CSE 5331 DBMS MODELS AND IMPLEMENTATION

11/2/2015

Transaction Manager and Deadlock detector

Rajaraman Govindasamy
1001165700
Team 5

Contents

Core Concept

The project mainly focus on implementing the transaction manager and deadlock detection. A transaction is an event which occurs on the database. Generally a transaction reads a value from the database or writes a value to the database. A read operation does not change the image of the database in any way. But a write operation, whether performed with the intention of inserting, updating or deleting data from the database, changes the image of the database. Any transaction has the following four properties *Atomicity, Consistency, Isolation, and Durability* collectively called as the ACID properties. The transaction can be of the following states, *ACTIVE, PARTIALLY COMMITTED, FAILED, ABORTED, COMMITTED, and TERMINATED*.

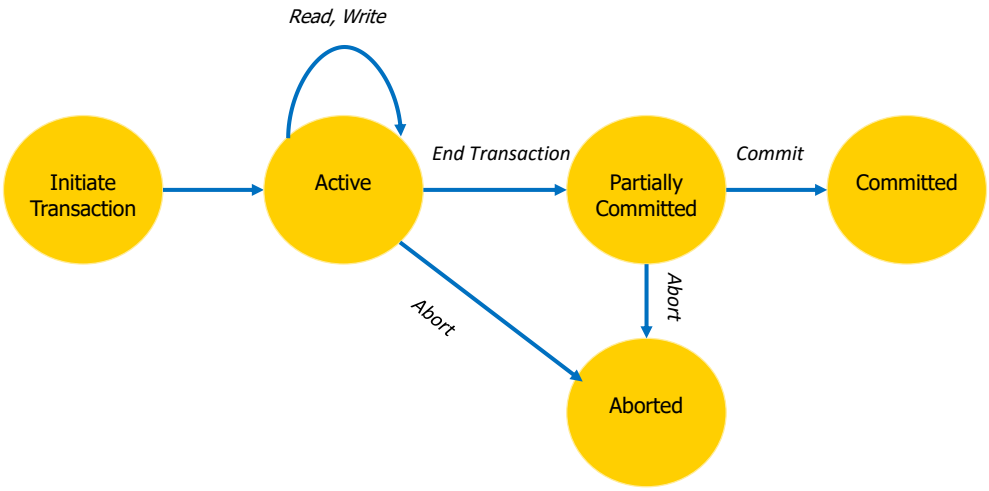

Figure 1: Transaction States

The transaction can be executed in different schedules or a collection of many transactions implemented as a unit. Depending upon the arrangements, a schedule can be of two types, serial and concurrent execution.

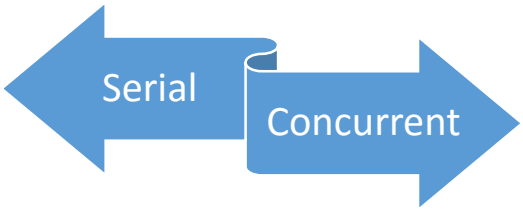

Figure 2: Serializability

The transactions are executed one after another, in a non-preemptive manner in serial mechanism and transactions are executed in preemptive, time shared manner in concurrent mechanism. When these transactions are trying to access the same data item, the instructions within the concurrent transactions must be ordered in some way so as there are no problem in accessing and releasing the shared item. This gives rise to address the conflict serializability problem, two instructions of two different transactions may want to access the same data item in order to perform a read/write operation. Conflict Serializability deals with detecting whether the instructions are conflicting in any way, and specifying the order in which these two instructions will be executed in case there is any conflict. A conflict arises if at least one (or both) of the instructions is a write operation.

Important rules for conflict serializability:

1. *If two instructions of the two concurrent transactions are both for read operation, then they are not in conflict, and can be allowed to take place in any order.*
2. *If one of the instructions wants to perform a read operation and the other instruction wants to perform a write operation, then they are in conflict, hence their ordering is important. If the read instruction is performed first, then it reads the old value of the data item and after the reading is over, the new value of the data item is written. It the write instruction is performed first, then updates the data item with the new value and the read instruction reads the newly updated value.*
3. *If both the transactions are for write operation, then they are in conflict but can be allowed to take place in any order, because the transaction do not read the value updated by each other. However, the value that persists in the data item after the schedule is over is the one written by the instruction that performed the last write.*

When a process indefinitely waits for a resource that is held by another process, then that situation is called deadlock. This leaves the transaction manager to either rollback or restart.

