Basi di dati modulo Tecnologie

VR474752

Tecnologie per i Dati

Transazioni

Cos’è una transazione

Una **transazione** è una sequenza di operazioni eseguite come un’unità indivisibile, usata per leggere e/o modificare dati in una base di dati. L’obiettivo è garantire la **correttezza** e la **robustezza** dell’accesso concorrente ai dati, specialmente in presenza di errori o guasti e l’**isolamento**.

Esempio tipico:

```
BEGIN TRANSACTION;

UPDATE conto SET saldo = saldo - 100 WHERE id = 1;
```

```
UPDATE conto SET saldo = saldo + 100 WHERE id = 2;
COMMIT;

END TRANSACTION;
```

Proprietà ACID

Le transazioni devono rispettare quattro proprietà fondamentali, note con l'acronimo **ACID**:

- **Atomicità**: La transazione è indivisibile. Se qualcosa va storto, tutte le operazioni devono essere annullate.
- **Consistenza**: Una transazione corretta non deve violare i vincoli di integrità.
- **Isolamento**: L'esecuzione di una transazione deve essere indipendente dalla contemporanea esecuzione di altre transazioni.
- **Persistenza (Durability)**: Dopo il `COMMIT`, i cambiamenti devono essere permanenti, anche in caso di crash.

Transazioni in SQL

Una transazione SQL si definisce con:

- `BEGIN TRANSACTION`
- una sequenza di operazioni (`INSERT`, `UPDATE`, `DELETE`, etc.)
- `COMMIT` (conferma) oppure `ROLLBACK` (annullamento)

Transazioni **ben formate** iniziano e terminano esplicitamente e non accedono più ai dati dopo `COMMIT` o `ROLLBACK`.

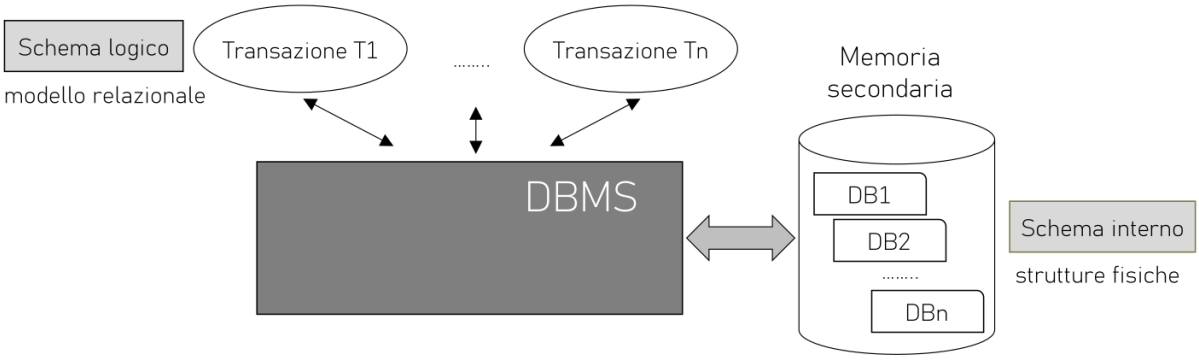
Architettura DBMS e gestione delle transazioni

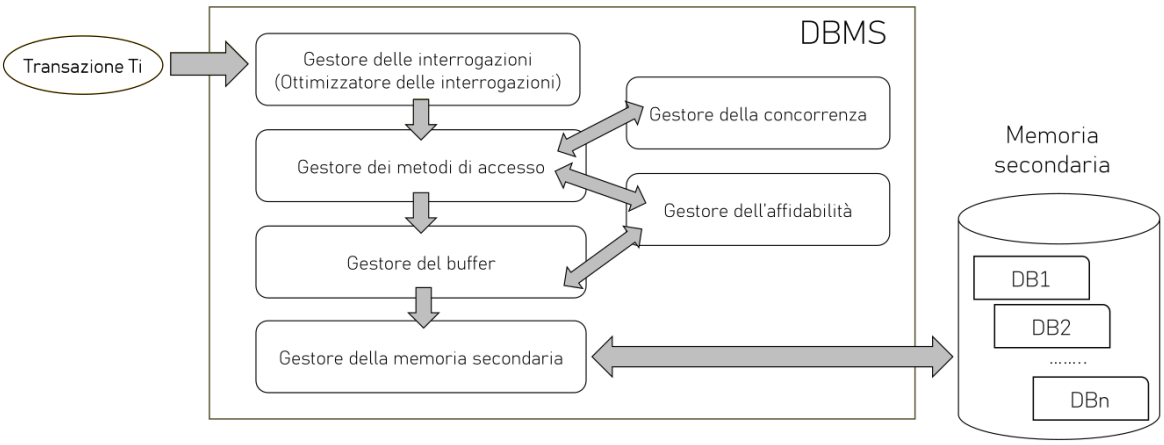
Un DBMS è un sistema software che gestisce grandi quantità di dati con le seguenti caratteristiche: sono grandi, condivisi e persistenti. Al suo interno troviamo diversi **moduli**, ognuno con un compito specifico:

- **Gestore della memoria secondaria**: è il modulo che gestisce lo spazio fisico su disco usato per memorizzare i dati. Lavora con pagine (blocchi).
- **Gestore del buffer**:è il modulo del DBMS che si occupa di spostare in memoria centrale, nel **buffer,** le pagine richieste dalle transazioni, così da evitare accessi diretti e frequenti al disco.
- **Gestore dei metodi di accesso**: è il modulo del DBMS che si occupa di **eseguire fisicamente le operazioni sui dati** previste nel piano di esecuzione, traducendole in:

s**cansioni** sequenziali, a**ccesso diretto** tramite indici (es. B+-tree), o**rdinamenti, join** tra tabelle, letture/scritture di specifici blocchi di dati su disco.

- **Gestore delle interrogazioni**: è il modulo del DBMS che si occupa di: **interpretare** la query scritta in SQL. **Tradurla** in un piano eseguibile. **Ottimizzarla** per ottenere il miglior tempo di risposta. **Passarla al motore di esecuzione**, che la realizza usando metodi fisici di accesso ai dati.
- **Gestore della concorrenza**: controlla l'esecuzione parallela delle transazioni. Serve per garantire consistenza e isolamento.
- **Gestore dell'affidabilità**: è il modulo del DBMS incaricato di garantire che i dati rimangano **corretti e coerenti anche in presenza di guasti**. Serve per garantire consistenza, atomicità e persistenza.





Strutture di Accesso ai Dati

Memoria secondaria e gestione dei dati

Le basi di dati gestite da un DBMS risiedono **in memoria secondaria** perché:

- Sono **troppo grandi** per essere contenute interamente in RAM.

- Devono essere **persistenti**, ovvero conservare i dati anche dopo lo spegnimento del sistema.

