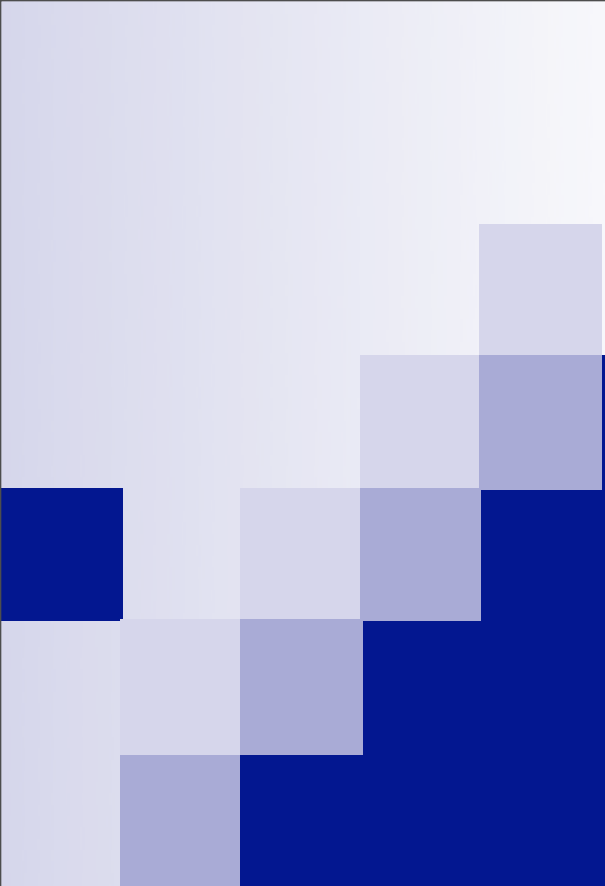
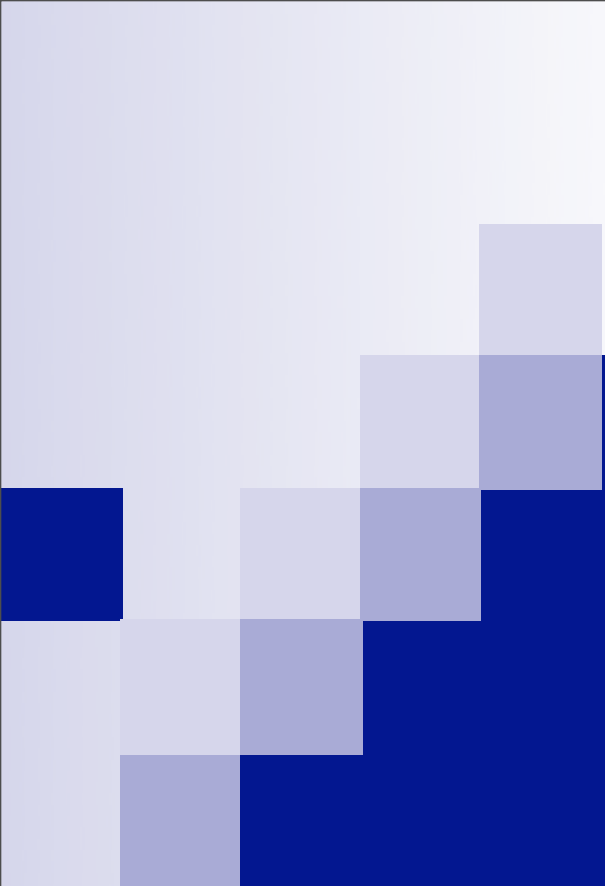


