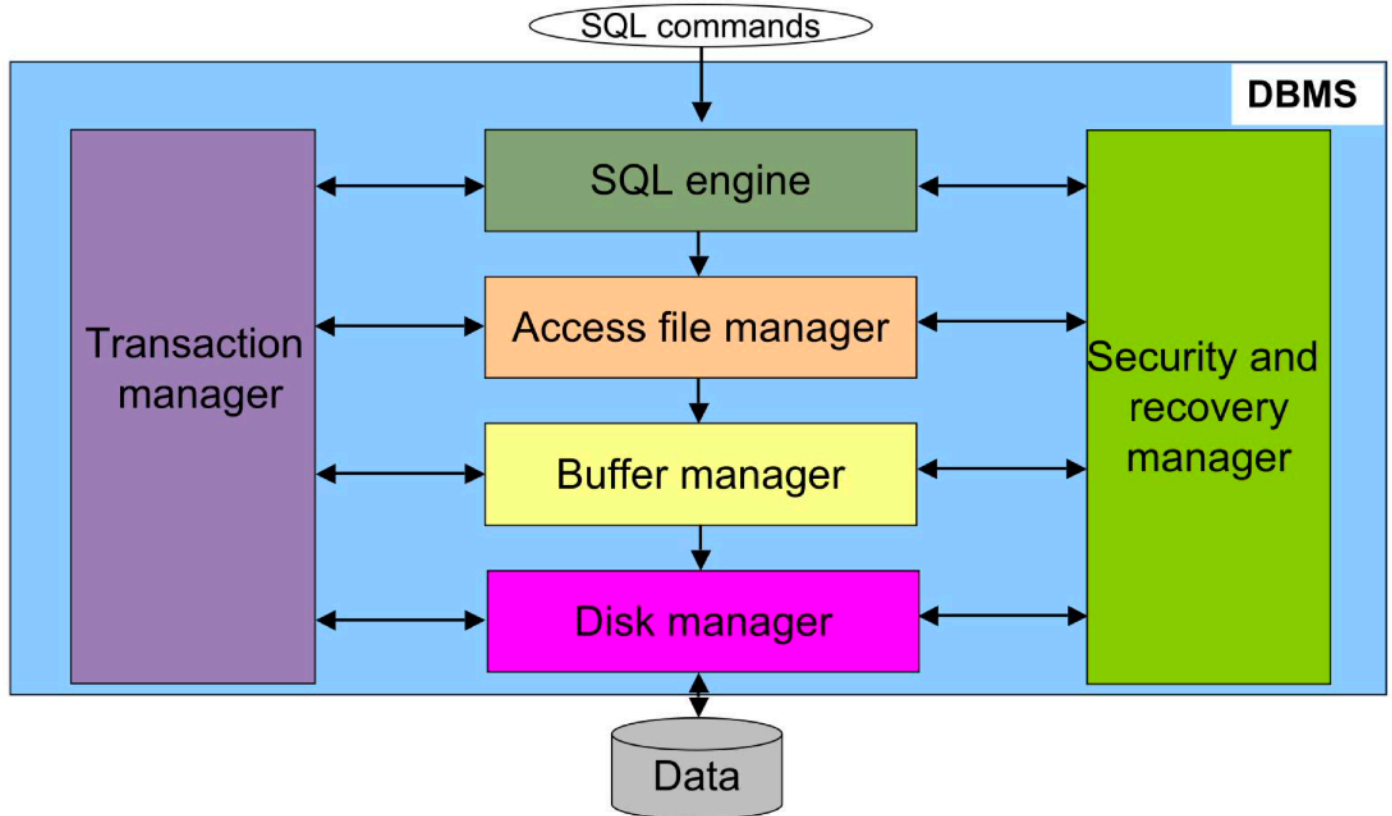


ORGANIZZAZIONE FISICA

La presente trattazione si propone di coniugare le nozioni acquisite nell'ambito del corso di Basi di Dati, limitatamente alla sezione curata dal Prof. Ing. A. Pellegrini, con i concetti appresi nel corso di Sistemi Operativi, tenuto dal Prof. F. Quaglia. Tale operazione si rende opportuna in quanto le tematiche affrontate nei due insegnamenti seguono direttrici concettuali affini.

Come è composto un DBMS?



Il DBMS interagisce con il mondo esterno attraverso comandi SQL, che vengono elaborati dal motore **SQL (SQL Engine)**.

Questo motore ha il compito di analizzare (Parsing) e interpretare le query ricevute, eseguendole in modo efficiente per garantire il corretto accesso e la gestione dei dati.

Il modulo **Access file manager** opera come un'interfaccia tra il **livello logico** (query SQL) e il **livello fisico** (file su disco).

Il **motore SQL** fornisce all'**Access File Manager** un insieme di operazioni basate sulla query ricevuta dall'utente, il cosiddetto **piano di esecuzione**, ovvero un insieme di operazioni da compiere per recuperare i dati.

L'**Access File Manager** traduce queste operazioni in comandi di basso livello per accedere ai file, in modo da capire se

leggere, scrivere, aggiornare o cancellare i dati presenti nel database.

Se più utenti accedono contemporaneamente agli stessi dati, il **Access File Manager** garantisce che non si verifichino anomalie utilizzando tecniche di **locking** per evitare che due utenti modifichino lo stesso dato in modo incoerente.

L' **Access File Manager** lavora in stretta collaborazione con il **Buffer Manager**.

Se i dati richiesti sono già in **memoria cache**, l'**Access File Manager** li recupera direttamente, evitando un accesso al disco e migliorando le prestazioni.

Se i dati non sono in cache, il **Access File Manager** richiede al **Disk Manager** di recuperarli dal disco.

Quindi l'**Access File Manager** interviene per tradurre la query in operazioni sui file.

Ad esempio: **SELECT titolo FROM film WHERE anno = 2020;**

I passi sono:

Apertura del file della tabella film.

Se la tabella è memorizzata come un file heap, la scansione avverrà sequenzialmente.

Se la tabella è memorizzata con un indice, verranno cercati solo i blocchi necessari.

Lettura dei blocchi di dati.

Il **Buffer Manager** verifica se i dati sono già in memoria cache.

Se i dati non sono in cache, il **Disk Manager** recupera i blocchi dal disco.

Applicazione dei filtri (WHERE anno = 2020).

I dati letti vengono confrontati con la condizione **anno = 2020**.

Solo le righe corrispondenti vengono mantenute.

Recupero della colonna richiesta (titolo).

Dopo aver filtrato i record, viene selezionata solo la colonna **titolo**.

Restituzione del risultato al motore SQL.

Il motore SQL riceve i dati filtrati e li restituisce all'utente.

Il **Disk Manager** è il componente del DBMS responsabile della **gestione fisica dei dati** su disco. Il suo compito principale è interagire con il file system per **leggere e scrivere i dati in modo efficiente**.

Riassumendo, il **Disk Manager** si occupa della lettura e scrittura fisica delle informazioni nel database.

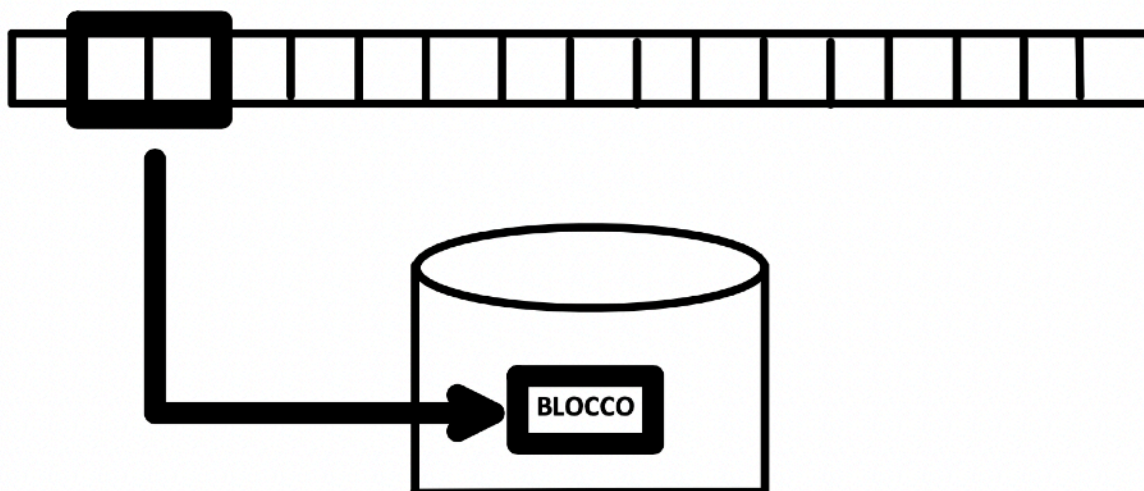
PAGINE, RECORD E BLOCCHI - Chiariamo

Il blocco è l'unità minima che il Disk Manager può leggere o scrivere dal disco alla RAM.

- Ciascun file è allocato sul dispositivo di memoria di massa come un insieme di blocchi (chiamati anche blocchi logici) non necessariamente contigui

Supponiamo di avere un contenuto di un file costituito da record: come vanno a finire questi record all'interno di blocchi di una memoria di massa?

Un blocco può ospitare una quantità superiore all'unità dei record.



All'interno di un blocco ci sono più pagine.

