

Transazione

Una transazione è un insieme di operazioni che trasformano il database da uno stato corretto a un altro stato corretto. Quindi, anche se una transazione può essere composta da più operazioni di lettura e scrittura, il sistema deve garantire che l'intero blocco venga eseguito in modo completo e corretto oppure annullato completamente in caso di errore.

Ad esempio, pensa a un bonifico bancario:

Operazione 1: sottraggo 100€ dal conto A.

Operazione 2: aggiungo 100€ al conto B.

Se qualcosa va storto tra le due operazioni (es. errore di rete), non posso permettere che 100€ spariscano dal conto A senza finire su B. Il sistema deve garantire tutto o niente.

Le transazioni nei database devono rispettare le **proprietà ACID**, un acronimo che sta per:

1. **Atomicità (Atomicity)**
2. **Consistenza (Consistency)**
3. **Isolamento (Isolation)**
4. **Durabilità (Durability)**

Queste proprietà garantiscono che il database rimanga **corretto e affidabile**, anche in caso di errori o interruzioni.

1. Atomicità (Atomicity) – "Tutto o niente"

Una transazione deve essere eseguita **completamente o per niente**.

- Se una transazione è composta da più operazioni, **o vengono eseguite tutte con successo oppure nessuna**.
- Se c'è un errore o un'interruzione (es. crash del sistema), il database deve **annullare tutte le operazioni** fatte fino a quel punto.

♦ Esempio:

Immagina un bonifico bancario:

- **Sottraggo 100€ dal conto A** ✓
- **Aggiungo 100€ al conto B** ✗ (Errore!)

Se la seconda operazione fallisce, l'intera transazione deve essere **annullata**, altrimenti i soldi sparirebbero dal conto A senza arrivare a B.

In SQL: Si usa `ROLLBACK` per garantire l'atomicità.

2. Consistenza (Consistency) – "Il database deve rimanere valido"

Una transazione porta il database **da uno stato corretto a un altro stato corretto**, rispettando tutte le regole definite (vincoli di integrità, chiavi primarie, chiavi esterne, ecc.).

- Una transazione **non deve mai lasciare il database in uno stato invalido**.
- Se una transazione viola un vincolo di integrità (es. saldo negativo su un conto bancario), il sistema deve impedirne l'esecuzione.

♦ Esempio:

Se un vincolo di database stabilisce che il saldo di un conto **non può mai essere negativo**, un bonifico di 200€ da un conto con solo 100€ **deve essere bloccato**.

In SQL: Il DBMS blocca automaticamente le operazioni che violano i vincoli di consistenza.

3. Isolamento (Isolation) – "Le transazioni non devono interferire tra loro"

Se più transazioni vengono eseguite contemporaneamente, devono **comportarsi come se fossero eseguite una alla volta**.

- Se due utenti stanno modificando lo stesso dato, il sistema deve gestire la concorrenza per **evitare anomalie**.
- Esistono diversi **livelli di isolamento** (es. **Read Uncommitted, Read Committed, Repeatable Read, Serializable**) che determinano quanto le transazioni possono "vedere" le modifiche di altre transazioni.

♦ Esempio:

Immagina due transazioni che leggono e aggiornano il saldo dello stesso conto **contemporaneamente**:

1. La transazione A legge il saldo (100€).
2. La transazione B legge lo stesso saldo (100€).
3. Entrambe sottraggono 50€.
4. Entrambe salvano il nuovo saldo (50€ anziché il corretto 0€!).

In **SQL**: Si usano **lock** o livelli di isolamento (`SET TRANSACTION ISOLATION LEVEL ...`).

4. Durabilità (Durability) – "I dati devono essere salvati in modo permanente"

Una volta che una transazione è **confermata (COMMIT)**, le sue modifiche devono essere **permanenti**, anche in caso di crash o spegnimento del sistema.

- I database usano **log di ripristino** e **archiviazione su disco** per garantire la durabilità.
- Anche se il server si spegne all'improvviso dopo un `COMMIT`, le modifiche devono essere **recuperabili**.

♦ Esempio:

Se confermi un bonifico e il server si spegne subito dopo, quando si riaccende il sistema deve **mantenere la modifica** (e non annullarla come se non fosse mai avvenuta).

In **SQL**: Il database usa **file di log (WAL - Write-Ahead Logging)** per garantire la durabilità.

Riassunto delle proprietà ACID

Proprietà	Descrizione	Esempio
Atomicità	Tutte le operazioni devono essere eseguite o nessuna.	Se un bonifico fallisce a metà, tutto viene annullato.
Consistenza	Il database deve rimanere in uno stato valido.	Non posso avere un saldo negativo se non è permesso.
Isolamento	Le transazioni non devono interferire tra loro.	Se due utenti aggiornano lo stesso saldo, il risultato deve essere corretto.
Durabilità	Una volta confermata una transazione, le modifiche sono	Dopo un riavvio del server, i dati salvati restano.

Proprietà	Descrizione	Esempio
	permanenti.	

Lock e Concorrenza

Per garantire tali proprietà occorre effettuare controlli su **affidabilità** (per atomicità e durabilità), **concorrenza** (per isolamento) e sull' **esecuzione** delle operazioni (consistenza), che vengono realizzati dal gestore delle transazioni.

