Transaction Chopping Implementation

Yun-Jung Cheng, Pei-Wen Fu

June 12, 2024

Abstract

Most of web applications store data in different locations, supporting quick local access for users around the world. To ensure a smooth user experience, the data storage system has to guarantee both serializability and low-latency. According to Lynx[1], a geo-distributed storage system, these can be easily achieved by utilizing transaction chopping technique. It works in a way that transactions are statically broken down into a sequence of hops, and each hop would modify data at one server. While executing transactions, the application will return control to the client after the first hop is completed, which often happens at a local datacenter. With the implementation of these methods, a system not only can preserve serializability but also attain low-latency. To evaluate this, we built a library book borrowing system. The results showed that the application successfully delivered low-latency operations.

1 Introduction

A distributed database is a database which stores data across different data centers. Typically, distributed databases operate on two or more servers on a computer network. Each location where a partition of the database is storing at is often called a node. For instance, a distributed database might have three nodes located in three different locations or nodes running on three separate machines in the same location, whereas a traditional database only has one node running on a single machine.

With users spread across various regions, web applications often use distributed data storage systems to maintain their data, providing a relatively effective operation. While a distributed database offers advantages such as high availability, high scalability, and reduced latency, a key challenge is the trade-off between data consistency and latency. It was believed one has to sacrifice consistency and serializability to achieve low latency. However, Lynx showed that it is possible to obtain both at the same time by implementing the following strategy.

Since operations of an application can be defined beforehand, the system is able to perform a static analysis on all possible operations. The analysis uses the theory of transaction chopping, which involves breaking down transactions into smaller hops. After the transactions are transformed to chains of hops, the construction of SC-graph is utilized to determine the feasibility of executing them piecewise while maintaining serializability.

It is shown that serializabiliity is assured if the SC-graph has no SC-cycles. If the graph consists of an SC cycle, further design should be made to ensure serializability. For example, the system may execute the transaction that causes a cycle with standard two-phase locking. Another factor that might affect serializability is the execution order within each server. When chains are executed piecewise, it is important to ensure origin ordering: if transactions T1 and T2 start at the same server, and T1 starts before T2, then T1 should execute before T2 at every server where they both execute.

While preserving serializability, this approach also ensures low latency. Instead of waiting for the entire transaction to complete, the system can return control to the client immediately after the first hop. Since the first hop typically executes in the local datacenter, it is often very fast. Consequently, both serializability and low latency are achieved under this mechanism.

To practically demonstrate this method, we built a book borrowing system for libraries that have geographically distributed branches, which will be explained in Section 3.

2 Background

2.1 Transaction Chopping

Transaction chopping is a technique used to improve the performance and manageability of database systems by breaking down large transactions into smaller, more manageable sub-transactions, referred to as hops. Each hop can be executed independently, allowing for better concurrency and reduced contention among transactions. This decomposition of transactions helps in achieving higher throughput and more efficient utilization of system resources.
The primary goals of transaction chopping are:

1. Concurrency: By breaking down a large transaction into smaller hops, multiple hops from different transactions can be executed concurrently, increasing overall system performance.
2. Manageability: Smaller hops are easier to manage and monitor, simplifying the process of detecting and resolving conflicts.
3. Scalability: Transaction chopping enables distributed systems to scale more effectively by distributing the workload of a large transaction across multiple nodes.

2.2 Transaction Chains

Transaction chains are a sequence of operations that ensure serializability in distributed systems. Each operation in a chain modifies data at a single node, allowing for piecewise execution while preserving the consistency and correctness of the overall transaction. Transaction chains are particularly useful in geo-distributed systems where latency and network partitioning can affect transaction execution.

2.3 Serializability

Serializability is a key property of database systems that ensures the outcome of executing transactions concurrently is the same as if the transactions were executed sequentially. This property is crucial for maintaining data consistency and integrity in the presence of concurrent transactions.

2.4 Directed Sibling Edges

Directed sibling edges are used to manage dependencies between sub-transactions (hops) within a transaction. By establishing directed edges between related hops, the system can ensure that hops are executed in the correct order, preventing conflicts and maintaining consistency.

3 Application Design

We applied the concept of transaction chopping to a book borrowing system. The application is designed for use in libraries with geographically distributed branches. We will first give an overview of the table schema and explain how we partition the data. Next, we will define the transactions. And finally, we show the SC-graph and how the cycles are formed.