DB - Cenni sulla gestione delle transazioni

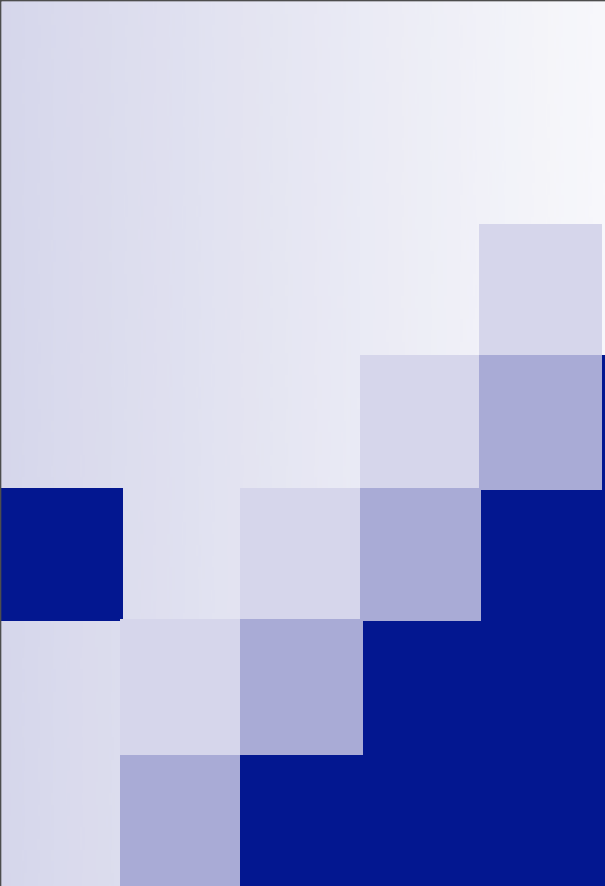


# Cenni sulla gestione delle transazioni in DBMS



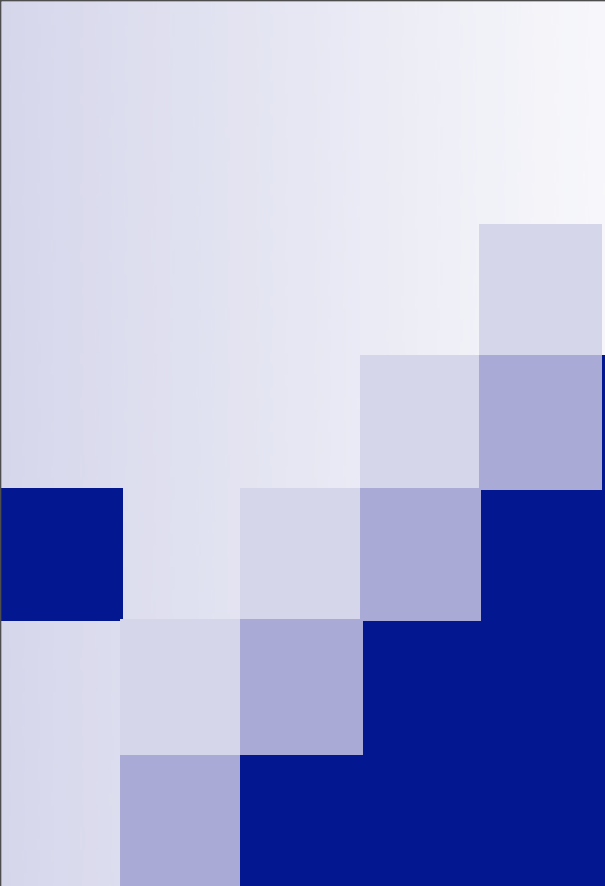
# Cenni sulla gestione delle transazioni in DBMS

