Transactions

Un DBMS è un ambiente multiutente in cui vari programmi interagiscono con i dati. Include componenti quali:

- *transaction manager*
- *concurrency control*
- *logging and recovery*
  e altri moduli per gestire memoria, esecuzione di query e sicurezza dei dati.

Un programma è una sequenza di:
read(X): legge un elemento del database chiamato X;
write(X): scrive un valore nell'elemento del database.

Una **transazione** rappresenta l'esecuzione del programma di un utente come unità indivisibile (insieme di operazioni di lettura e scrittura).
Problemi principali:

- esecuzione concorrente: più transazioni possono essere eseguite simultaneamente
  motivazioni:
  - gli accessi al disco sono lenti; la CPU non dovrebbe rimanere inattiva.
  - Per migliorare le prestazioni, più transazioni possono essere eseguite in modo concorrente, mantenendo la CPU occupata.
    - **Problema**: l'utente non deve percepire l'interleaving delle transazioni, cioè l'esecuzione simultanea di più operazioni.
- **Recupero da crash**: il database deve rimanere consistente anche in caso di errori o guasti.
  - **Problema**: il DBMS deve garantire che altre transazioni non siano influenzate dall'errore e deve lasciare il database in uno stato consistente.

Proprietà ACID

Per garantire l'esecuzione sicura e simultanea delle operazioni, ogni transazione deve rispettare le proprietà ACID:

Atomicità

- Una transazione deve essere completata integralmente o non avere alcun effetto sul database.
- **Gestione dell'abort**:
  - Cause di abort:
    1. Anomalie interne all'esecuzione (bloccate dal DBMS).
    2. Condizioni di eccezione rilevate dalla transazione (autosospensione).
    3. Crash di sistema (es. errori hardware, software o di rete).
  - In caso di abort, il DBMS:
    - Utilizza il file di log per annullare la transazione (rollback) e ripristinare lo stato consistente precedente.

Consistenza

- La transazione inizia e termina rispettando tutti i vincoli di integrità del database.

Isolamento

- Ogni transazione deve essere eseguita come se fosse l'unica attiva, indipendentemente dall'interleaving con altre transazioni.

Durabilità

- Una volta che una transazione è confermata (commit), le modifiche apportate diventano permanenti, anche in caso di crash.
  Una transazione:
- parte sempre in uno stato consistente dove tutti i vincoli referenziali sono soddisfatti;
- potrebbe passare attraverso uno stato intermedio di inconsistenza;
- termina sempre in uno stato consistente dove tutti i vincoli referenziali sono soddisfatti.

Schedulazione delle Transazioni

- **Definizione:** Una schedule rappresenta la sequenza cronologica di operazioni (read, write, commit, abort) eseguite da più transazioni.

Esempio

Dati:

- **T1**: `read1(A) → write1(A) → read1(C) → write1(C)`
- **T2**: `read2(B) → write2(B)`
  Schedule:
  `read1(A) → write1(A) → read2(B) → write2(B) → read1(C) → write1(C)`

Tipi di schedule:

1. **Completa:** Include commit o abort per ogni transazione.
2. **Seriale:** Tutte le operazioni di una transazione vengono eseguite consecutivamente (una transazione attiva alla volta).
3. **Serializzabile:** Non è necessariamente seriale, ma produce lo stesso risultato di una schedule seriale.

Anomalie nell'Esecuzione Interleaved

Quando più transazioni sono eseguite simultaneamente, possono verificarsi anomalie che lasciano il database in uno stato inconsistente.

Tipi di conflitti:

1. **Write-Read Conflict (Dirty Read):**
   - **T1** aggiorna `A` ma non esegue il commit.
   - **T2** legge `A` e utilizza un dato potenzialmente errato.