3.1 Table Schema

There are three tables: *Users*, *Books*, and *Loans*. The *Users* table stores each user with user_id, name, email, and their membership location. The *Books* table stores each book with book_id, title, author, publication_date, category, its current status (Available/Borrowed), and loan_id if it is borrowed. The *Loans* table stores each borrow record with loan_id, book_id of the book that is being borrowed, user_id of the user that borrows the book, borrow_date, due_date, and return_date if the loan has completed.

3.2 Partition Strategy

In our application, there are three library branches: Library A, Library B, and Library C. The *Users* table is partitioned based on the membership location. If a user is a member of Library A, then the data of that user will be stored in Library A, respectively. Similarly, the *Books* table is partitioned

based on where the book is held. Considering that a user is most likely to register for a membership at the closest library and borrow books from there, partitioning in this manner makes the first hop of accessing user and book data execute efficiently.

The *Loans* table, on the other hand, is duplicated in all three branches. For the reason that a user might as well borrow a book from a branch that is different from where their membership is held, if we partition *Loans* based on the membership location of the user who borrows the book, we will not able to gather all the results within one hop while tracking the borrow status. By duplicating *Loans* table in every node, we can ensure a low first-hop latency on either borrowing a book, returning a book, or tracking borrow status. Additionally, operations on the *Loans* table can happen in parallel among the three libraries, enhancing concurrency.

3.3 Transactions

There are seven predefined transactions in our application. We will explain what each operation does, then break them down into smaller hops.

3.3.1 Add New Book

The Add New Book transaction is executed when a librarian adds a new book to a library. It can be done in one hop:

1. Insert into *Books*

3.3.2 Remove Book

The Remove Book transaction is executed when a librarian removes a book from a library. It can be done in one hop:

1. Delete from *Books*

3.3.3 Borrow Book

The Borrow Book transaction is executed when a user borrows a book from a library. For example, if a user is borrowing a book from Library A, the operation can be broken down into three hops:

1. Insert the borrow record into *Loans* and update status in *Books* at Library A
2. Insert the borrow record into *Loans* at Library B
3. Insert the borrow record into *Loans* at Library C

3.3.4 Return Book

The Return Book transaction is executed when a user returns a book to a library. For example, if a user is returning a book to Library A, the operation can be broken down into three hops:

1. Update status in *Books* and update return date in *Loans* at Library A
2. Update return date in *Loans* at Library B
3. Update return date in *Loans* at Library C

3.3.5 Add User

The Add User transaction is executed when a librarian adds a user to a library. It can be done in one hop:

1. Insert into *Users*

3.3.6 Search User Information

The Search User Information transaction is executed when a librarian searches for information about a specific user. It can be done in one hop:

1. Read from *Users*

3.3.7 Track Borrow Status

The Track Borrow Status transaction is executed when a librarian tracks all the borrowed books from a library. It can be done in one hop:

1. Read from *Loans*

3.4 SC-Graph

The chopping algorithm takes a set of chopped transactions and constructs a graph, which is called SC-graph, where vertices represent transaction hops and edges represent relationships between hops. There are two types of edges: S-edges connect vertices of the same unchopped transaction, C-edges connect vertices of different transactions if they access the same item and an access is a write. An SC-cycle is a simple cycle containing a C-edge and an S-edge.

Figure 1 shows the SC-graph consists of the seven transactions in our application. For all transactions involving writing operations, instances of the same transaction may conflict. To display this issue, the SC-graph includes two instances of each transaction. And for read-only transactions, one instance suffices. Nonetheless, some of the conflicts between instances can be spurious. As we are inserting into a table, if every insertion will have a unique ID, there will never be a conflict. The same logic applies to deletion and update operations. Therefore, the only transaction that causes a C-edge between instances of itself is the Borrow Book transaction, as it is possible that two users want to borrow the same book at the same time.

Before we add the Track Borrow Status transaction into the graph, there is no SC-cycle. However, the SC-Graph of the seven transactions together leads to SC-cycles due to the addition of C-edges between hops that both access the *Loans* table. To resolve the cycles, we simply abort other transactions to handle the seventh transaction.

