CSE-5331-001 DBMS Models and Implementation

Project – 1

Two Phase Locking Protocol with cautious waiting and wait die method

Student Name: Surabhi Ingole Student ID: 1001875174

Student Name: Mehek Thaker Student ID: 1001869277

Abstract:

In this project, Two-Phase Locking Protocol with cautious waiting is implemented by Mehek Thaker and Two-Phase Locking with wait die method is implemented by Surabhi Ingole. We are using Hash Map as the data structure and plain-old java objects for defining the structure of Lock table and Transaction table. For input, we will be using .txt files and output will be displayed in the text format specifying what is happening at each line of the transaction passed as an input to the program. This simulation will perform the basic operations of begin, read, write, and end. It has different methods to execute these operations and to implement the blocking, aborting, and committing of transaction.

Programming language used: JAVA (SE version 16)

IDE used: Eclipse IDE

Files included:

Final_Documentation.docx

Main.java, Lock.java	Transaction.java,
TransactionManager.java	LockManager.java,
Operation.java	RigorousTwoPhaseLocking.java
input1.txt	output1.txt
input2.txt	output2.txt
input3.txt	output3.txt
input4.txt	output4.txt
input5.txt	output5.txt
input6.txt	output6.txt

Directions to execute the file:

Please install Eclipse IDE.

Currently, we have given input-1.txt as an input file. To give another input file, open the Main.java file and change the filename **String fileName = "./inputFiles/input1.txt"** (line 17) in both the programs.

To run the program: Please click on the run button.

Output will be in the same format shown in the respective output file attached.