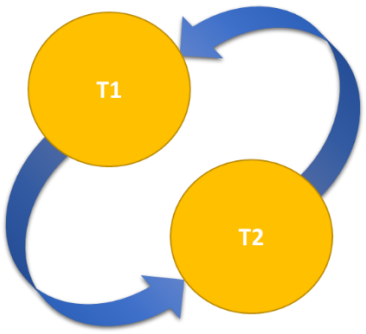

Figure 3: Deadlock

Overall Status

My basic approach towards this problem was top down starting from the analysis of the given base code which was needed to be in sync with the nomenclature and understand the coding complexity of the existing system. Understanding the flow of the code was really challenging where traversing through the zgt_tm, zgt_tx files for implementing the methods and reference to data structures in the header files were main focus to begin with the implementation. I started with the phase one for implementing the transaction manager where the major components were TxRead(), TxWrite(), CommitTx(), AbortTx() in zgt_tm.C file which calls the actual implemented methods readtx(), writetx(), aborttx(), committx() in zgt_tx.C file.

File Name: zgt_tx.C	
readtx()	This method is used to perform the read operation on a transaction. The read operation checks the lock status of the object using the set_lock() method and if not being used by any other transaction then performs the read operation after locking the object.
writetx()	This method is used to perform the write operation on a transaction. The write operation checks the lock status of the object using the set_lock() method and if not being used by any other transaction then performs the read operation after locking the object.
aborttx()	This method is used to perform the abort operation on a transaction. The abort operation releases all the locks using free_locks() method held by the transaction.
committx()	This method is used to perform the commit operation on a transaction. The commit operation also release all the locks using free_locks() method held by the transaction.
do_commit_abort()	This method performs the actual commit and abort operations.
set_lock()	This method locks the transaction object before doing any operation. This also checks if a object is locked or not.

File Name: zgt_tm.C	
TxRead()	This method is used to perform the read operation on a transaction. This method creates a thread and calls the readtx() method in zgt_tx.C file to perform the read operation.
TxWrite()	This method is used to perform the write operation on a transaction. This method creates a thread and calls the writetx() method in zgt_tx.C file to perform the read operation.

AbortTx()	This method is used to perform the abort operation on a transaction. This method creates a thread and calls the aborttx() method in zgt_tx.C file to perform the read operation.
CommitTx()	This method is used to perform the commit operation on a transaction. This method creates a thread and calls the committx() method in zgt_tx.C file to perform the read operation.

The above all methods were implemented. The methods were similar be implemented as the operation was varying between the read and write, and the same way for commit and abort. The second phase was the deadlock detection, the deadlock detection was very challenging and the same drained lot of time and work for analyzing. The general approach to traverse a graph and visiting a node was implemented in the traverse method as part of deadlock to execute. The transaction manager works fine with zero errors and runs n times in with all the test files using tmtest command.

Difficulty

The implementation of the thread concept dealing with semaphore P & V operations were quiet interesting as well challenging. Also, the deadlock detection was very challenging and the same drained lot of time and work for analyzing The initial setup was time consuming and the parameters setup in the make file was required to successfully compile the given code. The tmtest file required a change in file in the execution command, after fixing that the same was running fine and was producing results for n runs.

File descriptions

No new files were created.

Division of Labor

NA

Logical Errors

Below are few logical errors that I handled as part of the transaction manager implementation.

1. **Locking the transaction object**

   Locking a transaction was quiet challenging, finding the object and then proceeding to set lock only if the object is not being used by other objects. There were multiple times the proper object was not getting locked bringing the whole program to hung state.

2. **Conflict Serializability**

   The test files except conflict serializability was not executing and went to hung state during transaction manager implementation. This was because the current transaction object was not passed over the v operation which was causing that issue. After passing the current object the same was fixed and the transaction was properly released.