Il controllo sull'affidabilità è realizzato mediante un particolare file detto log che al suo interno mantiene l'insieme delle operazioni svolte sul DB mediante transazioni. (All'interno del file troviamo anche *checkpoint*, che indicano quali sono i dati già riportati in memoria dal DBMS dopo operazioni di commit)

Il log è memorizzato all'interno della memoria stabile (memoria che non si può danneggiare, astrazione).

Per facilitare il recupero delle informazioni il DBMS mantiene in memoria stabile anche il dump del DB, ovvero una copia completa di riserva dei dati e delle informazioni.

Gestione dei Guasti nei Database

La **gestione dei guasti** nei database è fondamentale per garantire che i dati rimangano **corretti e recuperabili** anche in caso di errori. Il **Recovery Manager** del DBMS si occupa di questa gestione attraverso strategie di backup, logging e ripristino.

1. Tipologie di Guasti

Esistono diversi tipi di guasti che possono compromettere il funzionamento del database:

1. Guasti Soft (Transitori)

- Errori di programma, crash del sistema operativo, perdita di tensione.
- Si perde il contenuto della **memoria principale (RAM)**, ma la **memoria secondaria (disco)** rimane intatta.
- **Soluzione:** Ripresa a **caldo** (warm restart).

2. Guasti Hard (Persistenti)

- Rottura del disco, danneggiamento dei file del database.
- Anche la **memoria secondaria** viene persa.
- **Soluzione:** Ripresa a **freddo** (cold restart) con **backup e log**.

2. Strategie di Recupero

Il **Recovery Manager** utilizza tre tecniche principali per gestire i guasti:

A. Logging (Write-Ahead Logging - WAL)

Il **log** è un file speciale che registra tutte le operazioni effettuate sulle transazioni **prima** che vengano scritte nel database.

Regola WAL: il log deve essere scritto **prima** che le modifiche reali vengano applicate al database.

♦ Struttura del log:

- **UNDO log** (annullamento): registra lo stato precedente di un dato (per ripristinare in caso di rollback).
- **REDO log** (ripetizione): registra lo stato aggiornato di un dato (per rifare operazioni incomplete dopo un crash).
- **Checkpoint:** un **punto di salvataggio** periodico per ridurre il tempo di ripristino.

Se il sistema si blocca **prima** del commit, il database deve usare **UNDO**.

Se si blocca **dopo** il commit, deve usare **REDO**.

B. Ripresa a Caldo (Warm Restart)

Usata nei **guasti soft**, quando la memoria principale è persa, ma il disco è integro.

♦ Fasi della ripresa a caldo:

1. **Individuazione dell'ultimo checkpoint nel log.**
2. **Costruzione degli insiemi:**
 - **UNDO** (transazioni non completate).
 - **REDO** (transazioni completate ma non scritte sul disco).
3. **Scansione a ritroso del log per annullare operazioni UNDO.**

4. **Scansione in avanti per rieseguire le operazioni REDO.**

C. Ripresa a Freddo (Cold Restart)

Necessaria nei **guasti hard**, quando anche la memoria secondaria è danneggiata.

- ◆ **Fasi della ripresa a freddo:**

1. **Recupero del database da un backup.**
2. **Riapplicazione del log per ripristinare le transazioni completate.**
3. **Esecuzione delle operazioni REDO registrate nel log.**

Gestione della Concorrenza e della Consistenza nei Database

Nei sistemi moderni, centinaia di transazioni possono essere eseguite contemporaneamente. Senza una gestione adeguata della **concorrenza**, potrebbero verificarsi **anomalie** che compromettono la **consistenza** del database.

1. Problemi della Concorrenza

Se più transazioni operano sugli stessi dati senza un adeguato controllo, possono verificarsi errori:

- ◆ **Perdita di Aggiornamenti (Lost Update)**

Si verifica quando due transazioni aggiornano la stessa variabile **senza coordinamento**, e una delle modifiche viene persa.

Soluzione: Locking o timestamp ordering.

- ◆ **Lecture Inconsistenti (Non-Repeatable Read)**

Si verifica quando una transazione legge un valore due volte e nel frattempo un'altra transazione lo modifica.

Soluzione: Livelli di isolamento come **Repeatable Read**.

◆ **Lecture Sporche (Dirty Read)**

Si verifica quando una transazione legge dati modificati da un'altra transazione **non ancora confermata**, che potrebbe poi essere annullata.

Soluzione: **Read Committed** o livelli di isolamento superiori.

◆ **Anomalie Phantom (Phantom Reads)**

Si verifica quando una transazione esegue una query e un'altra transazione modifica il set di risultati nel frattempo.

Soluzione: **Serializable Isolation**.

2. Strategie per la Gestione della Concorrenza

Per evitare queste anomalie, il **DBMS** utilizza **scheduler** e **protocolli di controllo della concorrenza**.

◆ **Scheduler e Serializzabilità**

Lo **scheduler** ordina l'esecuzione delle transazioni in modo che producano un risultato **equivalente** a quello di un'esecuzione **seriale** (cioè, una alla volta).

- **Scheduler Seriali:** le transazioni vengono eseguite una alla volta (sicuro ma inefficiente).
- **Scheduler Serializzabili:** consentono esecuzione concorrente ma garantiscono lo stesso risultato di uno seriale.

Scheduler View-Serializzabili (VSR)

La **view-serializzabilità** è un concetto che permette di verificare se uno **scheduler concorrente** è equivalente a uno **scheduler seriale**, basandosi sulle operazioni di lettura e scrittura eseguite sulle variabili del database.

