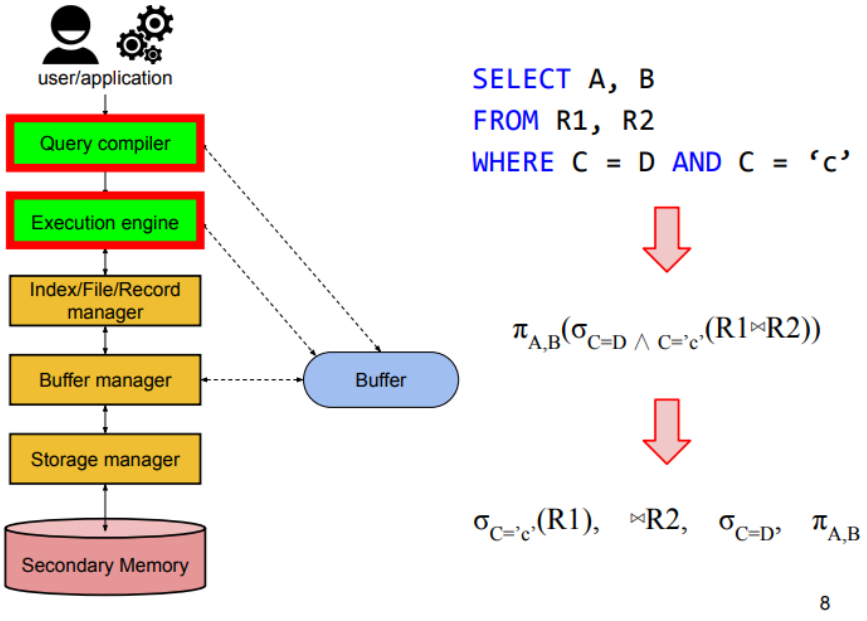
DBMS Architecture

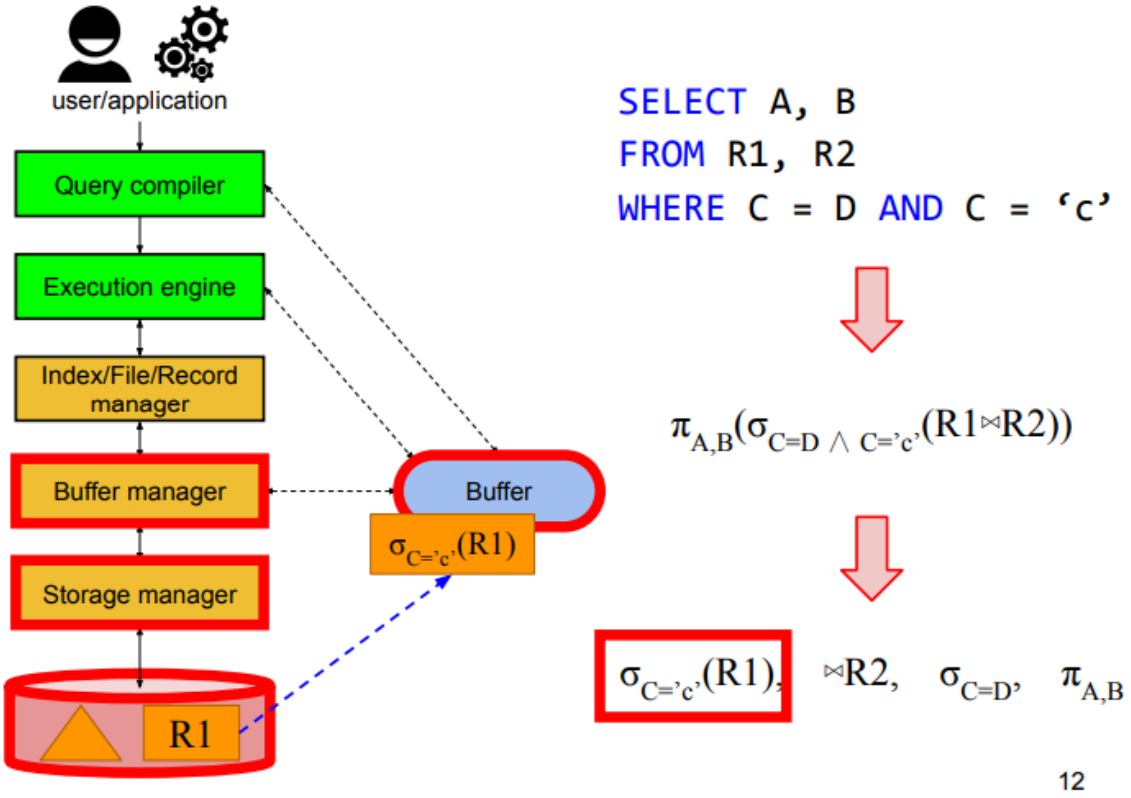
Un database management system (DBMS) è un software per creare, gestire e manipolare grandi quantità di dati in modo efficiente e persistente. L'architettura di un DBMS è composta da:

- **Query Processor**:
  è il componente che interpreta ed esegue la query SQL. Si compone di:
  - *parsing*: verifica la sintassi della query e costruisce un albero sintattico
  - *preprocessing*: verifica la semantica (es. se le tabelle e colonne esistono) e traduce la query in un albero di operazioni algebriche
  - *ottimizzazione*: trova la sequenza più efficiente di operazioni per eseguire la query (es. l'ordine delle tabelle di un join)
  - **Execution Engine**: esegue il piano ottimizzato interagendo con altri componenti, come la memoria e i moduli di gestione dei file.