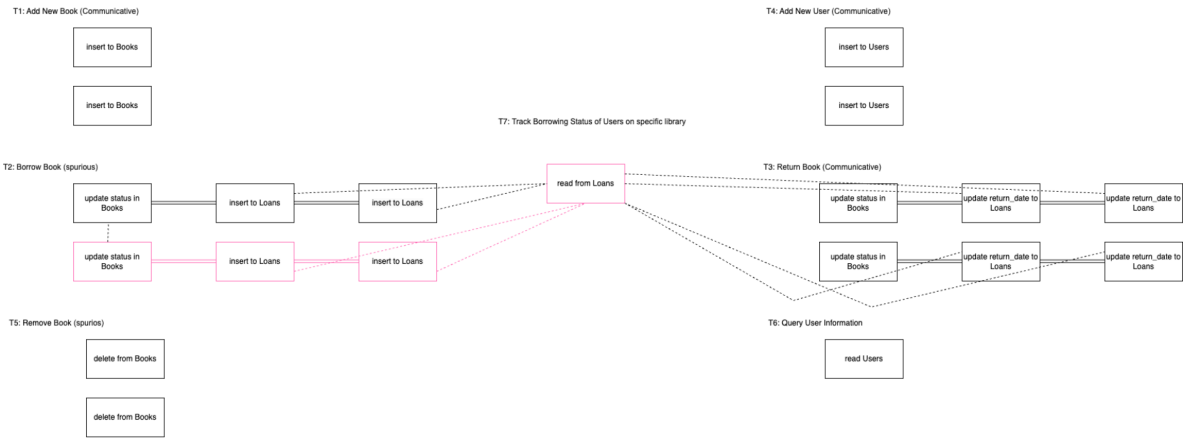

Figure 1: SC-graph

4 System Design

The system design for implementing the transaction chopping protocol involves decomposing transactions into smaller sub-transactions (hops) and managing dependencies using directed sibling edges.

This design enhances transaction processing efficiency by enabling concurrent execution and reducing contention.

4.1 Client

The client is responsible for initiating transactions and sending the hops to the appropriate servers. The system includes several roles:

1. Librarian 1: Operating at Library A
2. Librarian 2: Operating at Library B
3. Librarian 3: Operating at Library C
4. Member 1: Also operating at Library A

Each client can initiate various transactions. These transactions are broken down into smaller hops, which are then sent to the relevant servers for execution.

4.1.1 Client Design

1. Initialization: The client is initialized with a list of servers, a unique ID, and its location. It also has a maximum number of retries for failed hops.
2. Sending Hops: The client sends hops to servers. Each hop includes the action to be performed and any necessary parameters. The client begins by executing the first hop and waits for a response from the server before proceeding to the rest of hops.
3. Transaction Handling: The client handles different types of transactions (e.g., add user, add book, delete book, borrow book, return book) by creating a transaction object with a list of hops.

Listing 1: Client Class

```
class Client:
    def __init__(servers, id, location, max_retries):
        initialize servers, id, location, max_retries, lock

    def send_hop(server, hop, return_value, sequence_number):
        connect to server
        create hop_data with hop and return_value
        if sequence_number is not None:
            add sequence_number to hop_data
        send hop_data to server
        receive and return response

    def send_transaction(transaction):
        raise NotImplementedError
```

4.1.2 BaseClient Design

The BaseClient extends Client and implements the logic for sending transactions without origin ordering.

1. Sending Transactions: The client locks the process to ensure atomicity, executes the first hop, and then processes the remaining hops, retrying if necessary.
2. Handling Failures: If the first hop committed and the following hop fails, the client retries the hop up to a maximum number of retries. If the maximum retries are exceeded, the transaction is aborted.

4.1.3 OriginOrderingClient Design

The OriginOrderClient extends Client and ensures that transactions are processed in the correct order by assigning sequence numbers.

1. Getting Sequence Number: The client requests a sequence number from the server to ensure hops are processed in the correct order.
2. Sending Transactions: The client includes the sequence number in each hop to maintain order.
3. Handling Failures: Same as BaseClient.

Listing 2: OriginOrderingClient Class

```
class OriginOrderingClient extends Client:
    def get_sequence_number():
        connect to server
        request sequence number
        return sequence number

    def send_transaction(transaction):
        lock:
            get and set sequence number for transaction
            execute first hop
            if response is 'Failed':
                print transaction aborted
                return

            for each remaining hop in transaction:
                retries = 0
                while retries < max_retries:
                    execute hop with sequence number
                    if response is 'Success':
                        break
                    else:
                        retries += 1
                        print retrying hop
                        wait
                if retries == max_retries:
                    print transaction failed
                    return

            print transaction completed
```

4.2 Server

The server handles incoming hops, processes the actions, and maintains the database state. For this system, we assume there are three libraries, each managed by its own server. Each server is responsible for managing the database specific to its library and ensuring that the operations requested by the clients are executed properly.

