**Project 2**

**Implementation Of Transaction Manager**

(Course : Database Models & Implementation Techniques, CSE:5331-001)

*Submitted By,*

Group : 23

Group members :

Krishna Sankaramani Ramamoorthie

(UTA ID : 1001980344)

Archana Prakash Nikam

(UTA ID : 1001912520)

*Under guidance of,*

Prof. Sharma Chakravarthy

## TA: Billah, Umme Hafsa

T**able of Contents**

1. Introduction

2. Overall status of the project

3. Encountered Difficulties

4. File Description

5. Division of Labor

6. Handling of logical errors

7. Given input & output description

8. Analysis

9. References

**1. Introduction**

In a database management system, concurrency control is a method of controlling concurrent processes without causing conflicts. It guarantees that database operations are carried out reliably and concurrently to achieve the desired results without compromising the related Database's data integrity.

Below are the two types of locks used in locking protocol

**1. Shared Lock :** The data item can be shared between transactions when using the shared lock. A shared lock is also called as a read only lock.

**2. Exclusive lock :** An exclusive lock allows both reading and writing of a data item. This is mutually exclusive and cannot be held on the same data item concurrently. After the "write" transaction is complete, transactions may unlock the data item.

2. Overall status of the project :

In this project we have implemented a transaction manager that manages concurrency control using locking. We have implemented the strict two-phase locking (S2PL) protocol with shared locks for read and exclusive locks for write. The transaction manager handles locking and releasing of objects. Lock escalation i.e upgrading is not considered in this project. We have implemented a project using C. We have already given a skeletal code by professor.

Below are function implemented,

begintx(thrdArguments)

readtx(thrdArguments)

writetx(thrdArguments)

aborttx(thrdArguments)

committx(thrdArguments)

3. File description

We haven’t added any new files. We have implemented the code in zgt\_tm.c and zgt\_tx.c. The necessary operations required for transaction manager are coded in both the files. Our goal was to successfully execute each test case given along with project. These operations are implemented as below. Explanation of each method is given as below.

* **void \*begintx(void \*arg) :** In this method, intialised a transaction object. It is done after acquiring the semaphore for the tm and so that operation can proceed using the condition variable. When creating the tx object, we set the tx to TR\_ACTIVE and obno to -1, there is no semno as yet as none is waiting on this tx.
* **void \*readtx(void \*arg) :** In this method, we are starting the operation with mutex lock & getting atransaction id. We are checking a transaction status, weather it is in active, wait, commit or abort mode. And then we will call do\_commit\_abort() method. When transaction is active set transaction lock with 'S' lockmode for share memory lock. However if transaction is null we will display the error. We are using a switch case functionality to check eack case and its output.
* **void \*writeTx(void \*arg) :** This method is similar to readTx method as above using a switch case.We are checking a status of each transaction status as active, wait, commit, abort and then And then we will call do\_commit\_abort() method. When transaction is active setting transaction lock with 'X' lockmode for exclusive lock.
* **void \*abortTx(void \*arg) :** In this method we are starting an operation with mutex lock and getting a transaction id. If transaction is not null we will call do\_commit\_abort() method. Then finishing the operation with mutex lock.
* **void \*commitTx(void \*arg) :** Similar to abort method, in this method we are changing the status of current transaction to commit and calling a method do\_commit\_abort which handles the freeing of locks which are held by the transaction and remove transaction from Transaction Manager.
* **void do\_commit\_abort() :** By using this method, the lock is released and the transaction is deleted from the transaction manager. The methods Abort() and Commit() both uses do\_commit\_abort() method. It also checks the wait queue to see whether there are any additional transactions that need to be called because the current transaction has put them in waiting mode. Using zgt\_nwait(), we get a list of the transactions that are semaphore waiting and put it in a variable. The semaphore's zgt v() function is then used to sequentially release each semaphore. In the event of an exclusive lock, this will enable those transactions to apply to the released objects.
* **int set\_lock() :** This method determines when to give a transaction lock on a certain object. If the requested object already exists, we attempt to get the transaction that holds the lock as the object owner. Should that happen, the current transaction may obtain the lock to the object. If a transaction has the lock already, we must determine whether that transaction is the one that needs the lock in the first place. If so, the lock has already been granted to it. In the event that we surpass the Exclusive mode transaction, there is a possibility that the transaction that is awaiting Exclusive mode may be forced to wait indefinitely in the event that Shared lock inquiries keep coming in. Additionally, because another transaction already holds the object, the current transaction must add its node to the hash table for the same object and, assuming this is the first item it requires, refer the head of the transaction to this object node on the hash table. If it already points to other objects, it should iterate to the transaction's final object node and add the new item to the end of its list of objects. This transaction must wait if, for any reason, it is unable to obtain the lock. It changes to "W" status and sets a semaphore on the transaction that is waiting; for example, if transaction 2 is awaiting an object from transaction 1 then transaction 1's semaphore changes to "1". For the duration of the lock's acceptance, the given conditions are checked continuously in a loop. The process is completed by returning to read Tx or write Tx from where it was called after the lock has been given.
* **void perform\_readWrite() :** In this method, it updates the objarray[] value by incrementing 1 for write and decrementing 1 for read.