Una **pagina** è un'unità logica di memorizzazione nel database e contiene **più record** della stessa tabella. La dimensione di una pagina è fissa (es. 4 KB, 8 KB o 16 KB).

Il **record (o slot)** rappresenta una singola riga/tupla di una tabella.

All'interno di una **pagina**, i record sono archiviati sequenzialmente o con puntatori se la lunghezza è variabile. Quando una query richiede un record, il DBMS carica **l'intera pagina** in RAM per accedere al record richiesto.

Il DBMS carica sempre la pagina intera in RAM quando un record viene richiesto.

Se il DBMS usa una strategia basata sui blocchi, caricherà anche l'intero blocco che contiene la pagina.

L'approccio varia in base all'ottimizzazione del DBMS: alcuni caricano solo la pagina, altri preferiscono caricare blocchi interi per ridurre i futuri accessi al disco.

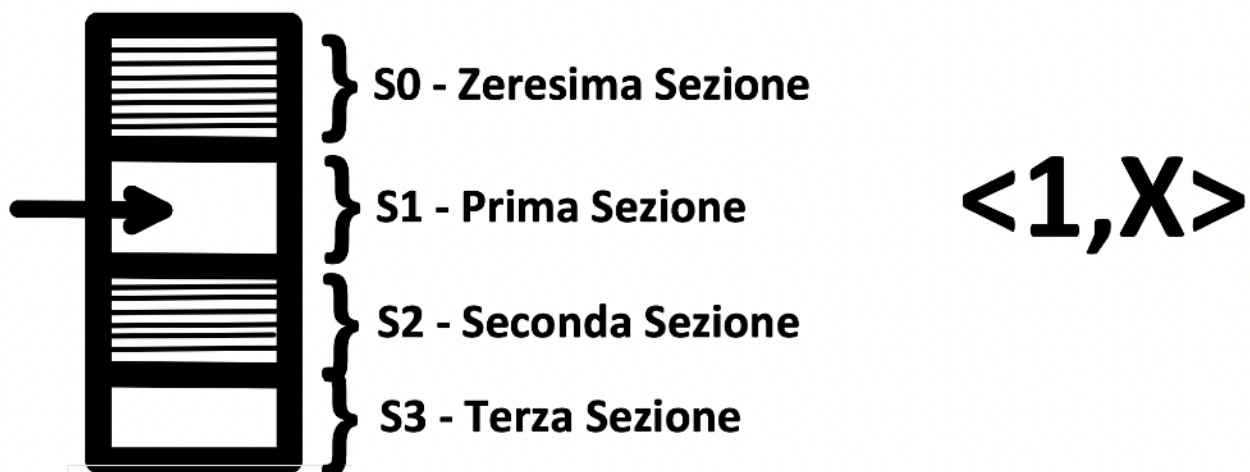
Approssimiamo la dimensione di un blocco a quella di una pagina, entriamo nella pagina.

Fisicamente la pagina è una collezione di slot(record) (uno per ciascun record). Ciascun record dentro la pagina ha un identificatore RID (Record ID):

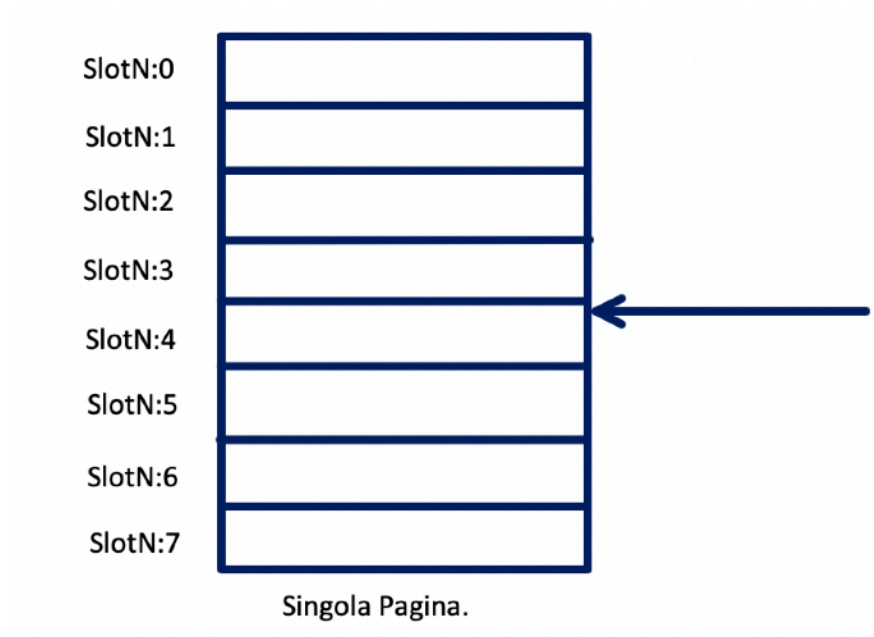
$$rid = \langle page\ id, slot\ number \rangle$$

Abbiamo una tabella "di primo livello" che è la directory della tabella delle pagine.

Il sistema operativo suddivide l'archiviazione in tante sezioni, ciascuna sezione contiene più pagine, ciascuna pagina contiene più record e ciascun record è numerato dallo slot Number.



Esempio di accesso: $\langle 1, 2, 4 \rangle$.
Prima sezione, pagina 2, slot 4.



Come Funziona l'Accesso ai Dati

Quando una query chiede un record, il DBMS segue questi passi:

1 Il **motore SQL** analizza la query e determina quali dati servono.

2 Il **Buffer Manager** verifica se la pagina è già in RAM.
Se la pagina è già in memoria, il record viene letto direttamente.

Se la pagina **non è in RAM**, il **Disk Manager** accede al disco.

3 Il **Disk Manager** recupera il blocco che contiene la pagina richiesta.

Se il DBMS lavora a livello di **pagina**, trasferisce solo quella specifica **pagina in RAM**.

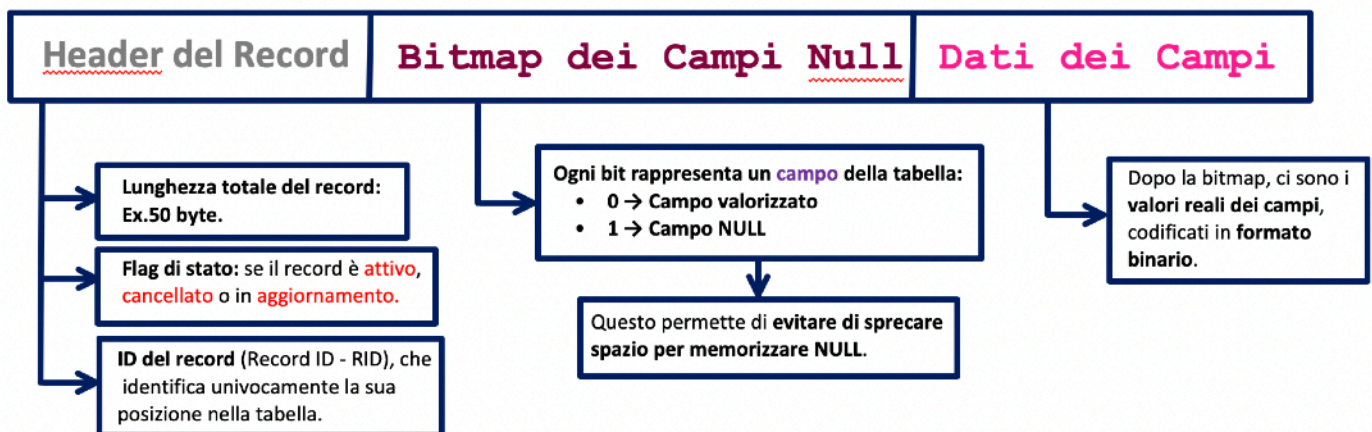
Se il DBMS lavora a livello di **blocco**, trasferisce tutte le pagine del **blocco in RAM**.

4 Il record viene estratto dalla pagina in RAM e restituito alla query.

Un record non viene memorizzato come testo, ma come una sequenza di byte, divisa in più campi.

Vediamo come è composto un singolo record.

| Header del Record | Bitmap dei Campi Null | Dati dei Campi |



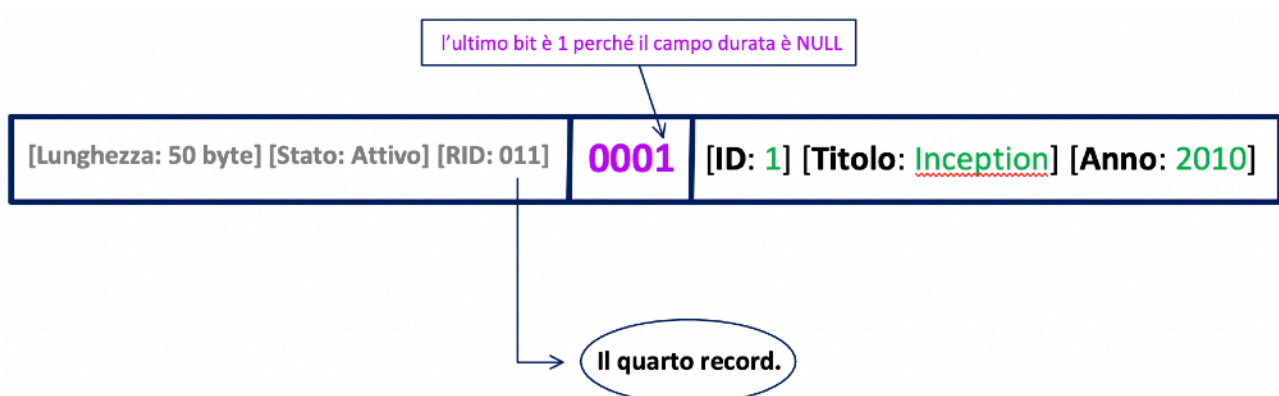
Se ho una tabella così composta:

```
Film(Id, titolo, anno, durata),
```

supponendo che all'interno di essa sia stata eseguita la query:

```
INSERT INTO Film (id, titolo, anno, durata) VALUES  
(1, 'Inception', 2010, NULL);
```

