Achieving Serializability with Low Latency in a Distributed Setting

Sarah Asad
University of California, Irvine
sasad2@uci.edu

Ruochen Wei
University of California, Irvine
ruochw7@uci.edu

ABSTRACT

This project implements a transaction processing system inspired by the principles outlined in the paper *Transaction Chains: Achieving Serializability with Low Latency in Geo-Distributed Storage Systems* [4]. The primary goal of this system is to emulate a geo-distributed database environment and demonstrate how transaction chains can be effectively utilized to achieve serializability with higher concurrency and lower latency. Our system simulates a distributed database environment using a local setup that spans across three virtual nodes, each represented as a CSV-based database table, thereby mimicking the behavior of a real-world geo-distributed storage system of a project management application of a company. Several transactions are also defined to simulate real-world use cases of the application, showcasing how the system handles various transactional scenarios in the distributed environment. The transactions are predefined with a corresponding SC-graph created. Four of them are SC-Cycles free, but an example dynamic transaction is created to introduce SC-cycles to the graph and based on that we propose a method to handle the specific case. The system uses a Server-Client implementation and threading to simulate concurrent client connections and concurrent transaction requests. It also facilitates transaction chains where each transaction is chopped and executed across multiple nodes as an independent local transaction, ensuring consistency and adherence to hop-based communication. Aimed at achieving serializability and avoiding race conditions, the system incorporates locking mechanisms that ensure mutual exclusion on shared resources to prevent conflicts of different transactions and maintains a consistent execution order for transactions originating from different clients, ensuring correctness in a distributed setup.

KEYWORDS

Distributed System, Serializability, Latency, Transaction Chains, Transaction Chopping, Hopping, SC-Cycles, SC-Graphs

ACM Reference Format:
Sarah Asad and Ruochen Wei. 2025. Achieving Serializability with Low Latency in a Distributed Setting . In . ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Based on the main concept of transaction chopping and SC-Cycles, we intend to propose an application with a simulated distributed database environment and implement a transaction processing component which handles some predefined transactions that satisfy the properties of transaction chains. Additionally, we propose a method to handle dynamic transactions that may not be executable concurrently with other transactions in a serializable manner. The main contents of the application are listed below:

First, we design a simple project management application for a company. The application contains three table schemas and four predefined transactions that correspond to the tasks and operations managed by the application. Each transaction contains at least one read and one write operation that execute across different servers and all transactions have a conflict with one other transaction. The table schemas are on different virtual node servers and are related with foreign keys, which contributes to the conflict operations of different transactions. Although different transactions conflict against each other, we create the SC-graph and make sure there are no SC-Cycles between them, making it possible for them to be executed in a serializable way as small pieces of independent transaction after chopping and assigned to different hops of servers.

Second, we propose a partitioning strategy and implement a transaction processing layer for the application across three nodes. This is implemented using a socket-based Client-Server structure. The application contains different classes related to each other, including Clients, Node Server, Transaction Manager, and Node Manager. Each Client, or user, chooses its target transaction from a list of predefined transactions and sends the transaction request to a Node Server, and will receive ACKs after each hop is executed. The Node Server receives requests from Clients, and sends the corresponding choices to Transaction Manager to get the chopped transaction objects and contains threading and locking methods that ensure geo-distributed execution concurrency and increase throughput and responsiveness while maintaining data consistency. The Transaction Manager gets the choice parameter passed from the Node Server, and creates the transaction with predefined hops. As for the actual file operations, the Node Manager handles the three virtual nodes and has methods to direct to the target table file and do the specific reads and writes on them.

Then, we give an example of dynamic transaction, which will introduce SC-Cycles to the SC-graph with the predefined transactions. Then we propose a method to handle the case by delaying the execution of the transaction and applying locks to eliminate the cycles that cause unserializability.

Lastly, we evaluate the latency and throughput of the system with different database sizes and number of concurrent transactions.

The test result shows that the execution time is polynomial, which indicates that our method is efficient.

2 BACKGROUND

Distributed Database systems are used in many of today's critical applications, from financial systems to could based services, A key challenge in these systems is ensuring that transactions which involve many operations across different nodes, execute in a way that ensures consistency. Serializability ensures that transactions appear to execute in a sequential order, even if they run concurrently. Achieving this in a geo-distributed environment presents many challenges primarily due to communication delays and the distributed nature of data. This is a growing area of research.