1. Concetti Chiave della View-Serializzabilità

◆ Relazione "Legge Da" (Read-From Relation)

- Se in uno **scheduler S**, una transazione **T1** esegue una scrittura $w_1(x)$, e successivamente un'altra transazione **T2** esegue una lettura $r_2(x)$, allora si dice che **T2 "legge da" T1**, se tra le due operazioni non ci sono altre scritture $w_k(x)$.
- In altre parole, **l'ultimo valore scritto prima della lettura deve essere quello letto**.

Esempio:

$w_1(X) \rightarrow r_2(X)$

- La transazione **T2 legge il valore di X scritto da T1**.
- Se in mezzo non ci sono altre $w(x)$, la relazione **"legge da"** è valida.

◆ Scrittura Finale (Final Write)

- Una scrittura $w_i(x)$ è una **scrittura finale** se è l'**ultima scrittura su X** nello scheduler.
- **L'ultima scrittura determina il valore definitivo che rimane nel database**.

Esempio di Scrittura Finale:

$w_1(X), w_2(X), w_3(X)$

- Qui, $w_3(X)$ è l'ultima scrittura su X → **è la scrittura finale**.

◆ Equivalenza tra Scheduler

Due scheduler sono **view-equivalenti** se rispettano queste condizioni:

1. **Stessa relazione "Legge Da"** → Se una transazione legge un valore da un'altra transazione in uno scheduler, deve farlo anche nell'altro.
2. **Stessa Scrittura Finale** → L'ultima scrittura di ogni variabile deve essere la stessa in entrambi gli scheduler.
3. **Le transazioni devono leggere gli stessi valori in entrambi gli scheduler**.

Se uno scheduler concorrente è **view-equivalente** a uno **scheduler seriale**, allora è detto **view-serializzabile**.

Conflict-Serializzabilità

La **conflict-serializzabilità** è un metodo per determinare se uno **scheduler concorrente** è equivalente a un **scheduler seriale**, basandosi sull'ordine delle operazioni che possono entrare in **conflitto**.

1. Cosa significa Conflitto tra Operazioni?

Due operazioni $op1$ e $op2$ di due diverse transazioni **confliggono** se:

1. **Agiscono sulla stessa variabile (es. x).**
2. **Almeno una delle due è una scrittura.**

Tipi di conflitto:

- **Lettura-Scrittura** ($r1(x) \rightarrow w2(x)$): una transazione legge un dato che un'altra sta modificando.
- **Scrittura-Lettura** ($w1(x) \rightarrow r2(x)$): una transazione modifica un dato prima che un'altra lo legga.
- **Scrittura-Scrittura** ($w1(x) \rightarrow w2(x)$): due transazioni scrivono sulla stessa variabile in ordine diverso.

Se l'ordine di queste operazioni **cambia** tra due scheduler, il **risultato del database può cambiare**.

💡 **Regola d'oro:** Se l'ordine delle operazioni in conflitto cambia tra due scheduler, allora i due scheduler **NON** sono equivalenti.

2. Come verificare la Conflict-Serializzabilità?

La **conflict-serializzabilità** può essere verificata in modo **efficiente** attraverso il **grafo di precedenza**.

Passaggi per costruire il grafo:

1. Creare un nodo per ogni transazione.
2. Aggiungere un arco diretto $T_1 \rightarrow T_2$ se esiste un conflitto tra T_1 e T_2 , e T_1 esegue l'operazione prima di T_2 .
3. Verificare se il grafo contiene cicli:
 - Se il grafo è aciclico, allora lo scheduler è **conflict-serializzabile**.
 - Se il grafo ha un ciclo, lo scheduler **non è conflict-serializzabile**.

2. View-Serializzabilità vs Conflict-Serializzabilità

♦ View-Serializzabilità

- Basata sulla logica delle **letture e scritture equivalenti**.
- Permette **più schemi di esecuzione** rispetto alla conflict-serializzabilità.
- Più **permissiva**, ma **difficile da verificare** (NP-completo).

♦ Conflict-Serializzabilità

- Basata sulle **dipendenze di conflitto** (letture e scritture simultanee sullo stesso dato).
- Più **facile da verificare** (basta un grafo di precedenza).

Differenza principale:

La view-serializzabilità considera l'**effetto logico** di un'operazione (chi legge cosa), mentre la conflict-serializzabilità si basa solo sugli **ordini delle operazioni**.

Gestione della Concorrenza nei DBMS: Locking e Problemi Correlati

1. Perché la Conflict-Serializzabilità Non Basta?

Il concetto di **conflict-serializzabilità** stabilisce che uno scheduler è valido se l'ordine delle operazioni delle transazioni rispetta determinati vincoli di conflitto. Tuttavia, questo modello **funziona solo se tutte le transazioni terminano con un commit**. Nella realtà, le transazioni possono essere annullate

(ROLLBACK), quindi serve un sistema più flessibile per garantire la correttezza dei dati. Per questo motivo, i DBMS implementano protocolli di **locking**.

2. Come Funzionano i Lock?

Un **lock** è un meccanismo che impedisce a più transazioni di modificare o leggere lo stesso dato contemporaneamente in modo non controllato. Quando una transazione vuole accedere a un dato, deve prima **richiedere un lock**. Se il lock è disponibile, la transazione può procedere. Al termine dell'operazione, il lock viene **rilasciato**, permettendo ad altre transazioni di accedere al dato.

