- **Dirty Bit** 
 - Indica se una pagina è stata modificata rispetto alla copia presente su disco.
 - Se **Dirty Bit = 1**, la pagina deve essere scritta su disco prima di essere sostituita.

Flusso dell'operazione di Fix

1. Verifica della presenza della pagina nel buffer

- Il **gestore del buffer** controlla se la pagina **P** è già memorizzata in un **frame F** del buffer pool.
- Se la pagina è presente, non c'è bisogno di leggerla dal disco.

2. Se la pagina è già nel buffer pool

- Viene incrementato il **pin counter** del frame **F** per indicare che la pagina è in uso e non può essere sostituita.

3. Se la pagina NON è nel buffer pool

- **Selezione di un frame libero o da rimpiazzare:**
 - Un frame **F'** viene scelto per contenere la nuova pagina, seguendo una **strategia di rimpiazzo** (es. LRU, FIFO, Clock Policy).
- **Gestione del dirty bit:**
 - Se il frame **F'** scelto contiene una pagina modificata (**dirty bit = 1**), allora la vecchia pagina viene **scritta su disco** (strategia di steal).
- **Caricamento della nuova pagina:**
 - La pagina **P** viene **letta dal disco** e scritta nel frame **F'**.
- **Aggiornamento del pin counter:**
 - Il **pin counter** della pagina **P** viene impostato a **1** per indicare che è in uso.

4. Restituzione dell'indirizzo della pagina

- Il Buffer Manager fornisce l'indirizzo del frame contenente **P**, pronto per essere utilizzato.

Unfix (rilascio di una pagina)

- Indica che una pagina non è più in uso.
- Se il contatore di utilizzo arriva a 0, la pagina può essere sostituita se necessario.
- La transazione rilascia la pagina da un frame.
- **il pin counter viene decrementato.**

Use (utilizzo di una pagina nel buffer pool)

- Indica che una pagina è attualmente in uso da un'operazione del DBMS.
- la transazione modifica il contenuto di una pagina:
il dirty bit viene impostato a true

Force (trasferimento sincrono verso la memoria secondaria)

- Scrive immediatamente una pagina modificata (dirty page) su disco.
- Viene usato per garantire la persistenza dei dati (ad esempio, in un'operazione di commit di una transazione).
- la transazione attende il completamento dell'operazione per andare in commit.

Flush (trasferimento asincrono verso la memoria secondaria)

- Scrive le pagine modificate su disco in modo **differito**, senza bloccare altre operazioni.
- Ottimizza le prestazioni riducendo le scritture frequenti su disco.

POLITICHE DI RIMPIAZZO.

Il **Buffer Manager** utilizza delle **strategie di rimpiazzo** per decidere quale pagina rimuovere dal buffer pool quando deve caricare una nuova pagina ma la memoria è piena.

STRATEGIA DI BASE.

Quando il **Buffer Manager** deve caricare una nuova pagina ma il buffer è pieno, deve scegliere un **frame da rimpiazzare** (la *vittima*).

- La scelta viene fatta tra i **frame con pin counter = 0**, cioè le pagine che **non sono attualmente in uso**.
- Se **non esiste alcun frame con pin counter = 0**:
 - La richiesta viene **messa in coda** fino a quando un frame diventa disponibile.
 - Se la situazione non si sblocca, **la transazione può essere abortita** per evitare un deadlock.

Se **tutte le pagine nel buffer sono in uso** (**pin counter > 0**), la transazione può essere bloccata finché una pagina non viene rilasciata (**unfix**). Se la situazione non si sblocca, **la transazione viene abortita**.

Immagina di avere un buffer pool con **3 frame occupati**:

Frame	Contiene Pagina	Pin Counter	Dirty Bit
F1	Pagina A	1	0
F2	Pagina B	0	1
F3	Pagina C	1	0

Se ora vogliamo caricare una nuova **Pagina D**:

1. Il Buffer Manager cerca un frame con **pin counter = 0** → trova **F2**.
2. Il Dirty Bit di F2 è **1**, quindi la **Pagina B viene scritta su disco** prima di essere sostituita.
3. La Pagina D viene caricata in **F2** e il suo **pin counter viene impostato a 1**.

LRU (Least Recently Used)

La politica **LRU (Last Recently Used)** è un algoritmo di rimpiazzo che tiene traccia dell'ordine di accesso alle pagine nel buffer e **sostituisce sempre la pagina utilizzata meno recentemente**.

Se il sistema usa LRU (Last Recently Used), per scegliere quale pagina eliminare sceglierà la pagina con il timestamp più vecchio, cioè quella usata meno recentemente.