Basato sulle slides di



# Cenni sulla gestione delle transazioni in DBMS

Basato sulle slides di  
**Monica Mordonini**

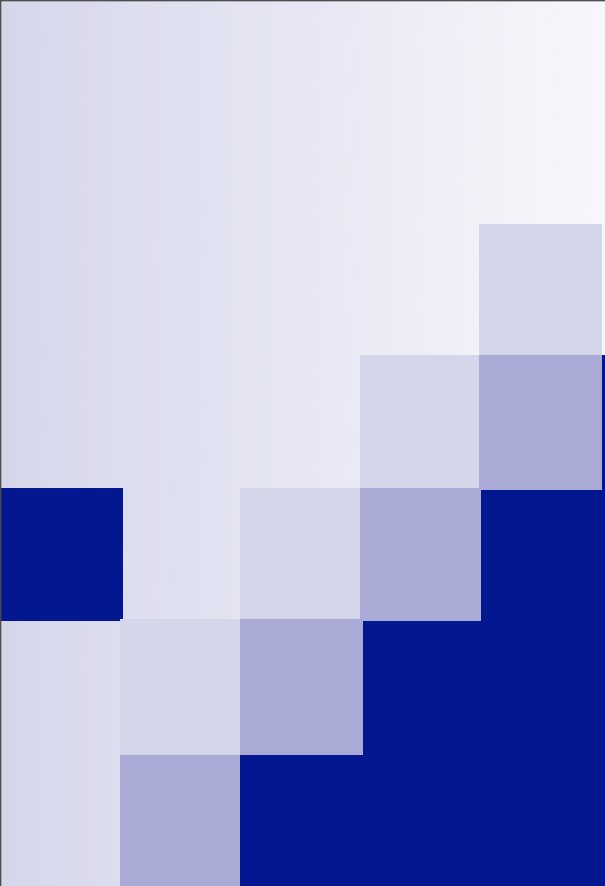


# Cenni sulla gestione delle transazioni in DBMS

Basato sulle slides di

**Monica Mordonini**

*(<http://www.ce.unipr.it/people/monica/>)*



# Cenni sulla gestione delle transazioni in DBMS

Basato sulle slides di

**Monica Mordonini**

*(<http://www.ce.unipr.it/people/monica/>)*

# Definizione di Transazione

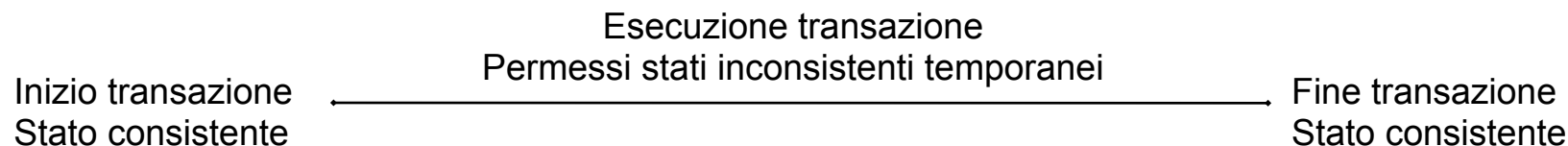
- Sequenza di operazioni sul DB cui si vogliono associare particolari caratteristiche di correttezza, robustezza, isolamento.
  - Esempi: Bonifico bancario, acquisto biglietto, prenotazione aerea, etc.
- Sistema transazionale: mette a disposizione un meccanismo per la definizione e l'esecuzione di transazioni.

# Transazioni

- Per mantenere le informazioni consistenti è necessario controllare opportunamente le sequenze di accessi e aggiornamenti ai dati.
- Gli utenti interagiscono con la base di dati attraverso programmi applicativi i quali usano le transazioni per garantire la correttezza del dato inserito/letto.
- Una transazione si può interpretare come un insieme parzialmente ordinato di operazioni di lettura e scrittura

# Transazione

- Ogni transazione è incapsulata tra due comandi:
  - Begin transaction (bot)
  - End transaction (eot)
- In una transazione devono essere eseguiti (solo una volta) almeno uno dei seguenti comandi:
  - Commit work (conferma l'operazione)
  - Rollback work (aborto)
- All'inizio ed alla fine della transazione il sistema e' in uno stato "consistente"
  - Durante l'esecuzione della transazione stessa il sistema puo' temporaneamente essere in uno stato inconsistente



# Transazione ben formata

- Proprietà che si manifesta a tempo di esecuzione
- Una transazione comincia con *begin transaction* e finisce con *end transaction*
- solo uno dei due comandi **commit work** o **rollback work** viene eseguito
  - le operazioni di aggiornamento *fisico* dei dati sono eseguite solo quando si fa una di queste due operazioni

# Transazioni - Proprietà

- L'insieme di operazioni che costituiscono una transazione deve soddisfare alcune proprietà, note come proprietà ACID:
  - **A**tomicità
  - **C**onsistenza
  - **I**solamento
  - **D**urabilità (Persistenza)

# Transazioni - Atomicità (1/2)

- E' detta anche proprietà *“tutto o niente”*
- Tutte le operazioni di una transazione devono essere trattate come una singola unità
  - o vengono eseguite tutte, oppure non ne viene eseguita alcuna
- L'atomicità delle transazioni è assicurata dal **sottosistema di ripristino (recovery)**.