La memoria secondaria è **più lenta** della memoria centrale e permette solo operazioni di **lettura/scrittura di blocchi interi**. Per ottimizzare l'accesso, si usa il **buffer**.

Il buffer e il gestore del buffer

Il **buffer** è un'area della memoria centrale organizzata in pagine, dove vengono temporaneamente caricati i blocchi della memoria secondaria, ogni pagina ha la dimensione di un numero intero di blocchi.

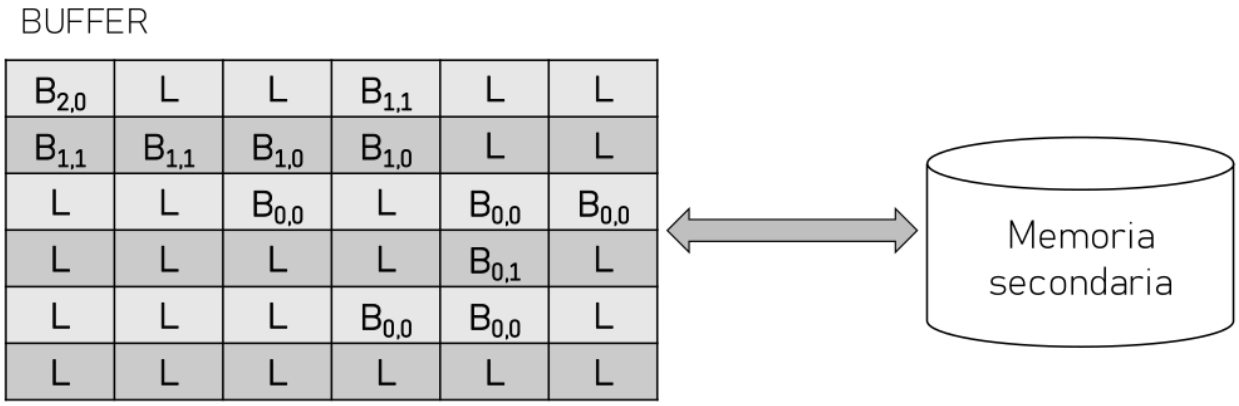
Lettura di un blocco:

- se il blocco è presente in una pagina del buffer allora non si esegue una lettura su memoria secondaria e si restituisce un puntatore alla pagina del buffer
- altrimenti si cerca una pagina libera e si carica il blocco nella pagina, restituendo il puntatore alla pagina stessa.

Scrittura di un blocco

- In caso di richiesta di scrittura di un blocco precedentemente caricato in una pagina del buffer, il gestore del buffer può decidere di differire la scrittura su memoria secondaria in un secondo momento.

Obiettivo: ridurre al minimo gli accessi alla memoria secondaria, sfruttando la **località** dei dati (i dati usati di recente tendono a essere riutilizzati) inoltre, una nota **legge empirica** dice che: "il 20% dei dati e acceduto dall' 80% delle applicazioni".

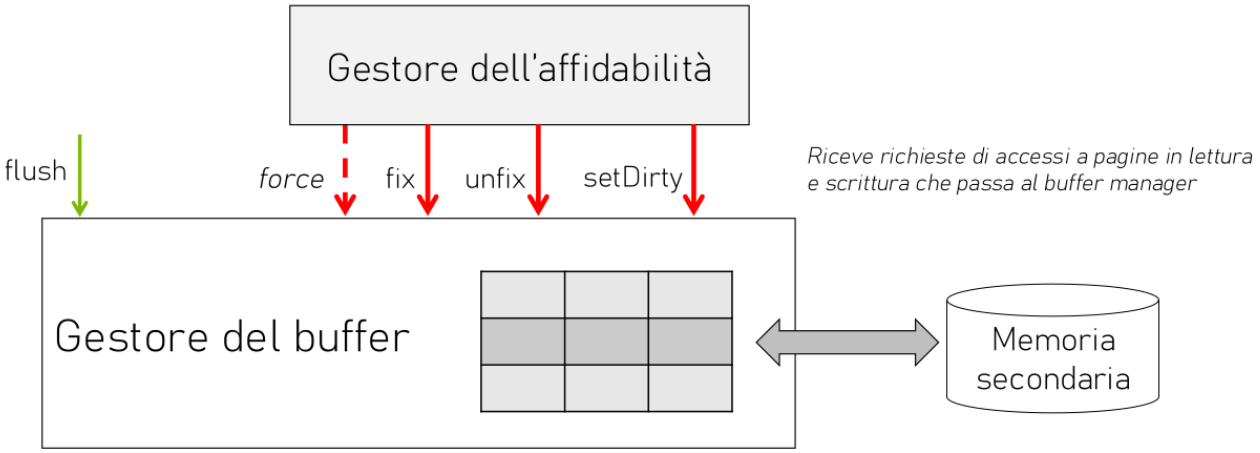


$B_{i,j}$ indica che nella pagina del buffer è caricato il blocco B, inoltre "i" indica che il blocco è attualmente utilizzato da i transazioni mentre "j" è 1 se il blocco è stato modificato e 0 altrimenti. L indica pagina libera.

Primitive fondamentali

Per gestire il buffer, il DBMS usa alcune **primitive operative**:

- `fix`: viene usata dalle transazioni per richiedere l'accesso ad un blocco e restituisce al chiamante un puntatore alla pagina contenente il blocco richiesto.
- `unfix`: viene usata dalle transazioni per indicare che la transazione ha terminato di usare il blocco. L'effetto è il decremento del contatore I che indica l'uso della pagina.
- `setDirty`: viene usata dalle transazioni per indicare al gestore del buffer che il blocco della pagina è stato modificato. L'effetto è la modifica del bit di stato J a 1
- `force`: viene usata per salvare in memoria secondaria in modo SINCRONO il blocco contenuto nella pagina (primitiva usata dal modulo Gestore dell'Affidabilità). L'effetto è il salvataggio in memoria secondaria del blocco e il bit di stato J posto a zero.
- `flush`: viene usata dal gestore del buffer per salvare blocchi sulla memoria secondaria in modo ASINCRONO. Tale operazione libera pagine "dirty" (il bit di stato viene posto a zero)



Politiche di gestione delle pagine

Il gestore del buffer deve decidere **quali pagine rimpiazzare** quando il buffer è pieno.

Politiche comuni:

- **LRU (Least Recently Used)**: si rimuove la pagina usata meno recentemente.
- **FIFO (First In First Out)**: si rimuove la pagina più "vecchia".

Per evitare la perdita di modifiche:

- Pagine marcate come `dirty` devono essere **scrivibili** su disco prima del rimpiazzo (`flush`).

Uso del contatore e del bit di stato

Ogni pagina ha:

- Un **contatore** `I`: quante transazioni la stanno usando.
- Un **bit** `J`: indica se è stata modificata (`J=1`) o no (`J=0`).

