

---

## Parte VII

### Gestione delle transazioni

---

## Funzioni del DBMS

- **Gestione dei dati**: cura la memorizzazione permanente dei dati ed il loro accesso
- **Gestione del buffer**: cura il trasferimento dei dati da memoria di massa a memoria centrale, e il caching dei dati in memoria centrale
- **Ottimizzazione delle interrogazioni**: seleziona il piano esecutivo di costo ottimo con cui valutare ciascuna interrogazione
- **Controllo della concorrenza**: garantisce la consistenza della base di dati in presenza di accessi concorrenti
- **Controllo dell'affidabilità**: garantisce il funzionamento del sistema in presenza di guasti

---

## Transazioni

- Unità elementari di lavoro sulla base di dati di cui si vogliono garantire proprietà di *correttezza*, *robustezza* e *isolamento*
- I DBMS prevedono meccanismi per gestire la definizione e l'esecuzione di transazioni
- Sintatticamente una transazione è contornata dai comandi **begin transaction** e **end transaction**; all'interno possono comparire i comandi di **commit work** e **rollback work**.
- Un comando di **commit** trasferisce gli effetti della transazione sulla base di dati
- Un comando di **rollback** (*abort*) annulla gli effetti della transazione e lascia inalterata la base di dati

---

## Transazioni: esempio

```
begin transaction
X := X - 10;
Y := Y + 10;
commit work;
end transaction
```

- Esegue il trasferimento di dieci unità da X a Y (p.es. conti correnti)
- Si vuole garantire che *vengano eseguite entrambe le azioni*, o *nessuna delle due*

## Transazioni ben formate

---

- Una transazione si dice *ben formata* se:
  - inizia con **begin transaction (bot)**;
  - termina con **end transaction (eot)**;
  - nel suo corso viene eseguito solo uno dei due comandi **commit work** o **rollback work**.
- In alcuni sistemi una coppia **begin transaction, end transaction** viene automaticamente eseguita dopo ciascun commit o abort.
- L'esecuzione risulta così automaticamente segmentata in una sequenza di transazioni ben formate.

## Proprietà ACIDE

---

- Dall'acronimo inglese **ACID: Atomicity, Consistency, Isolation, Durability**
- Sono le proprietà di cui vogliamo le transazioni godano
- **Atomicità**: ciascuna transazione è un'unità indivisibile di esecuzione
- **Consistenza**: l'esecuzione di una transazione non deve violare l'integrità della base di dati
- **Isolamento**: il risultato dell'esecuzione di una transazione deve essere indipendente dall'esecuzione di altre transazioni
- **Persistenza (Durability)**: gli effetti dell'esecuzione di una transazione andata in commit non devono essere persi

## Atomicità

---

- Per una transazione si possono avere *solo* due casi:
  - a) *Tutti* gli effetti della sua esecuzione sono trasferiti sulla base di dati
  - b) La transazione non produce *alcuna* variazione sulla base di dati
- Il DBMS deve poter effettuare particolari azioni per garantire l'atomicità
  - **undo**: se la transazione non va a buon fine viene *disfatto* il lavoro fatto fino ad allora
  - **redo**: se una transazione va in *commit*, gli effetti della sua esecuzione sono garantiti, e può essere necessario *rifare* il lavoro fatto

## Consistenza

---

- Se una transazione viola un vincolo di integrità, il sistema deve reagire
  - *correggendo* la violazione quando ci è o abbia senso;
  - *uccidendo* la transazione, cioè mandandola in *abort*
- Distinguiamo fra:
  - **vincoli immediati**: verifica immediata nel corso dell'esecuzione
  - **vincoli differiti**: verifica solo all'atto del *commit*
- Se c'è una violazione la *commit* non va a buon fine e la transazione viene abortita