Transaction chopping involves dividing transactions into smaller, independent sub-transactions, also called "chops" or "operations. These chops can then be executed in parallel across different nodes reducing contention and improving system throughput. However, to maintain serializability, dependencies between operations must be carefully managed. The system must ensure that the execution order of dependent operations align within a serializable schedule.

Whether different transactions can be executed concurrently across different nodes can be detected by creating a SC-graph of the currently executing transactions. In SC-graph, each transaction is represented by a series of vertices connected by solid edges as a string. The vertices are the transaction pieces executed on different hops after chopping, and the edges are S-edge. Vertices of different transactions can be connected by dashed edges if they access the same table and one of them contains a write, and the edges are C-edge. If SC-graph has a cycle with both S-edges and C-edges, the cycle is called an SC-Cycle. SC-Cycle indicates that the transactions cannot be executed concurrently since serializability is not guaranteed.

Another important concept is hopping, which refers to the movement of transaction execution across multiple nodes. In a geo-distributed setting, each transaction generally involves operations on data stored at multiple locations (nodes). To minimize latency, transactions are designed to hop between nodes with each node only processing part of the transaction. By limiting hops, the system reduces the impact of network delays while maintaining the integrity of transaction chains.

The concept of transaction chains integrates both ideas of transaction chopping and hops to achieve a balance between serializability and low latency. Each chain represents a sequence of operations distributed across nodes, with dependencies explicitly defined to maintain consistency.

This project adopts these principles to simulate a geo-distributed database environment. By managing transaction chopping and hopping, the system demonstrates how distributed transactions can be executed efficiently while adhering to strict consistency requirements.

3 RELATED WORK

For transactions which may contains lots of operations executed across lots of servers, Dennis Shasha [3] proposes an algorithm for chopping transactions into smaller pieces to improve performance, while preserving serializability. The key assumptions are that the user has knowledge of the set of transactions that may run during a certain interval, and can choose among isolation degrees, use multi version read consistency, and reorder instructions within transactions. The algorithm finds the finest chopping of transactions that preserves serializability, allowing for more concurrency and intra-transaction parallelism. The algorithm is efficient, running in O(n(e + m)) time, where n is the number of concurrent transactions, e is the number of edges in the conflict graph, and m is the maximum number of accesses per transaction.

Based on the concept of transaction chopping, Yang zhang -[4] proposed a geo-distributed system called Lynx providing serializable transactions with low latency. Lynx structures transactions as "transaction chains" - a sequence of hops, each modifying data on a single server. To achieve this, Lynx analysis transaction chains by building up a so called SC-graph to determine if they can be executed piecewise, with each hop running as a separate local transaction, while still preserving serializability. Then Lynx is able to return control to the application after the first hop and thus improves the QoS for time sensitive Web users.

Also, Andrea Cerone [1] investigates transaction chopping under Parallel Snapshot Isolation (PSI), a relaxed consistency model enabling more efficient large-scale implementations. They propose a new, more permissive criterion for safely chopping transactions under PSI compared to serializable models, and simplifies reasoning about PSI behavior.

But traditional transaction chopping methods such as Lynx [4] relies on static pre-programming and are not adaptable to runtime changes. People come up with new ideas to deal with dynamic transactions. Ning Huang [2] proposes Dynamic Transaction Chopping (DTC), a technique designed to address challenges in maintaining strong consistency while reducing latency in geo-replicated storage services. DTC introduces a dynamic chopper by changing data partitions and runtime conditions, and has a conflict detection algorithm, enabling real-time transaction division and conflict analysis. Experiments shows that DTC achieves higher piecewise execution and lower latency compared to static methods.

4 OUR PROJECT

In this section, we provide an overview of out application and system design, focusing on how the concepts of transaction chopping, transaction chains, and serializability are integrated to ensure correctness, efficiency, and consistency in a distributed database system.