Il record memorizzato nella pagina è il seguente:



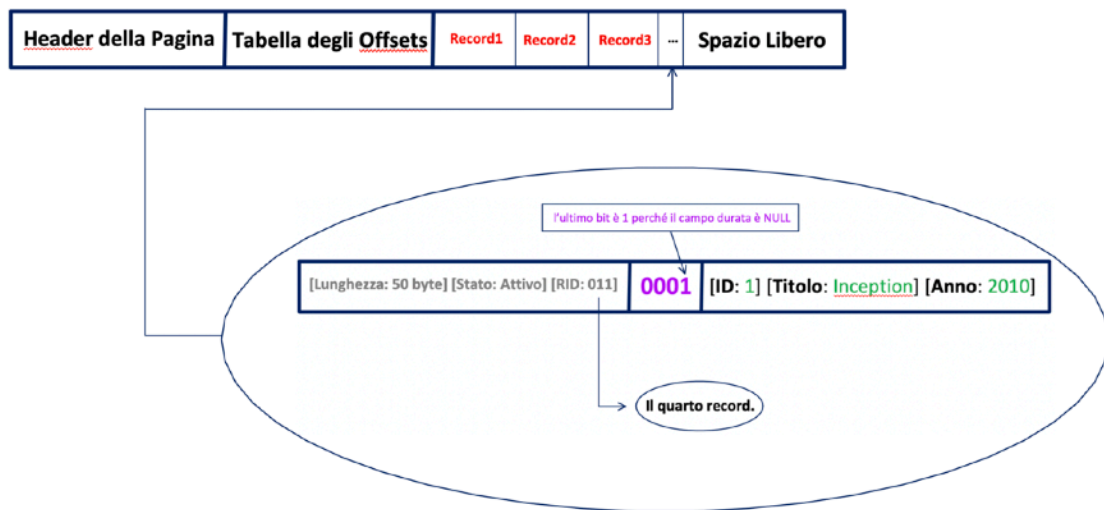
Ogni singolo record viene inserito all'interno di una pagina.
La pagina mantiene informazioni ai record che possiede.
Ogni **pagina** di un database relazionale è composta da tre sezioni
principali:

- 1 **Header della Pagina**: Contiene metadati sulla pagina (es. numero di record, spazio disponibile).
- 2 **Tabella degli Offsets**: Indica la posizione dei record nella pagina.
- 3 **Area dei Record**: Contiene i dati effettivi organizzati in un formato binario ottimizzato.

Composizione di una pagina.



Ciascun record viene inserito nella pagina:



La **Tabella degli Offsets** tiene traccia della posizione di ogni record.

Se un record viene **eliminato**, il suo spazio viene contrassegnato come **disponibile**: l'offset viene marcato come **spazio disponibile**.

Se un record cresce in dimensione e non c'è abbastanza spazio, viene **spostato** in un'altra pagina (**pagine overflow**).

Di seguito il record dentro la pagina:
Quando il DBMS cerca un record:

1. **Consulta la Tabella degli Offsets** per trovare la posizione del record nella pagina.

R1 → 50 significa che il primo record (R1) inizia alla posizione 50 all'interno della pagina.
R2 → 100 significa che il secondo record (R2) inizia alla posizione 100.
R3 → 150 significa che il terzo record (R3) inizia alla posizione 150.
Significato pratico:

- Quando il DBMS cerca un record, non scansiona tutta la pagina, ma usa gli **offset** per trovare rapidamente la posizione del record.

4 record

Tabella Offsets:

[R1 -> 50] [R2 -> 100] [R3 -> 150]

Record1 (ID: 2, Titolo: Maria, Anno: 2014)

Record2 (ID: 3, Titolo: The, Anno: 1999)

Record3 (ID: 4, Titolo: Filippi, Anno: 1999)

Record4 (ID: 1, Titolo: Inception, Anno: 2010)

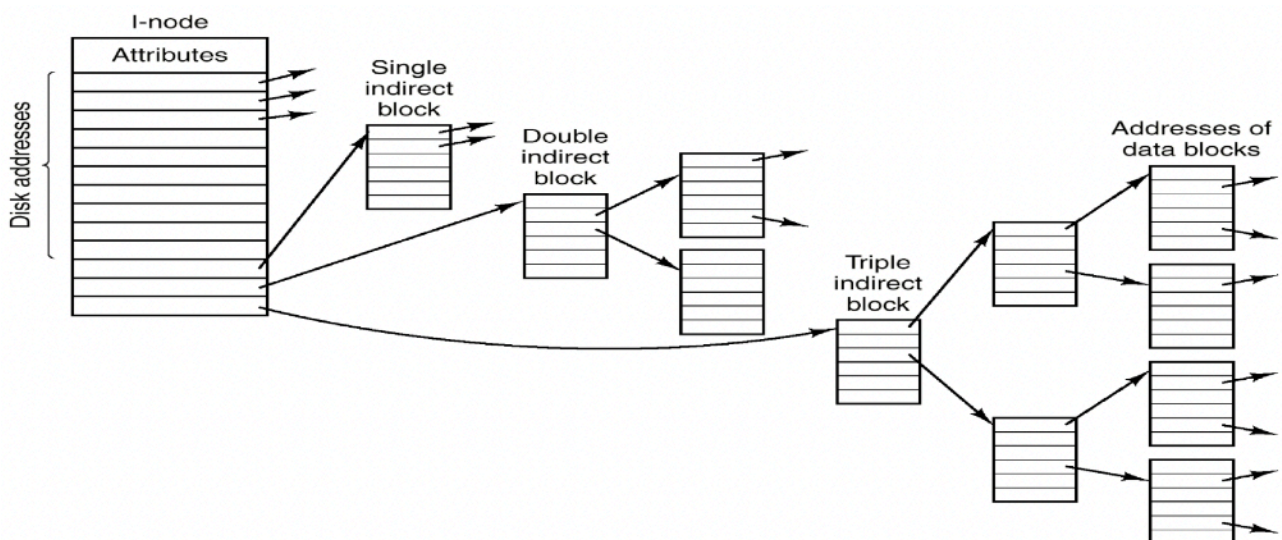
2. Legge l'**Header del Record** per verificare la lunghezza e se è attivo.

3. Salta la **Bitmap NULL** per capire quali campi sono valorizzati.

4. Legge i **dati dei campi** e li restituisce alla query.

Questo approccio permette un accesso **molto più rapido** rispetto a una lettura sequenziale di tutto il file.

Ma come vengono mantenute le informazioni dei **blocchi** all'interno del **sistema operativo**?



Ricordiamo che esiste un'indicizzazione a livelli multipli. I primi dieci elementi fungono da puntatori diretti alle esatte locazioni di memoria in cui risiedono, qualora presenti, i dati costituenti il file. Ciò implica che l'indirizzamento operato da tali elementi avviene in **maniera diretta**, ovvero le informazioni relative alla posizione dei blocchi di dati associati al file, all'interno del dispositivo di memoria di massa, vengono mantenute all'interno dell'**I-NODE** stesso. Tale struttura consente dunque di gestire file di dimensioni massime pari a

10 × Sizeof(Blocco) .

Nel caso in cui il file superi tale limite dimensionale, la mera indicizzazione tramite i dieci puntatori diretti risulta insufficiente. Sorge quindi il quesito: l'**I-NODE** dispone di ulteriori indici per sopperire a tale esigenza? La risposta è affermativa, e tali indici supplementari sono gli ultimi tre.

Il primo di questi è un **indice indiretto di primo livello**: esso non punta direttamente ai blocchi contenenti i dati del file, bensì a un blocco ausiliario che ospita un'ulteriore tabella di indici. Ciascuno di questi ultimi, a sua volta, fa riferimento a un **blocco dati** che custodisce effettivamente il contenuto del file.

Ritorniamo alle pagine e proseguiamo con la trattazione.

Nel contesto di questo corso, si dedica un'attenzione particolarmente elevata alla **progettazione di una base di dati**, poiché nella pratica reale una progettazione inadeguata può compromettere significativamente l'efficienza dell'accesso ai dati. In particolare, qualora una singola **tupla** ecceda le dimensioni di una **pagina di memoria** destinata a ospitarla, il sistema non sarà in grado di recuperare l'intera tupla mediante la lettura di un'unica pagina, generando così un'inefficienza che, idealmente, dovrebbe essere evitata. Per tale motivo, è imprescindibile concepire a **priori** una struttura della base di dati che rispetti principi di coerenza e ottimizzazione. Qualora il **database** contenga **tabelle di dimensioni eccessive**, caratterizzate da un **numero spropositato di attributi**, esso risulterà strutturalmente inadeguato. Una progettazione di tale natura, infatti, comprometterebbe la capacità del **DBMS** di ottimizzare le operazioni di accesso ai dati, riducendone le prestazioni ed esponendo il sistema a inefficienze evitabili.