3. Two-Phase Locking (2PL)

Uno dei protocolli più usati per la gestione dei lock è il **Two-Phase Locking (2PL)**, che divide l'uso dei lock in due fasi:

1. Fase di Crescita

- La transazione può acquisire nuovi lock, ma non può rilasciarne.

2. Fase di Rilascio

- Una volta che una transazione rilascia un lock, non può più acquisirne altri.

Questo metodo garantisce che le transazioni vengano eseguite in modo **serializzabile**, evitando letture sporche o perdite di aggiornamenti.

Strict Two-Phase Locking (Strict 2PL)

Una variante più sicura, in cui:

- **I lock vengono mantenuti fino al commit o rollback.**
- **Evita letture inconsistenti anche in caso di abort.**

Strict 2PL offre un livello di isolamento più elevato, ma può aumentare i tempi di attesa delle transazioni concorrenti.

4. Problemi dei Sistemi Basati su Lock

Livelock

Si verifica quando una transazione rimane in attesa indefinitamente perché altre transazioni continuano a ottenere il lock con priorità maggiore.

Soluzione:

- Implementare un meccanismo di **coda di priorità**, in cui più lunga è l'attesa, maggiore è la priorità della transazione in attesa.

Deadlock

Si verifica quando due o più transazioni si bloccano a vicenda, ognuna in attesa di un lock detenuto dall'altra.

Soluzioni per il Deadlock

1. **Timeout:** se una transazione rimane bloccata troppo a lungo, viene automaticamente annullata.
2. **Prevenzione:** le transazioni acquisiscono i lock in un ordine fisso (es. sempre X prima di Y).
3. **Rilevamento e Risoluzione:** il DBMS monitora le transazioni e, se rileva un ciclo di attesa, forza l'aborto di una transazione.

Normalizzazione nei Database

La **normalizzazione** è un processo che organizza i dati all'interno di un database per **eliminare ridondanze e garantire l'integrità dei dati**.

Lo scopo è ridurre la **ridondanza** e migliorare la **consistenza**, garantendo che i dati siano strutturati in modo efficiente.

Obiettivi della Normalizzazione

- ✓ Eliminare la ridondanza
- ✓ Evitare anomalie di inserimento, aggiornamento e cancellazione

✓ Migliorare l'integrità dei dati

Le forme normali (NF) sono regole che definiscono i livelli di normalizzazione. Le principali sono:

1. **Prima Forma Normale (1NF)**
2. **Seconda Forma Normale (2NF)**
3. **Terza Forma Normale (3NF)**
4. **Forma Normale di Boyce-Codd (BCNF)**

1. Prima Forma Normale (1NF) – "Niente attributi multipli"

Definizione:

Una relazione è in **1NF** se **tutti gli attributi contengono solo valori atomici** (cioè **senza valori multipli o liste**).

● Problema della 1NF:

- Colonne con più valori in una singola cella creano difficoltà nell'interrogazione dei dati.
- Si verificano **anomalie di aggiornamento, inserimento e cancellazione**.

Esempio di una tabella NON in 1NF:

ID_Studente	Nome	Materie
1	Marco	Matematica, Fisica
2	Luca	Informatica, Fisica
3	Anna	Chimica

- Il campo **Materie** contiene **più valori** → **Non è 1NF!**

Soluzione per ottenere la 1NF:

- **Separare i valori multipli in più righe.**

ID_Studente	Nome	Materia
1	Marco	Matematica
1	Marco	Fisica
2	Luca	Informatica
2	Luca	Fisica
3	Anna	Chimica

Ora ogni cella contiene un solo valore → **1NF raggiunta!** 

2. Seconda Forma Normale (2NF) – "Eliminare la dipendenza parziale dalla chiave"

Definizione:

Una relazione è in **2NF** se:

1. È in **1NF**
2. **Tutti gli attributi non chiave dipendono completamente dalla chiave primaria** (e non solo da una parte di essa).

Problema della 2NF:

- Se una chiave primaria è composta da **più attributi**, alcuni campi possono dipendere solo da una parte della chiave → **dipendenza parziale**.
- Ciò genera **ridondanza** e **anomalie di aggiornamento**.

Esempio di una tabella NON in 2NF:

ID_Studente	Materia	Nome_Studente	Professore
1	Matematica	Marco	Rossi
1	Fisica	Marco	Bianchi
2	Informatica	Luca	Verdi

ID_Studente	Materia	Nome_Studente	Professore
2	Fisica	Luca	Bianchi

Chiave primaria: (*ID_Studente, Materia*)

● Problema:

- **Nome_Studente** dipende solo da **ID_Studente** → **dipendenza parziale!**
- Se cambiamo il nome dello studente, dobbiamo modificarlo in più righe.

Soluzione per ottenere la 2NF:

Separiamo la tabella in due:

✓ Tabella Studenti

ID_Studente	Nome
1	Marco
2	Luca

✓ Tabella Materie

ID_Studente	Materia	Professore
1	Matematica	Rossi
1	Fisica	Bianchi
2	Informatica	Verdi
2	Fisica	Bianchi

Ora non ci sono più dipendenze parziali → **2NF raggiunta!** ✓

3. Terza Forma Normale (3NF) – "Eliminare la dipendenza transitiva"

Definizione:

Una relazione è in **3NF** se:

1. È in **2NF**
2. **Gli attributi non chiave dipendono solo dalla chiave primaria** (e non da altri attributi non chiave).

● Problema della 3NF:

- Se un attributo non chiave dipende da un altro attributo non chiave → **dipendenza transitiva**.

