PROGETTO 2 BASI DI DATI 2

Componenti gruppo:

Magno Alessandro 4478234

Software Utilizzati:

-PostgreSQL

-Eclipse IDE

Fonti:

-Libro: Sistemi di Gestione Dati

-Documentazione online Postgres, Oracle

-Soluzioni Laboratorio di Basi di Dati 2

-Apache Tomcat

Punto 1: Comprensione dello strumento

Postgres offre varie modalità di lock per controllare l'accesso simultaneo ai dati nelle tabelle. Alcune di queste modalità di lock vengono acquisite automaticamente da Postgres prima dell'esecuzione dell'istruzione, mentre altre vengono fornite per essere utilizzate dalle applicazioni. Tutte le modalità di lock (ad eccezione di AccessShareLock) acquisite in una transazione vengono mantenute per la durata della transazione. Oltre ai lock, vengono utilizzati i latches condivisi / esclusivi a breve termine per controllare l'accesso in lettura / scrittura alle pagine della tabella nel pool buffer condiviso. I latch vengono rilasciati immediatamente dopo il recupero o l'aggiornamento di una tupla.

Per quanto riguarda il lock escalation non viene supportato da Postgres e i livelli di isolamento vengono implementati utilizzando la snapshot isolation. Come risultato, i livelli di isolamento sono più restrittivi rispetto a quanto proposto dallo standard.

READ COMMITTED (= READ UNCOMMITTED)

- Come standard per quanto riguarda i lock ma in più utilizza snapshot isolation a livello di singolo comando (e restringe quindi quanto previsto dallo standard): ogni comando SQL viene eseguito sullo snapshot creato dalle transazioni committed fino a quel momento.
- Letture irripetibili e phantoms sono possibili a livello di transazione ma non a livello di singolo comando SQL.
- Serializzabilità non garantita

REPEATABLE READS

- Snapshot isolation ristretta a livello di transazione (e restringe quindi quanto previsto dallo standard): ogni transazione viene eseguita sullo snapshot creato dalle transazioni committed fino a quel momento e non può acquisire lock o modificare dati aggiornati da altre transazioni, dopo il suo inizio.
- Se due transazioni modificano lo stesso dato, e la prima effettua commit, la transazione che lo modifica dopo viene abortita e rieseguita.
- Nessuna anomalia possibile.
- Serializzabilità non garantita (snapshot isolation non la garantisce).

SERIALIZABLE

- Come REPEATABLE READS ma assicura serializzabilità introducendo verifiche aggiuntive (non cambia nulla in termini di lock ma se la serializzabilità non viene garantita, viene abortita la transazione).
- Maggiore overhead.

Punto 2: Transazioni in Java e JDBC

Premessa: le transazioni sono state eseguite su 90 conti (sia per questo punto, sia per quello successivo), perché il mio Postgresql server è stato configurato per accettare al massimo 90 connessioni e non sono riuscito ad incrementare questo massimo. L'errore generato con più di 90 connessioni è il seguente: FATAL: sorry, too many clients already. Inoltre, per effettuare connessioni distinte, ho utilizzato un pool di connection situato in una libreria esterna (org.apache.tomcat.jdbc.pool) che lascio in allegato.

1. **Transizioni singole**

   Il programma ritorna il seguente output:

   ```
   SQLState: 25P01
   Code: 0
   Message: Cannot rollback when autoCommit is enabled.
   ```

   Soluzione: Nel caso in cui l'autocommit viene settato a true (default), ogni comando viene eseguito come una singola transazione e quindi i valori vengo inseriti nella base di dati tranne l'ultimo, in quanto porta ad una violazione del vincolo di integrità. Se l'autocommit è settato a true, non è può effettuare il rollback.

   Nel caso in cui l'autocommit viene settato a false, tutti i comandi SQL compresi tra le istruzioni con.setAutoCommit(false) e con.close() costituiscono un'unica transazione, questo implica che vengono eseguiti tutti o non ne viene eseguito alcuno. In questo caso la transazione viene abortita, in quanto l'ultima istruzione viola il vincolo di integrità e la base di dati rimane vuota.

   In PostgreSql, per impostazione predefinita, i programmi SQL incorporati non sono in modalità di autocommit, pertanto COMMIT deve essere emesso esplicitamente quando lo si desidera.

2. **Transizioni concorrenti**

   Soluzione: Eseguendo il programma con input 100 100, creo 100 transazioni e un pool di 100 threads, dove ogni thread ne esegue una in maniera concorrente. Mentre nel secondo caso con input 100 1, ogni transazione viene eseguita una per volta in maniera sequenziale, risultando più lenta che la prima.

   Generando una connessione distinta per ogni thread, si simula l'accesso da parte di 100 utenti al database. Ciò comporta che il numero di transazioni eseguite con un'unica connessione è una, mentre nel caso di connessioni distinte sono 100.

   Per dimostrare i risultati, è stato eseguito un inserimento che viola il vincolo di chiave primaria. Nel primo caso (unica connessione) non è stata inserita alcuna tupla perché il pool di thread è stato considerato come un'unica transazione e siccome ha effettuato il rollback la transazione non è andata a buon fine. Nel secondo caso (connessioni distinte) è stata inserita la una tupla perché il pool di thread è stato considerato come 100 transazioni, di cui la prima terminata con

successo. Threads sono riferiti ai processi mentre le transazioni sono riferite ai dati.

