CSCI-GA.2434-001- Advanced Database Systems

Fall 2024

Mingyi Lim:

ml9027@nyu.edu

Replicated Concurrency Control and Recovery (RepCRec)

Summary

This project aims to implement a distributed database with serializable snapshot isolation, replication and failure recovery using the available copies method.

Design

The high level design of the database is as follows:

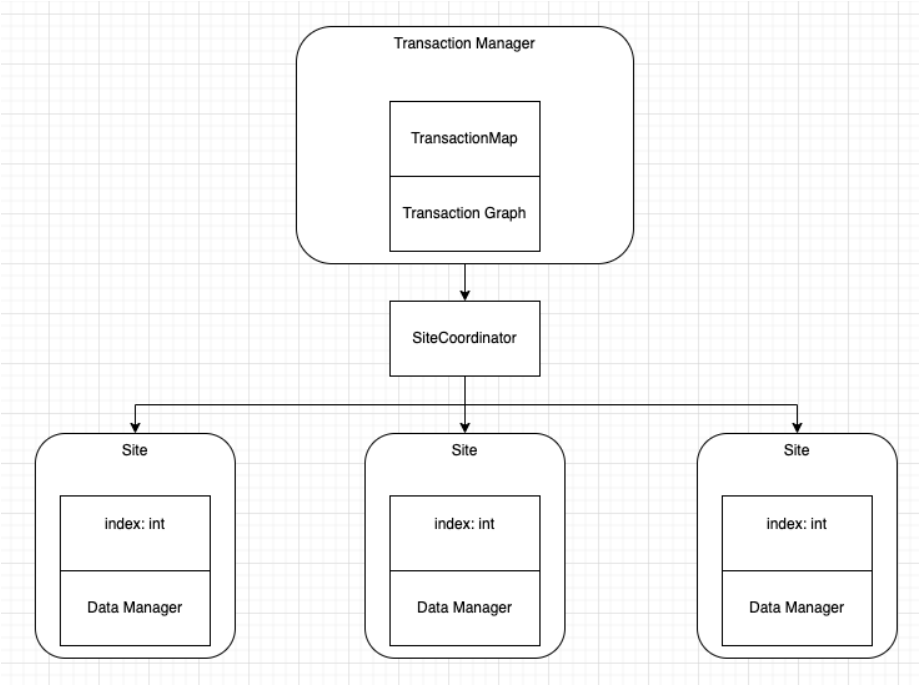

Figure 1: Entity diagram for distributed database

Transaction Manager

The transaction manager acts as the interface with the database. Contains the transaction details of active transactions, a transaction graph for cycle detection and contains the main logic for executing transactions.

A rough overview of the class is as follows:

```
TransactionManager {
    transaction_map: Map[int, Transaction]
    transaction_graph: Map[int, List[int]]
    site_coordinator: SiteCoordinator
    waiting_transactions: Set[Transaction]
}

def exec(transaction: Tx, operation: Operation) -> entrypoint for all operations. If a trans

def start(transaction: Tx) -> creates a transaction in the transaction_map and graph

def end(transaction: Tx) -> checks for RW cycles, write conflicts and site failures and trie

def read(transaction: Tx, key: int) -> Retrieves available sites for reads and attempts to r

def write(transaction: Tx, key: int, value: int) -> Attempts to write to all replicas of a s

def site_recover(site: int) -> starts executing operations on transactions waiting for speci
```

Transaction

Each transaction stores its start time, status completed operations and pending operations in case it is waiting for a site to be made available.

The rough model of a transaction is as follows:

```
Transaction {
    start_time: int
    status: TransactionStatus
    operations: Map[Operation, int]
    site_writes: Map[site_id, Write]
    pending_operations: List[dict, Set[int]]
}
```

Site Coordinator

The site coordinator keeps track of the uptime and history of each site, as well as it's current status. It also helps to retrieve relevant sites for the transaction manager.

The Site coordinator is also used for sending Up and down signals to the sites

It can be modelled as follows:

```
SiteCoordinator {
    sites: Map[int, Site]
    site_uptime: Map[Site, List[(int, int)]]
    transaction_manager: TransactionManager
}

def get_sites(key: int) -> List[(Site, History)]
def get_active_sites(key: int) -> List[(Site, History)]
def recover_site(site: int) -> Brings site back up, writes a new entry to site_uptime. Infor
def down_site(site: int) -> brings site down, closes the tuple on site_uptime. Deletes pendi
```

Site/DataManager

Sites are simply abstract representations of the data managers. The DataManager at each site keeps track of the values held within the site, as well as the commit history of each value. It also holds pending writes to values for each transaction (as volatile writes)

```
DataManager {
    commited_values: Map[int, List[(int, int)]]
    transaction_writes: Map(int, List[Write])
}

def commit_writes(transaction: int) -> Commits all writes from a transaction to committed_va
```

Testing

For the purpose of the project, we will include a discrete time simulation for incoming events. Commands will be mapped to the internal domain.

Commands related to the Transaction Manager will be sent to the transaction manager for processing

Commands related to the status of Sites will be sent to the SiteCoordinator

Constraints and Implementation

Available Copies

Writes

1. Writes are sent to all available copies which hold the write object.
2. If no site is available for writing, we will continue the transaction, but this will abort the transaction.
3. We validate on commit that all sites which were written to have been up since the write occured.
4. If no sites can service the or write request, abort the transaction.

Reads

1. For sites containing replicated data which have recovered, we don't allow reads directly from copies until a new write has occured.
2. We scan all copies until at least one meets the criteria. We can read from that site.
3. If no sites can service the read request currently, but some site might be able to, we wait for the site.
4. If no sites will ever be able to service the read request, we abort the transaction.

Snapshot Isolation

Writes

1. All commits on writes to values with a different latest committed value from when the write occurred will be aborted.

Reads

1. All reads will read from the last committed value before the transaction start time. Available copies approach will ensure that this can happen in a consistent manner.

Atomicity

1. All writes of a transaction are validated at commit time for
    1. available copies constraints
    2. Snapshot isolation constraints
    3. RW cycle constraints
2. Once all validations are concluded, all writes are committed atomically in a single time step.