# Transazioni - Atomicità (2/2)

- Non deve lasciare il database in uno stato intermedio inconsistente
  - un errore prima di commit deve causare l'UNDO delle operazioni fatte dall'inizio della transazione (bot)
- Possibili comportamenti:
  - Buon fine
  - aborto richiesto dall'applicazione (suicidio)
  - aborto richiesto dal sistema (morte) la base di dati

# Transazioni - Consistenza (1/2)

- una transazione deve agire sulla base di dati in modo corretto
- la transazione trasforma la base di dati da uno stato consistente (cioè che riflette lo stato reale del mondo che la base di dati deve modellare) ad un altro stato ancora consistente
- l'esecuzione di un insieme di transazioni corrette e concorrenti deve a sua volta mantenere consistente la base di dati

# Transazioni - Consistenza (2/2)

- La consistenza richiede che l'esecuzione della transazione non violi i vincoli di integrità definiti sulla base di dati.
- La verifica dei vincoli di integrità può essere:
  - immediata: durante la transazione (le operazioni devono essere inabiliate)
  - differita: quando il cliente ha deciso il commit nel qual caso l'intera transazione deve essere cancellata
- **conseguenza:** se lo stato iniziale è corretto lo è anche quello finale

# Transazioni - Isolamento (1/2)

- ogni transazione deve sempre avere accesso ad una base di dati consistente:
  - non può leggere risultati intermedi di altre transazioni.
- la proprietà di isolamento è assicurata dal sottosistema di **controllo della concorrenza** che isola gli effetti di una transazione fino alla sua terminazione

# Transazione - Isolamento (2/2)

- Richiede che ogni transazione venga eseguita indipendentemente dall'esecuzione di tutte le altre transazioni concorrenti
  - eliminare effetto domino: il rollback di una transazione causa il rollback delle altre

# Transazioni - Durabilità (Persistenza)

- i risultati di una transazione terminata con successo devono essere resi permanenti nella base di dati nonostante possibili malfunzionamenti del sistema
- il **sottosistema di ripristino (recovery)** garantisce la durabilità
  - può fornire misure addizionali, quali *back-up* su supporti diversi e *journaling* delle transazioni
  - ciò garantisce la durabilità anche a fronte di guasti ai dispositivi di memorizzazione
- indipendente dai guasti di sistema

# Transazioni e moduli di sistema

- Atomicità e Persistenza sono garantiti dal controllo dell'affidabilità (**sottosistema di ripristino**)
- L'isolamento è garantito dal **controllo di concorrenza**
- La consistenza è infine gestita dai compilatori DDL (*Data definition Language*) che introducono le opportune verifiche sui dati

# Transazioni e protocolli

- Le proprietà ACID vengono assicurate utilizzando due insiemi distinti di algoritmi o protocolli, che assicurano:
  - l'atomicità dell'esecuzione
    - Devono mantenere la consistenza globale della base di dati e quindi assicurare la proprietà di consistenza delle transazioni (anche concorrenti)
    - Necessita di protocolli di controllo della concorrenza
  - l'atomicità del fallimento

# Transazioni - Modello flat

- Facciamo riferimento al modello di transazioni più semplice (transazioni flat)
- prevede un solo livello di controllo
  - è il modello più usato nei DBMS commerciali
  - **BeginWork** dichiara l'inizio di una transazione flat
  - **CommitWork** indica il raggiungimento di un nuovo stato consistente
- I vari DBMS forniscono specifiche istruzioni SQL per il controllo delle transazioni

# Controllo della concorrenza

- **Scopo:**
  - Garantire l'integrità della base di dati in presenza di accessi concorrenti da parte di più utenti.
  - Necessità di sincronizzare le transazioni eseguite concorrentemente
- **Carico applicativo di DBMS: numero di transazioni per secondo (tps)**
- **I DBMS oggi sono in grado di eseguire migliaia di transazioni al secondo**

# Concorrenza: esempio

- base di dati che organizza le informazioni sui conti dei clienti di una banca
- il sig. Rossi e' titolare di due conti: un conto corrente (intestato anche alla sig.ra Rossi) e un libretto di risparmio rispettivamente con € 100 e € 1000.
- con la transazione T1 il sig. Rossi trasferisce € 150 dal libretto di risparmio al conto corrente
- contemporaneamente con la transazione T2 la sig.ra Rossi deposita € 500 sul conto corrente.

