Advanced Database Systems: Final Project
Replicated Concurrency Control and Recovery

Xintong Wang(N18322289), Dailing Zhu (N11754882)

December 7, 2017

Components:

1. Variable: a class that defines variable structure.

(1) Fields:

- **String variableID**: stores the name of the variable(which is Xi).
- **int value**: stores the value the variable contains(which is 10*i).
- **int index**: stores the index of the variable(which is i).

(2) Constructor:

- **protected Variable(int index)**: Initializes all three fields of a variable according to argument index.

2. Transactions: a class that defines transaction structure.

(1) Fields:

- **String transactionID**: stores the name of the transaction(which is Ti).
- **long timestamp**: stores the timestamp of the transaction(for deadlock detection).
- **String transactionType**: stores the type of the transaction(RW or RO).
- **HashMap<Integer ,List<String>> locks**: stores all locks a transaction currently has(creates each site a list of variables for which has been locked by this transaction).
- **String currentState**: Stores the current state of the transaction(active, waiting, aborted).
- **Hashtable<String, Integer> tempTable**: temporarily stores all the values a transaction has written to its local copy.

(2) Constructor:

- **protected Transaction(String id, String transactionType)**: Initializes transaction ID, transaction type and locks, and records the current timestamp.

3. Site: a class that defines site structure.

(1) Fields:

- **int siteIndex**: the index of the site.
- **boolean isUp**: stores the status of the site(is running or is failed).
- **HashMap<Variable, Boolean> variableList**: contains all the 20 vatiables and their state of existence.
- **HashMap<Variable, HashMap<Transaction, String>> lockTable**: Stores all locks on the site(which transaction adds which kind of lock on which variable).

(2) Constructor:

- **protected Site(int index)**: Defines variable list and lock table, initialize isUp to true, initializes site index, and put all variables that satisfied the condition into variable list.

(3) Key Methods:

- **boolean checkLockState(String variableID, String lockType)**: Check if there is a certain type of key on the certain variable.

  Input:

  String variableID, ID of the variable you're gonna check.

  String lockType, the type of lock you are looking for.
- **void recover()**: Set the value of isUP to true, recover the lock table, but only recover odd variables.

4. Graph: a class that defines conflict graph structure for deadlock detection.

(1) Fields:

- **Map<String, ArrayList<String>> adj**: Records the directed edge between two transactions.
- **HashMap<String, Boolean> vertices**: represents transaction.
- **ArrayList<String> cycleList**: collects all the vertices that forms a cycle.

(2) Constructor:

- **public Graph()**: Initializes adj, vertices, cycleList.

(3) Key Methods:

- **protected boolean isCyclic()**: decides if the current graph have a cycle or not.
- **boolean helper(String v, Set<String> recStack)**: help detect cycle, this methods realize DFS process.

  Input:

String v, the starting verticle.

Set<String> recStack, recursion stack.

5. DataManager(DM): a class that helps TM process transactions and it takes operations on fields of sites and variables.

(1) Fields:

- **List<Site> database**: stores all 10 sites with their variables. This database field works as the global database.

(2) Constructor:

- **protected DM()**: Initialize database.

(3) Key Methods:

- **fail(int siteIndex)**: Sets the state of certain site as down, and aborts all the transactions has accessed it before.

  Input:

  int siteIndex, index of the site that you are going to fail.
- **recover(int siteIndex)**: Sets the state of certain site as up, but only recovers odd variables in this site.

  Input:

  int siteIndex, index of the site that you are going to fail.
- **checkWriteLock(String variableID)** & **checkReadLock**(String variableID): check if a certain variable has been RL or WL by some transaction.

  Input:

  String variableID, ID of the variable we are going to check.
- **static void updateDatabase(String changedVariableID, int changedValue)**: If a transaction is going to be terminated, this function will be called in order to write all things in temp table into global database.

  Input:

  String changedVariableID, the variable the transaction has written.

  int changedValue, the value the transaction has written in.

