

PROGETTAZIONE FISICA

In questa sezione si passa dalla teoria alla pratica. Si raccomanda vivamente la lettura di questo documento, poiché è di fondamentale importanza e strettamente aderente alla realtà.

Nei sistemi di gestione di basi di dati (DBMS) si verifica la **concorrenza**, ovvero più client possono connettersi contemporaneamente all'istanza del DBMS ed eseguire operazioni sui dati. Questa situazione può causare problemi di **inconsistenza** nel database, compromettendo l'integrità e la correttezza delle informazioni memorizzate.

Nei DBMS, la correttezza dei dati non dipende solo da singole operazioni, ma dall'interazione tra operazioni concorrenti. A differenza della programmazione tradizionale, dove il codice agisce direttamente sui dati, nei DBMS scriviamo query senza controllare l'esecuzione a basso livello. La gestione della concorrenza è affidata al DBMS, purché gli segnaliamo le aree critiche e i possibili problemi sui dati. Grazie a strategie come l'isolamento delle **transazioni** e l'uso di **lock**, il DBMS garantisce la coerenza e l'integrità del database anche in presenza di **accessi simultanei**.

TRANSAZIONI E STORED PROCEDURE.

Una **transazione** è un'unità atomica di elaborazione che esegue una sequenza di operazioni sul database. Segue il principio "**all or nothing**", ovvero o tutte le operazioni della transazione vengono eseguite con successo (**commit**) oppure, in caso di errore, vengono annullate (**rollback**) per mantenere la coerenza del database.

Le **transazioni** vengono incorporate all'interno delle **stored procedure (backend del sistema, senza scomodare il client)**, che possono essere considerate come delle **API del DBMS**. Queste procedure consentono di invocare operazioni sul database **che sono persistenti nel DBMS** in modo strutturato ed efficiente, garantendo **atomicità, consistenza, isolamento e durabilità (ACID)** delle transazioni, oltre a ottimizzare le prestazioni riducendo il traffico tra applicazione e database.

Una store procedure viene scritta, compilata nel DBMS e vive all'interno di quest'ultimo: vive come execution planning precompilata.

Banalmente, si creano così:

Il comando **CREATE PROCEDURE** viene utilizzato per definire una **stored procedure** nel DBMS. Tuttavia, è importante notare che i DBMS non sono molto permissivi: di default, se si tenta di creare una procedura con un nome già esistente, l'operazione fallisce.

```
create procedure <proc-name> (p1, p2, ..., pn)
begin
    -- execution code
end;
```

Schema Riassuntivo Stored Procedure

```
CREATE
    [OR REPLACE]
    [DEFINER = { user | CURRENT_USER | role | CURRENT_ROLE }]
    PROCEDURE sp_name ([proc_parameter[,...]])
    [characteristic ...] routine_body

proc_parameter:
    [ IN | OUT | INOUT ] param_name type

type:
    Any valid MariaDB data type

characteristic:
    LANGUAGE SQL
    | [NOT] DETERMINISTIC
    | { CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA }
    | SQL SECURITY { DEFINER | INVOKER }
    | COMMENT 'string'

routine_body:
    Valid SQL procedure statement
```

Per ovviare si può utilizzare un comando opzionale:

create OR REPLACE procedure nome_procedura.

Viene creata una procedura, se già esistente viene rimpiazzata. Sostanzialmente sto dicendo al DBMS che voglio creare/aggiornare una qualche operazione.

L'etichetta **DEFINER** viene utilizzata per specificare l'utente o il ruolo con cui la procedura viene definita ed eseguita. Questo significa che:

- La procedura viene creata come se fosse stata definita da un determinato **utente**.
- È possibile associare la procedura a un **ruolo**, ossia un insieme di privilegi assegnati a più utenti, consentendo loro di eseguire la procedura con i permessi del ruolo definito.

In una **stored procedure** SQL, il numero di valori di ritorno non è limitato a uno: è possibile restituire **da 0 a N valori**.

I valori di ritorno vengono specificati fra i parametri.

In una **stored procedure** SQL, i parametri possono avere:

1. **Un nome** - Identificativo univoco all'interno della procedura.
2. **Un tipo di dato** - Definisce il formato del valore (es. INT, VARCHAR, DATE, ecc.).
3. **Una modalità di utilizzo**, che può essere:
 - **IN (input)** - Il valore viene passato alla procedura ma non può essere modificato.
 - **OUT (output)** - La procedura può assegnare un valore al parametro e restituirlo al chiamante.
 - **INOUT (input/output)** - Il parametro viene passato alla procedura, può essere modificato e il nuovo valore viene restituito.

All'interno di una **stored procedure**, è possibile specificare alcune **caratteristiche aggiuntive**, fornendo al DBMS informazioni utili per ottimizzarne l'esecuzione:

- Se una procedura viene dichiarata come **DETERMINISTIC**, significa che dato lo stesso insieme di parametri di input, il risultato sarà sempre identico. Questa informazione consente al DBMS di ottimizzare le prestazioni, ad esempio per il **Caching dei risultati**, ossia se la procedura viene chiamata frequentemente con gli stessi parametri, il DBMS può memorizzare il risultato in cache, evitando di ricalcolarlo ogni volta.
- Se una procedura viene dichiarata come **NO SQL**, significa che la procedura **non contiene alcuno statement SQL**.
- Se una procedura viene dichiarata come **READS SQL DATA**, significa che la procedura legge dati dal database ma non li modifica.
- Se una procedura viene dichiarata come **MODIFIES SQL DATA**, significa che la procedura **modifica i dati** all'interno del database.

Se nessuna di queste caratteristiche viene specificata, il DBMS **assume automaticamente** che la procedura:

1. **Non sia deterministica** (quindi può produrre risultati diversi con gli stessi input).
2. **Contenga istruzioni SQL.**
3. **Modifichi i dati** (comportamento predefinito).

Dichiarare esplicitamente queste caratteristiche aiuta il DBMS a ottimizzare l'esecuzione e a migliorare la gestione della concorrenza.