Steal e No-Steal

Se non ci sono pagine libere, il gestore può:

- **STEAL**: “rubare” una pagina usata da un’altra transazione ed eventualmente fare `flush`.
- **NO-STEAL**: sospendere la transazione corrente fino a che non si libera una pagina.

Gestore dell’affidabilità

È il modulo che garantisce:

- **atomicità:** che le transazioni non vengano lasciate incomplete
- **persistenza:** gli effetti di ciascuna transazione con conclusa con un commit siano mantenuti in modo permanente

Il log e la memoria stabile

Per garantire **affidabilità**, tutte le operazioni vengono registrate in un **log** su **memoria stabile** (memoria resistente ai guasti) .

Record di Transazione:

- Record di inizio transazione (`B(T)`), commit (`C(T)`), abort (`A(T)`).
- Record delle modifiche con stato prima (`BS`) e dopo (`AS`):
  - `I(T, O, AS)` → insert
  - `D(T, O, BS)` → delete
  - `U(T, O, BS, AS)` → update

Questo sistema consente il **ripristino (recovery)** del database in caso di guasto.

I record di transazione salvati nel LOG consentono di eseguire in caso di ripristino le seguenti operazioni:

- **UNDO**: per disfare un’azione su un oggetto O è sufficiente ricopiare in O il valore BS; l’insert/delete viene disfatto cancellando/inserendo O;
- **REDO**: per rifare un’azione su un oggetto O è sufficiente ricopiare in O il valore AS; l’insert/delete viene rifatto inserendo/cancellando O;

Gli inserimenti controllano sempre l’esistenza di O (non si inseriscono duplicati)

Record di Sistema:

- Operazione di **DUMP** della base di dati: record di DUMP, copia completa e consistente della base di dati.
- Operazione di **CheckPoint**: record CK(T1, …, Tn) indica che all’esecuzione del CheckPoint le transazioni attive erano T1, …, Tn
  1. Sospensione delle operazioni di scrittura, commit e abort delle transazioni
  2. Esecuzione della primitiva force sulle pagine “dirty” di transazioni che hanno eseguito il commit
  3. Scrittura sincrona (primitiva force) sul file di LOG del record di CheckPoint con gli identificatori delle transazioni attive
  4. Ripresa delle operazioni di scrittura, commit e abort delle transazioni

Regole di scrittura sul LOG

- Regola WAL (Write Ahead Log): i record di log devono essere scritti sul LOG prima dell’esecuzione delle corrispondenti operazioni sulla base di dati (garantisce la possibilità di fare sempre UNDO)
- Regola Commit-Precedenza: i record di log devono essere scritti sul LOG prima dell’esecuzione del COMMIT della transazione (garantisce la possibilità di fare sempre REDO)

Strutture fisiche e metodi di accesso

Il **DBMS**, una volta ottenuti i blocchi dal file system, organizza al suo interno i dati secondo **strutture fisiche** e li rende accessibili tramite **metodi di accesso**.

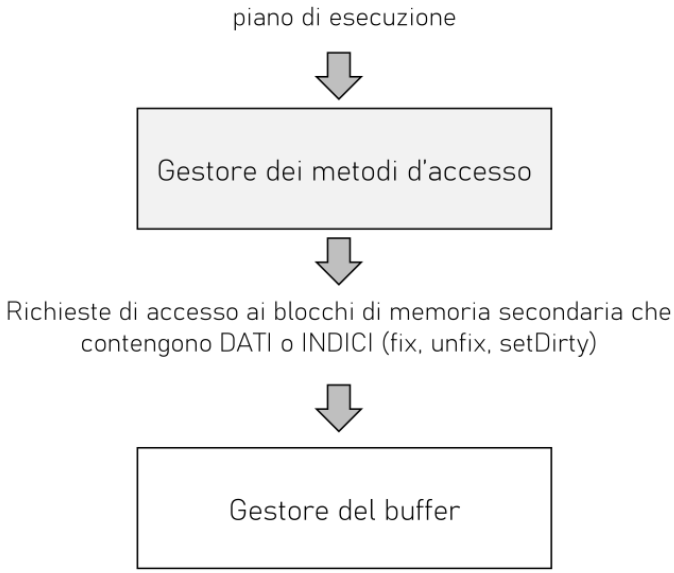
- Ogni **blocco** in memoria secondaria può contenere:
  - Tuple (cioè righe) di una tabella
  - Record di un indice
- I **metodi di accesso** sono moduli software che:
  - Implementano algoritmi di scansione (es. sequenziale, tramite indice)

- Gestiscono operazioni come ordinamento o join

Gestore dei Metodi di Accesso

E’ il modulo del DBMS che esegue il piano di esecuzione prodotto dall'ottimizzatore e produce sequenze di richieste di accessi ai blocchi della base di dati presenti in memoria secondaria.

Le richieste vengono inviate al gestore del buffer che si occupa di caricare i blocchi necessari in pagine di memoria centrale.

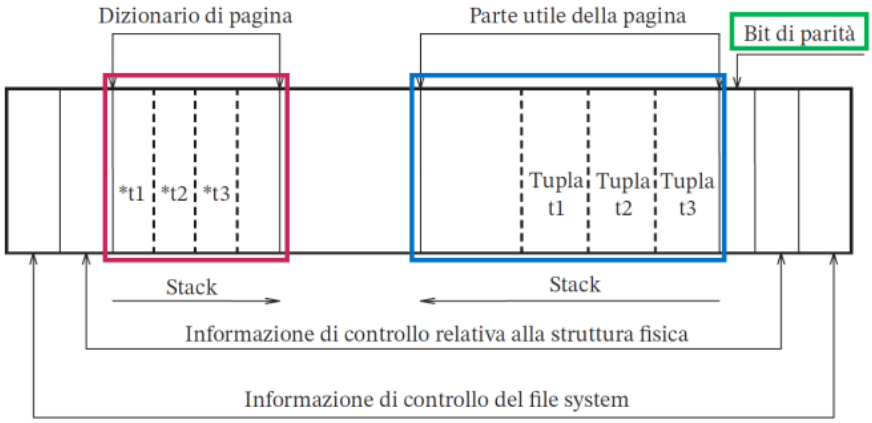


Organizzazione delle pagine

Una **pagina** è la copia di un blocco di memoria secondaria portata nel buffer.

