Advanced Database System

Akshat Kiritkumar Jain (aj3186), Ankit Sati (as14128)

Aim: To implement a distributed database, complete with multiversion concurrency control, deadlock avoidance, replication, and failure recovery.

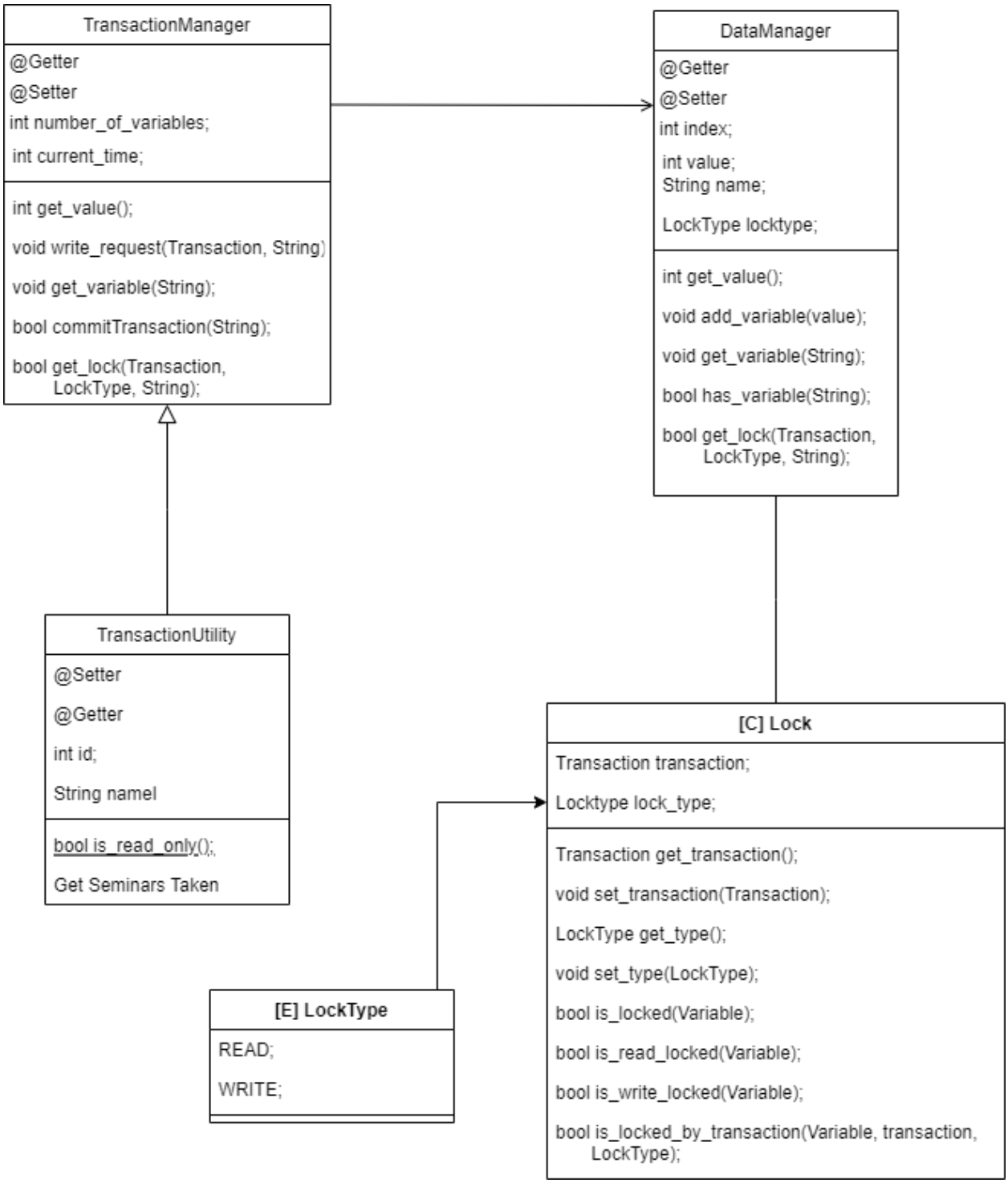
Module:

1. **Transaction_Manager**
    - Works as a hub for all the other modules where the attributes (GET/SET) of all the other modules can interact with each other.
    - This is the main class where we can create and invoke methods to trace the flow of all the transactions throughout the cycle.
    - All the methods that are inherent to different modules can be called in the transaction manager (dump()/ block()/ check()/transaction_begin()/ stop()/ end()). Then we can choose what is needed in order to complete a single transaction.
    - Check for possible deadlocks.
    - Best method available to escape the deadlock.

2. **Tansaction_Utility()**
   ->Tracing Transactions (Can add more)
    - This is a relatively simpler module which will mainly have the status field that will let us know the status of any current transaction.
    - This module can also be used to check the status of any previous transaction for fields like timestamps, flags, counters and status.
3. **Lock**
    - The prime feature of this module is to give us the information about the lock so that we can proceed efficiently.
    - We are yet to decide how we are going to keep the track by using a **Lock table** or resolve them in the **transaction manager**.
    - Whatever method we may choose, this module will have 3 fields being transaction_id, var, type_of_lock. We might have to add another field if we choose to maintain the lock table.
4. **Data Manager**
    - After the Transaction_Manager this is the most important module where all the data will be stored.
    - This is the module that keeps the site active at all times and also works as a data house to keep the track of all the deadlocks.
    - This module can be used to make a site active, as it has methods such as read(), fail()/recover() to deal with downtimes and also a boolean flag to notify the status.
    - Assigning the sites with a unique ID so that we can trace them as well as their states.
5. **Data Field**

- This module is common to all the sites as they will contain some datafield.
- Primarily used to store() the given data, trace() record a given value and finally update() the data in any instance of the object.

Module Diagram:



PseudoCode:

1. **Deadlock module**

   Deadlock detection is carried out by constructing a directed graph of the locks and determining whether the graph contains a cycle.
   Public void DetectDeadLock()
   Function - To detect whether there ist a deadlock in the live transaction and abort the youngest transaction.
   Depending upon our use of hashMaps we will choose between Kosaraju's or Tarjan's strongly connected components algorithm to detect cycles. When a cycle is found, we will stop the cycle's newest transaction. This algorithm will start at the beginning of the tick and run after a predetermined number of ticks.
   Code - depends on the use of the hashmaps

2. **Transactions Work_Flow**
   - begin(TransactionID)
     Function - create and initialize a new Transaction

   - beginRO(TransactionID)
     Function - create and initialize a new Transaction
     read_only = True

   - Read(TransactionID, Var)

     Function - call ReadLock(int TransactionID,int SiteID);

     read x on returned site;

     if False,

     call blockTransaction(Transaction) to block transaction.

3. **Phase Lock**

   As a transaction proceeds, a lock is acquired. As a result, each transaction has a LOCK object that stores all the locks it now holds as well as those it is attempting to obtain.
   Proc - write lock on data item X.

   Check if X is locked by other transactions.
   If Yes: Wait() No: Acquire the lock and add it to the personal LOCK object.
   Inform Transaction Manager
   pass() - Depends upon the use of hashmaps so will pass for now.

4. **Multi_Transaction_Consistency**
   Transaction 1 is a Read-Only transaction wanting to read data item X.
   Transaction 2 is a Write transaction that has updated data item X yet to commit changes.

   Event: T1 Read-Only(X)
   Data Manager
   Read the DATA (new_Values)
   We do this to ensure that a read-only transaction always reads the latest value of a data item
   without incurring any conflicts