- **Resource Manager**:
  Si occupa di gestire i file a livello fisico. E' responsabile della persistenza dei dati e dell'efficienza dell'accesso. Include:
  - *Index/File/Record Manager*: conosce la struttura dei dati e utilizza indici per velocizzare le ricerche;
  - *Buffer Manager*: carica i blocchi di dati dalla memoria secondaria alla memoria principale;

- *Storage Manager*: gestisce i file su disco, localizzandoli e leggendo i blocchi richiesti.



- **Transaction Manager**:
  si occupa delle transazioni, unità logiche di lavoro nel database spesso costituite da più query SQL. Esempio: transazione di trasferimento bancario:

```
START TRANSACTION;
UPDATE BANKACCOUNT SET Balance = Balance - 500 WHERE AccountNumber = 42177;
UPDATE BANKACCOUNT SET Balance = Balance + 500 WHERE AccountNumber = 12202;
COMMIT;
```

Concorrenza e Deadlock
In un sistema di database, più transazioni possono essere eseguite contemporaneamente, ma potremmo incontrare problematiche come:

- interferenza tra transazioni (es. una transazione modifica un dato mentre un'altra lo legge).
- inconsistenza: quando l'accesso simultaneo ai dati causa risultati non coerenti.
  Il **Concurrency-Control Manager** utilizza tecniche per garantire che le transazioni parallele non interferiscano negativamente tra loro:

1. *Locking* (blocco):
   - un lock è un meccanismo che impedisce ad altre transazioni di accedere a un dato finchè non viene rilasciato
   - due tipi principali:
     - *shared lock* (lettura): più transazioni possono leggere lo stesso dato
     - *exclusive lock* (scrittura): solo una transazione può modificare il dato
2. *schedulazione*: l'ordine di esecuzione delle transazioni viene determinato per garantire che i risultati siano equivalenti a una loro esecuzione sequenziale (serializzabilità).
   Un *deadlock* si verifica quando due o più transazioni sono bloccate in attesa di risorse detenute dall'altra. Esempio:

- Transazione T1 blocca il record X e vuole accedere al record Y.
- Transazione T2 blocca il record Y e vuole accedere al record X. Nessuna delle due può procedere.
  Il DBMS utilizza diverse strategie per risolvere questo tipo di conflitti:

1. **timeout**: una transazione viene terminata automaticamente se resta bloccata troppo a lungo.
2. **detection e rollback**
   - il sistema rileva i deadlock costruendo un grafo di attesa.
   - identificato il deadlock, una transazione viene abortita (rollback) per liberare le risorse.

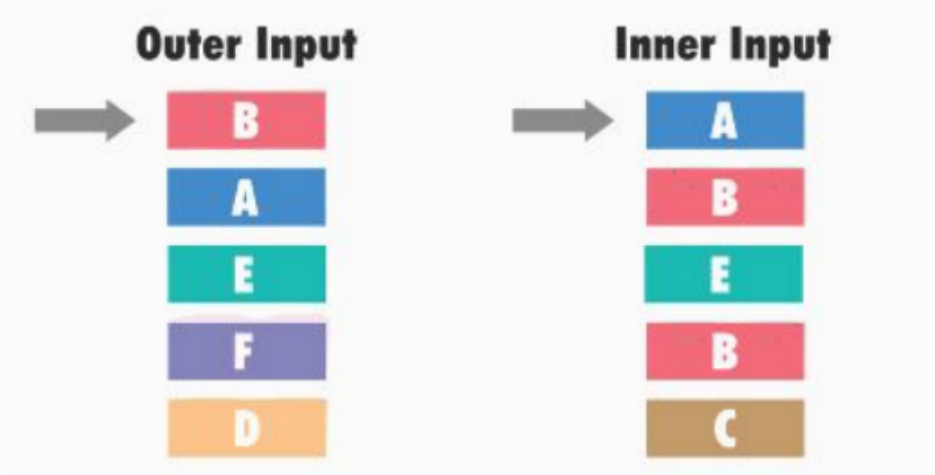
Logging and Recovery
Logging: Il *Log Manager* registra ogni modifica effettuata nel database, creando un registro che può essere utilizzato per recuperare il sistema in caso di guasto.
Recovery: Quando il sistema subisce un guasto. il *Recovery Manager* utilizza il log per riportare il database a uno stato consistente.

