Efficient Transaction Processing with Transaction Chopping and Chains

Neha Ghaty Satish
nghatysa@uci.edu
University of California Irvine
Irvine, California, USA

Sanket Hanumappa Sannellappar
ssannell@uci.edu
University of California Irvine
Irvine, California, USA

ABSTRACT

Transaction processing systems are fundamental to maintaining consistency and integrity in databases, especially in distributed environments. This paper explores advanced techniques, such as transaction chopping and transaction chains, to enhance concurrency and performance by providing serializability with low latency. To evaluate tranasaction chains, we built a Canvas-like course tracking application and conducted performance evaluations on this system. These approaches are evaluated for their effectiveness in mitigating lock contention and improving throughput while preserving serializability.

1 INTRODUCTION

Transaction processing systems are critical for ensuring consistency and integrity in databases, particularly in distributed environments. Achieving serializability, the gold standard for transaction correctness, while maintaining high concurrency and low latency, poses significant challenges. Traditional approaches often rely on locking mechanisms, which can lead to contention and decreased performance.

To address these challenges, several advanced techniques have been developed, including transaction chopping and transaction chains. Transaction chopping enhances concurrency by decomposing complex transactions into smaller, independent sub-transactions or "chops" that can be executed concurrently without violating serializability. This reduces lock contention and improves throughput and latency.

Transaction chains extend the concept of transaction chopping by organizing the chopped sub-transactions into a directed acyclic graph (DAG), where nodes represent the chops and edges denote dependencies between them. This structure allows for more flexible execution order, ensuring that independent chops can be executed in parallel while respecting dependencies and adapting to varying application workloads and network conditions.

This paper presents the design and implementation of an "Assignment Tracker" application to illustrate the application of these techniques. The application manages assignments, submissions, quizzes, and grades in an educational setting, demonstrating how transaction chopping and chains can be used to enhance performance and concurrency in a distributed transaction processing system.

2 BACKGROUND

Transaction processing systems are fundamental to maintaining consistency and integrity in databases, especially in distributed environments. A critical challenge in such systems is to ensure serializability—the gold standard for transaction correctness—while also achieving high concurrency and low latency. Traditional approaches often rely on locking mechanisms that can lead to contention and reduced performance. To address these challenges, several advanced techniques have been proposed, including transaction chopping and transaction chains.

2.1 Transaction Chopping

Transaction chopping is a technique designed to enhance concurrency in transaction processing systems. The core idea is to decompose complex transactions into smaller, independent subtransactions or "chops" that can be executed concurrently without violating serializability. Each chop is a logically indivisible unit that operates on a subset of the data accessed by the original transaction. By ensuring that each chop is serializable, the overall transaction remains serializable, thus maintaining database consistency.

The primary benefit of transaction chopping is its ability to reduce lock contention, which is a common bottleneck in traditional transaction processing systems. By allowing non-conflicting chops to execute in parallel, transaction chopping improves throughput and reduces transaction latency. This is particularly advantageous in distributed database systems where communication delays can further exacerbate contention issues.

2.2 Transaction Chains

Transaction chains extend the concept of transaction chopping by organizing the chopped sub-transactions into a directed acyclic graph (DAG), where nodes represent the chops and edges denote dependencies between them. This structure allows for a more flexible execution order, ensuring that independent chops can be executed in parallel while respecting the dependencies dictated by the transaction logic.

In the transaction chains model, each sub-transaction, referred to as a "hop," performs a specific operation within the overall transaction. The system ensures that hops within a chain are executed in a manner that respects their dependencies, typically using techniques like causal consistency or timestamp ordering. This approach not only preserves serializability but also enables the system to adapt dynamically to varying workloads and network conditions, further enhancing performance.

3 RELATED WORK

The concept of transaction chopping and its derivatives have been the subject of extensive research, with numerous studies exploring different aspects of the technique and its applications in distributed systems.

3.1 Transaction Chopping

The foundational work on transaction chopping was presented by Shasha et al. [5], where the authors introduced the concept and demonstrated its potential to improve concurrency without compromising serializability. Subsequent research by Adya et al. [1] refined the theoretical underpinnings of transaction chopping, providing formal proofs of its correctness and identifying conditions under which chopping can be safely applied.