Esempio.

```
set autocommit = 0;
```

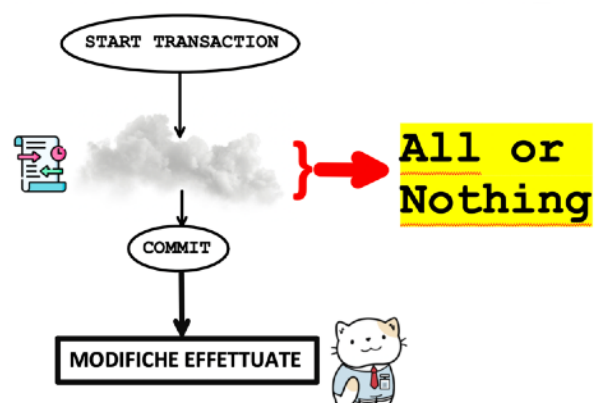
```
create procedure `procedure_name` (in input varchar(45), out output varchar(128))
begin
  declare exit handler for sqlexception
  begin
    rollback; -- rollback any changes made in the transaction
    resignal; -- raise again the sql exception to the caller
  end;

  set transaction isolation level repeatable read;
  start transaction;
  if ... then
    signal sqlstate '45001' set message_text = "Si è verificato un errore";
  end if;
  commit;
end
```

Questo è l'esempio di una stored procedure che utilizza una transazione.

Il **commit** rende permanenti le modifiche di uno statement SQL, ma l'**auto-commit** può essere problematico perché conferma automaticamente ogni operazione. Per garantire una semantica "**all or nothing**", si preferisce **disattivare l'auto-commit** e gestire le transazioni manualmente, eseguendo il commit solo dopo il successo di tutte (N) le operazioni. In caso di errore, si può effettuare un **rollback** per annullare le modifiche e mantenere la coerenza del database.

Il comando **START TRANSACTION** indica l'inizio di una transazione, garantendo che tutte le operazioni SQL all'interno della **stored procedure** vengano eseguite con la semantica "**all or nothing**". Questo significa che le modifiche verranno confermate solo con un **COMMIT**, oppure annullate con un **ROLLBACK** in caso di errore.



Se una **transazione viene annullata (abort)** a causa di un errore, **tutte** le operazioni eseguite all'interno di essa vengono annullate, anche se alcune erano già state completate con successo.

Ad esempio, se hai effettuato **10 inserimenti** e il decimo genera un errore:

- I **9 inserimenti precedenti non** vengono salvati.
- Il **database torna allo stato precedente alla transazione** grazie al **ROLLBACK**.

Questo garantisce l'**atomicità** della transazione, secondo il principio "**all or nothing**".

Quando si verifica un **errore** in un database SQL, è fondamentale **informare l'utente** dell'accaduto. Per farlo, i DBMS utilizzano **codici di errore**, che permettono di identificare il tipo di problema e gestirlo adeguatamente.

Come funzionano i codici di errore nei database SQL?

SQLSTATE - Un codice standard a 5 byte (i primi due per la classe) che indica la natura dell'errore.

- **3F000: invalid schema name**
- **30000: invalid SQL statement identifier**
- **23000: integrity constraint violation**
- **22012: data exception: division by zero**
- **45000: unhandled user-defined exception**

Per generare un codice di errore in SQL, si utilizza la parola chiave **SIGNAL**, che permette di **lanciare un'eccezione personalizzata** all'interno di una stored procedure.

-SIGNAL **SQLSTATE** '45000' -

Per generare un segnale che forzi la transizione dello stato di una query SQL verso un codice di errore, si utilizza il comando **SIGNAL**. Se il codice di errore generato è **diverso da uno stato corretto**, il DBMS interromperà l'esecuzione e restituirà un errore.

Esempio nell'esempio:

```
SIGNAL SQLSTATE '45000'  
SET MESSAGE_TEXT = 'Errore personalizzato';
```

Il comando **SIGNAL** consente di gestire errori sia all'interno che all'esterno di una transazione.

Nell'esempio precedente, il **SIGNAL** viene utilizzato durante una transazione: se si verifica un errore, il DBMS interrompe immediatamente l'esecuzione e notifica al client il codice di errore con il relativo messaggio. Tuttavia, la transazione rimane **in sospeso**, senza essere né confermata (**commit**) né annullata (**abort**), creando una situazione di **stallo**.

Per evitare il consumo eccessivo di risorse, il DBMS assegna a ogni transazione un **valore di timeout**. Se la transazione non viene completata entro questo tempo, viene automaticamente **annullata (abort)**, garantendo il corretto funzionamento del sistema e la disponibilità delle risorse.

Se voglio che, in caso di errore a livello applicativo, la transazione venga **immediatamente annullata** senza attendere il timeout, devo gestire il segnale di errore in modo da eseguire un **rollback** automatico.

Il concetto è simile a quello dei **gestori di segnali nei sistemi operativi**, dove a un determinato segnale corrisponde un **gestore** che definisce l'azione da intraprendere. Nei **DBMS**, possiamo definire un gestore di errore con **DECLARE HANDLER** e, in caso di errore, eseguire **subito un rollback**:

declare exit handler for sqlexception.

Questo è ciò che viene fatto nel codice.