4.1 Application Design

For our database, we designed a project management system consisting of three tables: **Employees**, **Projects**, and **Tasks**. To ensure scalability and manageability, each table was partitioned into its own node, with the data for each table stored in a separate CSV file. In Figure 1, we illustrate the schema of these tables and their relationships. Each **Project** has a unique ID and name, along with a manager, who is referenced from the **Employees** table. The manager's role is specified as "Manager," and the corresponding property, *ManagerID*, serves as a foreign key. Additionally, a manager may oversee multiple projects, allowing for a one-to-many relationship between **Employees** and **Projects**. For the **Tasks** table, each task

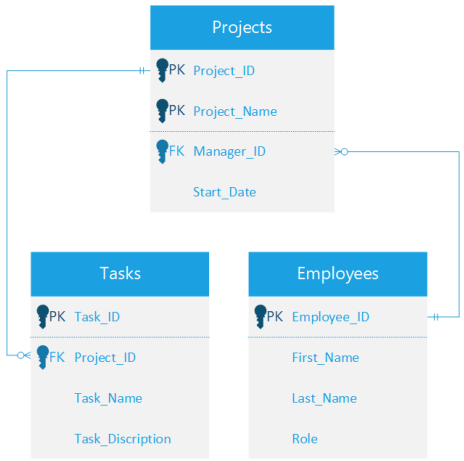

Figure 1: Project Management Database Schema

is assigned a unique ID and is associated with a specific project. The *ProjectID* property is used as a foreign key to establish this relationship. In the **Employees** table, each employee is identified by a unique ID, which serves as the primary key for that table.

Our system has four predefined transaction with predefined hops. These Transaction are listed below.

(1) Transaction 1: Add an employee working on a project.
- hop 1: Read from project
- hop 2: Insert employee

(2) Transaction 2: Add a Manager to a Project and add corresponding task.
- hop 1: Read from project
- hop 2: Insert a task
- hop 3: Insert an employee

(3) Transaction 3: Update a task of a project.
- hop 1: Read from project
- hop 2: Update task

(4) Transaction 4: Fire an employee working on a specific project.
- hop 1: Read from project
- hop 2: Delete employee

We ensured that these transactions are free from SC cycles, as illustrated in Figure 2, where the vertices represent transaction hops, and the edges denote relationships between those hops. No SC-cycle is introduced when executing the four predefined transactions. Each hop, denoted as Ti,j, represents the j-th hop of transaction i. In our design, each hop corresponds to a single read or write operation.

The S-edges connect hops within the same transaction, while the C-edges define dependencies between operations of different transactions. For example, the delete operation in Transaction 4, which modifies the **Employees** table, accesses the same data as the second hop in Transaction 1 and the third hop in Transaction 2, both of which involve insertions into the same table. Similarly, there is a C-edge between the second operations of Transaction 2 and Transaction 3, since the second operation of Transaction 3 may update the item inserted by the second operation of Transaction 2 in the **Tasks** table.

After adding all the S-edges and C-edges to the graph, we can see that there are no SC-Cycles within.

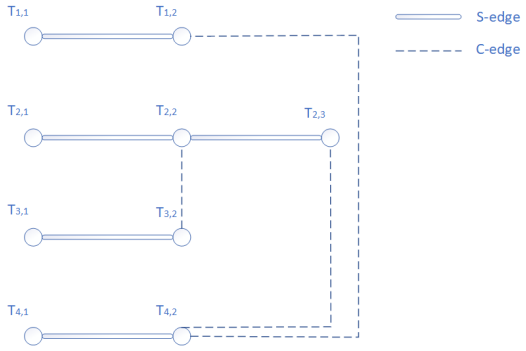

Figure 2: SC-Graph of Predefined Transactions

We also have a dynamic transaction that introduces an SC cycle into the original SC-graph. We later talk about how we deal with these cycles. This transaction is defined as below:

(1) Transaction 5: An Employee is assigned a project with a corresponding task.
- hop 1: Read from employee
- hop 2: Insert a project
- hop 3: Insert a task

The updated SC-graph is shown in Figure 3. The introduction of the dynamic transaction 5 adds two S-edges and three C-edges, which we will explain in detail.

The first operation of the dynamic transaction reads from the **Employees** table to identify a specific manager. This operation may conflict with the second operation of Transaction 1 and the second operation of Transaction 4, as both involve writing to the same table. The third operation of the dynamic transaction inserts a task into the **Tasks** table, which may conflict with the second hop of Transaction 3, as Transaction 3 performs a update on the same table.

After adding the corresponding edges to the graph, a SC cycle is introduced, which is highlighted in a different color to emphasize the conflict.