Esempio di una tabella NON in 3NF:

ID_Studente	Materia	ID_Professore	Nome_Professore
1	Matematica	101	Rossi
1	Fisica	102	Bianchi
2	Informatica	103	Verdi
2	Fisica	102	Bianchi

● Problema:

- **Nome_Professore** dipende da **ID_Professore**, non direttamente da **ID_Studente** → **dipendenza transitiva!**

Soluzione per ottenere la 3NF:

✓ Tabella Studenti-Materie

ID_Studente	Materia	ID_Professore
1	Matematica	101
1	Fisica	102
2	Informatica	103

ID_Studente	Materia	ID_Professore
2	Fisica	102

✓ Tabella Professori

ID_Professore	Nome_Professore
101	Rossi
102	Bianchi
103	Verdi

Ora non ci sono più dipendenze transitive → 3NF raggiunta! ✓

4. Forma Normale di Boyce-Codd (BCNF) – "Versione più rigorosa della 3NF"

Definizione:

Una relazione è in **BCNF** se per ogni dipendenza funzionale $A \rightarrow B$, A è una superchiave.

- **BCNF** è una versione più restrittiva della 3NF.
- Si applica quando una **chiave primaria** non è sufficiente a determinare tutti gli altri attributi.

💡 Se una tabella è in BCNF, è anche in 3NF.

BCNF (Boyce-Codd Normal Form) – Approfondimento 🚀

La **Forma Normale di Boyce-Codd (BCNF)** è una versione **più rigorosa della Terza Forma Normale (3NF)** e viene utilizzata per eliminare **ogni tipo di ridondanza** dovuta a **dipendenze funzionali**.

1. Definizione di BCNF

Una relazione è in **BCNF** se, per ogni **dipendenza funzionale** della forma:

$$A \rightarrow B$$

A è una **superchiave**, ovvero **un attributo o un insieme di attributi che determinano in modo univoco tutti gli altri attributi della tabella**.

Differenza tra 3NF e BCNF

Una relazione in **3NF** permette la presenza di **dipendenze funzionali in cui un attributo non chiave può dipendere da un altro attributo non chiave**, a patto che il secondo sia **determinato da una chiave candidata**.

BCNF invece **elimina anche queste dipendenze**, rendendo la struttura ancora più rigorosa.

2. Quando una relazione è in 3NF ma NON in BCNF?

Una tabella è in **3NF** ma non in **BCNF** quando:

1. È in **3NF** (quindi **senza dipendenze transitive**).
2. Ha una **dipendenza funzionale in cui il determinante NON è una superchiave**.

Es.

Dirigente, Progetto, Sede. Abbiamo progetto, sede superchiave ma dirigente -> sede. Non posso passare a BCNF ma 3NF soddisfatta

In questo caso la BCNF non è raggiungibile

3. Vantaggi e Svantaggi della BCNF

✅ Vantaggi

- **Elimina completamente la ridondanza** dei dati.
- **Migliora la coerenza dei dati**, evitando aggiornamenti incoerenti.
- **Previene anomalie di aggiornamento, inserimento e cancellazione**.

❌ Svantaggi

- La decomposizione può aumentare il numero di tabelle → **query più complesse e più join richiesti**.
- In alcuni casi, si può perdere **la dipendenza funzionale diretta tra attributi che erano nella stessa tabella**.

Dipendenza Funzionale nei Database 🔍

Le **dipendenze funzionali** sono un concetto fondamentale nei database relazionali e sono utilizzate per capire come gli attributi di una relazione (tabella) sono collegati tra loro.

1. Definizione di Dipendenza Funzionale

Si dice che un insieme di attributi **X determina un insieme di attributi Y** (notato come $X \rightarrow Y$) se, per ogni coppia di tuple (righe) in una tabella, se le due tuple hanno lo stesso valore per **X**, allora devono avere lo stesso valore per **Y**.

Informalmente:

E' un insieme di uno o più attributi i cui valori determinano i valori di un altro insieme di attributi (o uno) in modo univoco.

Esempio di dipendenza funzionale

Consideriamo questa tabella di studenti:

Matricola	Nome	Corso	Professore
1	Anna	Matematica	Rossi
2	Bob	Fisica	Bianchi
3	Carla	Matematica	Rossi

- **Matricola \rightarrow Nome** ✅

Ogni matricola identifica **univocamente** il nome dello studente.

Se due tuple hanno la stessa matricola, devono avere lo stesso nome.

- **Corso \rightarrow Professore** ✅

Ogni corso è tenuto da **un solo professore**.

Se due studenti frequentano lo stesso corso, avranno lo stesso professore.

Le dipendenze funzionali sono **fondamentali** per capire la struttura di un database e per ottimizzarlo eliminando ridondanze.

2. Tipi di Dipendenze Funzionali

Esistono tre tipi principali di dipendenze funzionali:

A. Dipendenza Funzionale Completa

Una dipendenza $X \rightarrow Y$ è **completa** se **tutti gli attributi di X sono necessari** per determinare Y. Se rimuoviamo un attributo da X, la dipendenza non è più valida.

B. Dipendenza Funzionale Parziale

Una dipendenza $X \rightarrow Y$ è **parziale** se esiste **un sottoinsieme di X** che è già sufficiente per determinare Y.

Ciò significa che alcuni attributi di X **non sono necessari** per determinare Y.