4.2.1 Server Design

1. Initialization: The server is initialized with a host, port, and database name. It listens for incoming connections.
2. Processing Hops: The server processes each hop by executing the specified action (e.g., add book, add user, borrow book) on the database. It returns the result to the client.

Listing 3: Server Class

```
class Server:
```

```
    def __init__(host, port, db_name):
        initialize host, port, db_name, socket

    def handle_client(conn, addr):
        raise NotImplementedError

    def execute_action(node, action, parameters, return_value):
        connect to database
        execute the action
        return 'Success' or 'Failed'

    def start():
        while true:
            accept connection
            start new thread to handle client
```

4.2.2 BaseServer Design

The BaseServer extends Server and implements the logic for handling clients.

1. Handling Clients: The server handles each client connection in a separate thread. It receives hops, processes them, and sends back the results.

4.2.3 OriginOrderingServer Design

The OriginOrderServer extends Server and ensures that hops are processed in the correct order using sequence numbers.

1. Maintaining Sequence Numbers: The server maintains a sequence number for each transaction to ensure order.
2. Handling Clients: Same as BaseServer.

Listing 4: OriginOrderingServer Class

```
class OriginOrderingServer extends Server:
    def __init__(host, port, db_name):
        initialize super class
        initialize sequence number

    def get_sequence_number():
        increment and return sequence number

    def handle_client(conn, addr):
        while true:
            receive data from client
            if data is empty:
                break
            if data is sequence number request:
                send sequence number to client
            else:
                process hop in order
                send result to client
```

5 Evaluation

The book borrowing application is implemented in Python. To illustrate the concept of three library nodes, we used Python SQLite to build three databases: Library A, Library B, and Library C. Each database contains three tables: *Users*, *Books*, and *Loans*.

We evaluate all seven transactions by calculating their throughput and latency. For each transaction, we execute two instances concurrently and calculate the average of the results.

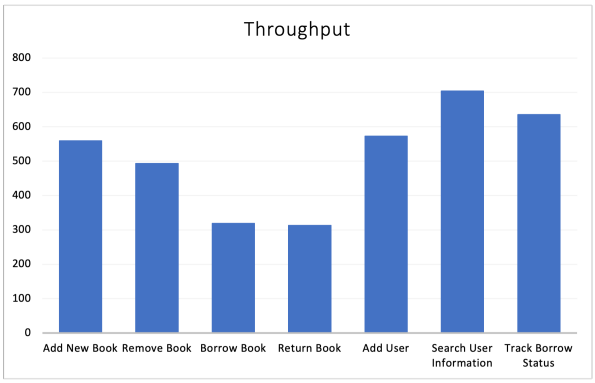

Figure 2: Throughput of operations

Figure 2 shows the operation throughput of the application. According to the bar graph, one-hop reading operations, including Search User Information and Track Borrow Status, have the highest throughput among all. For operations that write data, the throughput depends on the number of hops. Borrow Book and Return Book consist of three hops, leading to a lower throughput compared to other writing operations.

Operation	First-hop latency	Completion latency
Add New Book	3.574 ms	-
Remove Book	4.050 ms	-
Borrow Book	3.901 ms	6.274 ms
Return Book	4.114 ms	6.392 ms
Add User	3.493 ms	-
Search User Information	2.840 ms	-
Track Borrow Status	3.146 ms	-

Table 1: Latency of operations

Table 1 shows the first-hop latency and completion latency of the operations. All transactions return after the first hop, so first-hop latency corresponds to the user-perceived latency, which is generally low. Completion latency measures when the entire chain completes. For Borrow Book and Return Book, the updates of the other two nodes are executed after the first hop, resulting in a higher completion latency. Even so, the users can quickly get the completion response right after the first hop, rather than waiting for the entire transaction to complete.

6 Conclusion

This report details the implementation of a transaction chopping protocol and directed sibling edges in a distributed library management system. Transaction chopping decomposes large transactions into smaller, manageable sub-transactions (hops) to enhance concurrency and reduce contention. Directed sibling edges ensure dependencies between hops are correctly managed, maintaining data consistency and improving conflict resolution.Future work could explore adaptive transaction chopping techniques that adjust based on workload and system state, and applying these methods to real-world database systems for further validation. Overall, the implementation shows that transaction chopping and directed sibling edges are effective for enhancing performance and manageability in distributed database systems, paving the way for more efficient and scalable transaction processing solutions.

References

[1] Y. Zhang, R. Power, S. Zhou, Y. Sovran, M. K. Aguilera, and J. Li. Transaction chains: achieving serializability with low latency in geo-distributed storage systems. *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 276–291, 2013.