4.2 System Design

Our system was built using C++. In our implementation we use a Client-Server Architecture both of which are run on a local machine and establish a TCP socket connection.

```
struct operation {
    string type; // R, W, IT, IE, D
    int node; // node/table to operate on
};

struct transaction{
    int type; // 1, 2, 3, 4, 5
    // vector of operations (transaction chain)
    vector<operations> op;
};
```

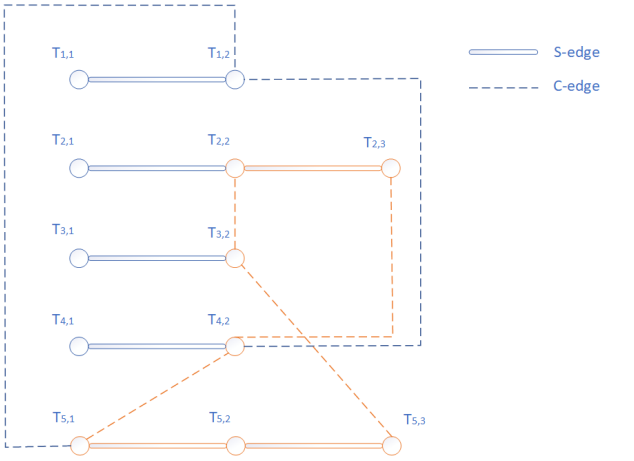

Figure 3: SC-graph with Dynamic Transaction 5 added which introduces a SC-cycle

We have a **Transaction Manager Class** that utilizes two structs, as described above. The first struct, *Operations*, contains two key variables: the operation type and the node identifier. The operation type can represent actions such as: R (read from a table), W (write to a table, i.e., update), IT (insert a row into the task table), IE (insert a row into the employee table), and D (delete a row from a table). The *Operations* struct also includes a *node* variable, which specifies the target table or node where the operation will be executed. The second struct, *Transactions*, includes a vector of operations, effectively simulating a chain of actions within the transaction and also has a type variable in dicating which transaction it is (1 through 5). These two structs work together to enable transaction chopping and chain creation. Each transaction is split into smaller, independent operations that interact with different nodes. These operations are then stored as a chain (via the operations vector) within each transaction. The Transaction Manager is responsible for assembling the predefined steps of a client-chosen transaction into a structured sequence. Below is a short code snippet demonstrating how this is implemented for the first two transaction choices, T1 and T2.

```
transaction pickTransaction(const string& clientChoice){
      transaction t; // create the transaction

      if(stoi(clientChoice) == 1){
         // T1: Insert employee corresponding to a project
         // hop 1: read from project table
         // hop 2: insert to emplyees table

         t.op.push_back({"R", 3});
         t.op.push_back({"IE", 1});
      }
      else if(stoi(clientChoice) == 2){
         // T2: add an employee to a project and
         // insert corresponding task
         // hop 1: read from projects
         // hop 2: insert task
         // hop 3: insert employee

         t.op.push_back({"R", 1});
         t.op.push_back({"IT", 2});
         t.op.push_back({"IE", 1});
      }
```

We also have a **Node Manager Class** which is responsible for performing all read and write operations on nodes/tables.

Client: After the client establishes a connection to the server, the server sends a *ready* message. Once the client receives this message indicating the server is ready, it initiates a loop that sends multiple transactions to the server in parallel using threading. Each transaction is randomly selected through a random number generator, which picks a number between 1 and 5, corresponding to the different transaction types. The transactions are then processed concurrently, ensuring that multiple operations are sent to the server simultaneously for more efficient execution.

Server: Once the server receives a transaction request from the client, it first sends that choice to the Transaction Manager's *PickClientChoice* method (shown above), which creates the transaction and inserts the predefined hops. After the server has the transaction, each transaction is added to its own queue.

Each transaction's operations are processed sequentially, meaning that within a transaction, operations such as reads and writes occur one after another. However, operations from different transactions can be executed concurrently, allowing for parallelism between transactions. For example, transactions T1 and T3, where T1 involves a read of "project" and a write of "employee", and T3 involves a read of "project" and a write of "task", reads can occur simultaneously as long as they do not conflict, due to a shared lock on the "project" resource. This means that the first read of each transaction can occur simultaneously.

Writes, on the other hand, require exclusive locks to prevent conflicts. When a transaction performs a write operation, the system ensures that no other transaction can modify the same resource at the same time. If multiple transactions attempt to write to the same resource, the server uses a mutex lock mechanism to serialize these operations, guaranteeing that writes happen one after another, even though the reads can continue to run in parallel.

Up until now, we have discussed only the four predefined transactions. To handle dynamic transactions (Transaction 5), which introduce an SC-Cycle, we determined that the simplest approach was to delay this transaction until all other transactions have been completed. This ensures that the system avoids potential conflicts or deadlocks while maintaining consistent execution flow. A snippet of how we implement this logic is shown in Figure 4.