## Isolamento

- Più transazioni possono essere elaborate concorrentemente
- Esse possono causare conflitti nell'accesso a dati comuni
- Si pensi a successioni di letture e scritture sciaguratamente intercalate
- Considerando l'esecuzione complessiva, tutto deve avvenire come se ciascuna transazione fosse eseguita isolatamente
- Il DBMS realizza meccanismi di controllo della concorrenza che garantiscono l'isolamento
- Problema con forti analogie con la sincronizzazione di processi concorrenti

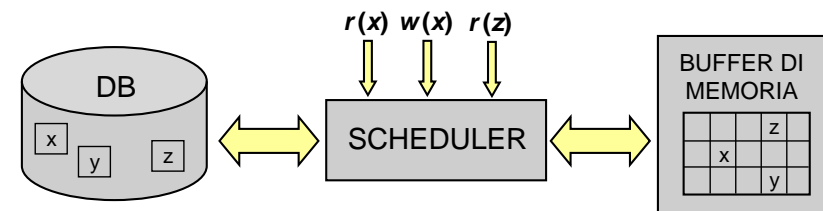
## Persistenza

- Quando una transazione va in *commit*, il DBMS garantisce la persistenza nel tempo degli effetti della sua esecuzione
- La persistenza deve essere garantita in presenza di:
  - Guasti di Sistema: crash del DBMS, o del sistema su cui esso gira
  - Guasti di Dispositivo: guasto dei dispositivi di memoria di massa, o delle linee in sistemi distribuiti
- Il DBMS deve conservare, in maniera *sicura* l'informazione necessaria a ripristinare gli effetti delle transazioni

## Proprietà ACIDE e moduli del DBMS

- L'architettura del DBMS è strutturata in modo da garantire il rispetto delle proprietà acide
- L'*atomicità* e la *persistenza* sono garantite dal *controllo di affidabilità*
- L'*isolamento* è garantito dal *controllo della concorrenza*, che regola l'accesso ai dati da parte delle transazioni
- La *consistenza* è garantita dai compilatori del DDL che introducono opportuni controlli che vengono poi eseguiti run-time dalle transazioni
- Prima dei DBMS esistevano comunque *sistemi transazionali* che supportavano l'esecuzione di transazioni

## Controllo della concorrenza



- Le transazioni accedono ai dati x, y, z tramite operazioni di lettura  $r(x)$  e scrittura  $w(x)$
- Ciascuna operazione comporta un trasferimento tra un *blocco* su disco e una *pagina* di memoria centrale
- Tutte le operazioni di lettura e scrittura vengono gestite dallo *scheduler* che determina se ciascuna richiesta può essere soddisfatta o meno

## Transazioni concorrenti: esempio

- Supponiamo che in una base di dati bancaria vengano sottomesse due transazioni di bonifico:
  - $T_1$  bonifico dal conto **A** al conto **C**
  - $T_2$  bonifico dal conto **B** al conto **C**
- Le transazioni operano (con significato evidente) sulle tabelle:

**Bonifico**(Cod\_bo,Conto1,Conto2,Importo,Data)  
**Conto**(Cod\_cc,Cod\_age,Saldo,Max\_scoperto)

## Sequenza di operazioni

- Ciascuna transazione tra l'istante di inizio e quello in cui chiede di andare in commit compie la stessa sequenza di accessi elementari alla base dati:
  1. **Letture in Conto del Saldo del conto debitore**
  2. **Aggiornamento in Conto del Saldo del conto debitore**
  3. **Letture in Conto del Saldo del conto creditore**
  4. **Aggiornamento in Conto del Saldo del conto creditore**
  5. **Inserimento in Bonifico di una nuova tupla**
- Se  $T_1$  e  $T_2$  vengono eseguite concorrentemente, e senza controllo, le loro azioni elementare si possono intercalare in un ordine qualsiasi con effetti nefasti