Pagina	Ultimo Accesso (Timestamp)
Pagina A	10:15:30
Pagina B	10:18:45
Pagina C	10:12:20
Pagina D	10:20:05

Nel nostro esempio, la **Pagina C (10:12:20)** verrà sostituita.

Come funziona?

- Ordina le pagine per tempo di ultimo accesso (utilizzando una coda di priorità);
- Viene sempre rimpiazzata la pagina acceduta **meno recentemente**;

Ogni volta che una pagina viene **acceduta**, il suo timestamp viene aggiornato. Quando il **buffer pool è pieno**, il sistema sceglie per la rimozione la pagina con il timestamp più vecchio (cioè la meno usata di recente). La pagina rimpiazzata viene scritta su disco **se è stata modificata** (Dirty Bit = 1), altrimenti viene semplicemente rimossa.

PRATICA COSTOSA:

LRU è un algoritmo efficace ma costoso in termini di calcolo e memoria, perché:

Mantenere un ordine di accesso è complesso 🕒

- Ogni volta che una pagina viene letta o scritta, bisogna **aggiornare la sua posizione** nella coda di priorità o in una lista ordinata.
 - Questo richiede **operazioni costose di aggiornamento** (ad esempio, spostare la pagina in cima alla lista).
- Nei database di grandi dimensioni, mantenere traccia dell'ordine di accesso a migliaia/milioni di pagine **aumenta il carico computazionale**. Ogni accesso a una pagina implica una **modifica nella struttura dati** che gestisce l'ordine delle pagine, rallentando il DBMS.

Clock Approximation

L'algoritmo **Clock Approximation** è una versione ottimizzata di **LRU (Least Recently Used)** che riduce il costo computazionale mantenendo comunque una buona efficienza nella gestione del **buffer pool**.

Si mantiene un "**orologio dell'ultimo utilizzo**" 🕒

- Ogni **frame** del buffer è disposto in un **ordine circolare** (come le ore di un orologio).
- Un **puntatore (current)** si muove tra i frame, come la lancetta di un orologio.

Ogni frame ha un bit "**referenced**" 🕒

- Se una pagina viene utilizzata, il suo **referenced bit** viene **impostato a TRUE (1)**.
 - Se non è stata usata di recente, viene impostato a **FALSE (0)**.
- Quando il Buffer Manager ha bisogno di rimpiazzare una pagina:**

- Controlla il frame puntato da **current**:
 - **Se il pin counter > 0** → la pagina è in uso, quindi si passa al frame successivo.
 - **Se pin counter = 0 e referenced = TRUE** → il bit referenced viene impostato a **FALSE**, ma il frame non viene ancora rimosso. Il puntatore avanza.
 - **Se pin counter = 0 e referenced = FALSE** → il frame viene scelto come **vittima** per essere rimpiazzato.

Ripete il processo finché non trova una pagina da sostituire

- Il puntatore **scorre ciclicamente** il buffer finché non trova un frame candidabile alla sostituzione.

Non tiene traccia esatta dell'**ultimo accesso** a una pagina, ma utilizza un **bit referenced** per capire se è stata usata di recente. Le pagine più utilizzate **hanno più probabilità di rimanere** nel buffer rispetto a quelle meno usate. È molto più efficiente di LRU classico perché evita costosi aggiornamenti dei timestamp.

Immagina un buffer pool con **5 frame** e il puntatore **current** che si muove tra di essi:

Frame	Referenced Bit	Pin Counter	Stato
F1	1	0	✅ Controllato
F2	1	0	🔄 Referenced → 0
F3	0	0	❌ Viene rimosso
F4	1	1	🔒 Occupato
F5	0	0	Disponibile

Il puntatore **current** controlla il primo frame **F1**: ha referenced=1, quindi lo **imposta a 0** e avanza.

Stessa cosa per **F2**.

Quando arriva a **F3**, vede che referenced=0 e pin counter=0 → **sceglie questa pagina per il rimpiazzo**.

Most Recently Used (MRU)

L'algoritmo **MRU (Most Recently Used)** è una politica di rimpiazzo che, al contrario di LRU (Least Recently Used), sostituisce sempre la pagina utilizzata più di recente.

LRU (Least Recently Used) cerca di mantenere in memoria le pagine usate più di recente, pensando che saranno riutilizzate.

Normalmente, LRU è più efficiente perché tende a mantenere in memoria le pagine più utilizzate. Tuttavia, **in alcuni casi particolari**, MRU può essere più vantaggioso.