- Se **T1** fa rollback, il dato letto da **T2** è invalido.
  - Given the following two transactions T_1 and T_2 and the schedule S:

    T_1: A := A + 100, B := B - 100

    T_2: A := A * 1.06, B := B * 1.06

  - **Please note**: T_1 writes a value in A that could make the DB inconsistent, and this value is read by T_2
  - **Problem**: A is written by T_1 and read by T_2 before T_1 commits

T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
	read(B)
	write(B)
	commit
read(B)	
write(B)	
commit	

2. **Read-Write Conflict (Unrepeatable Read):**
   - **T1** legge `A = 500`.
   - **T2** aggiorna `A` a 600 e conferma.
   - **T1** legge di nuovo `A` e trova 600, ma le letture di **T1** non sono coerenti.
     - Given the following two transactions T_1 and T_2 and the schedule S:

       T_1: A := A + 1

       T_2: A := A - 1

     - **Please note**: if T_1 repeats the reading of A, it will get a different value
     - **Problem**: A is written by T_2 before T_1 commits

T_1	T_2
read(A)	
	read(A)
	write(A)
	commit
write(A)	
commit	

3. **Write-Write Conflict (Lost Update):**
   - **T1** aggiorna `A` a 600.

- **T2** sovrascrive `A` a 450, perdendo l'aggiornamento di **T1**.
  - Given the following two transactions T_1 and T_2 and the schedule S:

    T_1: A := 1000, B := 1000

    T_2: A := 2000, B := 2000

  - **Please note**: the values of A and B are equal at the end of each of the two transactions
  - **Problem**: A and B have different values at the end of the schedule

T_1	T_2
write(A)	
	write(A)
	write(B)
	commit
write(B)	
commit	

4. **Phantom Anomalies:**

- Una transazione legge un insieme di dati (es. una tabella) e successivamente rileva che l'insieme è cambiato (es. inserimento/cancellazione di righe da un'altra transazione).

Aborted Transactions
In linea di principio, tutte le azioni di una transazione aborted devono essere cancellate. Tuttavia non è sempre possibile. Esempio:

T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
	read(B)
	write(B)
	commit
read(B)	
write(B)	
abort	

problema: T2 legge il valore di A che non avrebbe dovuto leggere, a causa di un write-read conflict
soluzione non soddisfacente: T2 dovrebbe anche essere abortito, ma violerebbe la proprietà di durabilità delle transazioni.

Transactions Parameters

- **Access Mode** imposta l'autorizzazione per modificare le tabelle utilizzate nella transazione:
  - **read only**: permette soltanto la lettura del DB, il tentativo di modificare il DB causa un errore
  - **read write**: consente le operazioni di lettura e scrittura
- **Statement Mode**: specifica le azioni da eseguire al termine di una transazione

- **Isolation Level**: specifica come gestire le transazione che modificano il DB:
  - **READ UNCOMMITTED**: la transazione richiede blocchi per scrivere oggetti ma non blocchi per la lettura. Consente *dirty read*.
  - **READ COMMITTED**: la transazione richiede blocchi per la scrittura e blocchi condivisi per la lettura. Previene *dirty read*.
  - **REPEATABLE READS**: blocca letture e scritture fino al commit.
  - **SERIALIZABLE**: blocca transazioni che potrebbero causare conflitti.

Level	Dirty read (Write-Read)	Unrepeatable read (Read-Write)	Lost update (Write-Write)	Phantom
READ UNCOMMITTED	may occur	may occur	may occur	may occur
READ COMMITTED	don't occur	may occur	may occur	may occur
REPEATABLE READS	don't occur	don't occur	don't occur	may occur
SERIALIZABLE	don't occur	don't occur	don't occur	don't occur

Concurrency Control Approaches

L'interleaving è necessario e preferibile per migliorare le performance, d'altronde non tutte le schedule sono possibili, alcune azioni devono essere riavvolte (roll back).
Diversi modi di controllare la concorrenza:

- **RESTRICTIVE**: (conflict-serializability) ogni transazione esegue due fasi:
  1. *acquisizione*: richiede blocchi
  2. *rilascio*: libera i blocchi dopo il commit

     Esempio:
     T1 acquisisce un blocco esclusivo su A e lo modifica.
     T2 deve aspettare il rilascio del blocco di A.