● Problema delle dipendenze parziali:

Se esistono dipendenze parziali, significa che **abbiamo ridondanza** nel database. Questo può portare a problemi nelle operazioni di aggiornamento.

C. Dipendenza Funzionale Transitiva

Una dipendenza $X \rightarrow Z$ è **transitiva** se esistono attributi Y tali che:

- $X \rightarrow Y$ ✓
- $Y \rightarrow Z$ ✓
- Quindi $X \rightarrow Z$ è implicita.

● Problema delle dipendenze transitive:

Significa che un attributo **non dipende direttamente dalla chiave primaria**, ma tramite un altro attributo.

Questo può creare **ridondanze** e anomalie nei dati.

3. Perché sono importanti le Dipendenze Funzionali?

Le dipendenze funzionali sono utilizzate per:

- ✓ **Verificare la qualità della progettazione di un database.**
- ✓ **Identificare ridondanze e anomalie nei dati.**
- ✓ **Aiutare nella normalizzazione, che riduce i problemi di aggiornamento e inserimento.**

4. Come verificare una dipendenza funzionale?

Per verificare se una dipendenza $X \rightarrow Y$ è valida:

1. **Controlliamo tutte le tuple della tabella.**
2. **Se due tuple hanno lo stesso valore per X, devono avere lo stesso valore per Y.**
3. **Se troviamo anche un solo caso in cui questa regola non è rispettata, allora $X \rightarrow Y$ non è una dipendenza funzionale valida.**

Calcolo Relazionale nei Database

Il **calcolo relazionale** è un metodo formale per esprimere query nei database relazionali basato sulla **logica del primo ordine**. È un approccio alternativo all'**algebra relazionale** e permette di formulare query in modo **dichiarativo**, specificando **cosa vogliamo ottenere** piuttosto che **come ottenerlo**.

Esistono due principali varianti del calcolo relazionale:

1. **Calcolo relazionale sui tuple (TRC - Tuple Relational Calculus)**
2. **Calcolo relazionale sui domini (DRC - Domain Relational Calculus)**

1. Calcolo Relazionale sui Tuple (TRC)

Il **calcolo relazionale sui tuple (TRC)** utilizza variabili che rappresentano **intere tuple** di una relazione. Le query in TRC specificano le proprietà che le tuple devono soddisfare.

Sintassi Generale:

$$\{t \mid \varphi(t)\}$$

Dove:

- **t** è una variabile che rappresenta una **tupla**.
- **$\varphi(t)$** è una formula logica che specifica le **condizioni** che t deve soddisfare.

✚ Esempio di TRC:

Consideriamo una relazione **Dipendenti(DID, Nome, Salario)**.

Se vogliamo trovare i nomi e i salari dei dipendenti che guadagnano più di **50000**, possiamo scrivere:

$$\{t.Nome, t.Salario \mid t \in Dipendenti \wedge t.Salario > 50000\}$$

✚ Interpretazione:

- La variabile **t** rappresenta una tupla della tabella **Dipendenti**.
- La condizione **t.Salario > 50000** filtra solo le tuple con salario superiore a 50.000.
- Il risultato conterrà solo i campi **Nome** e **Salario** delle tuple selezionate.

2. Calcolo Relazionale sui Domini (DRC)

Il **calcolo relazionale sui domini (DRC)** invece utilizza variabili che rappresentano **singoli valori** piuttosto che intere tuple. Le query in DRC specificano le condizioni che i **valori di dominio** devono soddisfare.

Sintassi Generale:

$$\{(x_1, x_2, \dots, x_n) \mid \varphi(x_1, x_2, \dots, x_n)\}$$

Dove:

- **x_1, x_2, \dots, x_n** sono variabili che rappresentano **valori singoli** (non tuple intere).
- **$\varphi(x_1, x_2, \dots, x_n)$** è una formula logica che specifica le condizioni che questi valori devono soddisfare.

📌 Esempio di DRC:

Usiamo la stessa relazione **Dipendenti(DID, Nome, Salario)**.

Per trovare i nomi e i salari dei dipendenti che guadagnano più di **50000**, scriviamo:

$$\{(N, S) \mid \exists DID(Dipendenti(DID, N, S) \wedge S > 50000)\}$$

📌 Interpretazione:

- **(N, S)** rappresenta le coppie di **Nome** e **Salario** da restituire.
- $\exists DID$ significa che **esiste un identificativo di dipendente (DID)** per cui la condizione è vera.
- **Dipendenti(DID, N, S)** specifica che **N** e **S** provengono dalla tabella **Dipendenti**.
- **S > 50000** impone il vincolo sul salario.

3. Differenze tra TRC e DRC 🏴‍☠️

Caratteristica	Calcolo sui Tuple (TRC)	Calcolo sui Domini (DRC)
Variabili	Rappresentano tuple intere	Rappresentano valori singoli
Sintassi	Usa variabili di tuple e condizioni su tuple	Usa variabili di dominio e condizioni su valori

4. Sicurezza delle Query nel Calcolo Relazionale

Non tutte le query scritte con il calcolo relazionale sono **sicure**.

Una query è **sicura** se:

1. **Restituisce un numero finito di risultati** (cioè non produce output infiniti).
2. **Utilizza solo valori esistenti nella relazione** (evitando risultati non definiti).

🔴 Esempio di Query Non Sicura:

$$\{t \mid \neg(t \in Dipendenti)\}$$

Questa query selezionerebbe **tutte le tuple che non sono in Dipendenti**, il che è **impossibile** da calcolare in un database finito!