Ogni pagina è composta da:

- **Informazioni utili**: i dati veri e propri (tuple)
- **Informazioni di controllo**:
    - Header e trailer (usati dal file system e dal DBMS)
    - Parte utile: contiene i dati
    - **Dizionario di pagina**: contiene puntatori a ciascun dato utile contenuto dalla pagina
    - **Bit di parità**: verifica l’integrità della pagina



Organizzazione interna:

- Le tuple possono essere memorizzate a partire da una fine della pagina
- Il dizionario cresce dalla parte opposta
- L'area centrale è lo spazio libero che si riduce quando si inseriscono dati

---

Gestione di tuple

- Se le **tuple hanno lunghezza fissa**, basta sapere l'offset iniziale e la dimensione.
- Se le **tuple hanno lunghezza variabile** (es. campi stringa o `NULL`):
  - Il dizionario memorizza l'offset di ogni tupla e di ogni attributo
  - È difficile dividere una tupla su più pagine (alcuni DBMS non lo permettono)
  - PostgreSQL adotta la tecnica **TOAST(The Oversized-Attribute Storage Technique)** per gestire attributi molto grandi

---

Operazioni su una pagina

C → CREATE , R → RETRIVE , U → UPDATE , D → DELETE

- **Inserimento**:
  - Se c'è spazio contiguo: inserimento diretto
  - Se non contiguo ma sufficiente: si riorganizza in memoria centrale
  - Se non c'è spazio: si alloca un nuovo blocco
- **Cancellazione**:
  - Sempre possibile
  - Spesso si marcano le tuple come "cancellate"

- **Modifica**:
    - Se la nuova tupla ha stessa lunghezza → modifica in loco
    - Se cambia dimensione → cancellazione + reinserimento

Accesso ai dati

- Accesso **diretto**: tramite chiave o offset nel dizionario
- Accesso **sequenziale**: per scansioni complete o parziali
- Accesso ad **attributi singoli**: tramite offset e lunghezza del campo

---

Strutture primarie di file

Queste strutture definiscono **come le tuple sono disposte nei blocchi** su disco.

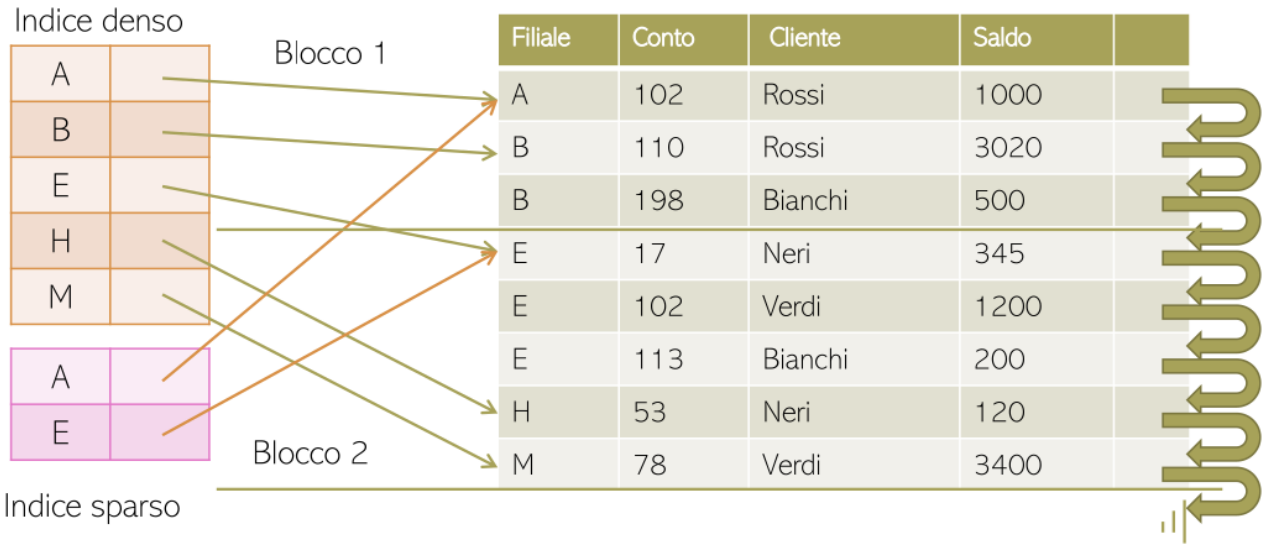
1. **Sequenziale disordinato (seriale)**:
    - Le tuple sono inserite in ordine di arrivo
    - Inserimenti molto efficienti
    - Ricerche lente (scansione completa)
    - Spesso usato insieme agli indici
2. **Sequenziale ordinato**:
    - Le tuple sono ordinate secondo un **campo di ordinamento** (non necessariamente la chiave primaria)
    - Vantaggi:
        - Efficienti operazioni come `ORDER BY`, range query
    - Svantaggi:
        - Inserimenti e aggiornamenti più costosi
3. **Sequenziale ad array**:
    - Possibile solo se le tuple hanno dimensione fissa
    - Tuple disposte come in una matrice (es. `nxm`)
    - Poco usato nei DBMS reali per difficoltà gestionali

Indici

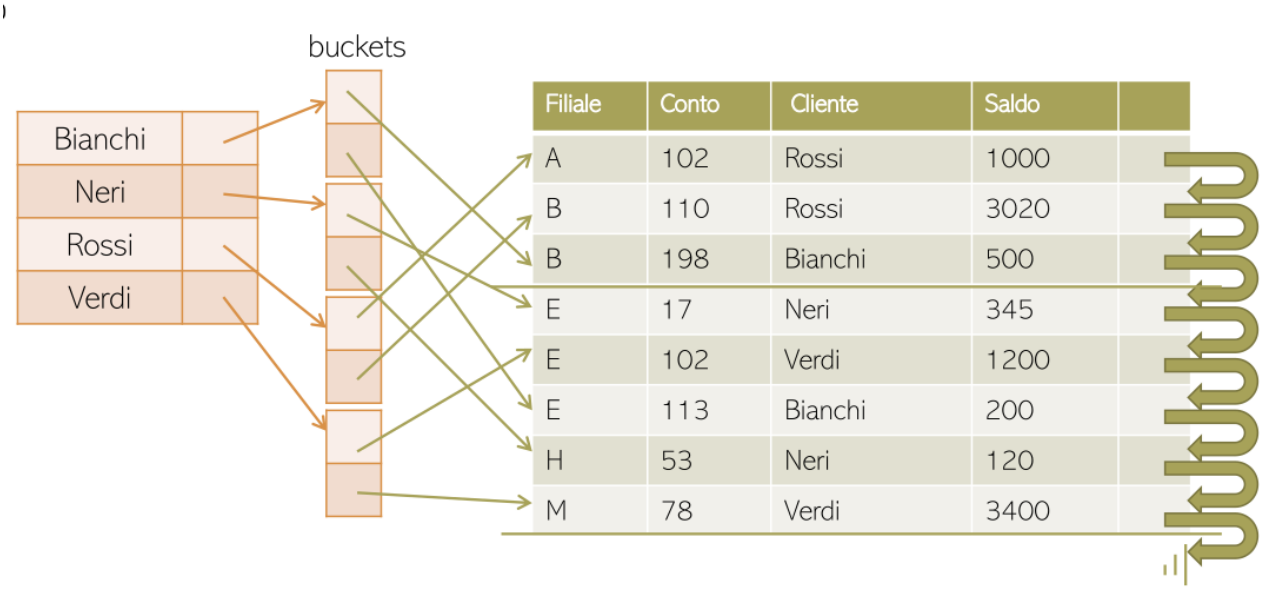
Per aumentare le prestazioni degli accessi alle tuple memorizzate nelle strutture fisiche (file sequenziale), si introducono strutture ausiliarie (dette strutture di accesso ai dati o INDICI). Tali strutture velocizzano l'accesso casuale via chiave di ricerca.