La base di dati deve essere normalizzata e concepita con una buona organizzazione delle tabelle e dei loro attributi.

PAGINE CON RECORD A LUNGHEZZA FISSA.

Fintanto che i **record** presentano una lunghezza fissa, la loro gestione risulta estremamente agevole. Qualora ogni **tupla** possieda una dimensione uniforme rispetto alle altre, l'organizzazione degli **slot** (ossia, i record) all'interno delle pagine di memoria si semplifica notevolmente, poiché è possibile suddividere e allocare lo spazio in maniera deterministica, assegnando a ciascuno slot una porzione predefinita della pagina. Questa proprietà deriva dal fatto che, in fase di definizione dello **schema di una base di dati**, è consuetudine specificare una **dimensione massima** per determinati tipi di dati, come le **stringhe**, la cui gestione potrebbe altrimenti risultare problematica. Tale accorgimento consente di preservare una struttura prevedibile all'interno delle pagine, facilitando l'accesso e l'organizzazione delle informazioni nel sistema di gestione della base di dati (**DBMS**).

Il **DBMS**, chiedendo ogni qual volta la dimensione di una stringa con il tipo **varchar**, spinge il progettista a creare implicitamente dei record a dimensione prefissata con l'unico scopo di **ottimizzare lo spazio**.

Ci sono, tuttavia, delle casistiche in cui non posso conoscere a priori la dimensione massima di un determinato attributo, e questo lo si può

notare con il tipo primitivo `text`, in quanto la logica sottesa a questo dato consiste nella scrittura arbitraria di un determinato record popolandolo con un numero indefinito di caratteri. In questo caso, il testo se troppo grande viene comunque organizzato a lunghezza fissa, ma il record rimanente viene inserito in più pagine, mantenendo un riferimento delle pagine che compongono quella tupla, aumentando gli accessi a disco.

In sostanza, quando un record contiene un campo di tipo `TEXT` o simili, il DBMS non lo memorizza interamente nella pagina principale ma usa una pagina di overflow:

Cosa succede?

- Il record principale rimane nella sua **pagina originale**.
- Il valore `TEXT` troppo grande viene spostato in un'altra pagina (**pagina di overflow**).
- Nel record principale viene memorizzato **un puntatore** alla pagina di overflow, così da poter recuperare il dato quando necessario.

Nota: Se il `TEXT` è ancora più grande e non entra in una singola pagina di overflow, il DBMS creerà **una catena di pagine di overflow**.

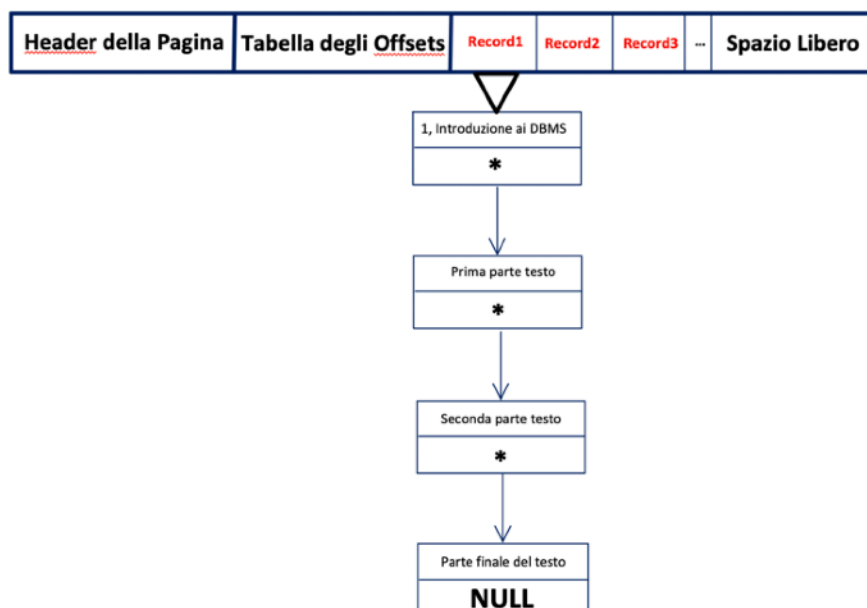
Il Puntatore è nel Record?

Sì, il record principale contiene il puntatore alla prima pagina di overflow.

Supponiamo di voler eseguire:

INSERT INTO articoli (id, titolo, testo)

VALUES (1, 'Introduzione ai DBMS', 'Questo è un articolo molto lungo... (molti KB di dati)');



È una **lista di trabocco** di pagine di overflow.
Quando una query richiede il valore TEXT, **il DBMS segue questi passaggi:**

- 1 Legge la pagina principale** per verificare se il valore è memorizzato direttamente lì o se c'è un puntatore a una pagina di overflow.
- 2** Se il valore è in overflow, **segue il puntatore alla prima pagina di overflow.**
- 3** Se il valore è molto grande, il DBMS **segue i puntatori tra le pagine di overflow** fino all'ultima pagina.
- 4 Ricostruisce il valore intero** e lo restituisce alla query.

Gli svantaggi sono ovvi:

Più accessi al disco → Se il valore TEXT è spezzato su molte pagine, la lettura è più lenta.

Frammentazione → I dati TEXT possono essere distribuiti su più pagine sparse nel disco.

Buona Pratica: Mai utilizzare il DBMS come un file system per memorizzare dati di grandi dimensioni come immagini, video, file PDF o documenti binari.

Soluzione: Memorizzare i file nel File System e Salvare il Percorso nel DBMS.

Come fare?

Salvare i file in un File System tradizionale (es. /uploads/immagini/ per immagini, /uploads/documenti/ per PDF, ecc.).

Nel database, **salvare solo il percorso del file, non il file stesso, e ci accedo dalla applicazione a livello applicativo.**

In questo modo abbiamo **eliminato il problema** della pagine di overflow: meno accessi al disco, query più veloci.

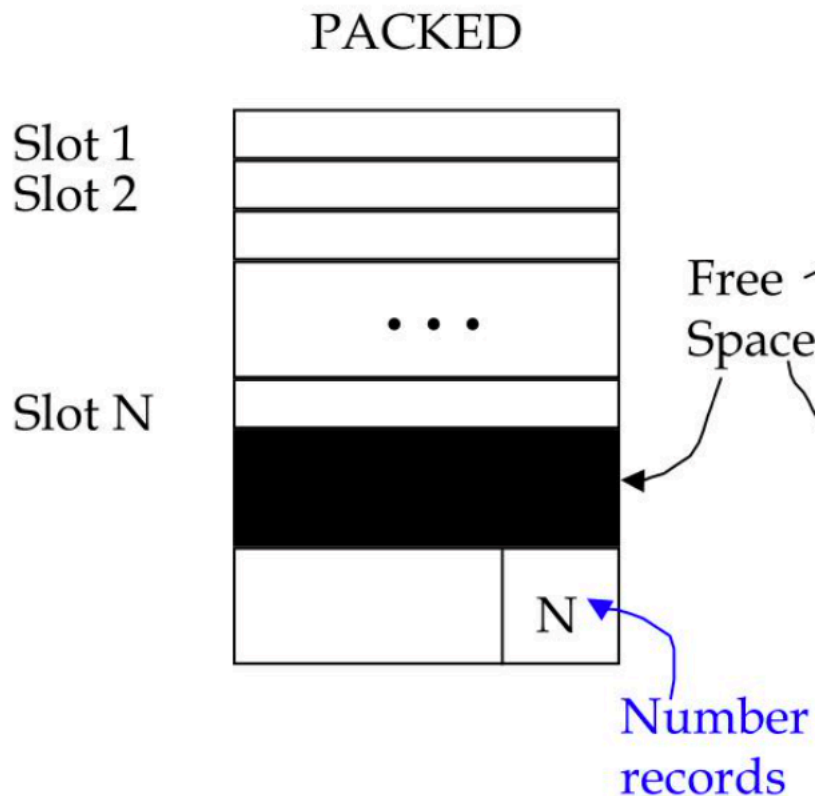
Come si organizzano dei record a dimensione fissa all'interno degli slot?

Posso utilizzare la mia pagina in **maniera impacchettata** o in **maniera non impacchettata.**

Maniera impacchettata

Gli slot assegnati a contenere determinati record sono tutti localizzati all'inizio della mia pagina.

Per trovare uno spazio libero è sufficiente mantenere a livello di pagina il numero dei record N che sono utilizzati.



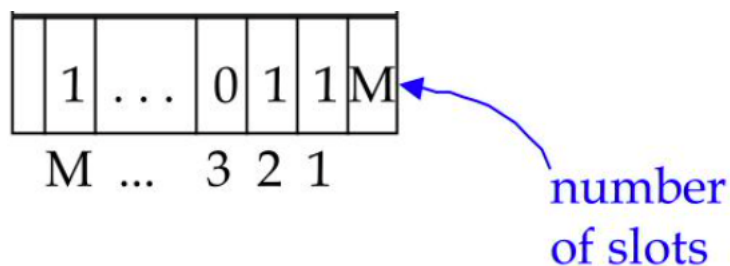
Facile, se sto utilizzando 1 solo record lo slot numero 2 sarà libero.

Se elimino una tupla devo prendere tutti gli slot successivi ad essa e spostarli di una posizione verso l'alto.

