Gestione delle transazioni

Tramite DDL e DML è possibile garantire una costruzione dell'architettura e insiemi di interrogazioni congrue ad un qualsiasi modello relazionale. Tuttavia, fino ad ora è stata attuata una semplificazione a livello strutturale, ossia non sono state considerate operazioni concorrenti e che le tabelle della basi dati possano divenire *sezioni critiche*. Prima di affrontare un contesto simile, si definiscono brevemente i termini citati.

Definizione informale
Con *concorrenza* si intende qualora più processi siano eseguiti nello stesso istante, causando alterazioni del sistema software di riferimento.

Definizione informale
Con *sezione critica* si descrive un dato o un insiemi di dati condivisi a più processi di diversa natura, i quali possono essere soggetti a modifiche dagli stessi, provocandone inconsistenza.

Alludendo ad un database siffatto qualora un utente finale, quindi che adoperi indirettamente la struttura dati, dovesse inserire valori inerenti all'acquisto di beni e servizi, il compito del DBMS consiste nell'aggiunta di un record di riferimento in una delle tabelle contenute, incrementando la quantità del prodotto selezionato. Tuttavia, essa rappresenta solo una delle possibili effettività che si possano concretizzare, poichè una realtà differente potrebbe consistere nel malfunzionamento del server in cui si accetti il pagamento del prodotto selezionato, senza elaborare la richiesta effettuata. Per cui occorre gestire le *transazioni*.

Transizione

Una **transizione** rappresenta un'unità di lavoro elementari che modificano il contenuto di una base di dati. La sintassi adoperata da modelli relazionali è posta come segue:

Start Transaction

...

Commit/Rollback work

Le *transazioni* sono comprese da una certa semantica, da una parte iniziale *Start Transaction* e da una sezione conclusiva *Commit/Rollback*. La differenza dei costrutti finali è relativa all'incidenza della transizione all'interno del database; *commit* indica il consolidamento dell'azione su disco qualora terminata la *transizione*, mentre *rollback* provoca la cancellazione di tutto ciò che sia limitato dalla sintassi in questione, per cui ripristina lo stato della base di dati prima del frammento di codice. Come è stato già illustrato una *transazione* pone molta importanza sulla propria sintassi, ma questa centralità è data dalla semantica su cui fonda, suddivisa in diverse caratteristiche, quali: **Trovare spiegazioni delle caratteristiche**

- *Atomicità*

  Definizione
  La transizione deve essere eseguita con la regola del tutto o del niente.

  La definizione pone che in relazione ad una qualsiasi azione adottata la transazione deve

provvedere al ripristino dello stato precedente all'attività modellata, indipendentemente se sia andata a buon fine o meno.

- *Consistenza*

  Definizione
  La transazione deve lasciare la base di dati in uno stato *consistente*, eventuali vincoli di integrità referenziale non devono essere infranti.

- *Isolamento*

  Definizione
  L'esecuzione della transazione deve essere indipendente dalle corrispettive.

- *Persistenza*

  Definizione
  Il commit di una transazione non deve essere perso.

  Rappresenta l'affidabilità della struttura dati, ...

Di seguito, sono illustrati due esempi che implementano i due contraddistinti costrutti finali, descritti come da modello.

```
START TRANSACTION
UPDATE SalariImpiegati
SET Conto = (Conto * 1.2)
WHERE (CodiceImpiegato = 010)
COMMIT WORK;
```

Figure 1: Esempio che implementa il costrutto commit

```
START TRANSACTION
UPDATE SalariImpiegati
SET Conto = (Conto * 1.2)
WHERE (CodiceImpiegato = 010)
IF (Conto > 1000) THEN COMMIT WORK;
ELSE ROLLBACK WORK;
```

Figure 2: Esempio che implementa il costrutto rollback

Un ulteriore elemento fondamentale consiste nella *gestione delle transazioni*. Nonostante, database relazionali sono macchiati dalla mancata possibilità di massimizzazione dei principi elencati precedentemente. Detto ciò, si suddividono due realtà implementative, le quali sono gestioni legate all'*affidabilità*, per cui garantiscono *atomicità* e *persistenza*, oppure gestioni specializzate sulla *concorrenza*, capaci di adottare *isolamento* in caso di esecuzione di più *transazioni*.

Alcuni DBMS apportano nelle loro architetture una *granularità* di concorrenza talmente elevata che si attua per singole celle interpellate per successive operazioni. In tale casistica

MySQL fornisce un'allusione al parallelismo, ma provvede ad una rapida e fulminea *sequenzialità*. Per cui è bene sottolinerare teoricamente il concetto di **schedule**.

Definizione informale
Dato un **insieme di transazioni** T_1, T_2, ..., T_n di cui ciascuna formata da un certo insieme di operazioni di scrittura (w_i) e lettura (r_i), si definisce **schedule** la sequenza di operazioni di lettura/scrittura di tutte le transazioni **cosi come eseguite sulla base di dati**.

Definizione informale
Uno schedule ***S*** si dice **seriale** se **le azioni di ciascuna transazione appaiono in sequenza**, senza essere interrotte da azioni di altre transazioni.

Per cui riassumendo uno *schedule* è detto *serializzabile* se le transazioni sono eseguite sequenzialmente e risultano completamente indipendenti l'una dall'altre. Tuttavia, tale approccio è altamente inefficiente, causando un tempo di attesa estremamente elevato; per cui occorre operare mediante attività parallele, ossia è necessario elaborare contemporaneamente più processi alla volta.