Indici su file sequenziali

- **INDICE PRIMARIO**: in questo caso la chiave di ordinamento del file sequenziale coincide con la chiave di ricerca dell'indice.



- **INDICE SECONDARIO**: in questo caso invece la chiave di ordinamento e la chiave di ricerca sono diverse.



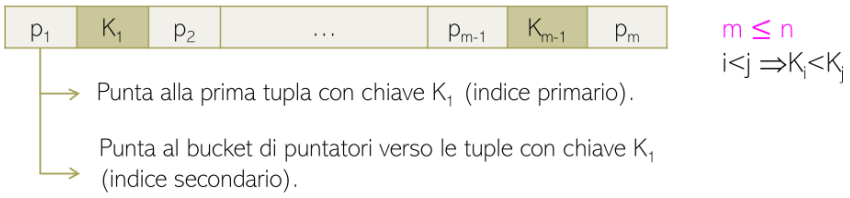
B+-tree e Hashing

B+-tree

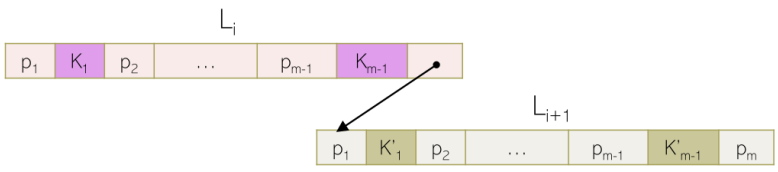
Il **B+-tree** è una struttura dati ad albero bilanciato usata nei DBMS per implementare **indici**. Serve a migliorare l'accesso ai dati, specialmente per ricerche basate su chiavi e operazioni di tipo `range`.

Struttura del B+-tree (fan-out=n)

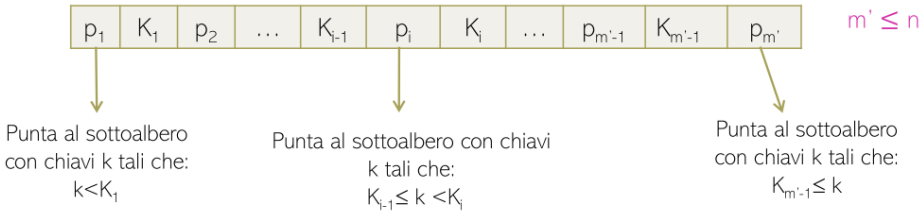
- **Nodo foglia**
  - può contenere fino a (n-1) valori ordinati di chiave di ricerca e fino a n puntatori



  - I nodi foglia sono ordinati.
  - Il puntatore pm del nodo Li punta al nodo Li+1 se esiste.



- **Nodo intermedio**



Vincoli di riempimento

- NODO FOGLIA (vincolo di riempimento con fan-out = n)
  - Ogni nodo foglia contiene un numero di valori chiave (#chiavi) vincolato come segue:

Arrotondamento all'intero superiore più vicino $\longrightarrow \lceil (n-1)/2 \rceil \leq \#chiavi \leq (n-1)$

- NODO INTERMEDIO (vincolo di riempimento con fan-out = n)
  - Ogni nodo intermedio contiene un numero di puntatori (#puntatori) vincolato come segue (per la radice non vale il minimo):

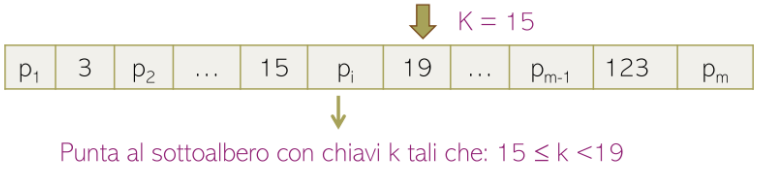
Arrotondamento all'intero superiore più vicino $\longrightarrow \lceil n/2 \rceil \leq \# puntatori \leq n$

Proprietà importanti:

- Tutte le foglie si trovano allo stesso livello.
- L'albero è **bilanciato**: ogni percorso radice-foglia ha uguale lunghezza.

Ricerca (con chiave K)

1. Si parte dalla **radice** e si cerca il il più piccolo valore di chiave maggiore di K
   - Se tale valore esiste (supponiamo sia Ki) allora seguire il puntatore Pi.
   - Se tale valore non esiste seguire il puntatore pm



2. Se il nodo raggiunto è un nodo foglia cercare il valore K nel nodo e seguire il corrispondente puntatore verso le tuple, altrimenti riprendere il passo 1.

Il costo è proporzionale alla **profondità** dell'albero:

$$prof_{B+tree} \leq 1 + \log_{\lceil n/2 \rceil}\left(\frac{\#valoriChiave}{\lceil (n-1)/2 \rceil}\right)$$

Inserimento (con chiave K)

1. Si trova la foglia in cui inserire la chiave K.
2. Se c'è spazio → si inserisce ordinatamente.
3. Se la foglia è piena → si effettua uno **split**:
    - Si creano 2 nodi foglia
    - I primi $\lceil (n-1)/2 \rceil$ nel primo e i rimanenti nel secondo
    - Inserire nel nodo padre un nuovo puntatore per il secondo nodo foglia generato e riaggiustare i valori chiave presenti nel nodo padre.
    - Se anche il nodo padre è pienolo SPLIT si propaga al padre e così via.

Lo **split** garantisce che l'albero resti bilanciato e che le proprietà siano rispettate.

Cancellazione (con chiave K)

1. Si localizza il nodo foglia con chiave da cancellare.
2. Si rimuove K dal nodo foglia col suo puntatore
3. Se il nodo ha ancora abbastanza chiavi → finito.
4. Se il nodo foglia scende sotto il vincolo di riempimento minimo → si effettua un **merge:**
    - Individuare il nodo fratello adiacente
    - Se hanno complessivamente al massimo n-1 valori chiave
        - si genera un unico nodo contenente tutti i valori
        - si toglie un puntatore dal nodo padre
        - si aggiustano i valori chiave del nodo padre
    - Altrimenti si distribuiscono i valori chiave tra i due nodi e si aggiustano i valori chiave del nodo padre
    - Anche il padre potrebbe subire merge → propagazione verso l'alto.