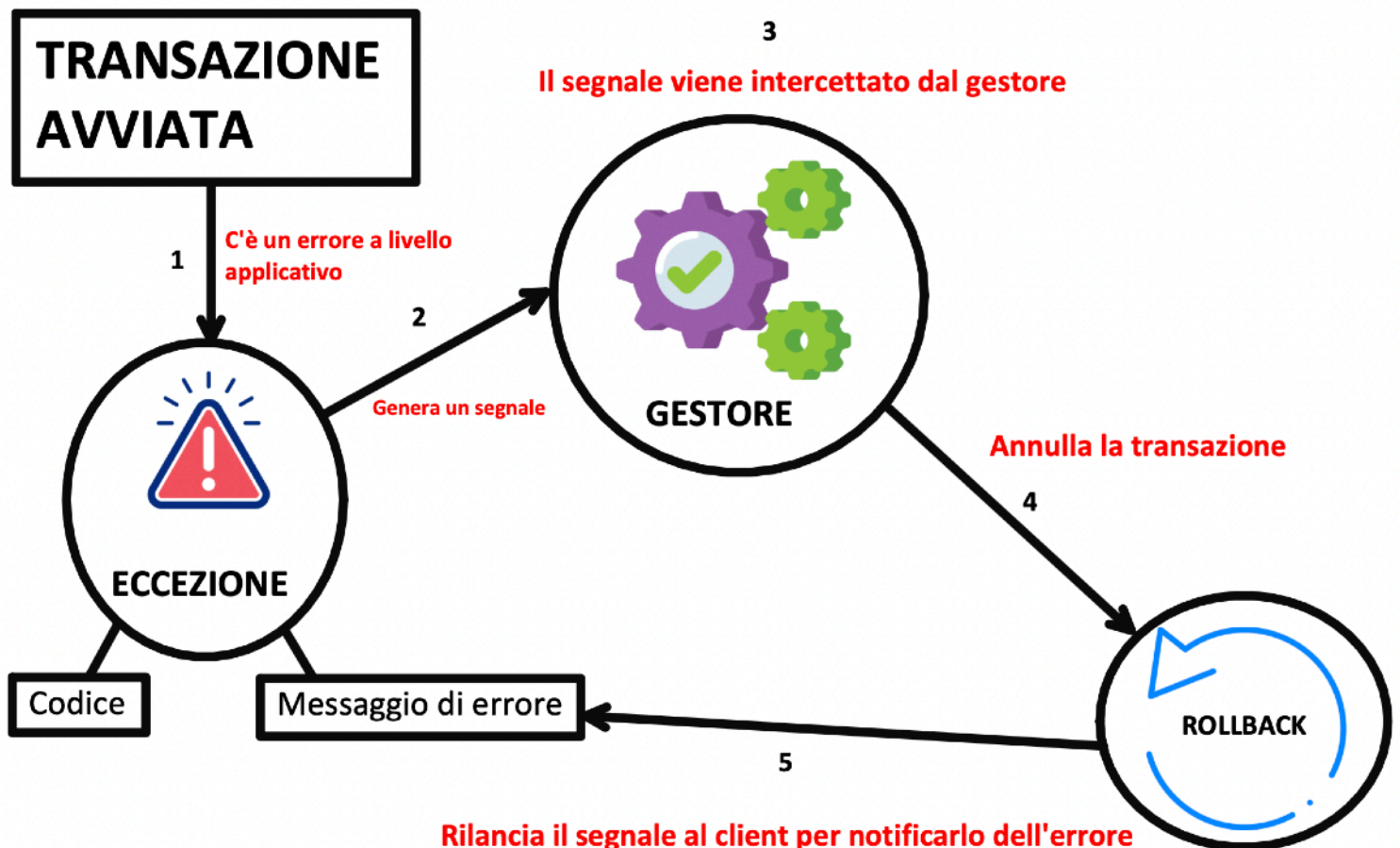


Se esiste, **l'handler prende il controllo** e anche questo è delimitato da un costrutto **BEGIN-END**.

All'interno del **gestore di errore**, eseguo un **ROLLBACK** per annullare immediatamente la transazione in caso di errore.

Successivamente, utilizzo **RESIGNAL** per informare l'utente, poiché **RESIGNAL** permette di rilanciare lo stesso segnale di errore che ha attivato il gestore.

Lo schema è il seguente:



Una volta che un segnale viene **catturato**, il **numero del segnale originale non esiste più**. In altre parole, il segnale viene **assorbito** dal gestore e **non viene notificato all'utente**, poiché il controllo passa interamente al gestore dell'errore.

Lo ripeto perché non è scontato: quando un segnale viene catturato, esso "muore". **RIP segnale**, da quel momento comanda il gestore.



Allo stesso modo, in un **DBMS**, se intercettiamo un errore con un gestore, quel codice **non arriverà mai all'utente** a meno che non lo rilanciamo esplicitamente con **RESIGNAL**.

L'ultima riga da spiegare è la seguente:

set transaction isolation level repeatable read.

(Caro DBMS... rileggerò più volte le stesse tuple [repeatable read] in questa transazione, vedi che devi fare).


Non essendoci primitive semaforiche, chiediamo al DBMS di gestire la sezione critica per noi.

Previene letture non ripetibili (*non-repeatable reads*):

Nessun'altra transazione può modificare i dati letti finché la transazione attuale non è terminata.

Tutte le letture fatte all'interno della transazione restano coerenti: Se esegui più volte la stessa query nella stessa transazione, vedrai sempre gli stessi risultati, anche se altri utenti modificano i dati nel frattempo.

Il DBMS ci fornisce una garanzia di **ISOLAMENTO DELLE TRANSAZIONI**.

 **Attenzione:** I livelli di isolamento sono un mondo complesso e spesso sottovalutato. Molti bug informatici derivano proprio da un'errata gestione dell'isolamento delle transazioni, perché alcuni sistemi non implementano correttamente certi livelli di isolamento, lasciando spazio a comportamenti imprevisti e **molti sviluppatori impostano i livelli di isolamento "a caso"**, senza comprendere le implicazioni su concorrenza e coerenza dei dati.



Risultato? Bug assurdi come transazioni che vedono dati inconsistenti, aggiornamenti che si sovrascrivono in modo inaspettato o letture sporche che mandano in tilt la logica applicativa.

Morale della storia: Se non scegli bene il livello di isolamento, potresti trovarti con un database che si comporta a cazzo di cane e genera errori difficili da debuggare.

Regola base della concorrenza nei DBMS:

Le transazioni che non accedono agli stessi dati non interferiscono tra loro e quindi possono essere eseguite in parallelo senza alcun problema.

Se due transazioni T1 e T2 operano su **righe diverse**, il DBMS le esegue in parallelo senza alcun blocco o attesa.

Quando si verifica il problema?

Se invece le due transazioni accedono agli stessi dati, potrebbe esserci un conflitto.

LIVELLI DI ISOLAMENTO TRANSAZIONALI.

Quando parliamo di livelli di isolamento, ci riferiamo a più transazioni concorrenti che accedono agli stessi dati. Quindi, gli attori principali sono due o più transazioni che insistono sullo stesso insieme di dati.

I **livelli di isolamento** in SQL si specificano con il comando:

SET TRANSACTION ISOLATION LEVEL <LIVELLO>;

Dove <LIVELLO> può essere uno dei seguenti:

SERIALIZABLE
REPEATABLE READ
READ COMMITTED
READ UNCOMMITTED

Partiamo da sotto.

READ UNCOMMITTED.

Permette di leggere dati non ancora committati: **NON BLOCCANTE**, ossia leggo il dato che trovo nel database, sia esso coerente o non coerente.