# Concorrenza: esempio

T1

Read(Lr)

$Lr = Lr - 150$

Write(Lr)

Read(Cc)

$Cc = Cc + 150$

Write(Cc)

Commit

T2

Read(Cc)

$Cc = Cc + 500$

Write(Cc)

Commit

La somma depositata da T2 è persa, (Lost update) e non si ottiene l'effetto voluto sulla base dati.

# Concorrenza: esempio

- l'esecuzione concorrente di più transazioni genera un'alternanza di computazioni da parte delle varie transazioni, detta *interleaving*
- l'interleaving tra le transazioni T1 e T2 nell'esempio produce uno stato della base di dati scorretto
- si sarebbe ottenuto uno stato corretto se le due transazioni fossero state eseguite l'una dopo l'altra, consecutivamente

# Anomaly 1: Update loss

- Consider two identical transactions:
  - $t_1 : r(x), x = x + 1, w(x)$
  - $t_2 : r(x), x = x + 1, w(x)$
- Assume initially  $x=2$ ; after serial execution  $x=4$
- Consider concurrent execution:

– Transaction $t_1$	Transaction $t_2$
bot	
$r_1(x)$	
$x = x + 1$	
	bot
	$r_2(x)$
	$x = x + 1$
$w_1(x)$	
commit	
	$w_2(x)$
	commit
- One update is lost, final value  $x=3$

## Anomaly 2: Dirty read

- Consider the same two transactions, and the following execution (note that the first transaction fails):

– Transaction $t_1$	Transaction $t_2$
bot	
$r_1(x)$	
$x = x + 1$	
$w_1(x)$	
	bot
	$r_2(x)$
	$x = x + 1$
abort	
	$w_2(x)$
	commit

- Critical aspect:  $t_2$  reads an intermediate state (dirty read)

# Anomaly 3: Inconsistent read

- $t_1$  repeats two reads:

– Transaction $t_1$	Transaction $t_2$
bot	
$r_1(x)$	
	bot
	$r_2(x)$
	$x = x + 1$
	$w_2(x)$
	commit
$r_1(x)$	
commit	

- $t_1$  reads different values for  $x$

## Anomaly 4: Ghost update

- Assume the integrity constraint  $x + y + z = 1000$ ;
  - Transaction  $t_1$ 
    - bot
    - $r_1(x)$
  
    - $r_1(y)$
  
  
    - $r_1(z)$
    - $s = x + y + z$
    - commit
  - Transaction  $t_2$ 
    - bot
    - $r_2(y)$
  
    - $y = y - 100$
    - $r_2(z)$
    - $z = z + 100$
    - $w_2(y)$
    - $w_2(z)$
    - commit
- In the end,  $s = 110$ : the integrity constraint is not satisfied
- $t_1$  sees a ghost update

# Concorrenza: lock

## ■ Idea base:

- ritardare l'esecuzione di operazioni in conflitto imponendo che le transazioni pongano dei blocchi (lock) sui dati per poter effettuare operazioni di lettura e scrittura
- due operazioni si dicono in conflitto se operano sullo stesso dato e almeno una delle due è un'operazione di scrittura
- una transazione può accedere ad un dato solo se ha un lock su quel dato

# Teoria del controllo della concorrenza

- Transazione: sequenza di operazioni di lettura e scrittura
- Ogni transazione ha un identificatore univoco assegnato dal sistema
- Assumiamo di omettere il begin/end transaction
- una transazione può essere rappresentata come:  $t_1: r_1(x) r_1(y) w_1(x) w_1(y)$

# Schedule

- Rappresenta la sequenza di operazioni I/O presentate da transazioni concorrenti
- esempio:  $S : r_1(x) r_2(z) w_1(x) w_2(z)$
- Il compito del controllo di concorrenza è di accettare alcuni schedule e rifiutarne altri

Lo schedule  $S$  è valido?