The reason we use a queue per transaction instead of a queue per node is because operations within a transaction rely on each other. For example, in transaction 1 where a employee is added based on a project, the second hop which inserts an employee into the employee table, relies on the first hop of reading from projects table.

After each operation is completed, an acknowledgment (ack) is sent to the client, and the client responds with an "Ok" message to confirm the completion of the operation. Once all operations for a transaction have been completed and acknowledged, the transaction is considered fully processed. This approach allows

```
Function ProcessClientRequest(clientSocket, transaction):
 Initialize empty queue for transaction responses
 Initialize empty list for delayed operations

 For each operation in the transaction:
     If the operation type is "5":
         - Add to the delayed operations list
         - Skip the current operation
     Else:
         - Determine the target node for the operation
         - If the operation is a read:
             - Lock the node with a shared lock
         - Else if the operation is a write:
             - Lock the node with an exclusive lock
         Try:
             - Execute the operation
             - Capture the result of the operation
         Catch (exception):
             - Set the response as an error message
         - Add the operation's response to the transaction response queue
         - Send the response to the client
         - Wait for the client's acknowledgment
         - Release the lock on the node after the operation is complete (shared lock for read, exclusive lock for write)
 After processing all operations:
     - For each delayed operation in the delayed operations list:
         - Determine the target node for the delayed operation
         - If the operation is a read:
             - Lock the node with a shared lock
         - Else if the operation is a write:
             - Lock the node with an exclusive lock
         Try:
         - Execute the delayed operation
         - Send the response to the client
         - Release the lock on the node after the delayed operation is complete (shared lock for read, exclusive lock for write)
 After all operations are processed:
     - Send a "Transaction Complete" message to the client
 Return "Transaction Complete"
```

Figure 4: Server Execution Pseudocode

the server to run operations for different nodes concurrently, improving throughput and responsiveness while maintaining data consistency through locking.

Figure 5 illustrates the client view when running a single transaction (Transaction 2) on the server which shows the acknowledgment sent from the server and the OK message sent from the client after each hop execution.

The high-level system architecture, including the communication between various classes and components, is illustrated in Figure 5. The overall system flow is detailed in Figure 6, which depicts the execution process across the entire system

```
Choose a Transaction to run
1) Insert an Employee working on specific Project
       hop 1: Read from Project
       hop 2: Insert Employee
2) Insert an Employee for a Project and add corresponding task
       hop 1: Read a project
       hop 2: Insert Task
       hop 3: Insert Employee
3) Update a Task Corresponding to a Project
       hop 1: Read a project
       hop 2: Update Task
4) Fire an Employee working on a specific project
       hop 1: Read project
       hop 2: Delete employee

Now sending clients choice
Choice sent to server: 2

Response from server ACK for operation 1: 230, 400, Reality Shift, 01-24-2022

Sending Response: OK

Response from server: ACK for operation 2: Task added successfully: 1133687232, 230, bug Fixes, identify and resolve issues in software

Sending Response: OK

Response from server: ACK for operation 3: Employee added successfully: 1133687232, Lily, Miller, Engineer

Sending Response: OK

Response from server: Transaction Complete
```

Figure 5: Transaction log When executing Transaction 2.

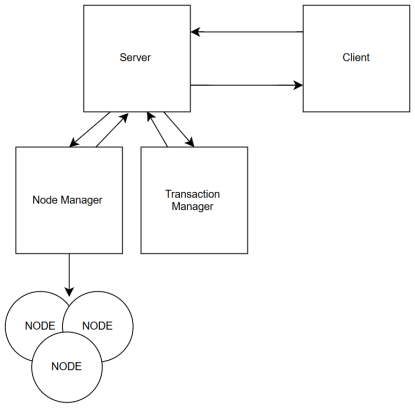

Figure 6: System Architecture: shows how different components communicate with one another

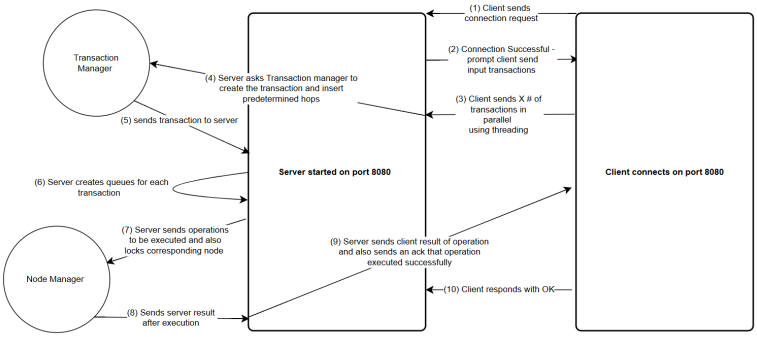

Figure 7: Shows the flow of the entire system. After the client responds to the server with OK (step 10), steps 7-10 are repeated for every operation in the transaction. This execution style is representative of hopping.