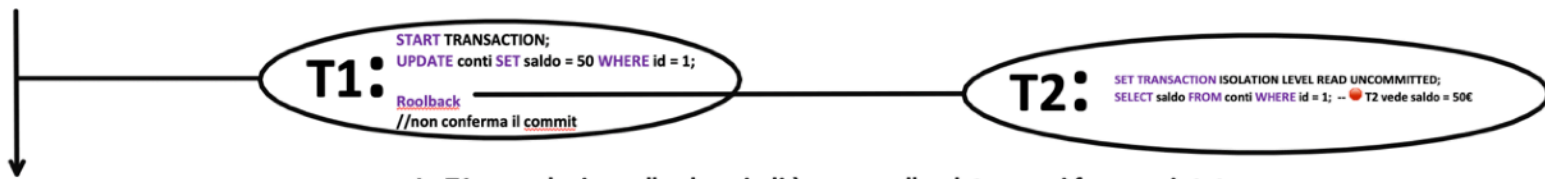
Potremmo leggere una scrittura generata da uno scrittore che non è ancora andato in commit: lettura non coerente.

Questo fenomeno si verifica poiché una transazione può leggere dati da una riga aggiornata da un'altra transazione che non ha ancora eseguito l'operazione di commit.

Supponiamo di avere la seguente tabella conti:

id	saldo
1	100€

Scenario con due transazioni concorrenti (T1 e T2).



La T1 va anche in rollback, quindi è come se l'update non ci fosse mai stato.
Però ora T2 vede 50, ossia la modifica di T1.

T2 ha già letto il valore 50€, che in realtà non esiste più!
T2 ha letto un dato che non è mai stato confermato → Dirty Read
Alla fine, il saldo vero resta **100€**, ma T2 potrebbe aver preso decisioni errate basandosi su un dato **sporco**.
Alla fine la tabella rimane uguale, ma T2 ha letto 50!

READ COMMITTED.

Una transazione può leggere solo dati che sono stati già confermati con **COMMIT**.

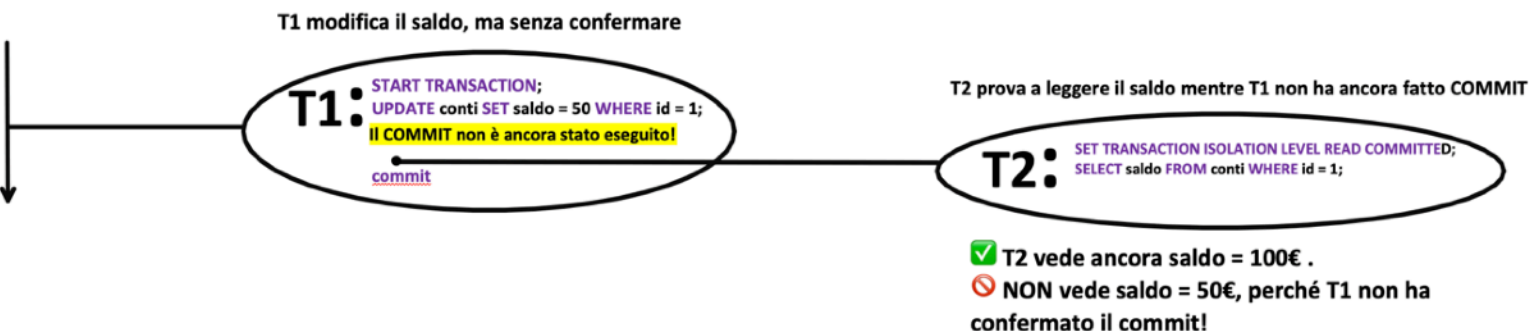
Evita le "dirty reads", ovvero letture di dati che potrebbero essere annullati con un rollback.

I lock associati ai dati che sono stati aggiornati in scrittura sono mantenuti fino alla fine della transazione, mentre i lock associati ai dati acceduti in lettura sono rilasciati alla fine della singola lettura.

Supponiamo di avere la seguente tabella conti:

id	saldo
1	100€

Scenario con due transazioni concorrenti (T1 e T2).



Dopo il commit, alla fine T2 esegue di nuovo la SELECT:

```
SELECT saldo FROM conti WHERE id = 1;
```



Ora T2 vede saldo = 50€, perché il COMMIT è stato eseguito.

Il dato aggiornato, nella read uncommitted potrebbe essere soggetto a rollback, in questo caso preveniamo questa situazione non permettendo la lettura del nuovo dato da un'altra transazione in concorrenza.

Il lock viene mantenuto fino al commit/rollback della transazione e poi rilasciato.

Tuttavia questo livello di isolamento non garantisce che una riletture dello stesso dato dia sempre lo stesso valore:

unrepeatable reads.

Se una transazione legge un valore e poi lo rilegge dopo un po', il valore potrebbe essere cambiato nel frattempo.



Commit finale da parte di t2.

Risultato: T2 ha letto due valori diversi per lo stesso dato nella stessa transazione → **Non-repeatable read.**

Evita le dirty reads (non si leggono dati che potrebbero essere annullati).

✓ **Migliore consistenza rispetto a READ UNCOMMITTED**, senza impattare troppo sulle prestazioni.

✗ **Non evita le "non-repeatable reads"** (i dati possono cambiare tra una lettura e l'altra nella stessa transazione).

REPEATABLE READ.

Il livello REPEATABLE READ garantisce che, durante una transazione, tutte le letture dello stesso dato restino sempre coerenti.

Se una transazione legge un dato, quel dato non può essere modificato da altre transazioni fino alla fine della transazione corrente.

Con questo livello di isolamento, vengono mantenuti i lock sia deidati acceduti in lettura sia in scrittura fino alla fine della transazione.

Immaginiamo di avere la seguente tabella conti:

id	saldo
1	100€

Scenario con due transazioni concorrenti (T1 e T2).



Ora T1 esegue una nuova SELECT (fuori dalla transazione precedente):

SELECT saldo **FROM** conti **WHERE** id = 1;

Ora vede saldo = 50€, perché la sua transazione precedente è finita.

