Project Proposal: Replicated Concurrency Control and Recovery

CSCI-GA.2434-001 Advanced Database Systems

Zien Yang(zy2236), Wenbo Song(ws1542)

December 2021 Updated

1 Introduction

This project implements a distributed database, complete with multi-version concurrency control, deadlock detection, replication, and failure recovery.

In this project, there are twenty variables and ten sites in total. For each transaction, the multi-version read consistency is used for the read-only transactions, otherwise the strict two-phase locking protocol is used to avoid non-serialization.

A graph data structure is used inside the transaction manager to keep track of the dependencies of current transactions. At the beginning of each tick, a graph traversal is needed for cycle detection to find potential deadlocks.

Available copy algorithm is used for data replication to enhance fault tolerance within the transaction manager.

2 Design Detail

2.1 IO handler

@Wenbo A class that parse the input file. It will ignore any line starts with //.

2.2 Transaction Manager

- Data Structure
  - A list of sites (Data Manager) with the length of ten.
  - A HashMap to map the transaction ids with theirs transaction objects
  - A queue to memorize the pending commands.
  - A timestamp to store the current time. The timestamp will increment after executing each command.
  - A boolean flag to indicate whether to print more debugging logs.
  - A class of transaction.
  - A class of command to be stored in the command queue
- Functions
  - **begin(transaction_id):** @Zien
    The function will create an instance of a read/write transaction.
  - **beginRO(transaction_id):** @Zien
    The function will create an instance of a read-only transaction.

- **read(transaction_id, variable_id): boolean** @Zien
  The function will go through all the available sites, and try to do a read operation base on the transaction_id and the variable_id. For read-only transactions, multiversion read consistency is used by calling the data manager's snapshot function. It returns True if any read operation is success; otherwise it returns False.
- **write(transaction_id, variable_id, value): boolean** @Zien @Wenbo
  The function will go through all the available sites, and try to do a write operation base on the transaction_id and the variable_id. It returns True when it successfully writes to all sites; otherwise it returns False.
- **dump():** @Zien
  The function will print out the committed values of the copies of all variables at all sites.
- **end(transaction_id):** @Zien
  The function will end a transaction by calling the abort/commit function based on the transaction's status
- **abort(transaction_id):** @Zien
  The function will abort the transaction to all of the sites. Then it will pop the transaction from the transaction HashMap.
- **commit(transaction_id):** @Zien
  The function will commit the transaction to all of the sites. Then it will pop the transaction from the transaction HashMap.
- **fail(site_id):** @Zien
  The function will turn down a site and change the status.
- **recover(site_id):** @Zien
  The function will recover an aborted site.
- **operate(args):** @Zien
  The function will execute the parsed commands by calling corresponding functions.
- **update_command_queue():** @Zien @Wenbo
  The function will pop the read/write command from the command queue if that command is successfully executed.
- **deadlock_detection():** @Zien @Wenbo
  The function will generate a wait-for graph for all sites. Then the function will use a depth-first search approach to find any deadlock conditions and abort the youngest transaction by finding the transaction with the latest timestamp.

2.3 Data Manager

- Data Structure
  - Site status.
  - A class of Variable, storing the value of a variable and its lock tables.
  - An hash-map to memorize the holding variables.
  - An hash-map to memorize the locking status of each variable with the corresponding transaction_id. If the site is down, the lock table will be erased.
  - A list of uncommitted variables.
- Function
  - **add_lock(variable_id, lock_type): boolean** @Wenbo
    The interface which is called by the Transaction Manager(TM) when the TM needs to add locks on a variable of this site. It returns True when the specified lock can be applied on the variable; otherwise it returns False.

- **release_lock(transaction_id): boolean** @Wenbo

  The interface which is called by the Transaction Manager(TM) when the TM needs to release locks of a transaction on this site.

- **local_write(variable_id, value, transaction_id): boolean** @Wenbo

  The interface which is called by the Transaction Manager(TM) when the TM needs to write to the variable. It returns True when the variable can be written to the specified value; otherwise, it return False.

- **if_can_write(variable_id, transaction_id): boolean** @Wenbo

  This function checks if this transaction can write into this variable or not.

- **read_value(variable_id): current value of a variable** @Wenbo

  The interface which is called by the Transaction Manager(TM) when the TM need to read the variable. It returns the value of the variable on this site.

- **commit(variable_id, value): boolean** @Wenbo

  Commit the changes made to a variable.

- **abort(transaction_id): boolean** @Wenbo

  Aborts the transaction on this site.

- **fail(): boolean** @Wenbo

  The lock table should be erased.

- **recover(): boolean** @Wenbo

  All non-replicated variables are available for reads and writes. All replicated variables are available for writing, but not reading.

- **snapshot(): a snapshot of the variables and their values** @Wenbo

  Returns a snapshot of the site for read request from a read-only transaction.

2.4 Component Relationship Figure

The design detail is fully explained in the above section. Therefore, the design figure shown below focuses on the relationship between different design components.