4.3 Evaluation

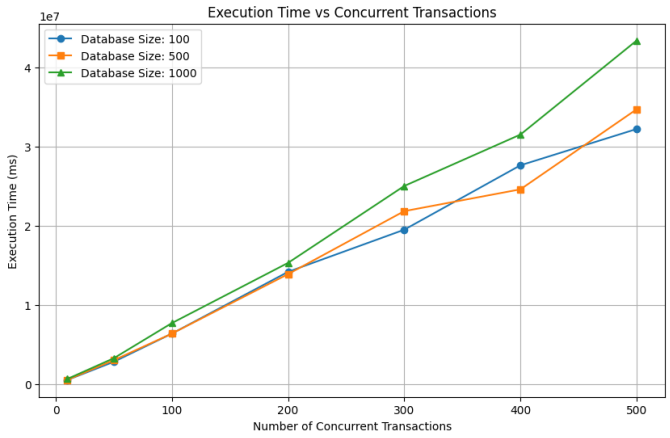

Figure 8: Shows execution time (in milliseconds) for concurrent transaction with varying number of transactions and database sizes.

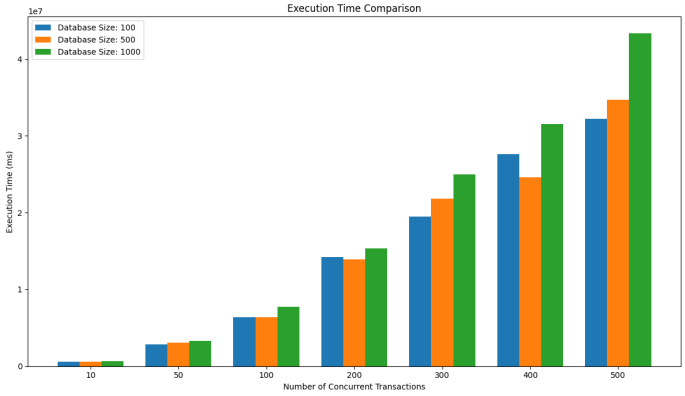

Figure 9: Concurrent Transaction Bar Chart

In this section, we conduct a comprehensive evaluation of our system's performance by analyzing its throughput and latency. This is achieved by varying two key parameters: the database size and the number of concurrent transactions. Through this analysis, we aim to gain deeper insights into how these factors influence the overall efficiency and responsiveness of the system.

Figures 8 and 9 illustrate the execution times for 10, 50, 100, 200, 300, 400, and 500 concurrent transactions across databases of varying sizes, with each table containing 100, 500, or 1,000 records. The graphs reveal that execution time increases linearly as the number of transactions rises. Additionally, transactions involving larger database sizes exhibit slightly longer execution times compared to those with smaller databases.

One important factor to consider is that the client randomly generates transactions to execute on the server, which may result in a higher frequency of less time-intensive transactions, such as Transactions 1 and 2. This behavior is evident in Figure 8, where 400 transactions on a database size of 500 are executed faster than 400 transactions on a database size of 100. Upon analyzing the results, it was observed that for 400 transactions with a database size of 500, the client frequently selected Transactions 1 and 2. These transactions require less execution time because an insert operation is computationally inexpensive; it simply appends a value to the end of a CSV file without needing to traverse the entire table. The variability in transaction selection should be carefully considered when interpreting the results.