3. **Isolamento**
   Soluzione: Il conto 0 non è detto che contenga 0 perché eseguendo le transazioni concorrentemente, può succedere che vengano letti valori non aggiornati. Per spiegare meglio, riporto un esempio di una possibile esecuzione (per semplificare utilizzo solo 2 transazioni):
   -transazione 1: acquisisce il lock sulla select -> R(balance) = 90
   -transazione 1: rilascia il lock e setta -> c = 90
   -transazione 2: acquisisce il lock sulla select -> R(balance) = 90
   -transazione 2: rilascia il lock e setta -> c = 90
   -transazione 1: acquisisce il lock sulla update
   (aggiorna balance = 90-1) e rilascia il lock
   -transazione 2: acquisisce il lock sulla update
   (aggiorna balance = 90 -1) e rilascia il lock

   In quest'ultimo passo si può vedere come la transazione 2 esegue l'update su un valore non aggiornato di balance.

   Il livello di isolamento impostato di default è il Read Committed, il quale permette letture inconsistenti e anomalia di phantom row.

   Manuale di postgresql, riporta che: questo livello di isolamento avvia ciascun comando con un nuovo snapshot che include tutte le transazioni committed fino a qull'istante, i comandi successivi nella stessa transazione vedranno comunque gli effetti della transazione concorrente committed. Il punto sopra indicato è se un singolo comando vede o meno una visione assolutamente coerente del database.

4. **Isolamento variazione**
   Soluzione: Eseguendo la transazione, si ottiene il conto 0 contente il valore 0 ed i conti da 1 a 100 contenti valore 1. Questo succede perché, tenendo presente che il livello di isolamento di default è Read Committed, ogni comando influisce solo su una riga predeterminata, quindi lasciare che veda la versione aggiornata della riga non crea alcuna incoerenza.

5. **Isolamento: modifiche**
   Soluzione: riseguendo il codice dell'esercizio 4 con il livello di isolamento read uncommitted si ottengo gli stessi risultati per il motivo spiegato precedente al punto 4. Il livello di isolamento read committed

è il default, spiegato al punto 4. Mentre per il livello di isolamento repeatable read si ottiene il seguente messaggio di errore (ERROR: could not serialize access due to concurrent update) perché una transazione con livello repeatable read non può modificare o bloccare le righe modificate da altre transazioni dopo il suo inizio.
Infatti, questo livello è diverso dal read Committed in quanto una query in una transazione repeatable read vede uno snapshot all'inizio della transazione, non all'inizio della query corrente all'interno della transazione. Il manuale di postgresql, per risolvere questo problema, consiglia, alle applicazioni che utilizzano questo livello, di essere preparate a riprovare le transazioni a causa di errori di serializzazione.
Con il livello di isolamento serializable si ottiene anche in questo un messaggio di errore (ERROR: coud not serialize access due to read/write dependencies among transactions) dovuto a errori di serializzazione.
In effetti, come riporta il manuale, questo livello di isolamento funziona esattamente come il precedente, tranne per il fatto che monitora le condizioni che potrebbero rendere l'esecuzione di un insieme concorrente di transazioni serializzabili comportarsi in modo incoerente con tutte le esecuzioni seriali di tali transazioni. Questo monitoraggio non introduce alcun blocco oltre a quello presente nel livello repeatable read, ma c'è un sovraccarico nel monitoraggio e il rilevamento delle condizioni che potrebbero causare un'anomalia di serializzazione provocherà un errore.

Rieseguendo il codice dell'esercizio 3, per i livelli read uncommitted e read committed(default) vale la spiegazione al punto 3, perché anche il livello read uncommitted permette letture inconsistenti. Per i livelli repeatable read e serializable si ottengono gli stessi messaggi di errore descritti in precedenza.

Punto 4: Tuning del livello di isolamento

-Pseudocodice

-T_1

W_1 (balance ← balance + c)

R_1 (branch ← account)

W_1 (bblance ← bblance + c)

-T_2

R_2 (balance ← account)

-T_3

R_3 (bblance ← branch)

R_3 (sum(balance) ← account)

-Livello di isolamento

Secondo me, per la prima transazione può essere impostato il livello di isolamento repeatable read perché essendoci update e select che vengono eseguite in maniera concorrente, si possono evitare problemi di letture inconsistenti. La seconda e la terza eseguendo soltanto select, può essere impostato come livello di isolamento il read committed, in quanto la loro esecuzione non dovrebbe incidere sulle update della prima transazione.

Il tempo di esecuzione in millisecondi è 108 ms, ok è uguale a 5. Le transazioni vengono eseguite con successo, aggiornando le tabelle con i valori corretti. Inoltre, non si ottengono errori di nessun tipo con il livello di isolamento repeatable read, il quale è stato impostato, utilizzando un savepoint, per riprovare le transazioni contenenti update come specificato nel manuale.

Per quanto riguarda altre alternative dei livelli di isolamento, si può ovviamente utilizzare il serializable, il quale, però, aumenta il tempo di esecuzione. Facendo alcuni test, ho notato che la seconda e la terza transazione, si può settare il livello repeatable read; che porta il tempo di esecuzione a 110 ms. Inoltre sempre queste ultime possono essere settate al livello read uncommitted, con tempo di esecuzione pari a 105 ms. Questo ultimo test, mi fa pensare che il loro comportamento sia indipendente dal livello di isolamento.