Pseudocode for Two-Phase Locking Protocol with cautious waiting method :

```
public static void main(){

        while(line != EOF) {
                if (operation == 'b') {
                        increment timestamp by 1;
                        Call Begin Function;}
                if (operation == 'r' || operation == 'w') {
                        Call Request Function;}
                if (operation == 'e') {
                        Call Commit Function;}
        }
}

public static void beginTransaction(){
        Create an object of transaction class;
        Set the status to "Active";
        Add the transaction to the hash table;
}

public static void requestLock(){
        Check the operation for read or write;
        Fetch the transaction from hash table based on transaction id;

                if (state == "Active")) {
                        Call Active method;
                } else if (state == Block")) {
                        Call Block method;
                } else if (transaction state == "Abort")) {
                        Print the state;
                } else if (transaction state == "Commit")) {
                        Print the state;
                }
}

public static void commitTransaction(){
        If (state == "Active){
                Set the state of the transaction to "Commit";
                Remove the locks that are present currently on the transaction;
```

```
                Update the transaction hash map for this record;}
} else if (state == “Block")) {
        Add the remaining operations to the waiting list;
        Update the record in transaction hash table;

} else if (state == “Abort")) {
        Print the log;
                }
}

public static void abortTransaction(){
        Set the transaction state to Abort;
        Release all the locks acquired by the transaction;
        Update the record in hash table;
}

public static void blockTransaction(){
                if (dataItem is already in lockTableMap) {
                        Fetch the dataItem;
                        Add the transaction id to the waiting list;
                        Update the lock hashtable;
                }
                if (dataItem not in lockTableMap){
                        Add the dataItem to the lock hash table;
        }
                Add the operation to the waiting list;
                Update the transaction hash table;}

public static void activeTransaction() {
                if (dataItem is already in lockTableMap)) {
                        Fetch the dataItem from the Lock hash table;
                        if (lockState == "Read" && operation == "Read") {
                                Call read_read method;
                        } else if (lockState == “Write" && operation == “Read") {
                                Call write_read method;
                        } else if (lockState == “Read" && operation == “Write"{
                                Call read_write method;
                        } else if (lockState == “Write" && operation == “Write") {
                                Call write_write method;
                        } else if (lockState == “” && operation == "Read") {
                                setLockState to "Read";
                                Add the transaction id to the readLockTransId list;

                        } else if (lockState == "" && operation == "Write") {
                                setLockState to "Write";
                                Set the writeLocktransactionId with the transaction id;
                        }
                        Add the record to lock hash table;
                } else {
                        if (operation == "Read") {
                                Fill the Lock table record with the dataItem, operation and transactionId;
                                Add the transaction id to the readLockTransId list;
```

```
                } else if (operation == "Write") {
                        Fill the Lock table record with the dataItem, operation and transactionId;
                }
                if (dataItem is not in the transaction){
                        Set the dataItem in transaction;}
                Add the record in transaction hash table;
                Add the dataItem record in Lock hash table
        }
    }

public static void releaseLock() {
        Fetch the dataItem from the lock hash table;

        if (lockState == “Write") || size of readLockTransId == 1) {
                Fetch the waiting list;
                Set lockState to “”;

                if (readLockTransId.size() == 1) {
                        Remove the entry from readLockTransId;
                } else {
                        Print that the lock is released;
                }
                Update the lock hash table;
                if (waiting list is empty) {
                        Remove the dataItem from lock hash table;
                } else {
                        while (waiting list is not empty) {
                                Remove waiting list
                                Fetch the transaction id from the hash table;
                                Call acquireLocks method;
                                Update the transaction hash table;
                                if (transaction state != “Commit") {
                                        return;
                                }
                        }
                }
                Remove the entry for that data item from the lock hash table;
        } else if (lockState == "Read") {
                Fetch the list of readLockTransIds;
                List<Integer> rtids = lockRecord.getReadLockTransId();
                Remove them all by looping over it;
                Update the lock hash table;
        }
    }

public static Transaction procureLock(){
        Fetch the list of waiting operations;
        Set its state to Active;
        Update the transaction hash table;
        if (list of waiting operations is not empty) {
                Print the log;
        }
```

```
            while (waitingList is not empty) {
                Get the operation removed from the list
                if (operation == "Read") {
                    Call request function with read parameter;
                } else if (operation == "Write") {
                    Call request function with write parameter;
                } else if (operation == “Commit”) {
                    Commit the transaction;
                }
            }
            Update the lock hash table with the dataItem;
            Return the transaction;
        }


public static Lock read_read(){
            Add the transaction id to the readLock Transaction Id list;
            if (dataItem is not present in the transaction dataItems){
                Set the dataItems value;
        }
            Update the transaction hash table with this record;
            return lock object;
        }

public static Lock read_write() {
            if (size of ReadLockTransId list == 1 && readLockTransId == transactionId) {
                setLockState to "Write";
                Delete that entry from the list;
                Set the writeLock Transaction Id with the current transaction id;
            } else if (size of readLockTransId == 1 && !readLockTransId == transaction Id) {
                Fetch the transaction with writeLock Transaction id from the Transaction hash table;
                if (two transactions are having the lock on the same dataItem) {
                        Set the state to “Abort”;
                        Update the transaction hash table;
                        Set lock state to “Write”;
                        Set the writeLock Transaction Id to the new transaction id;
                        Remove the transaction id with read lock from the readLock Transaction
id list;
                        Call releaseLock method on the dataItem with readLock;
                        RigorousTwoPhaseLocking.releaseLock(t1, dataItem);

                    } else {
                        Set the transaction state to Block;
                        Add the new write operation on the dataItem to the waiting operation;
                        Update the transaction hash table;
                        Add the transaction id to waiting list;
                    }

            } else if ( size of readLockTransId list > 1) {
                    Add them to readLock Transaction ID list and sort them in ascending order;
                    Fetch the first entry of the list;
                    if (first entry == transactionId of the current transaction) {
```

```
                    Fetch all the transactions with that transaction id from the transaction
hash table and abort all;
                    }
                    Clear the readLock transaction Id list;
                    Set the writeLock Transaction id with the transaction id of first entry;
            } else if (transactionId < transaction id of first entry) {
                    Fetch all the transactions with transactionId from the transaction hash
table and abort all;
                    }
                    Clear the readLock transaction Id list;
                    Set the writeLock Transaction id with the transaction id of first entry;
            } else {
                    Set the transaction state to “Block”;
                    Add the write operation on the dataItem in the waiting operation list;
                    Check all the transactions in readLock list of transactions
                    if (transactionId >= readTransId.get(trans id))
                            Increment to the next transaction id in the list;
                            if (transactionId < readTransId.get(trans id)) {
                              Remove that transaction id from the read Transaction id list;
                              Fetch that entry from the transaction hash table and abort it;
                            }
                    }
                    Add the transaction Ids to the list of read lock Transaction id;
            }
    }
    if (transaction fetched does not have that dataItem) {
            Set the dataItems in the transaction;
            Update the transaction hash table with the record;
    }
    return lock object;}

public static Lock write_read() {

    if (writeLockTransId == transactionId of the current transaction) {
            Set the lock state to “Read”;
    Reset the writeLock Transaction id to 0;
    Insert the transaction id in the list of readLock transaction id;
            if (transaction does not have that dataItem){
                    Set the dataItems in the transaction;
            }
            Update the transaction hash table;
    } else {
            Fetch the transaction from the transaction hash table;
            if (two transactions are having the lock on the same dataItem){
                    Set the transaction state to Abort;
                    Update the transaction hash table with the record changes;
                    Set writeLock transaction id to 0 and set the lock state to “Read”;
                    Add the transaction id to the readLock Transaction id list;
                    if (dataItem not present in the transaction fetched)
                            Add the dataItem to it;
                    Update the transaction hash table for the fetched record;
                    Release the lock on the dataItem;
```

```
                } else {
                        Set the transaction state to Block;
                        Add the dataItem with Read operation to the waiting operation list;
                        Add the transaction id fetched to the waiting list;
                        if (transaction fetched does not have that dataItem){
                                Add the dataItem to it;}
                        Update the transaction hash table for the fetched record;
                }
        }
        return lock object;

public static Lock write_write() {
        Fetch the transaction id with write lock from the transaction hash table;
        if (two transactions are having the lock on the same dataItem){
                Set the transaction state to abort;
                Update the transaction hash table with the record;
                Set the writeLock transaction id with the transaction id of the fetched transaction;
                Release the lock on the dataItem;
        } else {
                Set the transaction state to Block;
                Add the dataItem with write operation in the waiting operation list;
                Update the transaction hash table for the fetched transaction;
                Add this transaction id to the waiting list;
        }
        if (transaction fetched does not have that dataItem) {
                Set the dataItem in the fetched transaction;
                Update the transaction hash table for the fetched transaction;
        }
        return lock object;
```