Strict Two-phase Locking (Strict 2PL):

- se una transazione vuole leggere/scrivere un oggetto, deve richiedere un accesso esclusivo.
- dopo aver rilasciato un lock, la transazione non può chiederne altre
- quando viene eseguito il commit la transazione rilascia tutti gli accessi esclusivi
- Strict 2PL garantisce la serializzabilità, in particolare:
  - se le transazioni accedono a oggetti diversi, possono essere interleaved
  - altrimenti, se almeno due transazioni voglio l'accesso allo stesso oggetto e almeno una delle due vuole modificarlo, le due transazioni dovranno essere eseguite in serie
    il **Lock Manager** tiene traccia, per ogni oggetto del DB, dell'accesso in una *lock table*.

- **OPTIMISTIC**: esegue tutte le transazioni concorrenti e verifica la presenza di conflitti prima del commit
- **TIMESTAMPING**: assegna timestamps alle transazioni e compara tutti i valori per determinare l'ordine delle operazioni.

Deadlock Prevention

Una concorrenza locking-based potrebbe causare *deadlocks*. Solitamente i DBMS assegnano una priorità alle transazioni in base al tempo di inizio.
Se T1 richiede un lock posseduto da T2, il lock manager può:

- **wait-die**: se T1 è la transazione antecedente, T1 aspetta. Altrimenti T1 viene terminata e ripresa successivamente, con un random delay ma con lo stesso timestamp
- **wound-wait**: se T1 è la transazione antecedente, T2 viene terminata e ripresa successivamente con random delay, ma stesso timestamp. Altrimenti, T1 aspetta.
  Se i deadlock sono rari, il DBMS lascia che si verifichino e li risolve invece di adottare le policy per evitarli. Due approcci più comuni:
- il lock manager mantiene una struttura chiamata **waits-for graph**, che utilizza per identificare il cicli deadlock. Il grafo è periodicamente analizzato e i cicli deadlock sono risolti abortendo alcune transazioni.
- se una transazione aspetta per un periodo più lungo del timeout assegnato, il lock manager assume che la transazione è deadlocked e la abortisce.

Optimistic Concurrency Control

Il protocollo basato sul locking adotta un approccio pessimistico per prevenire i conflitti.
L'approccio ottimistico assume che le transazioni non vadano in conflitto (o che lo facciano raramente).
Validation è un approccio ottimistico dove le transazioni possono accedere ai dati senza lock, e al momento opportuno, controlla che le transazioni si siano comportate in modo seriale.
Le transazioni sono eseguite in tre fasi:

- **read**: la transazione è eseguita leggendo il dato dal DBMS e scrivendo in un area privata
- **validation**: prima del commit, il DBMS controlla che non ci siano stati conflitti. In tal caso, la transazione viene abortita e restartata automaticamente.
- **write**: se la fase di validation viene conclusa con successo, il dato scritto nell'area privata viene copiato nel DBMS.

Timestamping concurrency control

Timestamping è un altro approccio ottimistico che assegna per ogni transazione il timestamp TS del suo start time. Per ogni operazione a1 eseguita da T1:

- se l'operazione a1 è in conflitto con l'operazione a2 eseguita da T2 e TS1 < TS2, allora a1 deve essere eseguita prima di a2.
- se un'operazione eseguita da T viola questo ordine, la transazione T è abortita e ripresa con un TS maggiore.

Crash Recovery

Il **Logging and recovery manager** del DBMS deve assicurare:

- *atomicità*: operazioni eseguite da transazioni non-committed sono rolled back.
- *persistenza*: operazioni eseguite da transazioni committed devono persistere ad un crash di sistema

ARIES Recovery Algorithm

Advanced Recovery and Integral Extraction System è un algoritmo di ripristino eseguito dal Logging and Recovery Manager sugli arresti anomali di sistema. Tre fasi:

- **fase di analisi**: identifica le pagine sporche del buffer pool (cioè, le modifiche non ancora scritte sul disco) e le operazioni attive nel momento del crash
- **fase di redo**: partendo da un dato checkpoint nel log file, ripete tutte le operazioni e riporta il DB nello stato in cui si trovava al momento del crash
- **fase di undo**: cancella le operazioni delle transazioni che erano attive al momento del crash, ma che non erano committed, in ordine inverso.
  Principi:
- **write-ahead logging**: qualunque cambiamento ad un oggetto del DB deve prima essere registrato nel log file. Successivamente, il log file deve essere riportato sulla memoria secondaria. Infine, le pagine modificate possono essere aggiornate.
- **repeating history during redo**: Al riavvio, dopo un arresto, il sistema viene riportato allo stato in cui si trovava prima dell'anomalia. Le operazioni delle transazioni ancora attive durante il crash vengono cancellate.
- **logging changes during redo**: i cambiamenti fatti al DB durante l'annullamento delle transazioni sono registrati per assicurare che quell'azione non venga ripetuta nel caso di un altro arresto anomalo.