Quanto detto si traduce in *esecuzione concorrente*, la quale potrebbe causare un insieme di problematiche ricorrenti. Nuovamente, con il termine esecuzione concorrente si fa riferimento che più di un processo possano essere eseguiti nello stesso istante. Le avversità più ricorrenti risultano:

- *Perdita di aggiornamento*, avviene qualora transazioni operano su stessi dati, in cui una dell'entità non conclude la propria operatività in relazione ad uno specifico valore, per cui corrisposte elaborano lo stesso come se non fosse stata attuata alcuna modifica.
- *Lettura sporca*, avviene qualora utilizzato il costrutto *rollback*, dove nuovamente, una transazione operante su certi dati non concluda la propria attività, tale che corrisposte analizzino un valore viziato, il quale non dovrebbe concretamente subire alcuna variazione.
- *Lettura incosistente*, avviene qualora effettuato un *read* precedente ad una modifica che deve essere riferita su disco
- *Aggiornamento fantasma*, avviene qualora si operi su valori modificati da transazioni corrisposte, adoperando *read* su risultati non corretti

Concludendo, quindi si vuole realizzare uno *schedule* in grado di porre transazioni in maniera parallela e adottare meccanismi di controllo della concorrenza. Ciò è possibile tramite l'utilizzo dei *lock*.

Gestione della concorrenza

Per poter effettuare una qualsiasi operazione di lettura oppure di scrittura, è necessario aver precedentemente acquisito il **lock di riferimento**. Inoltre, vengono distinti due tipologie di lock, *lock di lettura* consente un accesso condiviso, per cui da parte di più processi, sulla stessa ricorsa in questione, e *lock di scrittura*, i quali ammettono *mutua esclusione*, ossia un

singolo elemento può avere il pieno controllo del dato desiderato.

I DBMS hanno a disposizione *lock manager* per poter fronteggiare alle problematiche poste tra transazioni, il quale ammette due singole azioni, ossia di *richiesta* oppure di *rilascio* del lock. Il compito del *lock manager* prevede di gestire i *lucchetti* riferite alle risorse contenute nel database e di rispondere adeguatamente alle richieste delle transazioni. Ciò avviene considerando un trio di strutture dati, poste in favore ad una risorsa x appartenente alla base di dati, quali:

- *State(x)*, rappresenta lo stato della risorsa, la quale può essere (Libero / w_lock / r_lock), dove rispettivamente la prima condizione indica che il dato richiesto è libero di essere manipolato, la seconda pone mutua esclusione poichè adoperato per azioni di modifica, infine la terza rappresenta un lock di lettura, il quale può essere condiviso ma non attuato per promuovere variazioni
- *Active(x)*, raffigura la lista contenente le transizioni *attive* sulla risorsa in questione
- *Queued(x)*, raffigura la lista contenente le transizioni *bloccate* sulla risorsa in questione

Davanti ad una richiesta, il *lock manager* esegue diversi step:

- Riceve una richiesta (r_lock / w_lock / unlock) da una transazione T, su un oggetto x
- Controlla la *tabella stato/azione*
- Se la risposta è *OK*, aggiorna lo stato della risorsa, e concede il controllo della risorsa alla transazione T
- Se la risposta è *NO*, inserisce la transazione T in una coda associata ad x, per cui aggiungendola alla lista *Queued*

Stabilito il comportamento generale su cui fonda un manager di lock, DBMS relazionali tipicamente adottano due tipologie contraddistinte di meccanismi per manipolare concorrenza tra transazioni.

Two Phase Lock
Il lock manager *Two Phase Lock*, non permette a transazioni che abbiano rilasciato i meccanismi di mutua esclusione, di acquisirne ulteriori. Per cui il paradigma implementato richiede che un qualsiasi processo operante debba prima apprendere tutti i lock necessari per la manipolazione delle risorse desiderate. Ogni schedule che adegua il 2PL è *serializzabile* e non incorre in effettività erronee dovute ad *aggiornamenti fantasma*, *letture inconsistenti* oppure *perdite di aggiornamenti*; tuttavia è ancora presente il problema della *lettura sporca*.

Strict Two Phase Lock
In un *Strict Two Phase Lock*, i lock di una transazione sono rilasciati solo dopo aver effettuato le operazioni di commit/rollback. Uno schedule che rispetta il S2PL eredita tutte le proprietà del 2PL, ed inoltre non presenta anomalie causate da problemi di *lettura sporca*.

Nonostante i due protocolli permettano di ovviare a problematiche legate all'esecuzione concorrente, sono accomunati da una specifica avversità, in certe circostanze potrebbero generare schedule con situazioni di *deadlock*. Per gestire le situazioni di deadlock causate dal lock manager, si attuano tre tecniche:

- *Uso dei timeout*, ogni operazione di una transazione ha un timeout entro il quale deve essere completata, pena annullamento della transazione stessa, ossia si adopera *rollback* nella sezione conclusiva
- *Deadlock avoidance*, prevenire le configurazioni che potrebbero portare ad un deadlock. Si può prevenire in due modi: attraverso l'utilizzo di lock/unclock su tutte le risorse allo stesso tempo oppure attraverso l'utilizzo di *time stamp* o di *classi di priorità* tra transazioni, nonostante possano provocare *starvation*
- *Deadlock detection*, utilizzare algoritmi per identificare eventuali situazioni di deadlock, e prevedere *meccanismi di recovery* dal deadlock. Si possono identificare attraverso l'utilizzo di *grafi delle risorse*, utilizzato per identificare la presenza di cicli. In caso fosse accertata la presenza, si attua *rollback* delle transazioni coinvolte nel ciclo in modo da eliminare la duplice dipendenza

...

...