Supponiamo di dover leggere **1 milione di pagine dal file**, ma il buffer pool ha spazio solo per **1000 frame**. Se usassimo **LRU**, le pagine appena lette rimarrebbero in memoria, ma siccome stiamo leggendo il file in **modo sequenziale**, non ci servono più. **MRU è migliore in questo caso** perché rimpiazza subito le pagine appena lette, liberando spazio per le nuove pagine della scansione.

Leggere un file in modo sequenziale significa scorrere le pagine del file una dopo l'altra, come leggere un libro pagina dopo pagina, dall'inizio alla fine.

Questo fenomeno è chiamato "**Sequential Flooding**":

Ogni nuova pagina letta sostituisce una pagina già presente, causando **troppe operazioni di I/O**.

MRU aiuta a ridurre questo problema.


MRU (Most Recently Used) rimpiazza subito la pagina più recente, liberando spazio per la prossima pagina da leggere.

- Questo è perfetto per la lettura sequenziale, perché **non abbiamo bisogno delle pagine lette in precedenza**.
- MRU **riduce il numero di operazioni di I/O**, migliorando le prestazioni.




Supponiamo che il database debba scansionare una tabella con 1 milione di pagine, ma il buffer pool ha solo 5 frame:

Operazione	Buffer Pool (LRU)	Buffer Pool (MRU)
Lettura Pagina 1	1	1
Lettura Pagina 2	1, 2	1, 2
Lettura Pagina 3	1, 2, 3	1, 2, 3
Lettura Pagina 4	1, 2, 3, 4	1, 2, 3, 4
Lettura Pagina 5	1, 2, 3, 4, 5	1, 2, 3, 4, 5
Lettura Pagina 6	Sostituisce Pagina 1	Sostituisce Pagina 5

Con LRU → La **pagina più vecchia (1)** viene eliminata. 

Con MRU → La **pagina più recente (5)** viene eliminata.  Nel caso di **lettura sequenziale**, MRU garantisce che il buffer venga sempre liberato per le nuove pagine senza sprecare spazio con pagine non più necessarie.

Quando usare MRU?

-  Quando si eseguono **scansioni complete di grandi dataset**.
-  Quando si elaborano **grandi file in modo sequenziale**.
-  Quando le pagine appena lette **non saranno più necessarie a breve**.

Quando NON usare MRU?

- Se i dati vengono **riutilizzati frequentemente**, come nelle cache delle query.
- Se il carico di lavoro prevede **molti accessi casuali ai dati**.

MRU è perfetto per scansioni sequenziali, dove i dati vengono letti una sola volta. In questi casi, LRU non funziona bene perché spreca spazio nel buffer, mentre MRU libera spazio in modo più intelligente!

Strategie di Steal e Force nel Buffer Manager

I DBMS moderni usano la combinazione Steal + No-Force perché:

Steal → permette di liberare spazio nel buffer pool rimpiazzando pagine modificate.

No-Force → riduce il numero di scritture su disco, migliorando le prestazioni.

Steal: permette di rimpiazzare anche pagine **modificate** (**dirty = true**).

No-Steal: le pagine modificate **non possono essere rimpiazzate** finché la transazione non è terminata.

Steal consente al DBMS di liberare spazio nel buffer rimpiazzando pagine modificate.

No-Steal blocca le pagine nel buffer fino al termine della transazione (potenzialmente inefficiente).

✓ Quando usare Steal?

- Se il buffer è piccolo e serve spazio per altre pagine.
- Se vogliamo gestire grandi transazioni senza tenerle interamente in RAM.

✓ Quando usare No-Steal?

- Se vogliamo semplificare il **recovery**, evitando che dati **non commitati** vengano scritti su disco.

Force: tutte le pagine modificate vengono scritte su disco al commit della transazione

No-Force: le pagine rimangono nel buffer e vengono scritte su disco solo in seguito

- **Force** assicura che tutti i dati modificati siano **subito salvati su disco** al commit.

- **No-Force** ottimizza le prestazioni evitando scritture frequenti su disco.

✓ **Quando usare Force?**

- Se vogliamo garantire che i dati siano **immediatamente persistenti dopo il commit**.
- Se abbiamo transazioni brevi e vogliamo ridurre la complessità del recovery.

✓ **Quando usare No-Force?**

- Se vogliamo **ridurre il numero di scritture su disco**, migliorando le prestazioni.
- Se possiamo accettare che i dati di una transazione rimangano in memoria per un po' prima di essere scritti su disco.