Questa organizzazione **consente molto velocemente** di **inserire nuove tuple**, ma richiede un **costo maggiore** in caso di **eliminazione frequente**.

Voglio migliorare questo costo di eliminazione, pertanto bisogna evitare di dover spostare tutti gli slot di una unità, ma a quel punto devo **tenere traccia** di **quali sono gli slot liberi** nel momento in cui voglio inserire nuovi record nella pagina.

Soluzione: utilizzo una semplice **BitMap**.



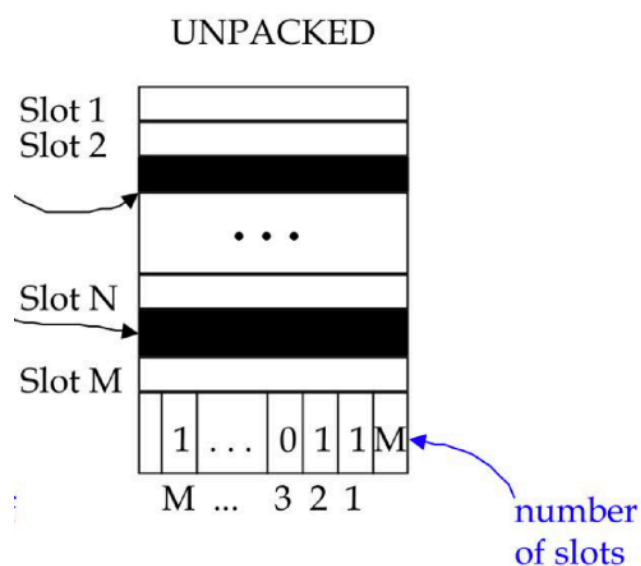
Utilizzando un bit per ciascuno slot posso effettuare una cernita accurata degli slot che sono **liberi** e di quelli che sono **occupati**.

Questa **BitMap** si trova all'interno dell'**Header della Pagina**.
Eliminare un record preciso significa azzerare uno specifico bit della BitMap, ossia rendere lo slot libero.

Inserire un record preciso significa guardare la BitMap, leggere tutti i bit finché non viene trovato un bit a 0, ossia uno slot libero in cui posso scrivere il record.

L'operazione di eliminazione diventa molto veloce,
l'operazione di inserimento diventa più lenta.

Questa è l'organizzazione non impacchettata.



Nota: Un DBMS ben costruito permette di utilizzare entrambe le strategie.

È il progettista che, dopo aver analizzato i requisiti della progettazione della base di dati, sceglie per il DBMS se utilizzare l'una o l'altra.

PAGINE CON RECORD A LUNGHEZZA VARIABILE.

Si costruisce una **directory di slot**.

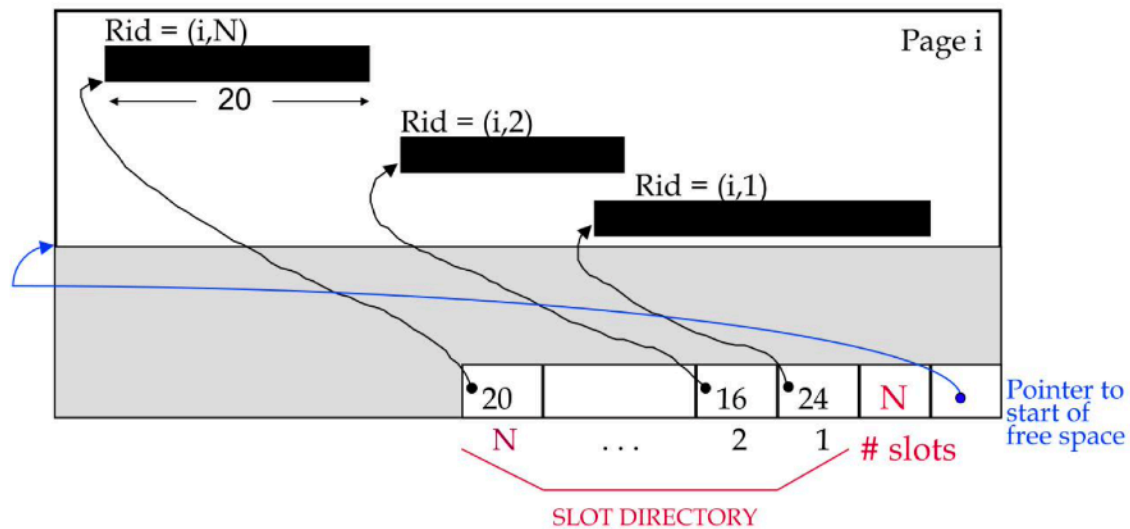
Devo capire a quale **offset** della mia **pagina** c'è il primo byte del record che sto cercando.

Come si fa?

Si inserisce il record i-esimo dall'inizio della pagina, e all'interno della directory di slot si scrive la dimensione in byte di quello specifico record.

Se voglio accedere ad un record numero X devo sommare tutte le taglie in

byte dei record precedenti per capire a quale **offset** della mia **pagina** c'è il primo byte del record che sto cercando.



Come faccio ad eliminare un record?

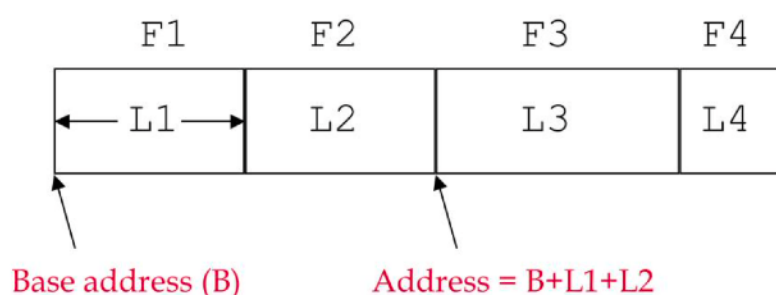
Cancellare un record significa impostare a -1 il valore dello slot corrispondente e spostare lo spazio del record nella "spazio libero" area.

Nel momento in cui il secondo **record** viene eliminato, invalidando il valore dello **slot** corrispondente, si genera uno spazio disponibile all'interno della **pagina di memoria**. Tuttavia, qualora si desideri successivamente inserire un nuovo record di **dimensione 17**, potrebbe non essere possibile collocarlo nella posizione precedentemente occupata dal record eliminato. Questo fenomeno è dovuto alla **frammentazione interna**, ovvero alla suddivisione disomogenea dello spazio all'interno della pagina, che impedisce un utilizzo efficiente delle risorse di memoria.

Quindi devo **ricompattare la memoria**.

Nota: in questo esempio la pagina è già stata caricata in RAM ed è giusto che sia così in quanto è stata letta dal disco alla memoria.

FORMATO DI UN RECORD A TAGLIA FISSA.



Dentro ci sono gli attributi e supponiamo di essere interessati al terzo.
Nelle operazioni di proiezione tipicamente non ci interessa tutta la tupla bensì il singolo attributo.

Se tutti gli attributi di un record hanno dimensioni fisse, è possibile calcolare la posizione esatta di un attributo senza leggere tutta la riga.

In che modo?

Ogni pagina del DBMS ha un header, che contiene informazioni sulla struttura dei record all'interno della pagina. Nei record a lunghezza fissa, gli attributi sono memorizzati in posizioni fisse, quindi il DBMS non ha bisogno di offsets per trovarli in quanto i record hanno una dimensione predefinita.

In record così: | ID (4B) | Nome (20B) | Età (4B) |

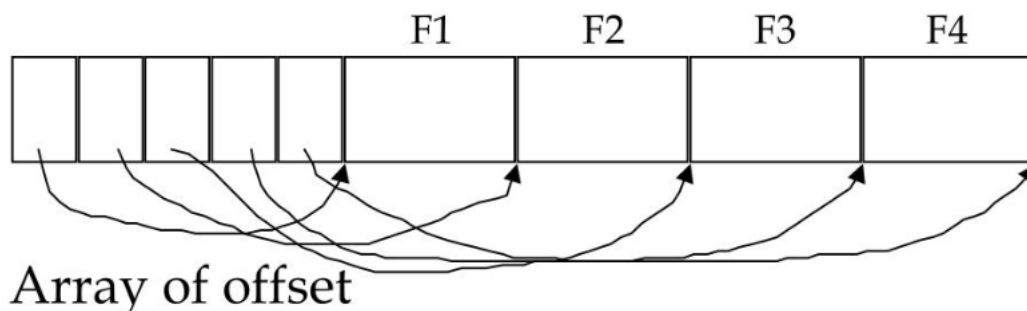
Se voglio leggere solo età, il DBMS può calcolare direttamente la sua posizione:

Posizione di `età` = Offset Iniziale + Dimensione di `id` + Dimensione di `nome` = **0 + 4 + 20 = 24**.

FORMATO DI UN RECORD A TAGLIA VARIABILE.

In questi tipi di record l'unico elemento fissato è il numero di attributi.

Nei record a lunghezza variabile ogni record ha dimensioni diverse, quindi il DBMS **non può sapere dove inizia il successivo** senza una tabella degli **offsets**.



Qui il DBMS deve leggere gli offsets per sapere dove inizia ogni record: non devo scandire tutto il record.