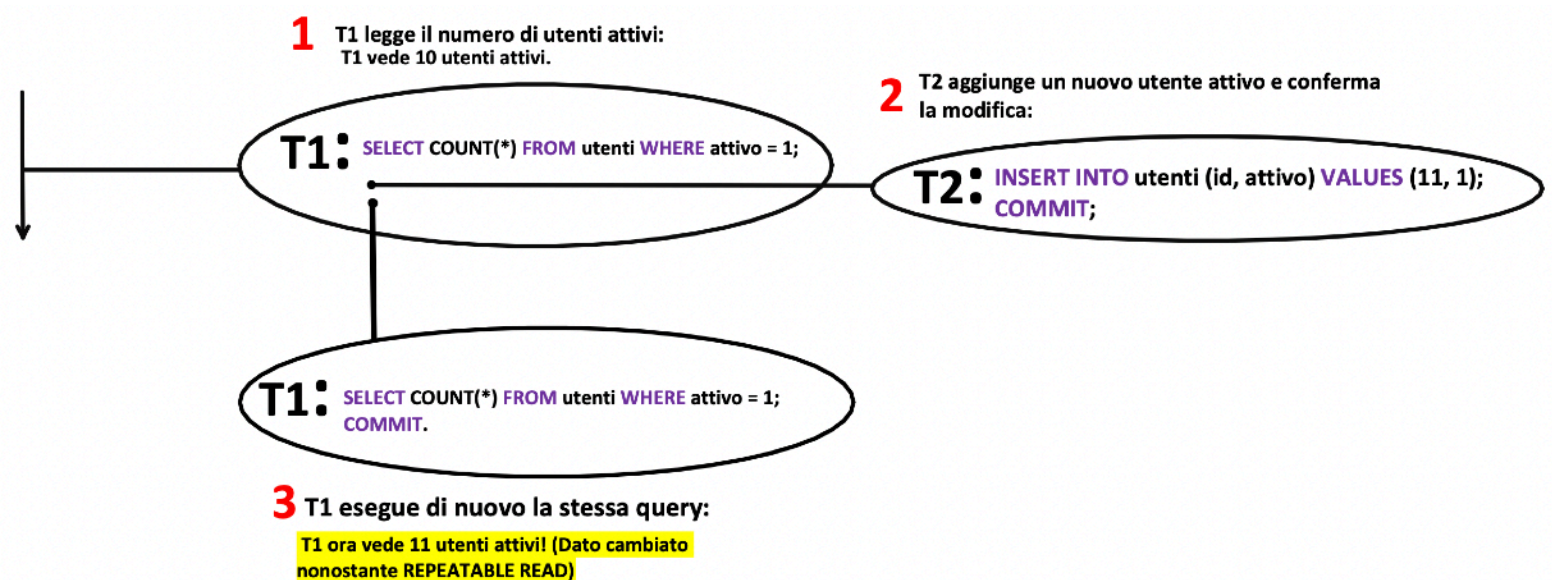
Lo statement di update e di commit di T2 deve essere serializzato dopo il commit di T1.

Non è possibile aggiornare la riga del database fin tanto che una transazione che ha letto quella riga non va in commit o non va in abort.

Uno dei problemi che possono uccidere il database è la lettura fantasma, che questo livello di isolamento può portare.

LETTURA FANTASMA.

Se una transazione legge un insieme di righe e poi un'altra transazione inserisce nuove righe, queste **potrebbero comparire nelle query successive** dentro la stessa transazione.



T2 poteva fare anche una update o una delete, sono sempre statement di aggiornamento, la situazione rimane uguale. Quando parliamo di **lettura/scrittura su una singola riga**, l'aggiornamento coinvolge **solo una riga specifica** e non modifica la struttura complessiva del dataset.

Caso particolare: operazioni su un range di dati.

Se un'operazione coinvolge **un range di dati** (ad esempio, tutte le righe che soddisfano una certa condizione), allora il DBMS deve:

1. Costruire un dataset temporaneo basato sulla query.
2. Applicare l'operazione (UPDATE, DELETE, INSERT) su quel dataset.
3. Aggiornare la tabella in base ai nuovi dati.

Ed è quello che è stato fatto ora.

Le **letture fantasma** (*phantom reads*) sono un problema proprio quando si lavora su **un range di dati**.

Quando una transazione esegue una query su un range
(WHERE saldo >= 100), il DBMS:

- 1 Cerca le righe che soddisfano la condizione → Scandisce indici o fa una scansione della tabella.
- 2 Costruisce un dataset temporaneo con i risultati.
- 3 Restituisce i dati alla transazione.
- 4 Se la transazione rilegge i dati più avanti, il DBMS ricrea il dataset sulla base dello stato attuale della tabella.

Problema: Se nel frattempo un'altra transazione **inserisce o cancella righe nel range**, il dataset **non sarà più lo stesso**, causando letture fantasma.

Perché il DBMS non blocca le nuove tuple automaticamente?

A differenza di un **lock su riga (row-level lock)**, che protegge una riga specifica, un **range di dati è dinamico** e non può essere "bloccato facilmente" con i normali meccanismi di lock.

I normali lock visti sin ora proteggono solo righe esistenti, quindi nuove INSERT o DELETE possono modificare il dataset tra due letture.

Se un INSERT o DELETE cambia i dati in un range di interesse per una transazione attiva, si verifica una lettura fantasma.

Anche un UPDATE può causare letture fantasma se modifica i dati in un range di interesse per una transazione attiva.

L'UPDATE, a differenza di INSERT e DELETE, non aggiunge o rimuove tuple, ma può farle entrare o uscire da un range di dati monitorato da una transazione attiva.