## Sequenza scorretta

1.  $T_1$  legge il Saldo del conto **A**
2.  $T_1$  aggiorna il Saldo del conto **A**
3.  $T_2$  legge il Saldo del conto **B**
4.  $T_2$  aggiorna il Saldo del conto **B**
5.  $T_1$  legge il Saldo del conto **C**
6.  $T_2$  legge il Saldo del conto **C**
7.  $T_2$  aggiorna il Saldo del conto **C**
8.  $T_1$  aggiorna il Saldo del conto **C**
9.  $T_1$  inserisce in Bonifico una nuova tupla
10.  $T_2$  inserisce in Bonifico una nuova tupla

## Sequenza scorretta (continua)

- L'esecuzione complessiva è scorretta
- Sia  $T_1$  che  $T_2$  leggono lo stesso valore per il saldo del conto **C**
- Alla fine il conto **C** risulterà incrementato del solo bonifico effettuato da  $T_1$ , e non di entrambi come sarebbe corretto
- In un DBMS il controllo della concorrenza avrebbe impedito il verificarsi di questa situazione
- $T_1$  e  $T_2$  avrebbero dovuto chiedere allo *scheduler* il permesso di accedere ai dati della base di dati
- Lo scheduler può in presenza di conflitti porre una transazione in attesa

## Serializzabilità

---

- Se le transazioni vengono eseguite in *maniera seriale*, cioè una per volta si hanno sempre esecuzioni corrette
- Questo comporterebbe però un degrado inaccettabile delle prestazioni
- Per gestire correttamente l'esecuzione concorrente, si fa pertanto riferimento al *criterio di serializzabilità*

*Viene garantito che l'esecuzione di più transazioni concorrenti abbia sempre lo stesso effetto finale che si sarebbe avuto se le transazioni fossero state eseguite in maniera completamente seriale, in un ordine qualsiasi*

## Locking

---

- Per garantire la serializzabilità quasi tutti i DBMS usano una tecnica denominata *locking* (blocco)
- Vengono bloccati man mano i dati che una transazione sta utilizzando, per poi rilasciarli quando la transazione va in *commit* o in *abort*
- Una transazione che vuole accedere a dati bloccati da un'altra, è costretta ad un'attesa
- Grande miglioramento di prestazioni rispetto ad un'esecuzione completamente seriale
- Interferiscono, e vengono rallentate solo transazioni che accedono a dati comuni

## Two Phase Lock (2PL)

---

- **Two Phase Lock (2PL)**: tecnica di locking adottata raticamente da tutti i DBMS commerciali
- Questa distingue fra due tipi di lock:
  - **Lock condiviso**: viene messo se la transazione accede ai dati in lettura
  - **Lock esclusivo**: viene messo se la transazione accede ai dati in scrittura
- Transazioni che vogliono accedere agli stessi dati solo in lettura non sono necessariamente rallentate
- Unico vincolo: una transazione non può più chiedere lock quando ne ha già rilasciato uno

## Locking gerarchico

---

- Uno svantaggio del locking è che limita l'esecuzione concorrente
- Il locking gerarchico consente di bloccare solo le parti effettivamente necessarie della base di dati
- Questo riduce la probabilità di blocco delle transazioni
- Diversi livelli di *granularità di lock*:
  - L'intero database
  - Un'intera tabella
  - Una singola tupla
  - Un campo di una tupla

## Sequenza corretta

1.  $T_1$  legge il Saldo del conto **A** e blocca la relativa tupla di Conto
2.  $T_1$  aggiorna il Saldo del conto **A**
3.  $T_2$  legge il Saldo del conto **B** e blocca la relativa tupla di Conto
4.  $T_2$  aggiorna il Saldo del conto **B**
5.  $T_1$  legge il Saldo del conto **C** e blocca la relativa tupla di Conto
6.  $T_2$  cerca di leggere il Saldo del conto **C** (bloccato da  $T_1$ ), e viene bloccata
7.  $T_1$  aggiorna il Saldo del conto **C**
8.  $T_1$  inserisce in Bonifico una nuova tupla bloccando la tabella
9.  $T_1$  va in commit e rilascia i blocchi, sbloccando così  $T_2$
10.  $T_2$  legge il Saldo del conto **C** e blocca la relativa tupla di Conto
11.  $T_2$  aggiorna il Saldo del conto **C**
12.  $T_2$  inserisce in Bonifico una nuova tupla bloccando la tabella
13.  $T_2$  va in commit e rilascia i blocchi