Pseudocode for Two-Phase Locking Protocol with wait-die method:

All the other functions would be the same except the conditions in the following function. As wait-die method works with timestamp.

```
public static Lock read_write() {
        if (size of ReadLockTransId list == 1 && readLockTransId == transactionId) {
                setLockState to "Write";
                Delete that entry from the list;
                Set the writeLock Transaction Id with the current transaction id;
        } else if (size of readLockTransId == 1 && !readLockTransId == transaction Id) {
            Fetch the transaction with writeLock Transaction id from the Transaction hash table;
            if (timestamp of fetched transaction > timestamp of already running transaction){
                        Set the state to “Abort”;
                        Update the transaction hash table;
                        Set lock state to “Write”;
                        Set the writeLock Transaction Id to the new transaction id;
                        Remove the transaction id with read lock from the readLock Transaction
id list;
                        Call releaseLock method on the dataItem with readLock;
```

```
                    RigorousTwoPhaseLocking.releaseLock(t1, dataItem);

                } else {
                    Set the transaction state to Block;
                    Add the new write operation to the waiting operation;
                    Update the transaction hash table;
                    Add the transaction id to waiting list;
                }
            } else if (size of readLockTransId list > 1) {
                Add them to readLock Transaction ID list and sort them in ascending order;
                Fetch the first entry of the list;
                if (first entry == transactionId of the current transaction) {
                    Fetch all the transactions with that transaction id from the transaction
hash table and abort all;
                    }
                    Clear the readLock transaction Id list;
                    Set the writeLock Transaction id with the transaction id of first entry;
                } else if (transactionId < transaction id of first entry) {
                    Fetch all the transactions with transactionId from the transaction hash
table and abort all;
                    }
                    Clear the readLock transaction Id list;
                    Set the writeLock Transaction id with the transaction id of first entry;
                } else {
                    Set the transaction state to "Block";
                    Add the dataItem with write operation in the waiting operation list;
                    Check all the transactions in readLock list of transactions
                    if (transactionId >= readTransId.get(trans id))
                        Increment to the next transaction id in the list;
                        if (transactionId < readTransId.get(trans id)) {
                          Remove that transaction id from the read Transaction id list;
                          Fetch that entry from the transaction hash table and abort it;
                        }
                    }
                    Add the transaction Ids to the list of read lock Transaction id;
                }
            }
            if (transaction fetched does not have that dataItem) {
                Set the dataItems in the transaction;
                Update the transaction hash table with the record;
            }
            return lock object;}

public static Lock write_read() {

            if (writeLockTransId == transactionId of the current transaction) {
                Set the lock state to "Read";
            Reset the writeLock Transaction id to 0;
            Insert the transaction id in the list of readLock transaction id;
                if (transaction does not have that dataItem){
                    Set the dataItems in the transaction;
                }
```

```
                Update the transaction hash table;
        } else {
                Fetch the transaction from the transaction hash table;
                if (timestamp of fetched transaction > timestamp of already running transaction){
                        Set the transaction state to Abort;
                        Update the transaction hash table with the record changes;
                        Set writeLock transaction id to 0 and set the lock state to "Read";
                        Add the transaction id to the readLock Transaction id list;
                        if (dataItem not present in the transaction fetched)
                                Add the dataItem to it;
                        Update the transaction hash table for the fetched record;
                        Release the lock on the dataItem;
                } else {
                        Set the transaction state to Block;
                        Add the dataItem with Read operation to the waiting operation list;
                        Add the transaction id fetched to the waiting list;
                        if (transaction fetched does not have that dataItem){
                                Add the dataItem to it;}
                        Update the transaction hash table for the fetched record;
                }
        }
        return lock object;

public static Lock write_write() {
        Fetch the transaction id with write lock from the transaction hash table;
        if (timestamp of fetched transaction > timestamp of already running transaction){
                Set the transaction state to abort;
                Update the transaction hash table with the record;
                Set the writeLock transaction id with the transaction id of the fetched transaction;
                Release the lock on the dataItem;
        } else {
                Set the transaction state to Block;
                Add the dataItem with write operation in the waiting operation list;
                Update the transaction hash table for the fetched transaction;
                Add this transaction id to the waiting list;
        }
        if (transaction fetched does not have that dataItem) {
                Set the dataItem in the fetched transaction;
                Update the transaction hash table for the fetched transaction;
        }
        return lock object;
```