Supponi di avere:

id	saldo
1	100€
2	200€
3	300€

T1 avvia una transazione e legge tutte le righe con saldo ≥ 200€:

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;  
START TRANSACTION;  
SELECT * FROM conti WHERE saldo >= 200;
```

id	saldo	
1	100€	
2	200€	
3	300€	

T1 vede 2 righe (`id = 2, 3`).

T2 modifica una riga che prima non apparteneva al range di T1 (può farlo perché il lock è libero):

```
UPDATE conti SET saldo = 250 WHERE id = 1;  
COMMIT;
```

Ora
il record con `id = 1` **entra nel range** (con il lock) di T1 (`saldo >= 200`)!

T1 rilancia la stessa `SELECT` all'interno della sua transazione:

```
SELECT * FROM conti WHERE saldo >= 200;
```

Ora T1 vede 3 righe (`id = 1, 2, 3`)!

Verificata una **lettura fantasma**, perché la transazione ha visto cambiare il dataset tra due letture.

Problema: T1 ha letto inizialmente **solo 2 righe**, ma dopo la modifica di T2 ne vede **3**, anche se non ha mai fatto un **COMMIT**!

In questo contesto, il lock viene applicato a livello di riga (*row-level locking*). Il lock riguarda esclusivamente le righe che rientrano nel **range di dati** selezionato. Pertanto, quando una transazione esegue una **SELECT**, vengono acquisiti **N lock**, uno per ciascuna riga coinvolta, e questi permangono fino al **commit** della transazione.

Tuttavia, sorge una criticità: se una seconda transazione esegue un **UPDATE** su una riga che inizialmente **non apparteneva al range selezionato**, il lock su quella riga **non è presente**, poiché non era vincolata dalla selezione precedente. Di conseguenza, l'operazione di modifica è consentita.

Questo implica che, pur mantenendo il lock sulle righe originariamente lette, il sistema **non impedisce modifiche su dati esterni al range iniziale** (**INSERT**, a maggior ragione dato che **non esiste e il lock è libero, quindi crea una nuova riga e la "locka"**, una riga non può essere lockata se non c'è :|), i quali, successivamente, potrebbero rientrarvi a seguito di un aggiornamento.

id	saldo	
1	100€	
2	200€	
3	300€	



Su MySQL, **REPEATABLE READ** è il livello di isolamento predefinito.



L'unico livello di isolamento che impedisce le letture fantasma è **SERIALIZABLE**, perché **blocca le operazioni di inserimento ed eliminazione** su un range di dati finché la transazione non è completata.

SERIALIZABLE.

Quando una transazione legge un range di dati, nessun'altra transazione può inserire, modificare o eliminare righe in quel range fino al commit della transazione attuale.

Questo è il livello più restrittivo e garantisce la massima coerenza dei dati. Per farlo, utilizza un range lock, cioè **blocca non solo le righe esistenti, ma anche l'intero intervallo di dati coinvolto nella query.**

Tutti i lock vengono mantenuti fino alla fine della transazione e ogni volta che una SELECT utilizza uno specificatore di tipo WHERE, viene acquisito anche il range lock.

ESEMPIO FONDAMENTALE PER CAPIRE:

id	saldo
1	100€
2	200€
3	300€

Immaginiamo di avere nel database una tabella conti così composta:

E supponiamo di avere due transazioni T1 e T2.

T1:Seleziona tutte le righe con saldo \geq 200€.

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;  
START TRANSACTION;  
SELECT * FROM conti WHERE saldo >= 200;
```

Il DBMS applica un range lock, bloccando tutte le righe che rientrano nel range e l'intervallo dove potrebbero essere inserite nuove righe.

id	saldo
1	100€
2	200€
3	300€

T2: Prova a inserire una nuova riga con saldo = 250.

```
START TRANSACTION;  
INSERT INTO conti (id, saldo) VALUES (4, 250);
```



BLOCCATA.

T2 deve **aspettare che T1 finisca** prima di poter inserire la riga.
Questo perché l'INSERT rientra nel range bloccato da T1
(saldo >= 200).

T2: Prova a modificare saldo = 150 per id = 1.

```
UPDATE conti SET saldo = 250 WHERE id = 1;
```



BLOCCATA.

Anche se la riga id = 1 non era nel range iniziale, l'UPDATE la sposterebbe dentro (saldo >= 200), quindi il DBMS impedisce la modifica fino al commit di T1.

T2: Prova a cancellare una riga nel range bloccato.

```
DELETE FROM conti WHERE saldo = 200;
```



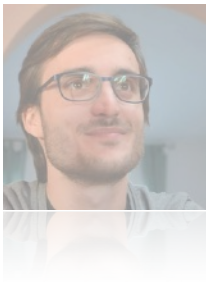
BLOCCATA.

La riga con **saldo = 200** è già bloccata da T1, quindi nessun'altra transazione può modificarla o eliminarla finché T1 non termina.

Nessuna transazione concorrente può inserire, modificare o cancellare dati nel range fino al commit.

Evita le letture fantasma, garantendo una consistenza totale.

Attenzione: **riduce la concorrenza** e può portare a blocchi e rallentamenti nelle operazioni parallele.



*"Non usare il livello di isolamento **SERIALIZABILE** a casaccio. Non è una bacchetta magica: metterlo ovunque solo perché non sai quale scegliere non è la soluzione, in questo modo distruggiamo le performance all'aumentare del numero di utenti."*

Prof. Ing. A. Pellegrini .

Ricorda: Se più transazioni non accedono agli stessi dati e non operano sullo stesso range, possono essere eseguite in parallelo senza interferenze.

Morale: Il vero problema nasce solo quando le transazioni lavorano sugli stessi dati o range, causando blocchi, attese o anomalie di concorrenza.

È buona pratica ricordarlo.

Ora approfondiamo la trattazione, superando i confini del corso di Basi di Dati per addentrarci in concetti più avanzati, non necessari ai fini dell'esame. Dopo aver compreso i fondamenti del controllo della concorrenza e delle transazioni nei DBMS, è il momento di esplorare le **soluzioni più sofisticate**.

Sistemi a Controllo di Concorrenza Multiversione (MVCC, Multi-Version Concurrency Control).

I sistemi a controllo di concorrenza multiversione (MVCC - Multi-Version Concurrency Control) sono un meccanismo avanzato utilizzato dai DBMS moderni per gestire accessi concorrenti senza bloccare le letture.

Ogni transazione vede una "versione" isolata dei dati, senza interferire con altre transazioni.

Invece di applicare lock esclusivi sui dati, il DBMS mantiene più versioni di una riga, permettendo alle letture di avvenire senza aspettare le scritture. Quando un dato viene modificato, il DBMS **non sovrascrive immediatamente la riga originale**, ma crea una **nuova versione della riga**, mantenendo quella vecchia accessibile per le transazioni che la stanno ancora leggendo.

Esempio pratico:

id	saldo
1	100€

Lo schema che rappresenta il funzionamento è il seguente:

T1: Legge il saldo di id = 1

```
START TRANSACTION;  
SELECT saldo FROM conti WHERE id = 1;
```

T1 vede saldo = 100€.