Anche qui, l'albero si **mantiene bilanciato**.

Strutture ad accesso calcolato (hashing)

Si basano su una funzione di hash che mappa i valori della chiave di ricerca sugli indirizzi

di memorizzazione delle tuple nelle pagine dati della memoria secondaria;

$h: K \rightarrow B$ K: dominio delle chiavi, B: dominio degli indirizzi

Problema delle strutture ad accesso calcolato sono le collisioni, cioè valori di chiave di ricerca diversi, portano allo stesso valore di indice.

Se T è il numero di tuple previsto nel file, F è il fattore di blocco (quante tuple per blocco) e f è il fattore di riempimento(frazione di spazio fisico mediamente utilizzata in ciascun blocco): il file può prevedere un numero di blocchi B pari a $B = T/(f \times F)$.

È pesante cambiare la funzione di hashing dopo che la struttura d'accesso è stata riempita; si deve ricostruire l'indice da capo.

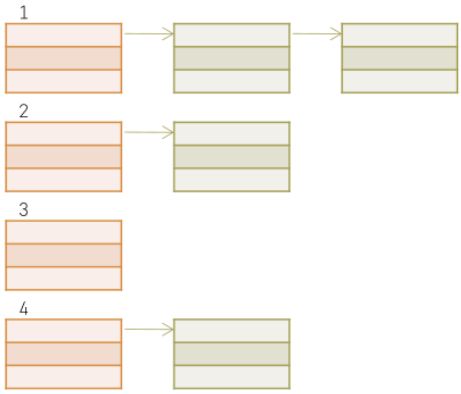
Costi Ricerca, Inserimento e Cancellazione

- Calcolare b = h(f(K)) (costo zero)
- Accedere al bucket b (costo: 1 accesso a pagina)
- Accedere alle n tuple attraverso i puntatori del bucket (costo: m accessi a pagina con m<=n)

Collisioni

Si verifica quando, dati due valori di chiave K1 e K2 con K1 K2, risulta: h(f(K1)) = h(f(K2))

Per gestire tale situazione si prevede la possibilità di allocare Bucket di overflow, collegati al Bucket di base.



Confronto Hashing vs B+-Tree

Ricerca

Selezioni basate su condizioni di uguaglianza → A = cost: Hashing (senza overflow buckets): tempo costante

Selezioni basate su range → A>cost1 AND A<cost2: B+-tree: tempo logaritmico per accedere al primo valore dell'intervallo, scansione dei nodi foglia fino all'ultimo valore compreso nel range.

Inserimenti e cancellazioni

•Hashing: tempo costante + gestione overflow

•B+-tree: tempo logaritmico nel numero di chiavi + split/merge

---

Concorrenza

Anomalie dell’esecuzione concorrente

Quando due o più transazioni accedono agli stessi dati senza sincronizzazione, possono verificarsi **anomalie**:

1. Perdita di aggiornamento

Due transazioni leggono un valore e lo aggiornano. L’ultima che scrive sovrascrive il lavoro dell’altra.

Esempio:

```
T1: bot r1(x); x = x + 100; w1(x); commit; eot
T2: bot r2(x); x = x - 50; w2(x); commit; eot
```

Se eseguite in parallelo senza controllo → uno degli aggiornamenti va perso.

---

2. Lettura sporca

Una transazione legge un dato **modificato da un’altra transazione non ancora confermata** (commit). Se la seconda viene annullata, la prima ha letto un valore **non valido**.

Esempio:

```
T1: bot r1(x); commit; eot
T2: bot r2(x); x=x+1; w2(x); … abort;
```

T1 ha letto un valore che **non è mai esistito realmente** nel database confermato.

---

3. Lettura inconsistente (non repeatable read)

Una transazione legge lo stesso dato due volte e ottiene due valori diversi, perché un’altra transazione lo ha modificato nel frattempo.

Esempio:

```
T1: bot r1(x)→2; r1’(x)→ 3; commit; eot
T2: bot r2(x)→2; x=x+1; w2(x)→ 3; commit; eot
```

---

4. Aggiornamento fantasma

Durante una scansione, una transazione vede cambiare il risultato della propria query a causa di modifiche effettuate da un'altra transazione.

Esempio:

Vincoli: x+y+z = 100

```
T1:bot r1(y)→20; r1(x)→20; r1(z)→50; s = x+y+z → 90; eot
T2:bot r2(y)→20; y = y + 10; r2(z)→60; z = z – 10; w2(y)→30; w2(z)→50; commit;
eot
```

---

5. Inserimento fantasma

Simile al caso precedente, ma focalizzato sull'inserimento di **nuove tuple** che soddisfano il predicato della query.

Schedule

Uno **schedule** è una sequenza di operazioni (lettura/scrittura) di più transazioni. Può essere:

- **Seriale**: tutte le operazioni di una transazione vengono eseguite prima di passare alla successiva. → s2: r1(x) w1(x) r2(x) w2(x) è SERIALE
- **Non seriale**: le operazioni sono mescolate. → s1: r1(x) r2(x) w2(x) w1(x)NON è SERIALE

Obiettivo: consentire **schedule non seriali** che **mantengano la correttezza** (cioè siano equivalenti a un qualche schedule seriale).

Serializzabilità

Perché serve la serializzabilità

Abbiamo visto che eseguire più transazioni contemporaneamente può causare **anomalie**. L'obiettivo è trovare un criterio per decidere **quali esecuzioni parallele sono sicure**.

Idea centrale: uno schedule è **corretto** se produce lo stesso effetto di uno **schedule seriale** (cioè con transazioni eseguite una alla volta).

Schedule e serializzabilità

Uno schedule è **serializzabile** se il risultato finale è equivalente a quello di uno schedule **seriale**.

Ci sono due principali criteri di equivalenza:

1. **View-serializzabilità (VSR)** – più generale
2. **Conflict-serializzabilità (CSR)** – più semplice da verificare

---

View-Serializzabilità (VSR)

Due schedule sono **view-equivalenti** se:

1. Ogni operazione di **lettura** in entrambi legge lo stesso valore (dalla stessa scrittura).
2. L'insieme delle **scritture finali** (cioè le ultime scritture su ogni dato) è identico.

Per calcolare la view-serializzabilità si definiscono:

- **LEGGE_DA**: per ogni lettura si indica da quale scrittura precedente deriva.
- **SCRITTURE_FINALI**: si identifica l'ultima scrittura su ogni variabile.

VSR è **corretto ma difficile da verificare automaticamente**, perché richiede l'analisi di tutte le possibili permutazioni seriali.