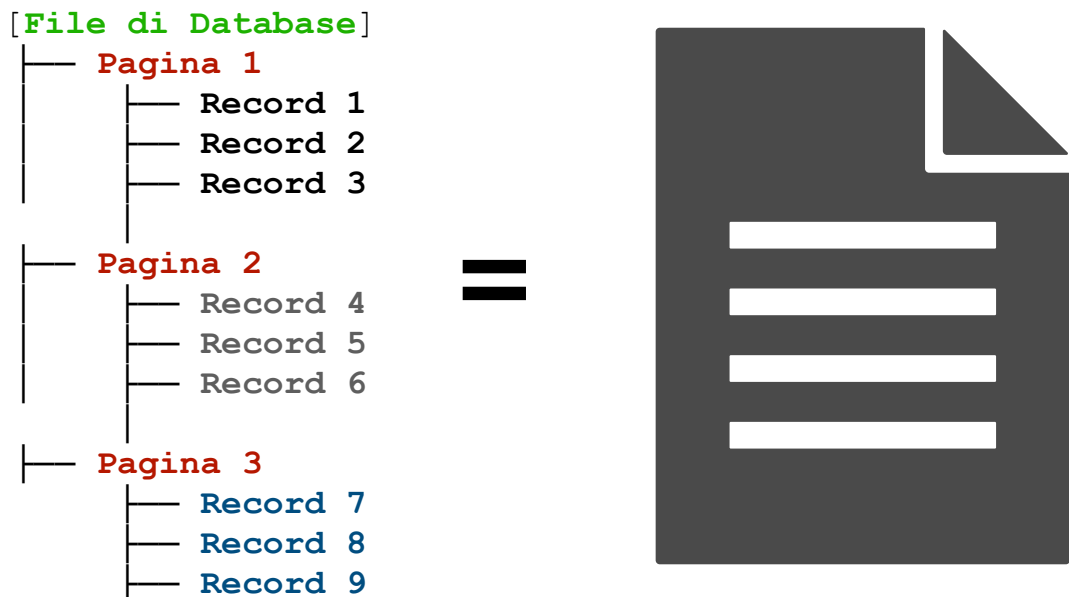
I FILE.

Un file è una collezione di pagine, ciascuna contenente una collezione di record.

Questo è un concetto fondamentale per capire come i database organizzano e **gestiscono i dati su disco**.

Le pagine appartengono a un file, ma il DBMS può leggerle solo attraverso i blocchi.

Esempio:



Quindi, **i file esistono solo a livello logico**, mentre i **blocchi servono per trasferire i dati tra disco e RAM**.

Nei DBMS relazionali tradizionali, come **MySQL**, **PostgreSQL** e **SQL Server**, ogni tabella è memorizzata in uno o più file.

Un file contiene una collezione di pagine, e ogni pagina contiene record della stessa tabella e all'interno di un file ci sono tuple della stessa relazione.

```
[File "clienti.db"]
├── Pagina 1
│   ├── Record 1 (ID=1, Nome="Mario", Età=25)
│   └── Record 2 (ID=2, Nome="Luca", Età=30)
└── Pagina 2
    ├── Record 3 (ID=3, Nome="Anna", Età=28)
    └── Record 4 (ID=4, Nome="Elena", Età=35)
```

Piccolo cenno: È possibile avere più tabelle nello stesso file attivando un **Tablespace Condiviso**, ossia un'area di archiviazione dove più tabelle condividono lo stesso file su disco. In altre parole, i dati di più tabelle vengono memorizzati nello stesso file fisico, anziché avere un file separato per ogni tabella. Questo **può essere configurato** per contenere i dati di **una o più tabelle**.

Cosa Succede Quando il DBMS Accede a un File?

1. Il DBMS Riceve una Query.

SELECT nome **FROM** clienti **WHERE** id = 10;

Il motore SQL analizza la query e determina dove si trovano i dati nel file di database.

2. Il DBMS Cerca la Pagina nei Suoi Buffer.

Prima di accedere al disco, il **Buffer Manager** verifica se la **pagina contenente il record richiesto è già in RAM**.

Se la pagina è in RAM, il DBMS la usa direttamente (evitando un accesso al disco).

3. Il DBMS Determina il Blocco su Disco.

Il **gestore del file system** del DBMS identifica il **file di database** che contiene la tabella richiesta.

Consulta la **mappa delle pagine*** del file per trovare in quale blocco si trova la pagina contenente il record richiesto.

Un blocco è l'unità fisica di trasferimento tra disco e RAM e può contenere più pagine.

*La **mappa delle pagine** è una struttura interna del **DBMS** che tiene traccia della posizione delle pagine nei file di database. Serve per sapere **quale pagina si trova in quale blocco su disco** e permette di accedere ai dati in modo efficiente.*

Ogni file di database contiene la sua mappa delle pagine, perché deve sapere dove si trovano le pagine che lo compongono.

Questa struttura consente al DBMS di localizzare rapidamente i dati senza dover leggere tutto il file sequenzialmente.

Esempio di mappa delle pagine nel DBMS:

Tabella clienti → Pagina 12

Mappa delle pagine nel DBMS:

Pagina 12 → Blocco 1500 su disco

Pagina 13 → Blocco 1510 su disco

Il DBMS ora sa che deve leggere il Blocco 1500 dal file su disco.

4. Il DBMS Carica il Blocco in RAM

Il **Disk Manager** invia una richiesta di lettura al **File System** del sistema operativo.

Il File System recupera il Blocco 1500 dal file di database e lo carica in RAM.

Il blocco può contenere **più pagine**, quindi il **Buffer Manager** estrae la pagina specifica (es. Pagina 12).

5. Il DBMS Estrae il Record dalla Pagina

La **Pagina 12** viene decodificata per estrarre il record richiesto.

Se il record ha un campo di tipo **TEXT** o **BLOB**, potrebbe essere necessario accedere a una **pagina di overflow**.

Il DBMS trova il record giusto all'interno di una pagina grazie alla **Tabella degli Offsets**.

✓ Usa gli offset per accedere direttamente alla posizione del record, **evitando scansioni lente**.

✓ **Confronta il valore della chiave primaria** per assicurarsi che sia il record corretto.

Esempio: Struttura di una Pagina con Offsets

Offsets:

[R1 → 50B]	(Record 1 inizia a byte 50)
[R2 → 100B]	(Record 2 inizia a byte 100)
[R3 → 150B]	(Record 3 inizia a byte 150)

Se il DBMS cerca ID = 2 (indice primario (**Clustered Index**)), sa che il record 2 è a offset 100, ma:

Deve leggere il record a offset 100.

Deve controllare il valore del campo ID.

Se ID = 2, restituisce il record.

Se ID ≠ 2, passa al record successivo.

Quindi l'offset velocizza l'accesso, ma non garantisce che sia il record giusto.

Se la ricerca non è basata su una chiave primaria (come ID), il DBMS non può usarlo per trovare direttamente la pagina. Deve quindi adottare una strategia diversa.

Quindi le ricerche con chiave primaria sono molto veloci.

Se c'è un indice secondario → Il DBMS lo usa per trovare il record più velocemente.

Tipi di indici secondari:

B-Tree Index (più comune) → Organizza i dati in una struttura ad albero bilanciato.

Hash Index → Ottimo per ricerche di uguaglianza (WHERE email = 'x').

Full-Text Index → Ottimizzato per ricerche su testi lunghi.

Come si creano:

```
CREATE INDEX idx_cognome ON clienti(cognome); -- B-Tree
Index
CREATE INDEX idx_email_hash ON utenti(email) USING HASH; --
Hash Index
CREATE FULLTEXT INDEX idx_testo ON articoli(testo); -- Full-
Text Index
```

In generale, gli indici secondari sono strutture indipendenti dalla tabella e vengono salvati separatamente dai dati della tabella.

Esempio di struttura del file system in MySQL:

```
/var/lib/mysql/
├─ database/
│   ├─ clienti.ibd          # Dati della tabella
│   └─ clienti_idx.ibd     #Indice secondario "idx_nome"
```

Qui l'indice secondario ha un proprio file .ibd.

6. Il DBMS Restituisce il Risultato

Dopo aver letto il record, il DBMS invia il risultato al **motore SQL**, che lo formatta per l'utente.

Se i dati vengono usati spesso, il DBMS **mantiene la pagina in RAM** per velocizzare future richieste.

Il DBMS deve supportare diverse operazioni sui file, tra cui:

- Inserimento, eliminazione e aggiornamento di un record.
- Lettura di un record dato il suo Record ID (RID).
- Scansione di tutti i record per trovare quelli che soddisfano una condizione (es. WHERE nome = 'Mario').

Tipi di Organizzazione dei File nei DBMS

SIMPLE E INDEXED: ZOOM SUI RECORD.

Simple: Heap File.

I record nelle pagine vengono inseriti in ordine casuale senza una particolare organizzazione e senza nessun criterio particolare.