6. TransactionManager(TM): a class that implement all the actions a transaction may take.

(1) Fields:

- **List<Transaction> runningTransaction**: collects all transactions that are active at the moment.
- **LinkedHashMap<Transaction, ArrayList<String>> waitingAction**: collects all transactions that are waiting for the commit of other transactions.
- **Graph waitingGraph**: Collects all the elements in waitingAction in order to detect deadlock.

(2) Constructor:

- **protected TM()**: Initializes runningTransaction and waitingActions.

(3) Key Methods:

- **Transaction begin(String transactionID, String transactionType)**: Initialize a new transaction with its type(RW or RO), and add it to activeTransaction list.

  Input:

  String transactionID, name of the new transaction.

  String transactionType, type of the new transaction.
- **boolean read(String transactionID, String onReadVariableID)**: implement read action of transactions. Check if the variable is able to be read or not, if not, add this transaction to waitingList, then do the deadlock detection.

  Input:

  String transactionID, ID of the transaction that is going to do the read action.

  String onReadVariableID, ID of the variable that is going to be read.
- **boolean write(String transactionID, String onChangeVariableID, Integer onChangeValue)**: implement write action of transactions. Check if the variable is able to be written or not, if not, add this transaction to waitingList, then do the deadlock detection.

  Input:

  String transactionID, ID of the transaction that is going to do the write action.

  String pnChangeVariableID, ID of the variable that is going to be written.

  Integer onChangeValue, value that the transaction is going to write in.
- **void end(String transactionID)**: implement the end action of transactions. This method will commit all the available write actions of the transaction first, then kill this transaction.

  Input:

String transactionID, ID of the transaction that is going to be terminated.

- **boolean deadLockDetection()**: check if there is a cycle in graph.
- **Boolean killYoungest()**: if deadlock is detected, then kills the youngest transaction in this cycle.
- **void addToWaitingAction(String variableID, String actionType, String newValue, Transaction transaction)**: adds transaction to waitingList if it has to wait for another transaction releasing its lock.

  Input:

  String variableID, variable that the transaction is trying to read or write.

  String actionType, read or write action.

  String newValue, the value that the transaction is trying to write in if this action is write.

  Transaction transaction, the transaction that is going to be put in waitingList.
- **void resumeWaitingAction()**: dequeue the first available transaction that is available to run.

Algorithms Implementation:

1. Two Phase Locking:

We defines each transaction a lock table which contains what kind of lock it has set on which variable. In that case, when a transaction is trying to read or write a variable, we first check if this variable has already been locked by someone else. If yes, then this transaction must wait until the lock be released by other one. Locks will be released only when the transaction is committed. When we commit a transaction, we do not only delete it from active list or waiting list, we will also clean its lock table, which represent releasing all keys it holds.

2. Available Copies Approach:

When a transaction writes a variable, it writes on every available copy of the variable. When a transaction reads a variable, it reads the first available copy of the variable. That is what available copies approach do. When a site is down, then we will delete every variable it contains and abort every transaction that has accessed this site, after that all the read and write actions will not take this site into account. When a site is recovered, all the transactions that have been blocked during the time of failure are able to run again. However, we will only recover those odd

index variables(since they don't have copies), so that when a transaction is trying to read a variable, it will also not take this recovered site into account.

3. Multiversion Read Consistency:

For read-only transactions, they will not set locks when reading variables. We define a temp table for each transaction, so when a transaction writes a value on a variable, it writes this value into its temp table instead writing it into database. We will change values on database according to this temp table when the owner transaction of this table has committed. In that case, we can guarantee that a read-only transaction always reads an up-to-date value of variable without any conflict.

4. Deadlock Detection:

We build a directed graph structure representing conflict graph in order to detect if a deadlock(cycle) exists. In order to realize deadlock detection, we first create an object called Graph, when each time a transaction is trying to read or write a variable, if it has to wait since another transaction is holding the key, then we put this transaction into waiting list and put the relationship of waiting(which transaction is waiting for which transaction) into waiting graph. Then we begin to check if the waiting graph is cyclic using DFS. If no, we will move to the next action, else we will kill the youngest transaction in the cycle. The searching of cycle will repeat until no cycle exists in waiting graph.