Conflict-Serializzabilità (CSR)

Due operazioni sono in **conflitto** se:

- Agiscono sulla **stessa variabile**
- Almeno una è una **scrittura**
- Provengono da **transazioni diverse**

Tipi di conflitto:

- `ri(x)` ↔ `wj(x)`
- `wi(x)` ↔ `rj(x)`
- `wi(x)` ↔ `wj(x)`

Verifica della CSR:

1. Si costruisce il **grafo dei conflitti**:
    - Nodi = transazioni
    - Arco da Ti a Tj se un'operazione di Ti precede ed è in conflitto con un'operazione di Tj
2. Lo schedule è **CSR** se il grafo è **aciclico**

CSR è **più restrittiva** ma può essere verificata **in tempo polinomiale** → molto usata nei DBMS reali.

Esempio

Schedule:

```
S = r1(x), w1(x), r2(x), w2(x)
```

Conflitti:

- `r1(x)` ↔ `w2(x)` → T1 → T2

- `w1(x)` ↔ `r2(x)` → T1 → T2
- `w1(x)` ↔ `w2(x)` → T1 → T2

Grafo:

- Nodo T1 → Nodo T2
  → Aciclico → **CSR**

CSR ⇒ VSR, ma NON vale il contrario

Ogni schedule CSR è anche VSR, ma non tutti i VSR sono CSR.

Implicazioni:

- CSR è sufficiente per la correttezza, ma non necessaria.
- VSR è più potente ma più difficile da usare in pratica.

Ottimizzazione delle Interrogazioni

Motivazione

Una query SQL è **dichiarativa**: l'utente specifica *cosa vuole*, non *come* ottenerlo.
Il compito di scegliere il *come* è affidato al **DBMS**, che deve generare un **piano di esecuzione efficiente**.

Questo processo si chiama **ottimizzazione delle interrogazioni**.

Fasi della compilazione di una query

1. **Analisi lessicale e sintattica**:
   - La query SQL viene trasformata in un **albero sintattico**.
2. **Traduzione in algebra relazionale**:
   - Viene costruita un'espressione in **algebra relazionale** che rappresenta il significato della query.
3. **Ottimizzazione algebrica**:
   - Si trasforma l'algebra per ridurre il costo computazionale (senza cambiare il significato).
4. **Ottimizzazione dipendente dai metodi di accesso**:

    - Si sceglie il **metodo fisico** più efficiente per ogni operazione (scansione, join, uso indici, ecc.).
5. **Generazione del piano di esecuzione**:
    - Si ottiene un **programma eseguibile** dal DBMS.

Ottimizzazione algebrica

Si basa su **regole di trasformazione** dell'algebra relazionale, simili a semplificazioni matematiche.

Principali tecniche:

- **Anticipo delle selezioni** (*selection push*): eseguire i filtri il prima possibile per ridurre le dimensioni intermedie.
- **Anticipo delle proiezioni** (*projection push*): selezionare solo gli attributi necessari sin da subito.
- **Riorganizzazione degli operatori**: ad esempio, applicare il join prima di un ordinamento.

Queste trasformazioni **non richiedono informazioni sul costo**, si basano solo sulla logica dell'algebra.

Ottimizzazione fisica: metodi di accesso

Una volta definito cosa fare, bisogna decidere **come farlo**.
Il DBMS può scegliere diversi **metodi fisici** per accedere ai dati, a seconda delle strutture disponibili.

1. Scansione

- Il metodo più semplice: si leggono tutte le pagine di una relazione.
- Costo : numero di pagine relazione R = `NP(R)`
- Varianti:
    - Scansione con proiezione
    - Scansione con selezione
    - Scansione con modifica (insert/delete/update)

---

2. Ordinamento (Sort)

Serve per:

- `ORDER BY`
- Eliminare duplicati (`SELECT DISTINCT`)

- `GROUP BY`

Algoritmo usato: **Z-way Sort-Merge** (tipicamente `Z=2`):

- **Fase 1: Sort interno** → si leggono una alla volta le pagine della tabella; le tuple di ogni pagina vengono ordinate facendo uso di un algoritmo di sort interno (es. QuickSort); ogni pagina così ordinata, detta anche "run", viene scritta su memoria secondaria in un file temporaneo.
- **Fase 2: Merge esterno** → si fondono le "run" ordinate in più passaggi.

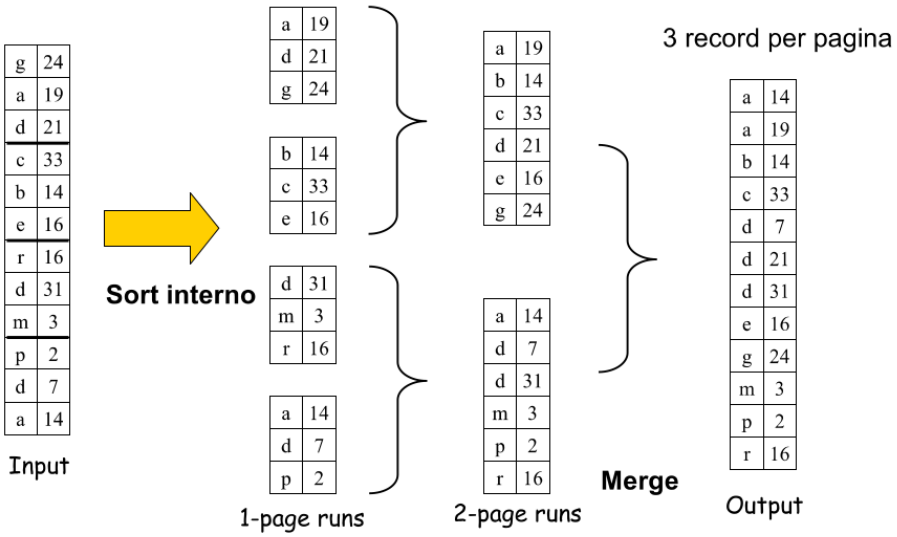
Costo approssimato:

$2 * NP * (1 + \log_2(NP))$

Dove `NP` è il numero di pagine da ordinare.

Esempio

Caso base a due vie (Z = 2), e supponiamo di avere a disposizione solo 3 buffer in memoria centrale (NB = 3)



---

3. Accesso diretto tramite indice

Se esiste un **indice** su un attributo usato nella WHERE, si può accedere **direttamente** ai dati.

Tipi di selezione supportati:

- `A = v`: uguaglianza → indice **hash** o **B+-tree**
- `A >= v1 AND A <= v2`: intervallo → solo con indice **B+-tree**
- `A = v1 AND B = v2`: si usa l'indice sulla condizione **più selettiva**

- `A = v1 OR B = v2`: se esistono entrambi gli indici, si uniscono i risultati (eliminando duplicati), se manca anche un solo indice → serve scansione sequenziale.