Esempio di heap file:

[**Heap File**] (collezione di pagine)

|— **Pagina 1** → [R2, R4, R1]
|— **Pagina 2** → [R7, R3]
|— **Pagina 3** → [R6, R8, R5]

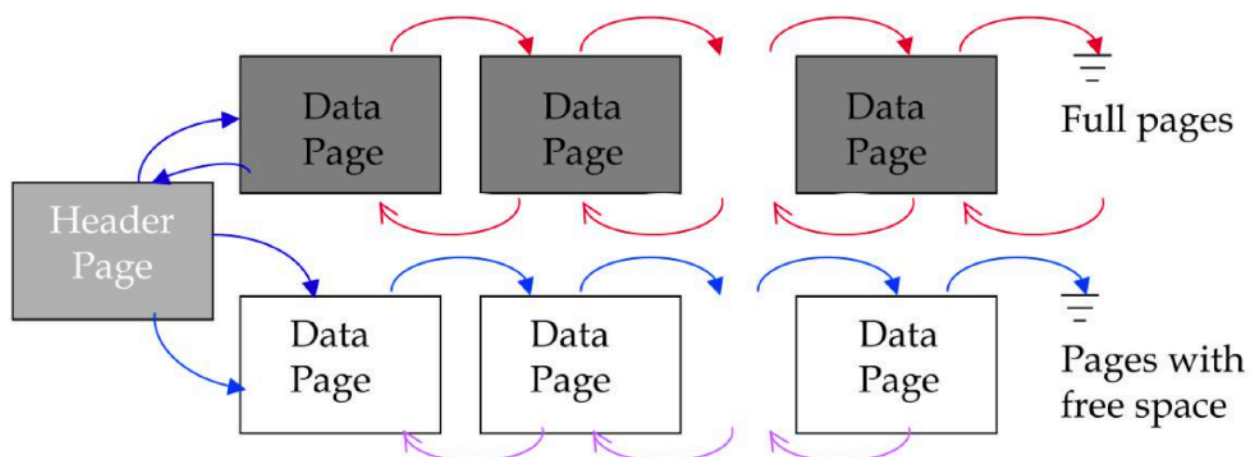
Vantaggi:

Inserimento veloce → I record vengono semplicemente inseriti nella prima posizione libera disponibile.

Struttura semplice → Nessuna gestione complessa di indici o ordinamenti.

Ottimo per caricare grandi quantità di dati velocemente. Bisogna conoscere, per ogni pagina del file, quale di queste hanno uno spazio libero a disposizione per inserire un record.

E si usava una **rappresentazione tramite liste**.



Esistono due liste: una lista che raccoglie **pagine piene** e una lista che raccoglie pagine con spazio libero.

Quando è necessaria una nuova pagina, il DBMS chiede al gestore del disco.

La nuova pagina viene aggiunta all'elenco delle pagine con spazio libero (●).

Se una pagina si riempie completamente, viene spostata nella lista delle pagine piene (●).

Se una pagina diventa completamente vuota, viene rimossa dall'elenco e deallocata.

Qualunque tipo di ricerca non è efficace, bisogna scorrerle tutte.

Cos'è la Header Page nei DBMS?

La **Header Page** è una pagina speciale all'inizio di un file di database o di una struttura di pagine che **mantiene informazioni di gestione** sulle altre pagine.

SCRITTURA RECORD.

Il DBMS consulta la lista delle pagine con spazio libero (●).

Trova la prima pagina disponibile e inserisce il record.

Se la pagina si riempie completamente, viene spostata nella lista delle pagine piene (●).

Se non ci sono pagine disponibili, il DBMS alloca una nuova pagina e la collega alla lista.

LETTURA RECORD.

Il DBMS inizia dalla Header Page, che contiene il puntatore alla prima pagina della lista.

Segue i puntatori alle pagine successive, scorrendo la lista fino a trovare il record richiesto.

Se il record è in una pagina piena (●), il DBMS legge direttamente il dato.

Se è in una pagina con spazio libero (●), segue il collegamento e legge il record.

ELIMINAZIONE RECORD.

Il DBMS individua la pagina contenente il record tramite un indice o una scansione.

Elimina il record e aggiorna la tabella degli offsets.

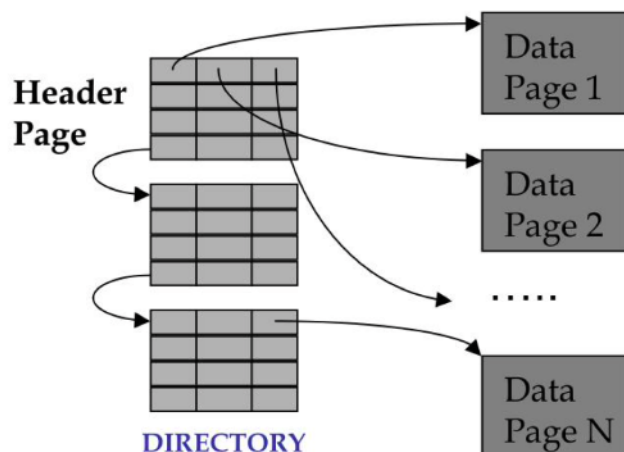
Se la pagina diventa parzialmente vuota, viene aggiunta alla lista delle pagine con spazio (●).

Se la pagina diventa completamente vuota, viene deallocata e rimossa dall'elenco.

Esiste anche una **rappresentazione tramite directory**.

La **rappresentazione con directory** è un metodo per organizzare e gestire le pagine in un file di database, migliorando l'efficienza nella ricerca di pagine con spazio libero per nuovi record.

Scopo principale → **Gestire le pagine dati in modo più efficiente** rispetto alle liste collegate, riducendo la necessità di scansioni.



Una directory è una tabella che contiene un elenco di pagine dati.

Ogni entry della directory memorizza:

- 1 Un **puntatore** alla pagina dati corrispondente.
- 2 Un indicatore dello spazio disponibile nella pagina (es. un **contatore**).

[Esempio di rappresentazione a Directory]

```
-----  
| Entry 1 → Puntatore a Pagina 1 (80% piena) |  
| Entry 2 → Puntatore a Pagina 2 (50% piena) |  
| Entry 3 → Puntatore a Pagina 3 (30% piena) |  
-----
```

Il DBMS può trovare rapidamente una pagina con spazio libero senza dover scansionare tutte le pagine!

La **directory** è più **efficiente** rispetto alle **liste collegate** perché la ricerca di una pagina con spazio libero è più veloce: perché il numero di accessi è lineare rispetto alla dimensione della directory, non alla dimensione della relazione. Ma vogliamo migliorare ancora di più.

Come Funziona l'inserimento dei Dati con la Directory?

Il DBMS segue questi passaggi:

- 1 **Consulta la directory** per trovare una pagina con spazio disponibile.
- 2 **Usa il puntatore** della directory per accedere direttamente alla pagina dati.
- 3 **Inserisce il record nella pagina.**
- 4 **Aggiorna il contatore** dello spazio libero nella directory.
- 5 Se la pagina si **riempie completamente**, viene segnalata come "**piena**" nella directory.

Il modo migliore per supportare la ricerca è avere un ordinamento dei record, quindi un **ordinamento sulle pagine**.

Se all'interno del file memorizzo pagine ordinate dove all'interno di ciascuna pagina ho record ordinati, effettuare la ricerca è potenzialmente più efficace.

Se ho un ordinamento sui record posso decidere addirittura di saltare un'intera pagina.

L'ordinamento che vado ad attuare ai record delle pagine si basa su una **chiave di ricerca**: ossia un insieme di attributi sul quale viene costruito un ordinamento lessico grafico.

Si mettono in ordine i record in funzione del valore di un certo attributo.

Le pagine anche vengono ordinate secondo questo ordinamento.

L'ordine delle pagine riflette l'ordinamento dei record.

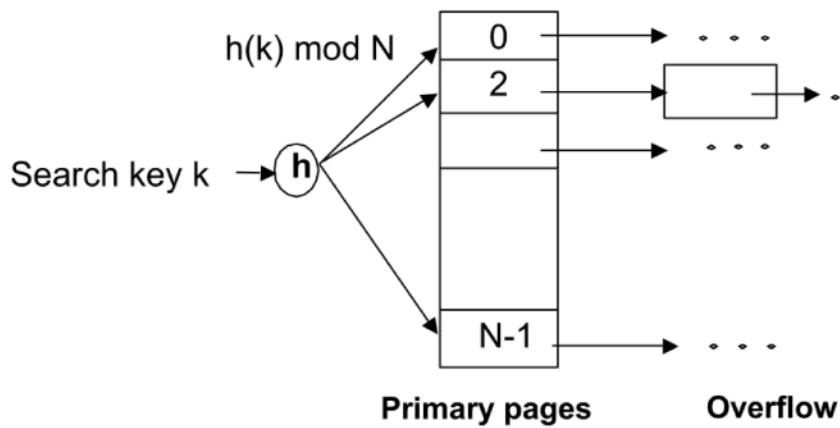
Una organizzazione efficiente si basa sul famoso:

Hashed file



Obiettivo principale → **Trovare rapidamente un record senza dover scansionare l'intero file!**

Negli **Hashed File**, le pagine del file vengono suddivise in gruppi chiamati **bucket**.



Ogni **bucket** contiene:

- 1 **Una pagina primaria** → Contiene i record associati a quel bucket.
- 2 **Pagine di overflow (se necessarie)** → Se la pagina primaria è piena, i record in eccesso vengono memorizzati in pagine di overflow collegate.

La **funzione di hash, applicata alla chiave di ricerca**, permette di individuare direttamente il **bucket corretto, restituendoci un indice!**

Questo sarà l'indice del Bucket dove potrò inserire i dati!

Se sto inserendo Pippo, calcolo la funzione di hash di pippo, e posso ottenere ad esempio l'indice numero 3: il record pippo deve essere inserito all'interno della pagina numero 3.

Dopodiché all'interno di quella pagina i record saranno ordinati.