T1 rilancia la SELECT

```
SELECT saldo FROM conti WHERE id = 1;
```

T1 continua a vedere 100€, mentre nuove transazioni vedranno 150€ dopo il commit di T2.

T2 aggiorna il saldo e conferma la modifica

```
START TRANSACTION;  
UPDATE conti SET saldo = 150 WHERE id = 1;  
COMMIT;
```

T2 ha creato una nuova versione con saldo = 150€, visibile solo per le nuove transazioni.

Fino al commit di T2, tutte le altre transazioni vedranno ancora la versione vecchia (100€).

T3: Legge il saldo di id = 1

```
SELECT saldo FROM conti WHERE id = 1;
```

T3 vede saldo = 150€.

- Se T2 ha modificato il valore ma non ancora confermato (COMMIT), allora vedrà il valore aggiornato (150€) all'interno della sua transazione.
- Questo è un comportamento standard nei DBMS con MVCC: una transazione vede le proprie modifiche interne, anche prima del commit.

Questo è il principio chiave di MVCC: ogni transazione vede uno "snapshot" coerente del database fino al commit!

Non è basato su lock.

Alla fine di questo viaggio, la domanda a cui vogliamo rispondere è: **quale livello di isolamento utilizzare?**

I livelli di isolamento più elevati impongono lock più stringenti sui dati, il che può generare situazioni di stallo tra transazioni concorrenti.

Più lock → più attese tra transazioni concorrenti.

- **SERIALIZABLE**, ad esempio, utilizza **range lock**, bloccando intere sezioni di dati.
- Se due transazioni cercano di accedere a dati sovrapposti, si verificano attese reciproche.

Esempio pratico di deadlock causato da SERIALIZABLE.

Immaginiamo due transazioni (T1 e T2) che operano sulla tabella conti:

id	saldo
1	100€
2	200€

T1 blocca il conto id = 1 e aspetta di aggiornare id = 2

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;  
START TRANSACTION;  
UPDATE conti SET saldo = 150 WHERE id = 1;  
🔒 T1 blocca id = 1
```

T2 blocca il conto id = 2 e aspetta di aggiornare id = 1

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;  
START TRANSACTION;  
UPDATE conti SET saldo = 250 WHERE id = 2;  
🔒 T2 blocca id = 2
```


Ora T1 prova ad aggiornare `id = 2`, ma è bloccato da T2

```
UPDATE conti SET saldo = 200 WHERE id = 2;
```

🛑 T1 è in attesa perché `id = 2` è bloccato da T2

T2 prova ad aggiornare `id = 1`, ma è bloccato da T1

```
UPDATE conti SET saldo = 300 WHERE id = 1;
```

🛑 T2 è in attesa perché `id = 1` è bloccato da T1

Risultato: T1 e T2 sono bloccate in attesa reciproca → Deadlock!

Più alto è il livello di isolamento, più aumenta il rischio di deadlock. Serve un compromesso tra coerenza e concorrenza per evitare rallentamenti e stalli nel sistema.

Soluzioni per ridurre il rischio di deadlock.

Usare livelli di isolamento più bassi (**READ COMMITTED** o **REPEATABLE READ**) quando possibile.

Evitare lock troppo restrittivi su ampi range di dati.

Evitare di mantenere una transazione aperta più del necessario.

Usare timeout sulle transazioni per rilevare e risolvere deadlock: **SET innodb_lock_wait_timeout = 5;**

L'unico che conosce quale livello di isolamento andare ad utilizzare è l'implementatore della store procedure.

Effetto collaterale: SERIALIZABLE può "bloccare" le altre transazioni.

Anche se un'altra transazione sta usando un livello più basso (**READ COMMITTED**), potrebbe comunque rimanere **in attesa** se cerca di accedere a dati che sono bloccati da una transazione **SERIALIZABLE**.

Esempio pratico

T1 usa **SERIALIZABLE** e legge tutti i conti con saldo $\geq 200\text{€}$

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;  
START TRANSACTION;  
SELECT * FROM conti WHERE saldo >= 200;
```

Il DBMS blocca l'intero range, impedendo qualsiasi modifica a queste righe.

T2 prova a modificare una riga nel range, ma usa **READ COMMITTED**

**SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
START TRANSACTION;**

UPDATE conti **SET** saldo = 300 **WHERE** id = 2;

🛑 **Bloccata!** Anche se T2 ha un livello di isolamento più basso, **non può eseguire l'UPDATE** finché T1 non termina la sua transazione!

Effetto collaterale: anche se il DBMS **non promuove formalmente T2 a SERIALIZABLE**, nella pratica **T2 si comporta come se fosse SERIALIZABLE** perché rimane bloccata in attesa.

Se molte transazioni lavorano **sugli stessi dati**, l'uso di **SERIALIZABLE** crea un **effetto domino**, riducendo la concorrenza.

Nei sistemi con **alto carico**, usare **SERIALIZABLE** **può causare colli di bottiglia**, rallentando tutte le transazioni.

Evitare **SERIALIZABLE** quando non è strettamente necessario.

2PL.

2PL non è un livello di isolamento da impostare direttamente, ma una **tecnica usata dai DBMS per gestire la concorrenza**.

Scegliendo **REPEATABLE READ** o **SERIALIZABLE**, il DBMS applica 2PL per garantire l'isolamento delle transazioni.

Il **Two-Phase Locking (2PL)** è un protocollo fondamentale per garantire l'**isolamento e la consistenza** delle transazioni nei DBMS. Il suo principio chiave è **imporre un ordine fisso di accesso ai dati**, evitando **deadlock e inconsistenze**.

Regola chiave: Una volta che una transazione rilascia un lock, non può acquisirne di nuovi!

Esempio di Two-Phase Locking.

Immaginiamo di avere la tabella conti:

id	saldo
1	100€
2	200€

T1 inizia la transazione e acquisisce un lock su id = 1

Fase 1: Growing Phase (Acquisizione Lock)

- 1 T1 inizia e acquisisce un lock su id = 1 per leggerlo**

```
START TRANSACTION;
SELECT saldo FROM conti WHERE id = 1;
🔒 Lock su riga 1 (lettura)
Out: (1, 100€)
```

- 2 T2 prova a leggere id = 2 e prende il lock di lettura**