3. **Thread concept**

   The thread concept dealing with semaphore P & V operations were challenging. There were many instances where complete understanding of even the small functions were necessary including the header files.

4. **File execution**

   Execution of the test files for n runs was throwing error as the tmtest file was having the different file instead of zgt_test. On resolving, I was able to test the files for n runs successfully.

Successful Output:

rxg5700@omega:~/FALL_2015_project_2_code_given/src

```
EXECUTION:  4963
EXECUTION:  4964
EXECUTION:  4965
EXECUTION:  4966
EXECUTION:  4967
EXECUTION:  4968
EXECUTION:  4969
EXECUTION:  4970
EXECUTION:  4971
EXECUTION:  4972
EXECUTION:  4973
EXECUTION:  4974
EXECUTION:  4975
EXECUTION:  4976
EXECUTION:  4977
EXECUTION:  4978
EXECUTION:  4979
EXECUTION:  4980
EXECUTION:  4981
EXECUTION:  4982
EXECUTION:  4983
EXECUTION:  4984
EXECUTION:  4985
EXECUTION:  4986
EXECUTION:  4987
EXECUTION:  4988
EXECUTION:  4989
EXECUTION:  4990
EXECUTION:  4991
EXECUTION:  4992
EXECUTION:  4993
EXECUTION:  4994
EXECUTION:  4995
EXECUTION:  4996
EXECUTION:  4997
EXECUTION:  4998
EXECUTION:  4999
Removing Shared Memory, Semaphores and Messages...
User : rxg5700
Removing SEMAPHORE : 1304690714
No SHARED MEMORY to remove.
No MESSAGE(s) to remove.
...Done...
[rxg5700@omega src]$
```

rxg5700@omega:~/FALL_2015_project_2_code_given/src

```
EXECUTION:  2463
EXECUTION:  2464
EXECUTION:  2465
EXECUTION:  2466
EXECUTION:  2467
EXECUTION:  2468
EXECUTION:  2469
EXECUTION:  2470
EXECUTION:  2471
EXECUTION:  2472
EXECUTION:  2473
EXECUTION:  2474
EXECUTION:  2475
EXECUTION:  2476
EXECUTION:  2477
EXECUTION:  2478
EXECUTION:  2479
EXECUTION:  2480
EXECUTION:  2481
EXECUTION:  2482
EXECUTION:  2483
EXECUTION:  2484
EXECUTION:  2485
EXECUTION:  2486
EXECUTION:  2487
EXECUTION:  2488
EXECUTION:  2489
EXECUTION:  2490
EXECUTION:  2491
EXECUTION:  2492
EXECUTION:  2493
EXECUTION:  2494
EXECUTION:  2495
EXECUTION:  2496
EXECUTION:  2497
EXECUTION:  2498
EXECUTION:  2499
Removing Shared Memory, Semaphores and Messages...
User : rxg5700
Removing SEMAPHORE : 1305968663
No SHARED MEMORY to remove.
No MESSAGE(s) to remove.
...Done...
[rxg5700@omega src]$
```

rxg5700@omega:~/FALL_2015_project_2_code_given/src

```
T2            Committx

------------------------------------------------------------
TxId    Txtype  Operation       ObId:Obvalue:optime     LockType        Status          TxStatus
T1      W       BeginTx
T1              ReadTx          1:-1:277                ReadLock        Granted          P
T2      W       BeginTx
T1              ReadTx          2:-1:277                ReadLock        Granted          P
T2              WriteTx         5:0:235                 WriteLock       Granted          P
T2              WriteTx         6:1:235                 WriteLock       Granted          P
T2              ReadTx          6:0:235         ReadLock        Granted          P
T1              WriteTx         3:1:277                 WriteLock       Granted          P

T2            Committx
T1              WriteTx         4:1:277                 WriteLock       Granted          P
T1              ReadTx          1:-2:277                ReadLock        Granted          P
T1              WriteTx         2:0:277                 WriteLock       Granted          P
T1              WriteTx         4:2:277                 WriteLock       Granted          P
T1              WriteTx         4:3:277                 WriteLock       Granted          P

T1            Committx
T2              ReadTx          5:-1:235                ReadLock        Granted          P

------------------------------------------------------------
```