Ma se possiedo una funzione di hash che mi indica in quale Bucket compaiono determinati record, posso calcolare la funzione di hash anche per supportare la ricerca, così facendo conosco la posizione dei miei record di interesse a quale Bucket fanno riferimento e, pertanto, **evito di effettuare la scansione full lineare.**



Procedo a colpo sicuro ad ispezionare il Bucket di interesse.

Se inserisco tanti record con lo stesso valore nella chiave di ricerca la pagina primaria si riempirà favorendo il pumping della lista di trabocco (**overflow pagine**).

FUNZIONAMENTO:

Come Funziona l'Inserimento in un Hashed File?

L'inserimento in un **Hashed File** avviene utilizzando una **funzione hash** che assegna ogni record a un **bucket** specifico. Se il bucket ha ancora spazio disponibile, il record viene inserito direttamente. Se il bucket è pieno, il record viene salvato in **pagine di overflow**.

Esempio: **Inserire un record nella tabella clienti**

```
INSERT INTO clienti (codice_fiscale, nome) VALUES  
('ABC123XYZ', 'Luca');
```

Passo 1: Applicare la Funzione Hash sulla Chiave di Ricerca.

Il DBMS prende la **chiave di ricerca** (es. `codice_fiscale`) e applica una **funzione hash** per determinare il bucket di destinazione.

$h('ABC123XYZ') \bmod 5 \rightarrow \text{Bucket } 2$

Il record sarà assegnato al **Bucket 2**.

Passo 2: Controllare lo Spazio nella Pagina Primaria.

Il DBMS controlla se la **pagina primaria del bucket** ha spazio libero.

Caso 1: Spazio disponibile  Il record viene inserito direttamente nella **pagina primaria** del Bucket 2.


Caso 2: Pagina primaria piena  Il record viene inserito in una **pagina di overflow** collegata al bucket.

Passo 3: Inserimento del Record

Esempio di Struttura dei Bucket prima dell'inserimento


Bucket 0 → Pagina Primaria (spazio libero)

Bucket 1 → Pagina Primaria (piena) → Overflow 1

Bucket 2 → Pagina Primaria (spazio libero) 

 Il DBMS inserisce `codice_fiscale = 'ABC123XYZ'` direttamente nel **Bucket 2**.

Esempio di Struttura dopo l'inserimento

Bucket 2 → Pagina Primaria (occupata al 70%) 

 Il record è stato inserito senza bisogno di overflow!

Passaggi per la ricerca eseguiti dal DBMS:

```
SELECT * FROM clienti WHERE codice_fiscale = 'ABC123XYZ';
```

- 1 Applica una funzione hash sulla chiave **codice_fiscale**.
- 2 Trova il bucket corrispondente usando il valore $h(k) \bmod N$.
- 3 Accede direttamente alla pagina primaria del bucket.
- 4 Se il record è lì, viene restituito immediatamente.
- 5 Se la pagina primaria è piena, segue i puntatori alle pagine di overflow.

✓ L'accesso è diretto e molto veloce rispetto a una scansione sequenziale!

LE JOIN SONO BELLE MA NON BISOGNA ABUSARNE.

Per determinare la struttura dati più adeguata all'organizzazione efficiente dei **record** all'interno delle **pagine di memoria**, è essenziale condurre un'analisi dei **modelli di costo**, con particolare riferimento ai **tempi di esecuzione** delle operazioni.

Dal punto di vista **logico**, la **normalizzazione** delle tabelle rappresenta una strategia ottimale, in quanto assicura coerenza e minimizzazione della ridondanza dei dati. Tuttavia, tale approccio può compromettere significativamente le **prestazioni del DBMS**, soprattutto nei casi in cui l'esecuzione di operazioni di **JOIN** richieda numerose scansioni, aumentando il costo computazionale delle query. Immaginiamo una JOIN, e l'attributo dove voglio effettuare l'uguaglianza non è l'attributo con cui ho ordinato i miei record all'interno del file, in quel caso devo andare a guardare **tutto il mio file**.

Il DBMS deve confrontare i record di due o più tabelle per trovare quelli che soddisfano la condizione della JOIN.

Se non ci sono indici, il DBMS potrebbe dover **scansionare completamente** le tabelle, aumentando il costo computazionale.

Come Sopperire alla JOIN?

Creazione e utilizzo di **indici appropriati**.

Se una JOIN viene eseguita su colonne senza indice, il DBMS è costretto a scansionare interamente le tabelle.

Esempio di Ottimizzazione con Indici NON CLUSTERED.

```
CREATE INDEX idx_ordini_cliente ON ordini(id_cliente);  
CREATE INDEX idx_clienti_id ON clienti(id);
```

Ora il DBMS può accedere ai record velocemente, senza scansioni inutili.

Verifica se il DBMS sta usando gli indici con EXPLAIN

```
EXPLAIN SELECT * FROM ordini  
JOIN clienti ON ordini.id_cliente = clienti.id;
```

 Se il piano di esecuzione mostra "Using Index", significa che gli indici stanno accelerando la JOIN!

Un indice Non-Clustered crea una struttura separata che mappa i valori indicizzati ai record della tabella.

I dati nella tabella restano disordinati, ma il DBMS può accedere velocemente ai record grazie alla struttura dell'indice.

Puoi avere più indici Non-Clustered sulla stessa tabella.

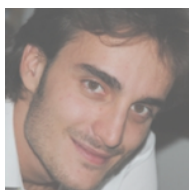
```
CREATE INDEX idx_clienti_nome ON clienti(nome);
```

L'indice sarà ordinato alfabeticamente per **nome**, ma i dati nella tabella resteranno nell'ordine originale.

Come viene memorizzato l'indice?

[Indice Ordinato]		[Tabella Originale]
-----		-----
Anna → ID 3	-->	2 Marco
Luca → ID 1	-->	1 Luca
Marco → ID 2	-->	3 Anna
Paolo → ID 4	-->	4 Paolo

Quando cerchi **nome = 'Luca'**, il DBMS usa l'indice ordinato e trova subito l'ID, senza scansionare tutta la tabella!



"Se le vostre query utilizzano tante join state uccidendo il DBMS. La normalizzazione è eccellente ma uccide le prestazioni nel momento in cui il DBMS effettua tante scansioni per le Join".

-Prof. Ing. A. Pellegrini (1987)

Quando l'Indice Viene Usato nelle JOIN?

L'indice viene usato se la JOIN è su una colonna con un indice (Clustered o Non-Clustered) e l'uguaglianza è chiara.

```
CREATE INDEX idx_ordini_idcliente ON ordini(id_cliente);
CREATE INDEX idx_clienti_id ON clienti(id);

SELECT *
FROM ordini
JOIN clienti ON ordini.id_cliente = clienti.id;
```

Ora il DBMS accede velocemente ai dati senza scansionare l'intera tabella!

L'indice potrebbe non essere usato se:

- 1 La colonna della JOIN ha troppi valori duplicati** → Se quasi tutti i record hanno lo stesso valore (es. stato = 'Italia'), l'indice è meno utile.
- 2 Il DBMS decide che una scansione completa è più efficiente** → Se la tabella è piccola, potrebbe leggere tutto più velocemente che usare l'indice.
- 3 La JOIN usa funzioni o trasformazioni sulla colonna indicizzata.**

Quale Tipo di Indice è Meglio per le JOIN?

Tipo di Indice	Vantaggi nelle JOIN	Quando usarlo?
Clustered Index	Ordina fisicamente i dati, JOIN molto veloci	Sulla colonna della PK (es. <code>id</code>)
Non-Clustered Index	Struttura separata, evita scansioni	Per JOIN su chiavi esterne (<code>id_cliente</code>)
Composite Index	Ottimo per più colonne usate nella JOIN	Se la JOIN usa più campi insieme (<code>id_cliente, data</code>)

Esempio di Indice Composito Ottimizzato:

```
CREATE INDEX idx_ordine_cliente_data ON ordini(id_cliente, data);
```

Ora le JOIN che coinvolgono **id_cliente** e **data** saranno più veloci!

Se fai una JOIN su una chiave esterna (**FOREIGN KEY**), conviene indicizzarla per migliorare le prestazioni.

```
CREATE INDEX idx_ordini_idcliente ON ordini(id_cliente);
```

```
SELECT *  
FROM ordini  
JOIN clienti ON ordini.id_cliente = clienti.id;
```

Ora la JOIN usa un indice e il DBMS può accedere velocemente ai record! Se invece la colonna **id_cliente** non ha un indice, il DBMS dovrà scansionare tutta la tabella.

Quindi tutto questo per dire che cercare per chiave primaria (**PRIMARY KEY**) è quasi sempre la soluzione più veloce perché:

1. La chiave primaria ha sempre un indice Clustered (se supportato dal DBMS).
2. Il DBMS può usare una ricerca binaria ($O(\log n)$) invece di una scansione completa ($O(n)$).
3. L'accesso ai dati è diretto e ottimizzato dal motore del database.

Se la chiave primaria è Clustered, i dati sono **ordinati fisicamente** nel file di database, rendendo l'accesso ancora più rapido.

```
SELECT * FROM clienti WHERE id = 10;
```

Come Funziona l'Accesso alla PK?

[Indice Clustered Ordinato]

```
ID: 1 → Offset 10  
ID: 2 → Offset 20  
ID: 3 → Offset 30  
ID: 4 → Offset 40
```

Se cerchi $ID = 3$, il DBMS va direttamente all'offset corretto $(O(\log n))$.