Domande Orale

Indice

Transazioni

Transazione: complesso di operazioni (di *lettura* e *scrittura*) che portano il DB da uno stato corretto ad un altro stato corretto.
Un sistema transazionale è in grado di eseguire transazioni per un certo numero di applicazioni concorrenti.
(In SQL è possibile definire una transazione mediante uno specifico comando `start`, non è invece necessario definire l'`end` ⇒ nel blocco va poi utilizzato almeno un comando tra il `commit work` (permette terminazione corretta) e il `rollback work` (permette di abortire la tranisizione)).

ACID

La transazione gode di 4 proprietà:

- **Atomicità**: le operazioni che costituiscono la transazione compongono un blocco unico che deve essere eseguito nella sua interezza (Il DB non può essere lasciato in uno stato intermedio). In caso di guasti o errori prima del termine della transazione, le azioni sui dati già svolte devono essere annullate (operazione detta `UNDO`) e poi successivamente ripetute (`REDO`). ⇒Tali operazioni sono realizzate dal sistema di recovery (Ha il compito di riportare il DB in uno stato corretto in caso di guasti).
  Se la transazione ha successo, le modifiche vengono riporate in modo

permanente dal DBMS nella memoria fisica (stato corretto: stato in cui il DB non presenta modifiche dovute a transazioni incomplete).

> ⚠ NOTA: >
>
> Mediante rollback work è la transazione che richiede l'annullamento delle operazioni per qualche motivo. Talvolta però, questa richiesta può essere inoltrata direttamente dal sistema che non riesce ad eseguire le operazioni.

- **Consistenza**: la transazione rispetta i vincoli di integrità del DB. In questo modo si garantisce che lo stato di arrivo sia corretto.
- **Isolamento**: la transazione è isolata rispetto alle altre transazioni concorrenti.
- **Durabilità**: le transazioni che terminano con una operazione di commit vengono mantenute in memoria fisica in modo permanente.

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)

> ⚠ NOTA: >
>
> Le modifiche spesso vengono riportate prima sul log e solo dopo nella memoria secondaria contenente il DB. (modalità differita)

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

Le operazioni di risoluzione vengono svolte dal **recovery manager**.

- *Guasti "soft"*: errori di programma, crash di sistema, perdita di tensione. In questo caso si perde il contenuto della memoria principale, ma rimane preservato il contenuto della memoria secondaria (gestite tramite ripresa a caldo);
- *Guasti "hard"*: si perde anche il contenuto della memoria secondaria. Rimane perciò preservato solo il contenuto della memoria stabile (`log` e `dump`).
  Ripresa a caldo: 4 fasi
- Individuazione ultimo *checkpoint* nel log
- Costruire insiemi `UNDO` (operazioni da disfare) e `REDO` (operazioni da rifare)
- Scandire il log a ritroso, cancellando tutte le operazioni inserite nell'insieme UNDO
- Avanzare nuovamente nel log implementando le operazioni dell'insieme REDO
  Ripresa a freddo: i dati vengono ripristinati mediante il dump. A questo punto vengono effettuate di nuovo tutte le operazioni registrate nel log fino al momento del guasto seguendo lo schema di una ripresa a caldo.

Concorrenza

I sistemi odierni devono essere in grado di governare centinaia di transazioni al secondo che possono essere eseguite anche nello stesso arco temporale.
Se la concorrenza non venisse gestita correttamente, potrebbero verificarsi gravi problematiche legate alla consistenza dei dati.

Consideriamo la seguente situazione: una transizione T_1 prova a modificare il valore di una certa variabile X, però, prima che questo venga trascritto in memoria fisica, viene schedulata una seconda transizione T_2 che legge X e la modifica. In questo caso il risultato della transizione T_1 viene perso, perché T_2 leggerà il valore iniziale di X e sarà questo ad essere modificato.
Un'altra situazione critica è quando la transizione T_1 prova a leggere due volte il valore di una certa X. Se tra queste due operazioni viene schedulata una T_2 che modifica X, T_1 leggerà per X due valori distinti senza averla modificata.

Si possono infine generale anche altre anomalie come gli inserimenti fantasma (T_1 legge stipendi impiegati, poi viene schedulata T_2 che inserisce un nuovo impiegato.

T_1 a questo punto viene caricata di nuovo in CPU per essere completata e calcola la media degli stipendi $\Rightarrow$ non teniamo conto di inserimento fatto da T_2).

Il gestore delle transizioni si affida per risolvere tali problematiche a specifici *scheduler*, come quelli seriali. In questo caso le operazioni delle transizioni vengono eseguite una alla volta in maniera successiva. (Una volta terminata T_1 si passa poi a T_2) $\Rightarrow$ se T presenta un elevato numero di operazioni, avremo occupazione non efficiente della CPU.
Gli scheduler, in generale, possono provocare le problematiche precedentemente citate. Una parte di essi, però, produrrà il risultato corretto, in questo caso sono detti serializzabili (ovvero producono sul DB gli stessi effetti di uno scheduler seriale). Abbiamo bisogno di un metodo per stabilire tale equivalenza. (Nella seguente trattazione supponiamo di dover gestire solo transazioni andate in commit)

Scheduler View-Serializzabili (VSR)

Relazione "`legge da`": dato un'ordinamento di scheduler S, si dice che una operazione di lettura $ri(x)$ `LEGGE_DA` un'operazione di scrittura $wj(x)$ se $wj(x)$ *precede* $ri(x)$ in S e non sono presenti altre $wk(x)$ tra le due operazioni citate.
Scrittura finale: $wi(x)$ è scrittura finale, se è l'ultima operazione di scrittura su x che appare in S. Due scheduler sono *view-equivalenti* se hanno la stessa relazione "`legge da`" e le stesse scritture finali.
Uno scheduler è **view-serializzabile** se è *view-equivalente* ad uno scheduler seriale.
PROBLEMA: stabilire view-serializzabilità di uno scheduler è un problema NP-completo.

Scheduler Conflict-Serializzabili

In uno scheduler S due azioni distinte a_i ed a_j sono in conflitto tra loro se operano sullo stesso oggetto x ed almeno una è di **write**.
Due scheduler si dicono **conflict-equivalenti** se contengono le *stesse operazioni* ed i conflitti appaiono nello stesso ordine.
Uno scheduler è **conflict-serializzabile** se è *conflict-equivalente* ad uno scheduler seriale.
Per verificare la *conflict-serializzabilità* basta utilizzare un algoritmo su grafi $(grafo \Rightarrow (nodo = transazione)$, (arco orientato da T_i a T_j se c'è almeno un conflitto