## Sequenza corretta (commenti)

- Questa esecuzione delle transazioni  $T_1$  e  $T_2$  potrebbe essere generata da uno scheduler 2PL
- Corrisponde ad una esecuzione seriale in cui  $T_1$  viene eseguita completamente prima di  $T_2$
- $T_2$  viene bloccata solo quando tenta di leggere il saldo di C, precedentemente bloccato da  $T_1$
- $T_2$  viene poi sbloccata quando  $T_1$  viene mandata in commit e rilascia tutti i lock
- I lock vengono presi a livello di tupla (granularità di lock fine)
- Lock a livello di tabella avrebbero comportato un minore livello di concorrenza (efficienza nell'esecuzione)

## Deadlock

- Come tutte le tecniche di blocco, il *two phase lock* può condurre ad una situazione di blocco critico (*deadlock*)
- Il deadlock è una situazione di attesa circolare
- Si verifica, ad esempio, quando due transazioni detengono in uso esclusivo ciascuna la parte di base di dati su cui l'altra è in attesa
- Il deadlock viene gestito dai DBMS in due modi
  - Prevenzione: il DBMS mantiene un grafo di allocazione e verifica che non vi siano cicli
  - Timeout: si pone un limite alla durata delle transazioni

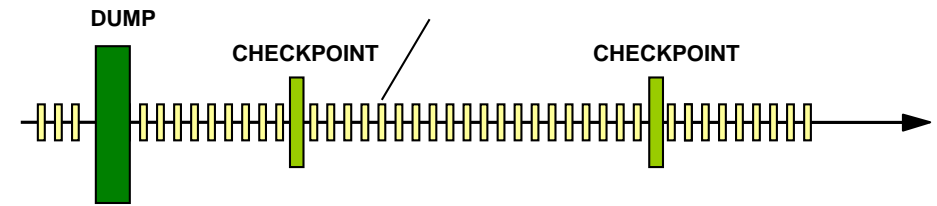
## Controllo del deadlock con timeout

- Viene stabilito un tempo massimo di esecuzione (*timeout*) per ciascuna transazione
- Se una transazione supera il timeout essa il DBMS assume che essa sia in deadlock
- La transazione viene mandata in *abort*, questo risolve la situazione di stallo
- Successivamente la transazione (incolpevole) viene mandata in *restart*
- *Timeout lunghi* provocano il persistere di situazioni di stallo
- *Timeout brevi* causano un eccessivo numero di restart

## Controllo di affidabilità

- Garantisce le proprietà di atomicità e persistenza
- Il controllore di affidabilità è responsabile di:
  - *Mantenere un log (giornale) che registra tutte le azioni svolte sulla base di dati*
  - *Realizzare l'esecuzione della primitiva **redo**, che ripristina gli effetti (perduti) di un'azione già svolta*
  - *Realizzare l'esecuzione della primitiva **undo**, che annulla gli effetti di un'azione svolta*
  - *Realizzare l'esecuzione dei comandi di **begin transaction** , **commit work** , **rollback work***
  - *Realizzare le operazioni di ripresa a caldo e ripresa a freddo di ripristino dopo malfunzionamenti*

## Organizzazione del log



- Record di transazione: registrano le azioni elementari compiute dalle transazioni
- Checkpoint: registrano la situazione delle transazioni attive
- Dump: viene effettuata una copia completa della base di dati