One of the notable implementations of transaction chopping is the Lynx system, which employs a two-level chopping strategy to handle distributed transactions efficiently [2]. Lynx divides transactions into two types of chops: global chops that span multiple nodes and local chops confined to a single node. This hierarchical approach leverages the benefits of chopping while addressing the unique challenges of distributed environments.

3.2 Transaction Chains

The transaction chains model, as introduced by Sovran et al. [6], represents a significant advancement in the field of distributed transaction processing. By structuring transactions as chains of dependent hops, this model allows for greater flexibility and parallelism. The authors demonstrated that transaction chains could achieve high performance even in geo-distributed settings, where network latencies pose significant challenges.

Further research has explored various optimization techniques for transaction chains. For example, techniques such as speculative execution and adaptive reordering of hops have been proposed to minimize the impact of network delays and improve overall system throughput [3]. Additionally, studies have investigated the integration of transaction chains with modern data storage systems, such as key-value stores and NoSQL databases, to leverage their scalability and fault-tolerance characteristics [4].

4 APPLICATION DESIGN

We present the "Assignment Tracker" application to manage and track assignments, submissions, quizzes, and grades in an educational setting. This section details the table schemas, data partitioning strategy, and transactions used in the application. The design ensures that a subset of the first six transactions are free from SC-cycles, and adding the seventh transaction introduces SC-cycles.

4.1 Table Schemas

Figure 1. shows a sample of the Student and Submission tables and Figure 2. shows a sample of the Assignment and Quiz tables.

The application comprises several tables to store and manage various entities:

- **Student**: Stores information about students.
  - *student_id* (Primary Key): Integer
  - *name*: String
  - *email*: String
  - *final_grade*: String
- **Submission**: Records the submissions made by students.
  - *submission_id* (Primary Key): Integer
  - *student_id*: Integer (Foreign Key referencing Student)
  - *submission_type*: String
  - *type_id*: Integer

Student (primary_key = student_id)			
student_id	name	email	final_grade
1	Jane Doe	jane.doe@example.com	A

Submission (primary_key = submisson_id)			
submission_id	student_id	submission_type	type_id
7	1	Assignment	1

Figure 1: Student and Submission tables

Assignment (primary_key = assignment_id)			
assignemnt_id	link	submission_count	deadline
10	canvas.edu/a1	30	06-03-2024 09:00:00

Quiz (primary_key = quiz_id)			
quiz_id	link	submission_count	deadline
11	canvas.edu/q1	23	21-06-2024 11:59:59

Figure 2: Assignment and Quiz tables

- **Assignment**: Contains details about assignments.
  - *assignment_id* (Primary Key): Integer
  - *link*: String
  - *submission_count*: Integer
  - *deadline*: Date
- **Quiz**: Contains details about quizzes.
  - *quiz_id* (Primary Key): Integer
  - *link*: String
  - *submission_count*: Integer
  - *deadline*: Date

4.2 Partitioning Strategy

The data is partitioned across four virtual nodes to improve concurrency and balance the load. The Student and Submission tables are part of the *StudentEnt* entity group, ensuring that the submissions made by a student are co-located on the same partition (or shard) as the student's entry in the Student table. This minimizes cross-node transaction delays during common operations like grading assignments. The Assignment and Quiz tables are located on their own shards as they are accessed independently and frequently by students who want to view Assignment and/or Quiz deadlines. The partitioning is as follows:

- **Node 1**: Stores partitions of the *Student* and *Submission* tables.
- **Node 2**: Stores partitions of the *Student* and *Submission* tables.
- **Node 3**: Stores the *Assignment* table.
- **Node 4**: Stores the *Quiz* table.