5. Potenza Espressiva: Calcolo Relazionale vs Algebra Relazionale

♦ Il calcolo relazionale e l'algebra relazionale sono equivalenti in termini di **potenza espressiva**.

Ciò significa che qualsiasi query che possiamo esprimere con uno, può essere espressa anche con l'altro.

Principale differenza:

- **L'algebra relazionale è procedurale** → dice **come** ottenere i dati.
- **Il calcolo relazionale è dichiarativo** → dice **quali dati vogliamo**, senza specificare il metodo per ottenerli.

Indici nei Database

Gli **indici** nei database sono **strutture dati** che permettono di velocizzare le ricerche su una tabella. Funzionano come un **indice di un libro**: invece di scorrere tutte le pagine, possiamo trovare rapidamente la sezione desiderata.

1. Perché servono gli indici?

Senza un indice, il database per trovare un record deve scorrere **tutte le righe della tabella** (**scansione completa** o *full table scan*), il che può essere molto lento se la tabella ha milioni di righe.

✦ **Esempio senza indice:**

Se cerchiamo un cliente per **ID**, il database deve controllare **ogni riga una per una** fino a trovare il risultato.

✦ **Esempio con indice:**

Se abbiamo un indice sull'**ID del cliente**, il database può trovare subito il record senza scansionare tutta la tabella.

2. Come funzionano gli indici?

Gli indici funzionano creando una **struttura dati separata** che memorizza i valori di una colonna (o più colonne) insieme a un **puntatore alla posizione del record** nella tabella.

Gli indici più comuni sono basati su **B-Tree**, che permette di eseguire ricerche molto efficienti in **tempo logaritmico $O(\log n)$** .

Esempio: Indice su una tabella Clienti

Tabella **Clienti**:

ID	Nome	Città
101	Marco	Roma
102	Anna	Milano
103	Luca	Napoli

Se creiamo un **indice su ID**, il database costruisce una struttura simile a questa:

```
Indice (B-Tree)
-----
101 → Posizione riga 1
102 → Posizione riga 2
103 → Posizione riga 3
```

Quando cerchiamo **ID=102**, il database usa l'indice per trovare subito la posizione del record, invece di scansionare tutte le righe.

3. Tipi di Indici nei Database

Esistono diversi tipi di indici, ognuno con un uso specifico:

A. Indice Primario (Primary Index)

- Creato **automaticamente** sulla **chiave primaria** di una tabella.
- Ogni tabella può avere **un solo** indice primario.

- **Es:** PRIMARY KEY (ID)

B. Indice Unico (Unique Index)

- Impedisce la duplicazione dei valori in una colonna.
- Simile alla chiave primaria, ma **può esistere più di un indice unico per tabella**.
- **Es:** CREATE UNIQUE INDEX idx_nome ON Clienti(Nome);

C. Indice Composito (Composite Index)

- Basato su **più colonne** invece di una sola.
- Utile per query che filtrano su più campi contemporaneamente.
- **Es:** CREATE INDEX idx_cliente ON Clienti(Nome, Città);

4. Quando usare (o non usare) gli indici?

◆ Usare gli indici quando:



- ✓ La tabella è **grande** e si fanno molte ricerche.
- ✓ Si effettuano frequentemente **filtri e ordinamenti su una colonna**.
- ✓ Le query coinvolgono **join tra tabelle**.

● Non usare troppi indici quando:

- ✗ La tabella è **piccola** (scansionare poche righe è veloce anche senza indice).
- ✗ Si fanno **molti INSERT/UPDATE/DELETE** (gli indici rallentano le operazioni di scrittura perché devono essere aggiornati).
- ✗ Le query restituiscono **troppi risultati** (un indice su una colonna con pochi valori distinti non è utile).

6. Vantaggi e Svantaggi degli Indici

Vantaggi ✓	Svantaggi ✗
Migliorano la velocità delle query SELECT	Rallentano INSERT, UPDATE, DELETE

Vantaggi 	Svantaggi 
Rendono più veloci le ricerche e i filtri	Occupano spazio aggiuntivo su disco
Ottimizzano l'esecuzione delle JOIN	Troppi indici possono confondere l' Ottimizzatore di Query

Implementazione fisica

L'**implementazione degli indici** è un aspetto fondamentale per ottimizzare l'accesso ai dati nei database. Esaminiamo le quattro principali tecniche di indicizzazione: **B-Tree, Heap, Hash e ISAM (Indexed Sequential Access Method)**.

1. B-Tree e B+ Tree

Il **B-Tree** è una struttura di dati ad albero bilanciato utilizzata nei database per garantire accesso efficiente ai dati.

Struttura del B-Tree

- Ogni nodo ha **fino a P figli** e **P-1 chiavi** ordinate.
- Le chiavi determinano il percorso da seguire per trovare un valore.
- L'albero rimane **bilanciato**, quindi la profondità è sempre **logaritmica rispetto al numero di record**.

Caratteristiche

- ✓ **Ricerca logaritmica**: $O(\log N)$ per trovare un valore.
- ✓ **Inserimenti ed eliminazioni efficienti**: il bilanciamento evita degrado delle prestazioni.
- ✓ **Ottimo per ricerche intervallo**: grazie alla struttura ordinata.

Operazioni

1. **Ricerca**: segue il percorso guidato dalle chiavi nei nodi fino a trovare il valore desiderato.