Ottimizzare il JOIN

Il **JOIN** è una delle operazioni più costose nelle query relazionali, soprattutto quando le relazioni coinvolte sono grandi.

Algoritmi principali per il JOIN

1. Nested-Loop Join

Idea: per ogni tupla della relazione esterna `R`, si scansiona l'intera relazione interna `S` per trovare le tuple che soddisfano la condizione di JOIN.

```
for each tR in R:
    for each tS in S:
        if tR e tS soddisfano il join:
            aggiungi (tR, tS) al risultato
```

Costo base:

```
NP(R) + NR(R) * NP(S)
```

Dove:

- `NP(R)` = pagine di R
- `NR(R)` = numero di tuple di R
- `NP(S)` = pagine di S

Variante con indice:

Se `S` ha un **indice** sull'attributo di join:

- Non si legge tutta `S`, ma solo le pagine puntate dall'indice.

Costo:

```
NP(R) + NR(R) * (ProfIndice + NR(S)/VAL(A, S))
```

- `VAL(A, S)`: numero di valori distinti dell'attributo `A` in `S`.

Utile se `S` è grande e l'indice è molto selettivo.

---

2. Block Nested-Loop Join

Variante del Nested-Loop Join che legge **più tuple di R in blocco**, riducendo il numero di accessi a `S`.

Costo:

```
NP(R) + NP(R) * NP(S)
```

- Se si può bufferizzare S, il costo si riduce a:

```
NP(R) + NP(S)
```

Più efficiente del Nested Loop semplice, ma richiede **più buffer**.

---

3. Merge-Scan Join

Requisiti:

- Gli insiemi di tuple in input sono ordinati sugli attributi di join
- Il join è un equi-join

Procedura:

- Si scorrono entrambe le relazioni in parallelo.
- Se le chiavi corrispondono → si produce la tupla di output.

Costo:

```
NP(R) + NP(S)      (input già ordinati)
```

Veloce e lineare, ideale per `ORDER BY`, `GROUP BY` e `DISTINCT`.

---

4. Hash-based Join

Valido solo per equi-join.

Idea:

- Si partiziona R e S usando una **funzione hash** sull'attributo di join.
- Si costruisce una hash table con una delle due relazioni (di solito la più piccola).
- Si accoppiano solo le **partizioni con hash uguale**.

Costo:

- Costruzione: `NP(R) + NP(S)`
- Matching (peggiore): `NP(R) * NP(S)`, ma in media molto meno

Molto efficiente per dataset grandi, non richiede ordinamento o indici.

Scelta del piano di esecuzione

Per ogni query, il DBMS considera:

- **Operatori alternativi** per ciascuna operazione (scan, join, ecc.)
- **Ordine delle operazioni** (grazie alla proprietà associativa del join)
- **Strutture disponibili**: indici, ordinamenti, statistiche

Processo:

1. Genera una lista (o albero) di **piani alternativi**
2. Calcola una **stima del costo** per ciascuno (in I/O)
3. Sceglie il **piano con costo minimo**

Informazioni usate per stimare i costi

Memorizzate nel **Data Dictionary** del DBMS:

- `CARD(T)`: cardinalità (#tuple)
- `SIZE(T)`: dimensione media delle tuple
- `VAL(A, T)`: numero di valori distinti di un attributo
- `MIN(A, T)`, `MAX(A, T)`: intervallo di valori
- `NP(T)`: numero di pagine

Questi dati aiutano a:

- Stimare la **selettività** dei predicati
- Calcolare la **riduzione** delle dimensioni intermedie
- Valutare il numero di accessi a disco

Architettura di MongoDB

Architettura generale

MongoDB è progettato per la **scalabilità** e l'**affidabilità**, grazie a due meccanismi chiave:

Replicazione

- Crea copie identiche dei dati su più server (nodi).
- Ogni **replica set** contiene:
    - Un **nodo primario** (gestisce le scritture)
    - Uno o più **nodi secondari** (copie sincrone/asinc.)

- In caso di guasto, si elegge un nuovo primario tramite **algoritmo Raft**.
- Utilizza un **oplog** (log delle operazioni) per sincronizzare i dati tra nodi.

Sharding

- Suddivide il dataset in **parti (chunk)** distribuite su più server (**shard**).
- Ogni shard è un **replica set**.
- Usato per scalare orizzontalmente.
- Il router **mongos** indirizza le query verso gli shard corretti.
- I **config server** mantengono metadati su sharding e chunk.

Replicazione in dettaglio

Replica set:

- Una copia completa del dataset.
- Solo il **nodo primario** può ricevere scritture.
- I **secondari** replicano i dati dall'oplog.

Elezione del primario:

- Avviene tramite votazione (algoritmo Raft).
- Scatta in caso di:
    - Timeout nella comunicazione
    - Inizializzazione
    - Aggiunta/rimozione nodi

Altri ruoli:

- **Arbitro**: partecipa alle elezioni ma non conserva dati.
- **Nodi nascosti/ritardati**: usati per backup/reporting.

Scritture: write concern

- `w`: numero minimo di nodi che devono confermare una scrittura (`0`, `1`, `majority`, `n`)
- `j`: scrittura confermata su disco (journaling)
- `wtimeout`: tempo massimo prima di considerare fallita la scrittura

Letture: read concern

- `local`: dato più recente sul nodo

- `available`: senza attesa (minima latenza)
- `majority`: solo dati confermati da maggioranza
- `linearizable`: massimo livello di consistenza
- `snapshot`: lettura coerente in una transazione

Sharding in dettaglio

- Ogni **shard** è un replica set che contiene una parte dei dati.
- **Shard key**: campo (o più) usato per distribuire i dati in **chunk**.
- MongoDB supporta:
    - **Ranged sharding** (range di valori della shard key)
    - **Hashed sharding** (hash della shard key)

Bilanciamento:

- Un processo chiamato **balancer** redistribuisce i chunk tra gli shard per evitare squilibri.
- Gestione di **jumbo chunk** e **resharding** per adattarsi a nuovi pattern.

Proprietà ACID in MongoDB

- **Atomicità**
    - Le operazioni su un **singolo documento** sono **sempre atomiche**.
    - Per operazioni multi-documento si usano **transazioni** (da MongoDB 4.x in poi).
- **Consistenza**
    - Supporta **eventual consistency** (default)
    - Supporta **strong consistency** con `readConcern: majority` e `writeConcern: majority`
- **Isolamento**
    - Supporta vari livelli con `readConcern`, `writeConcern`, e `session`
    - `snapshot` → isolamento per transazioni
    - `majority` → letture coerenti tra transazioni
- **Durability** (Persistenza)
    - Meccanismo **Write-Ahead Log** su disco ogni 100 ms