# Schedule serializzabile

- Uno schedule seriale non è efficiente anche se elimina alla fonte molte anomalie
- Schedule serializzabile:
  - Uno schedule che produce lo stesso risultato di uno seriale a partire dalle stesse transazioni ma in condizioni di concorrenza

# Primitive di lock

- Tutte le operazioni di lettura e scrittura devono essere protette tramite l'esecuzione di tre diverse primitive:
  - r\_lock
  - w\_lock
  - Unlock
- Lo scheduler che riceve una sequenza di richieste di esecuzione di queste primitive da parte delle transazioni ne determina la concorrenza

# Transazioni ben formate

- Stato di un oggetto:
  - Libero
  - r-locked (bloccato da un lettore)
  - w-locked (bloccato da uno scrittore)
- Ogni read di un oggetto deve essere preceduto da `r_lock` e seguito da `unlock`
- Ogni write di un oggetto deve essere preceduto da `w_lock` e seguito da `unlock`

# Tabella dei conflitti

	libera	r-locked	w-locked
r-lock	si	ok	no
w-lock	si	no	no

- **si**: blocco della risorsa il programma procede
- **no**: il programma va in attesa che la risorsa venga sbloccata
- **ok**: il contatore dei lettori viene incrementato per ogni lettore e decrementato dopo operazione unlock
  - una risorsa può essere letta da più lettori, ma non modificata mentre si legge

# Locking a due fasi

- Una transazione dopo aver rilasciato un lock non può acquisirne degli altri:
  - prima fase: la transazione acquisisce tutti i lock che le servono (fase crescente)
  - seconda fase (calante) i lock acquisiti vengono rilasciati

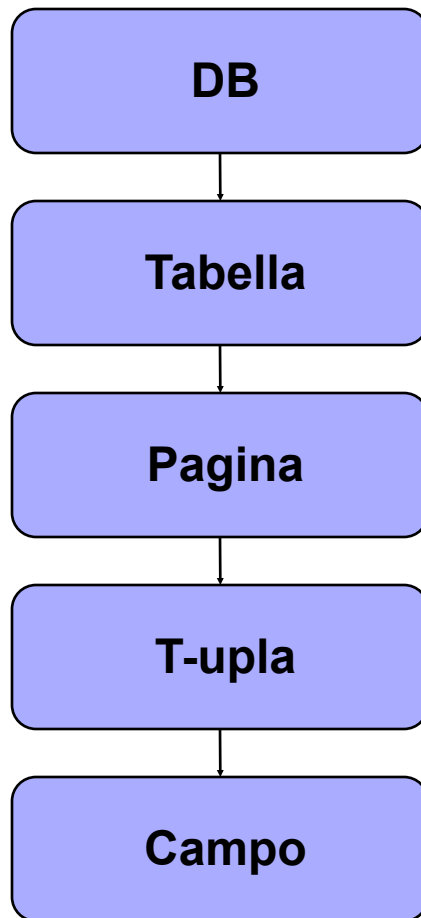
# Assunzioni

L'uso di:

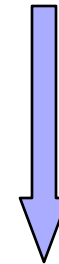
- Transazioni ben formate
- Politica dei conflitti (come da tabella)
- Locking a due fasi

*-> implicano la serializzabilità*

# Granularità del locking



Granularità  
ridotta



Maggiore  
concorrenza

# Problema del deadlock

**T1**

w-lock(a)

w-lock(b)

read/write(a)

read/write(b)

unlock(a)

unlock(b)

**T2**

w-lock(b)

w-lock(a)

read/write(a)

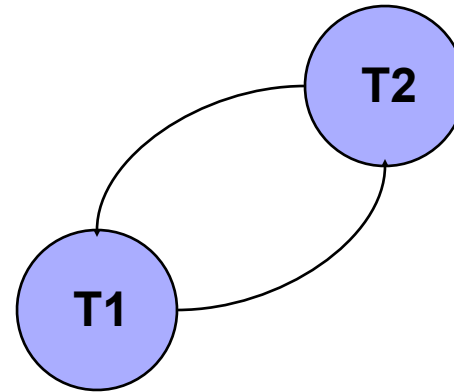
read/write(b)

unlock(b)

unlock(a)

# Insorgenza del deadlock

- T1 esegue w-lock(a)
- T2 esegue w-lock(b)



- T1 attende una risorsa controllata da T2
- T2 attende una risorsa controllata da T1

# Tecniche risolutive del deadlock

## ■ Time-out

- Un'attesa eccessiva è interpretata come deadlock (per clustered database)

## ■ Prevenzione

- evitare alcune condizioni di attesa

## ■ Determinazione

- ricerca dei cicli sul grafo di attesa