**N.B.** Il log è registrato su memoria stabile (e.g. dischi RAID)

## Record di transazione

---

- Contengono informazioni dettagliate su ciascuna azione elementare compiuta da una transazione
  - *Inizio della transazione*
  - *Commit della transazione*
  - *Abort della transazione*
  - *Dettaglio di ogni aggiornamento*
  - *Dettaglio di ogni inserimento*
  - *Dettaglio di ogni cancellazione*
- Per ciascun aggiornamento, inserimento e cancellazione vengono conservati i nuovi ed i vecchi valori di tutti gli attributi interessati (**before-state** e **after-state**)

## Checkpoint e dump

---

- **Checkpoint**: viene scritto ad intervalli periodici:
  - Contiene la lista di tutte *transazioni attive*
  - Prima del checkpoint il DBMS *riallinea buffer con memoria di massa*
- **Dump**: viene generato molto meno spesso del checkpoint:
  - Contiene una copia completa della base di dati
  - Effettuato col DB fuori linea e nessuna transazione attiva
  - Effettuato su supporti rimovibili e conservato in luogo diverso
  - Operazione molto costosa in termini di tempo e indisponibilità del sistema

## Allineamento del buffer

---

- In realtà le transazioni operano su dati che si trovano nel *buffer*, cioè in memoria centrale
- La memoria di massa viene riallineata solo ogni tanto, non spesso per motivi di efficienza
- Quando una transazione va in *commit*, non è garantito che gli aggiornamenti siano stati trasferiti alla base di dati
- In caso di crash il contenuto della memoria viene perduto, e con esso gli aggiornamenti effettuati solo nel buffer
- Buffer e base di dati vengono periodicamente riallineati negli istanti di checkpoint
- Sono garantite tutte le transazioni andate in commit prima del checkpoint

## Regole nella gestione del log

---

- **WAL (Write Ahead Log)**: impone di scrivere la parte **before-state** di un record di log *prima* di effettuare la corrispondente azione sulla base di dati. Garantisce la possibilità di eseguire correttamente una **undo**
- **Commit-Precedenza**: impone di scrivere la parte **after-state** dei record di log *prima* di effettuare il commit. Garantisce di poter rifare correttamente le scritture di una transazione andata incommit, ma con pagine non ancora trasferite su memoria di massa

**N.B.** *Queste regole garantiscano che in qualunque momento capiti il guasto sia possibile effettuare la ripresa correttamente*

## Tipi di guasti

---

- **Guasti di sistema**: dovuti a errori nel software, oppure anche ad eventi esterni quali la mancanza dell'alimentazione
  - Portano alla caduta del sistema operativo (*crash* di sistema)
  - Perdita della memoria centrale, ma non di quella di massa
  - Perdita del *buffer*
- **Guasti di dispositivo**: riguardano i dispositivi di memoria di massa
  - Perdita, totale o parziale, del contenuto della base di dati

*I guasti non devono poter interessare il log, che deve essere scritto su memoria stabile (tipo dischi RAID)*

## Ripresa a caldo

---

- Si effettua in corrispondenza a *guasti di sistema*
- Si cerca nel log l'*ultimo checkpoint*
- Sono a rischio solo le transazioni andate in *commit* dopo questo checkpoint
- Non c'è garanzia di *riallineamento tra memoria e dischi*
- Tutte queste transazioni devono essere *ricostruite*, in base al contenuto del log
- Le altre transazioni presenti nel checkpoint vengono *abortite*, cioè i loro effetti parziali vengono *disfatti*
- Dopo la ripresa queste transazioni vengono *mandate in restart*

## Ripresa a freddo

---

- Si effettua in corrispondenza a *guasti di dispositivo*
- Consta di tre fasi:
  - I. *Si ripristina completamente la base di dati a partire dal dump più recente*
  - II. *Si ripercorre il log a partire dal dump fino all'istante di guasto effettuando tutte le operazioni ivi registrate*
  - III. *Si effettua una ripresa a caldo a partire dall'ultimo checkpoint, come se si fosse appena verificato un guasto di sistema*