Tecniche di JOIN

Il join è un'operazione computazionalmente costosa, quindi il DBMS usa diverse tecniche per ottimizzarla.

Nested-loop JOIN

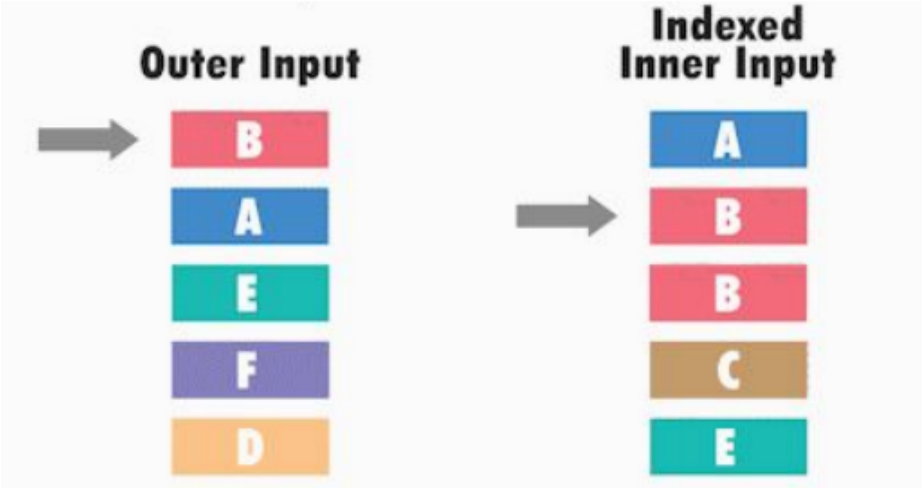
- più semplice, ma meno efficiente.
- per ogni riga della tabella R (outer), vengono scansionate tutte le righe della tabella S (inner)
  PRO: non richiede precondizioni. CONTRO: estremamente lento per tabelle grandi.
  (es. R ha righe 1000 righe e S ne ha 500, vengono effettuate 1000 * 500 = 500000 confronti)



Single-loop JOIN

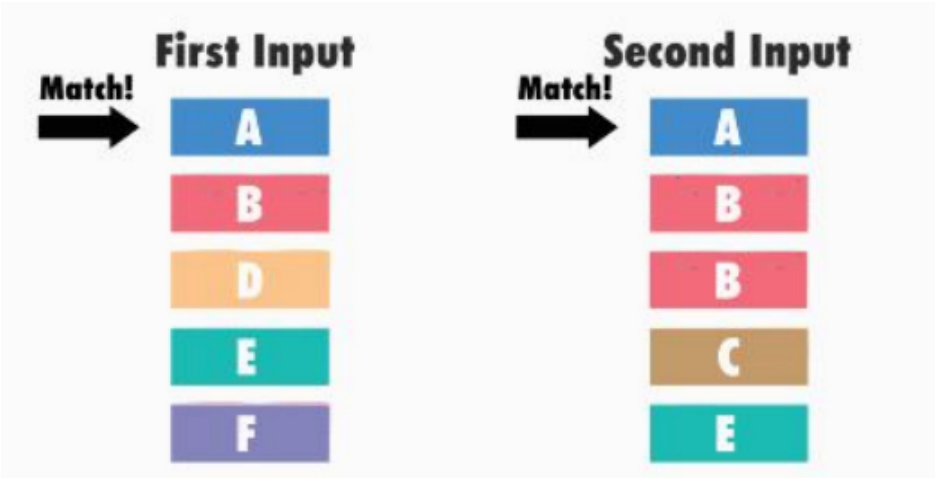
- usa un indice (o struttura hash) su una delle due tabelle.
- per ogni riga di una tabella (es. S), l'indice viene utilizzato per cercare direttamente le righe corrispondenti nell'altra tabella (es. R).
  Se esiste un indice sull'attributo di join di R, l'accesso a ogni riga è diretto, riducendo il numero totale di confronti.
  PRO: più efficiente rispetto al nested-loop se gli indici esistono.

CONTRO: richiede indici preesistenti.



Sort-merge JOIN

- ordina entrambe le tabelle sugli attributi di JOIN.
- scansiona le tabelle ordinate una sola volta, abbinando le righe con valori uguali.
  PRO: ottimo per tabelle grandi, specialmente quando devono essere ordinate.
  CONTRO: richiede una fase di ordinamento iniziale se le tabelle non sono già ordinate.



Hash-based JOIN

- usa una funzione di hashing per suddividere entrambe le tabelle in *bucket*.
- le righe nei bucket corrispondenti vengono abbinate.
  Esempio: con una funzione hash che divide una tabella in 10 bucket, ogni bucket viene processato separatamente, riducendo il numero di confronti.
  PRO: efficiente per tabelle grandi

CONTRO: richiede memoria sufficiente per gestire i bucket.