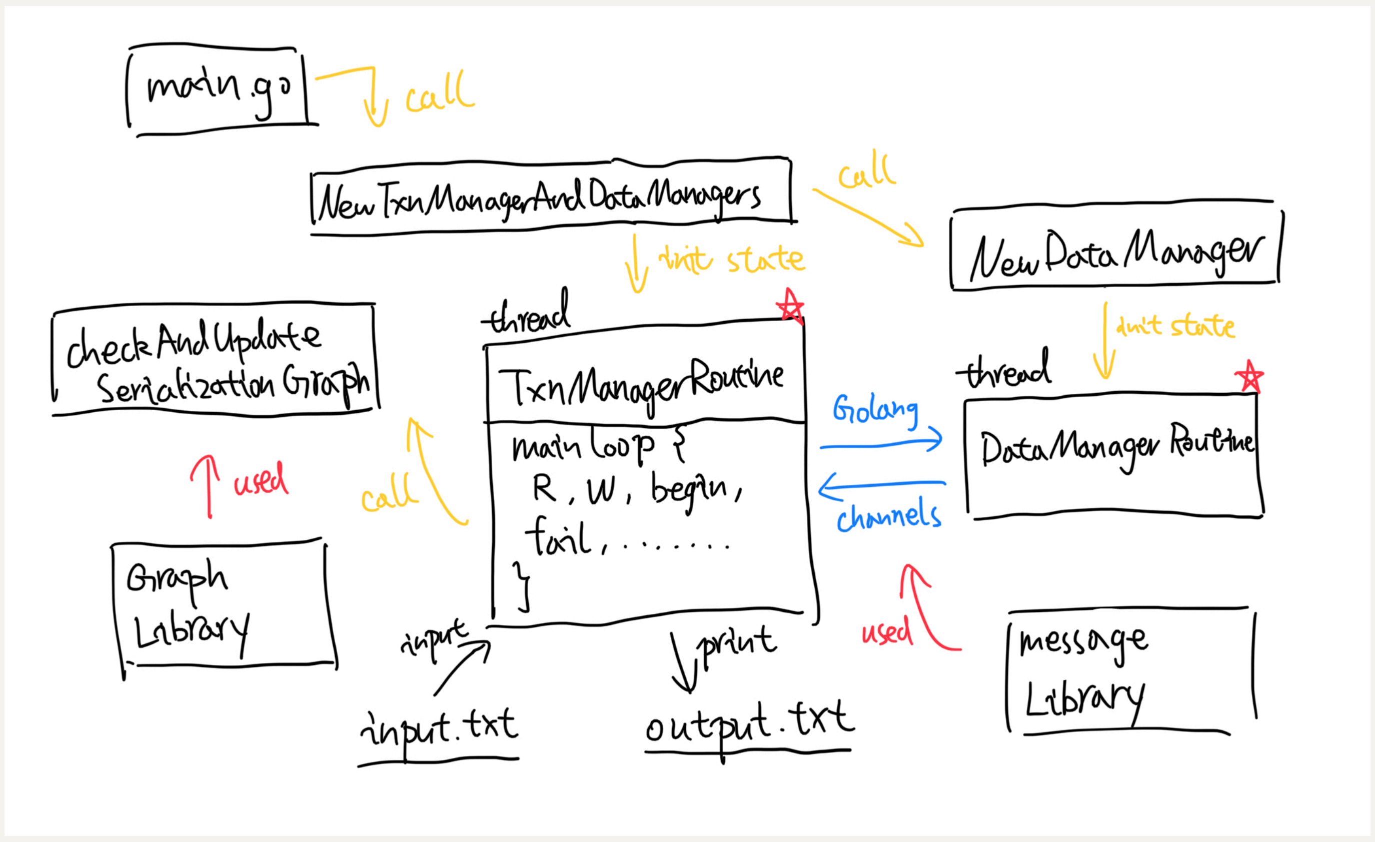
Serializable Snapshot Isolation with Available Copies Documentation

Coulson Liang



Major Functions

`func (s *TxnManagerState) TxnManagerRoutine(inputFilePath string)`

`TxnManagerRoutine` has only the input file path as an argument, and has no return. It is supposed to be called on a initialized `TxnManagerState` struct. This function contains the main infinite loop which read the inputs from the given input file, parse them to database commands (read, write, commit, fail, ... etc) line by line, and dispatch each command to its own code of handling, and eventually print out the outputs to the output.txt file.

- **Read**: Upon a read, the first thing is check the transaction's write set to enforce *read your own writes*. If it is not in the write set, we proceed. Basically, **the TxnManager maintains the information of which data site commited which version at which time for every key**, all store in `s.txnRecords`. With those information, we can populate the `versionNeeded` variable, which is the timestamp of the version of the key that last commited before this transaction starts. Then, we loop through the DataManagers to send read request to ask for this particular version of the key-value pair. If one of them send back `msg.Read` instead of `msg.Error`, this means we get that value. If none of them send back the result, we will decide if we go to wait or abort.
- **Write**: this part is simple, we will try to send the write information to all up DataManagers if the key is replicated, or only one if it is not replicated. At this point the DataManager will save this write pair but will not put it into the table. We only update the actual table at commit time. Also, we will update the `txnRecord`'s corresponding write set, this is for future *read your own write*.

- **Book Keeping**: It is also important that we always keep traking of which DataManager each transaction accessed during its execution, so that if one of them failed after acces, the transaction will be aborted.

`func NewTxnManagerAndDataManagers() TxnManagerState`

`NewTxnManagerAndDataManagers` has no argument, returns a initialized `TxnManagerState` struct. This is the constructor of `TxnManagerState`, and this also calls the functions that initializes DataManagers and starts their Go routines, so by the time this function return, the entire database system should be read to use.

`func NewDataManager(managerNumber int, commandChan chan *msg.Command, responseChan chan *msg.Response, initTime uint64) DataManagerState`

`NewDataManager` takes information about the new DataManager as arguments, then return a initialized `DataManagerState` struct. This is the constructor of `DataManagerState`.

`func (s *DataManagerState) DataManagerRoutine()`

`DataManagerRoutine` has no argument or return, it consists a infinite loop of receiving commands from TxnManager through a go channel, process the command, then reply through a channel.

`func (g *Graph) HasCycleWithConsecutiveRWEdges() bool`

`HasCycleWithConsecutiveRWEdges` is a method of the `Graph` struct, it takes no argument and returns a boolean. This detects if there is a cycle containing consecutive RW-edges in the given graph.

`func (s *TxnManagerState) checkAndUpdateSerializationGraph(txnNumber int) bool`

`checkAndUpdateSerializationGraph` is a method of the `TxnManagerState` struct, it takes an txnNumber of the transaction we want to verify, and then return a bool. This function will try to add new edges according to the transaction's record, and update the conflict graph if all those edges can be added without adding consecutive RW-edges to the graph, then return true. If there will be consecutive RW-edges, this function don't change the graph and return false, so that the `TxnManagerRoutine` will abort this transaction.

`msg` package

This defines all the types of the key-value pairs and all the command and response types that are used for the communication between the TxnManager and the DataManagers.

`graph` package

Yes, I implemented my own graph package, which support edge labling and cycle finding. Those are the foundation of the `checkAndUpdateSerializationGraph` functions, which is used for checking the serializability of the transactions at commit time.