Functions that will be used in the implementation:

i) beginTransaction ()

This function will set a transaction status as "Active" and will add that transaction to hash table.

ii) requestLock()

This function will conduct the necessary checks before giving a read or write lock to the transaction. It also checks whether the state is active, aborted, blocked, or committed.

iii) commitTransaction()

This function first checks whether the state is active. If it is active, this function commits the transaction and releases all the locks. In case of block state, this function adds the remaining operations to the waiting list and update the records accordingly. In case of abort state, this function just prints the result.

iv) abortTransaction()

This function set the transaction state to abort and releases all the locks. After the releasing the locks, it will update the record in hash table.

v) blockTransaction()

This function will check if the data item is present in lock table or not. If it is present, it will add transaction id to the waiting list and update the lock hash table. If item is not present, it will add data item to table and update the transaction table.

vi) activeTransaction()

This function will check if the data item is present in lock table or not. If it is present, according to its state value, it will call other utility functions to check read/write conflicts otherwise it will add the record into lock hash table.

vii) releaseLock()

This function releases all the locks and resets the lock state on that dataItem. It will check if the waiting list of operations is empty, in this case it will remove the dataItem from the lock table otherwise it will call the acquire lock method and updates the transaction hash table.

viii) procureLock()

This function will set the state of listed waiting operations to Active and update the transaction hash table. If the list of waiting operations is empty, then it will print the result otherwise it will perform the operation and update the lock hash table with the data item and return the transaction.

ix) read_read()

This function will check whether the data item is present in the transaction data items after adding the transactions id to the readlock transaction id list. If it is not present, it will set value for data items and updates the transaction hash table with the same record and return the object.

x) read_write()

This function will promote a read lock to a write lock. It will perform necessary checks based on the deadlock prevention protocol used (either cautious waiting or wait die), according to which it will abort or block the transaction.

xi) write_read()

This function will demote a write lock to a read lock. It will perform necessary checks based on the deadlock prevention protocol used (either cautious waiting or wait die), according to which it will abort or block the transaction.

xii) write_write()

This function will perform necessary checks based on the deadlock prevention protocol used (either cautious waiting or wait die), according to which it will abort or block the transaction.

Basic Summary of the code:

Input file is a .txt file, with each new line having an operation. The program starts by reading the input file and removing the trailing spaces from the line.

Then when this processed input is passed to the main function, respective functions are called depending on the type of operation.

Each of the operations are processed using the wait-die and cautious-waiting method. Each sub-function also prints small description of the process happened on the data item and stating whether it procured the lock, got successfully committed or aborted.

Each of the input is stored using the form inputnumber.txt and the corresponding output is outputnumber.txt.

REFERENCES:

1. https://github.com/manubhats/TwoPhaseLocking
2. https://github.com/aishwaryarajan75/Two-Phase-Locking