2. Inserimento:

- Se il nodo ha spazio, si aggiunge la chiave.
- Se il nodo è pieno, si divide (split) creando nuovi nodi.

3. Eliminazione:

- Se un nodo diventa troppo scarico, può essere unito (merge) con un nodo adiacente.

B+ Tree

Il **B+ Tree** è una variante del B-Tree dove:

- **Solo le foglie contengono i record.**
- **I nodi interni contengono solo chiavi e puntatori.**
- **Le foglie sono collegate tra loro**, permettendo scansioni sequenziali efficienti.

Vantaggi del B+ Tree rispetto al B-Tree

- ✓ **Migliore scansione sequenziale** grazie ai puntatori tra le foglie.
- ✓ **Struttura più compatta nei nodi interni** perché non contengono record.
- ✓ **Usato nella maggior parte dei DBMS** per query su intervalli.

2. Heap File (File Disordinato)

L'**heap file** è la struttura più semplice per memorizzare dati in un database.

Struttura

- I dati vengono **inseriti in ordine casuale** senza una struttura specifica.
- L'accesso ai dati richiede una **scansione completa** (full scan) a meno che non si utilizzino indici secondari.

Caratteristiche

- ✓ **Inserimento rapido**: i dati vengono scritti alla fine del file.
- ✗ **Ricerche lente**: senza indici, la ricerca richiede una scansione completa.
- ✓ **Buono per operazioni batch o dati con accesso casuale**.

Operazioni

1. **Ricerca:** scansiona tutti i record fino a trovare il valore desiderato.
2. **Inserimento:** aggiunge il record alla fine del file.
3. **Eliminazione:** il record viene contrassegnato come cancellato e lo spazio può essere riutilizzato.

Ottimizzazione

- **Uso di indici** per accelerare le ricerche.
- **Periodiche riorganizzazioni** per compattare lo spazio eliminando record cancellati.

3. Hash Index

Gli **indici hash** mappano le chiavi in indirizzi di memoria secondaria usando una **funzione hash**.

Struttura

- La funzione hash trasforma un valore chiave in un indirizzo di blocco.
- Se due chiavi hanno lo stesso hash (**collisione**), il DBMS usa tecniche di gestione come **liste di overflow** o **riposizionamento lineare**.

Caratteristiche

- ✓ **Accesso diretto molto rapido** per ricerche basate su uguaglianza (*WHERE id = X*).
- ✗ **Non efficiente per ricerche su intervalli** (*WHERE id > X*).
- ✓ **Ottimo per carichi di lavoro con accesso diretto frequente**.

Operazioni

1. **Ricerca:**
 - Calcola l'hash della chiave.
 - Accede direttamente al blocco corrispondente.
2. **Inserimento:**
 - Usa la funzione hash per trovare un blocco libero.
 - Se c'è una collisione, usa liste di overflow o tecniche di risoluzione.
3. **Eliminazione:**

- Rimuove il record e gestisce eventuali liste di overflow.

Gestione delle collisioni

- **Indirizzamento aperto**: cerca la prossima posizione disponibile.
- **Liste di overflow**: usa un'area separata per i record che collidono.
- **Doppia funzione hash**: applica una seconda funzione hash in caso di collisione.

4. ISAM (Indexed Sequential Access Method)

L'**ISAM** è un metodo di indicizzazione basato su un **indice statico**.

Struttura

- I dati sono ordinati fisicamente in un file sequenziale.
- Esistono **tre livelli**:
 - i. **Indice primario** (sparse) che punta a blocchi di dati.
 - ii. **Indice secondario** (denso) che contiene una voce per ogni record.
 - iii. **File dati** contenente i record reali.

Caratteristiche

✓ **Ottimo per ricerche su intervalli grazie alla struttura ordinata.**

✗ **Statico**: non si adatta bene a molte modifiche, perché l'indice non si aggiorna dinamicamente.

✓ **Veloce per accesso sequenziale e scansione di range di dati.**

Operazioni

1. Ricerca:

- L'indice primario indirizza velocemente al blocco giusto.
- L'indice secondario trova il record esatto.

2. Inserimento:

- I nuovi record vanno in **aree di overflow**, mantenendo l'ordine.
- Periodicamente, i file vengono **ricalcolati** per eliminare le aree di overflow.

3. Eliminazione:

- I record cancellati rimangono finché il file non viene riorganizzato.

Ottimizzazione

- **ISAM Dinamico**: versioni moderne come **B-Tree** permettono aggiornamenti dinamici senza riorganizzazioni periodiche.
- **Bilanciamento del file** per ridurre le aree di overflow.

5. Confronto tra gli Indici

Tecnica	Vantaggi	Svantaggi	Caso d'uso ideale
B-Tree	Ricerca $O(\log N)$, aggiornamenti efficienti, gestione dinamica	Più costoso in memoria rispetto a ISAM	Accesso efficiente per ricerche e ordinamenti
B+ Tree	Scansione sequenziale veloce, bilanciato, ottimizzato per DBMS	Maggiore overhead rispetto a ISAM	Query su intervalli e grandi dataset
Heap	Inserimenti rapidi	Ricerche lente senza indici	Dati senza frequenti ricerche
Hash	Accesso diretto rapido, ottimo per chiavi primarie	Non funziona per ricerche su intervalli	Lookup diretto su chiavi univoche
ISAM	Buono per accesso sequenziale e ordinato	Non supporta aggiornamenti dinamici bene	Archivi con pochi aggiornamenti