Log File

Il log file tiene traccia di tutte le azioni eseguite dal DBMS. E' fisicamente organizzato in registrazioni memorizzate in uno storage stabile, che dovrebbe resistere ad incidenti / guasti dell'hardware. I log records sono ordinati sequenzialmente con un id unico, *Log Sequence Number* (LSN).
La parte più recente del log file, detta **log tail**, è conservata in **log buffers** e salvata periodicamente in storage stabili. Il log file e i dati vengono scritti su disco con gli stessi meccanismi.

Logging Data Structures

La **dirty page table** tiene i record di tutti gli ID delle pagine che sono state modificate e non ancora scritte in uno storage stabile, e della LSN dell'ultimo log entry che l'ha causato.

Log File Record

<LSN, Transaction ID, Page ID, Redo, Undo, Previous LSN>

- i campi Transaction ID e Page ID identificano la transazione e la pagina
- i campi Redo e Undo tengono le informazioni sulle modifiche che il log record salva e sul come annullarle
- il campo Previous LSN è una reference al precedente log record creato per la stessa transazione. Permette il roll back di transazioni abortite.

Creazione di Log File Record

Un record è scritto nel Log File per ognuno dei seguenti eventi:

- **page update**: un record deve essere scritto in memoria stabile prima di modificare effettivamente i dati della pagina. Mantiene sia il vecchio che il nuovo valore della pagina per rendere possibili le

operazioni di undo e redo.

- **commit**: un record traccia che una transazione è stata completata con successo e il log tail viene scritto nello stable storage.
- **abort**: un record traccia una transazione abortita, e la transazione undo viene ripresa
- **end**: dopo un commit/abort, sono necessarie alcune operazioni di finalizzazione al fine della quale viene scritta una registrazione finale
- **undoing operation**: durante un recovery o durante l'undoing delle operazioni, viene un scritto un tipo speciale di file record, il Compensation Log Record (CLR). Un record CLR non viene mai ripristinato e traccia che un operazione e stata già annullata

Checkpoint

Un checkpoint è uno snapshot dello stato del DB. I checkpoint riducono il tempo di ripristini. Invece di dover eseguire l'intero log file, è sufficiente eseguire all'indietro fino ad un checkpoint. ARIES crea checkpoints in tre step:

- **begin-checkpoint**: il record viene scritto nel log file
- **end-checkpoint**: il record, contenente *dirty page table* e *transaction table*, viene scritto nel log file
- alla fine dell'end-checkpoint il record viene scritto nello stable storage, un record, **Master Record**, contenente LSN del *begin-checkpoint* viene scritto in una parte conosciuta dello stable storage Questo tipo di checkpoint viene detto **fuzzy checkpoint** e non è costoso in termini di performance. Non interrompe le normali operazioni del DBMS e non richiede la scrittura delle pagine del buffer pool.

ARIES Crash Recovery

- **analysis phase**:
  - determina la locazione del log file da dove comincia la fase di redo, ovvero l'inizio dell'ultimo checkpoint
  - determina quale pagine del buffer pool contiene il dato modificato che non era ancora stato scritto al momento dell'anomalia.
  - identifica le transazioni che erano in corso al momento dell'anomalia
- **redo phase**: dalla *dirty page table*, ARIES identifica il minimo LSN di una dirty page. Da qui, ripete le azioni fino all'anomalia, nel caso in cui non erano già state rese persistenti
- **undo phase**: i cambiamenti di una transazione uncommitted devono essere annullate in modo da ripristinare il DB in uno stato consistente.