3. Encountered difficulty :

One of the laborious tasks in our project was figuring out the concurrency control concept's relationship to the semaphore logic and thread flow. For best effectiveness, several factors have to be taken into account when acquiring and releasing locks. It was challenging and confusing because there were several executions. I was quite challenging to debug. Due to thread notions, it was difficult to debug code while the console was overflowing with logs. It took some time for us to begin capturing the flow of code. It took some time to realize that the objarray contains the value that will be altered upon read or write.

4. Division of labor :

|  |  |  |
| --- | --- | --- |
| **Sr No** | **Name** | **Task worked on** |
| 1 | Sankaramani Ramamoorthie, Krishna | 1. Implemented read, write, set lock transaction operations.  2. Worked on debugging & resolving logical errors. |
| 2 | Archana Prakash Nikam | 1. Implemented abort and commit transaction operations.  2. Worked on test cases and documentation part. |

5. Handling of logical errors:

* Encountered segmentation fault due to the program tries to access a memory location that is not allowed to access. This error was solved by careful execution of pointer in the code.
* Hanging due to lock conditions and error because of transaction holding incorrect transaction values. We where able to solve this logical error by debugging and analyzing the code
* Read write into logfile since the output where not correctly written into the logfile. We where able to solve it by implementing the correct pointers value to the respective locations.

6. Given input & output description

Input:

The input has 3 columns the first is the type of transaction, second is the transaction number and the third is the value to read or write.

Output:

The output is a logfile where each row explains the transaction process and the columns are transaction id, transaction type, the type of operation, the object Id,value, and optime, type of Lock, and the transaction status.

7. Analysis

Input no\_conflists\_2Txs.txt

Running zgt\_test the test works

Running tmtest 100,1000,5000 there was no hanging the test completed successfully all the executions.

Input Interleaved\_RW.txt

Running zgt\_test the test works but in the logfile the updates are overwritten by the upcoming transactions Hence it experiences the lost update problem of concurrency problem.

Running tmtest 100,1000, 500 there was no hanging the test completed successfully all the executions.

Input Multi\_ROTxs.txt

Running zgt\_test the test works since there only read only transactions so they can be shared by the transactions

Running tmtest 100,1000, 500 there was no hanging the test completed successfully all the executions.

Input disj\_multi\_accesses.txt

Running zgt\_test the test works for the read and write of the 1st transaction and completes it by committing and computed the 2nd transaction.

Running tmtest 100,1000, 500 there was no hanging the test completed successfully all the executions.

Input ddlk\_2Txs.txt

Running zgt\_test the test hangs due to deadlock because both of the transactions are waiting on the other.

Running tmtest 100,1000, 500 there was no hanging the test completed successfully all the executions.

Input ddlk\_3Txs.txt

Running zgt\_test the test hangs due to deadlock because the three transactions are waiting on one another.

Running tmtest 100,1000, 500 there was no hanging the test completed successfully all the executions.

Input test\_abort.txt

Running zgt\_test the test works for the read and write of the transactions and aborts successfully and wipes out all its effects.

Running tmtest 100,1000, 500 there was no hanging the test completed successfully all the executions.

Input RW\_disjoint.txt

Running zgt\_test the test works as running multiple read write transactions with no deadlock

Running tmtest 100,1000, 500 there was no hanging the test completed successfully all the executions.

Input RW\_pot\_ddlk.txt

Running zgt\_test the test works as running multiple read write transactions with no deadlock

Running tmtest 100,1000, 500 there was no hanging the test completed successfully all the executions.

Input unlikely\_ddlk.txt

Running zgt\_test the test works the transaction is serializable and executed successfully.

Running tmtest 100,1000, 500 there was no hanging the test completed successfully all the executions.

7. References

1. cse4331-5331\_fall-22project2\_v2.pdf

2. TxMgr-proj2\_help\_v1.pdf

3. Lecture slides

4. Book : Ramakrishnan - Database Management Systems 3rd Edition.pdf