```
START TRANSACTION;
SELECT saldo FROM conti WHERE id = 2;
🔒 Lock su riga 2 (lettura)
Out: (1, 100€)
```

- 3 T1 aggiorna id = 1 e prende un lock di scrittura**

```
UPDATE conti SET saldo = 150 WHERE id = 1;
🔒 Lock di scrittura su riga 1
Out: (1, 150€) per id=1
```

- 4 T2 prova a modificare id = 1 ma è bloccata!**

```
UPDATE conti SET saldo = 300 WHERE id = 1;
🚫 BLOCCATA! (T1 ha il lock su id = 1)
Stato tabella: (1, 150€) per id=1
```

Finché T1 non rilascia il lock, T2 non può procedere.

◆ Fase 2: Shrinking Phase (Rilascio Lock)


- 5 T1 esegue il COMMIT, rilasciando tutti i lock**

```
COMMIT;
```

 Lock su riga 1 rilasciato

6 Ora T2 può eseguire l'UPDATE

```
UPDATE conti SET saldo = 300 WHERE id = 1;
```

 Ora può procedere









```
COMMIT;
```

```
Output = (1,300).
```

- **Durante la Growing Phase**, T1 e T2 acquisiscono lock sulle righe che vogliono modificare.
- **T1 deve rilasciare tutti i lock prima che T2 possa continuare.**
- **Questo impedisce che una transazione modifichi dati mentre un'altra li sta usando**, evitando inconsistenze.

Se hai una transazione che sta inserendo dati (**INSERT**) in una tabella e un'altra transazione che deve leggere (**SELECT**) dalla stessa tabella, il comportamento dipende dal livello di isolamento impostato.

Cosa succede tra **INSERT** e **SELECT** in base al livello di isolamento?

Livello di Isolamento	Il SELECT vede l' INSERT prima del commit?	Rischi di blocco o inconsistenza
READ UNCOMMITTED	 Sì, legge i dati anche se la transazione non ha fatto COMMIT	 Possibili dirty reads (può vedere dati che poi verranno annullati con ROLLBACK)
READ COMMITTED	 No, legge solo dati confermati (COMMIT)	 Nessun rischio di dirty reads, ma possibili unrepeatable reads
REPEATABLE READ	 No, legge sempre gli stessi dati per tutta la transazione	 Evita unrepeatable reads, ma può avere letture fantasma
SERIALIZABLE	 No, può addirittura bloccare nuovi INSERT nel range della query	 Massima consistenza ma rischio di deadlock

Immaginiamo la tabella **ordini**:

id	prodotto	prezzo
1	Laptop	1000€

Caso 1: READ UNCOMMITTED (pericoloso)

- 1** T1 inizia e inserisce un nuovo ordine, ma non fa ancora COMMIT

```
START TRANSACTION;  
INSERT INTO ordini (id, prodotto, prezzo) VALUES (2, 'Mouse',  
50);
```

- 2** T2 legge la tabella e vede il nuovo record

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;  
SELECT * FROM ordini;
```

T2 vede il record con id=2 (anche se T1 non ha fatto COMMIT!)

- 3** T1 fa ROLLBACK e l'inserimento sparisce!

```
ROLLBACK;
```

Problema: T2 ha visto un record che ora **non esiste più** → **dirty read!**

Caso 2: READ COMMITTED (più sicuro)

- 1** T1 inizia la transazione e inserisce un nuovo record

```
START TRANSACTION;  
INSERT INTO ordini (id, prodotto, prezzo) VALUES (2, 'Mouse',  
50);
```

- 2** T2 legge la tabella, ma **NON** vede ancora il nuovo record

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;  
SELECT * FROM ordini;  
-- T2 vede solo il record con id=1
```

- 3** T1 fa COMMIT, ora il dato è visibile per tutti

```
COMMIT;
```

- 4** Ora T2 rifà la SELECT e vede il nuovo record

```
SELECT * FROM ordini;  
Ora T2 vede anche id=2
```

 **Vantaggio:** T2 vede solo dati confermati, evitando dirty reads.

Caso 3: **SERIALIZABLE** (rischio di blocco)

1 T1 inizia e inserisce un record

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;  
START TRANSACTION;  
INSERT INTO ordini (id, prodotto, prezzo) VALUES (2, 'Mouse',  
50);
```

2 T2 prova a leggere la tabella, ma il DBMS lo blocca in attesa che T1 finisca

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;  
SELECT * FROM ordini;  
-- 🚫 Bloccato fino al commit di T1!
```

3 T1 fa COMMIT, solo ora T2 può leggere

```
COMMIT;
```

Più lento.

In molti DBMS, quando una transazione esegue un **SELECT**, il sistema applica un Read Lock (Shared Lock - S), che consente ad altre transazioni di leggere gli stessi dati contemporaneamente.

Più transazioni possono effettuare letture senza bloccarsi a vicenda perché un read lock può essere condiviso con altri read lock.

T1: START TRANSACTION;

T1: SELECT saldo FROM conti WHERE id = 1; -- 🔒 Read Lock su id=1

T2: START TRANSACTION;

T2: SELECT saldo FROM conti WHERE id = 1; -- ✅ Consentito! Read Lock condiviso

Se una transazione esegue solo letture (**SELECT**), può generalmente utilizzare il livello di isolamento più basso senza problemi.

Perché?

- **Le letture non modificano i dati**, quindi non rischiano di causare inconsistenze nel database.
- **I livelli di isolamento bassi (READ UNCOMMITTED, READ COMMITTED) offrono migliori prestazioni**, evitando blocchi e attese inutili.

Il mondo dei **lock** e delle **transazioni** è un ambito estremamente complesso e dettagliato, in cui l'ingegnere deve essere in grado di intervenire con competenza e precisione. Per quanto questo documento possa apparire esaustivo, **non potrà mai sostituire l'esperienza diretta e la conoscenza che si acquisiscono attraverso la pratica sul campo**. Pertanto, questo materiale deve essere considerato come **un supporto teorico** che, per una comprensione più approfondita, va integrato con una **videolezione di riferimento**, che offrirà un ulteriore livello di approfondimento e applicazione pratica. A tal proposito, verrà allegata la videolezione del **Prof. Pellegrini**, che arricchirà e completerà lo studio dell'argomento.