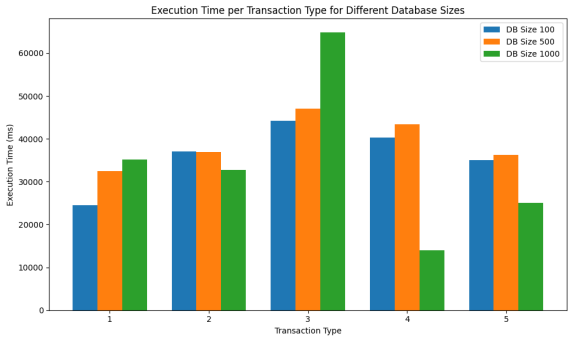

Figure 10: Execution time of different transaction types across three database sizes (100, 500, and 1000 records). The chart illustrates how the execution time varies with transaction type and database size, with larger databases generally resulting in longer execution times for most transaction types

Next, we analyze the time taken per transaction. All transactions in our system begin with a read operation from the *Projects* table. As the database size increases, the time required for reading grows slightly. This is because a read operation involves traversing the table to locate the specific project ID being queried.

Transactions 1 and 2, which involve inserts, are relatively quicker. Transaction 1 reads from the *Projects* table and performs an insert into the *Employees* table. This is efficient since inserts simply append data to the end of a CSV file without requiring a full table traversal. Similarly, Transaction 2 includes a read operation followed by two

inserts, which also maintain efficiency due to the minimal overhead of the insert operations.

In contrast, Transaction 3 takes longer to execute. This transaction begins with a read operation, as previously discussed, but also includes an update to the *Tasks* table. Updates require locating the relevant row by traversing the table, modifying the specific value, and then rewriting the updated data back to the file, which adds additional computational overhead.

Transaction 4, which performs a delete operation, also involves more processing time. The delete operation requires traversing the table to locate the row matching the provided ID, removing the row, and rewriting the table without the deleted entry. .

It is important to note that reads, inserts, and deletes do not always exhibit a direct correlation with increased execution time. In certain cases, if the target row for a read, insert, or delete operation is located early in the table, a full traversal may not be required. This can result in faster execution times, as the system can quickly access or modify the relevant data without needing to scan the entire table. This behavior emphasizes that the position of the data within the table can significantly impact the efficiency of these operations.

This analysis highlights how the complexity and type of operations in a transaction directly influence its execution time.

5 CONCLUSION

This project simulates a geo-distributed database environment across three virtual nodes, trying to simulate the behavior of a real-world distributed storage system of a project management application.

Through the execution of several transactions, the system demonstrates its capacity to handle different use cases in a distributed environment while ensuring high concurrency. The SC-graph analysis identifies scenarios both free of SC-Cycles and those that introduce SC-Cycles, offering valuable insights into the dynamic challenges faced in transaction processing. To address these challenges, the project proposes a method for managing transactions with SC-Cycles, enhancing the system's robustness and scalability.

Built on a server-client architecture with a multi-threaded implementation, the system supports concurrent client connections and transaction requests. By setting up a transaction processing layer, the system ensures that each transaction is effectively chopped into smaller pieces distributed across multiple nodes, maintaining consistency while adhering to hop-based communication principles. Furthermore, locking mechanisms are used to ensure serializability, prevent race conditions, and uphold correctness in a highly concurrent distributed environment. The evaluation of execution time demonstrates that the system operates with polynomial complexity, validating the efficiency of the proposed handling method.

This project demonstrates how theoretical principles from distributed systems research can be applied to practical implementations, providing a solid foundation for exploring more advanced transaction processing techniques in distributed databases.

REFERENCES

[1] Andrea Cerone, Alexey Gotsman, and Hongseok Yang. "Transaction Chopping for Parallel Snapshot Isolation". In: *Distributed Computing*. Ed. by Yoram Moses. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 388–404. ISBN: 978-3-662-48653-5.

[2] Ning Huang et al. "DTC: A Dynamic Transaction Chopping Technique for Geo-Replicated Storage Services". In: *IEEE Transactions on Services Computing* 15.6 (2022), pp. 3210–3223. DOI: 10.1109/TSC.2021.3089819.

[3] Dennis Shasha et al. "Transaction chopping: algorithms and performance studies". In: *ACM Trans. Database Syst.* 20.3 (Sept. 1995), pp. 325–363. ISSN: 0362-5915. DOI: 10.1145/211414.211427. URL: https://doi.org/10.1145/211414.211427.

[4] Yang Zhang et al. "Transaction chains: achieving serializability with low latency in geo-distributed storage systems". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP '13. Farminton, Pennsylvania: Association for Computing Machinery, 2013, pp. 276–291. ISBN: 9781450323888. DOI: 10.1145/2517349.2522729. URL: https://doi.org/10.1145/2517349.2522729.