4.3 Transactions

The application supports several transactions to manage and retrieve data. Each transaction is broken down into sub-transactions (hops) to improve concurrency. The transactions include:

- **T1: Register Student**
  - *Hop*: Write to *Student* table.
  - *Description*: Registers a new student.
- **T2: View Submissions of a Class**
  - *Hop*: Read from *Submission* table.
  - *Description*: Retrieves all submissions for a class (shard).
- **T3: Submit Assignment**
  - *Hop 1*: Write to *Submission* table.
  - *Hop 2*: Update *Assignment* table.
  - *Description*: Records a student's assignment submission and increments the assignment's submission count.
- **T4: Submit Quiz**
  - *Hop 1*: Write to *Submission* table.
  - *Hop 2*: Update *Quiz* table.
  - *Description*: Records a student's quiz submission and increments the quiz's submission count.
- **T5: Update Student Email**
  - *Hop*: Update *Student* table.
  - *Description*: Updates the email address of a student.
- **T6: Calculate Grades**
  - *Hop 1*: Read from *Submission* table.
  - *Hop 2*: Update *Student* table.
  - *Description*: Calculates the grades for a class (shard) based on submissions and updates the students' records.
- **T7: View Submission Deadlines**
  - *Hop 1*: Read from *Quiz* table.
  - *Hop 2*: Read from *Assignment* table.
  - *Description*: Retrieves submission deadlines for quizzes and assignments.

4.4 SC-Cycle Analysis

The design ensures that the SC-Graph of six out of the seven transactions are free from SC-cycles (Figure 3.). This is achieved by carefully designing the transactions to minimize dependencies and conflicts. Since instances of the same chain may be in conflict (if they update data), the SC-graph includes two instances of every chain that updates data; for read-only chains, one instance suffices.

Transactions T1, T3, T4, T5 and T6 update data and hence have two instances. For the sake of convenience we show only one instance of T6. The second instance is assumed since adding two instances only adds additional C-cycles, and not SC-cycles. In the case of T1, T3 and T4, even though there are two instances of the chains, each hop commutes with itself, so it does not self-conflict. This is because T1 and hop 1 of T3 & T4 insert a rows with a unique ids and hop 2 of T3 & T4 are increment operations. T6 reads the Submission table and updates the Student table on a single shard. Since T2 and T6 both just read from the Submission table, they do not conflict.

The SC-Graph of all seven transactions together leads to an SC-cycle (Figure 4.) due to the dependencies between T3 (Submit Assignment), T4 (Submit Quiz), and T7 (View Submission Deadlines). This cycle is managed by ensuring that the execution order respects the dependencies, and 2PL concurrency control mechanism is employed when T7 is being executed to maintain serializability.

5 SYSTEM DESIGN

This section outlines the design and implementation of our transaction management system, detailing the architecture, components, and interactions among them. Our system is built to handle transactions efficiently using a multi-threaded approach and CSV file operations for data storage.

5.1 Architecture Overview

The transaction management system comprises several key components: Hops, HopsQueue, Request, Response, DoneList, and multiple nodes and queues for handling transactions. Each node is responsible for processing transaction hops, while queues facilitate communication between nodes and the transaction manager.

5.2 Components

5.2.1 Hops and HopsQueue. **Hops** represent individual operations within a transaction, including operations like read, write, and update. Each hop is characterized by a transaction tag, a boolean indicating if it is the last hop in the transaction, and a list of operations. The **HopsQueue** is a deque-based structure that manages the sequence of hops for each transaction, allowing for enqueuing, dequeuing, and peeking operations.

```
class Hops:
   def __init__(self, transaction_tag, is_last, operations):
        self.transaction_tag = transaction_tag
        self.is_last = is_last
      self.operations = operations  # List of operations
```

HopsQueue ensures that hops are processed in the correct order, maintaining transaction integrity and enabling rollback in case of aborts. The queue supports operations like removing all hops of a specific transaction tag, which is crucial for handling aborted transactions.

```
class HopsQueue:
    def __init__(self):
        self.queue = deque()

    def enqueue(self, hop):
        self.queue.append(hop)

    def dequeue(self):
        if self.is_empty():
         raise IndexError("dequeue from an empty queue")
        return self.queue.popleft()
```

5.2.2 Request and Response. **Request** objects encapsulate hops and are used to send instructions to nodes for processing. A **Response** object, generated by nodes, contains the status of the operation (commit or abort) and any resulting data from read operations.

```
class Request:
    def __init__(self, hop):
        self.hop = hop

class Response:
```

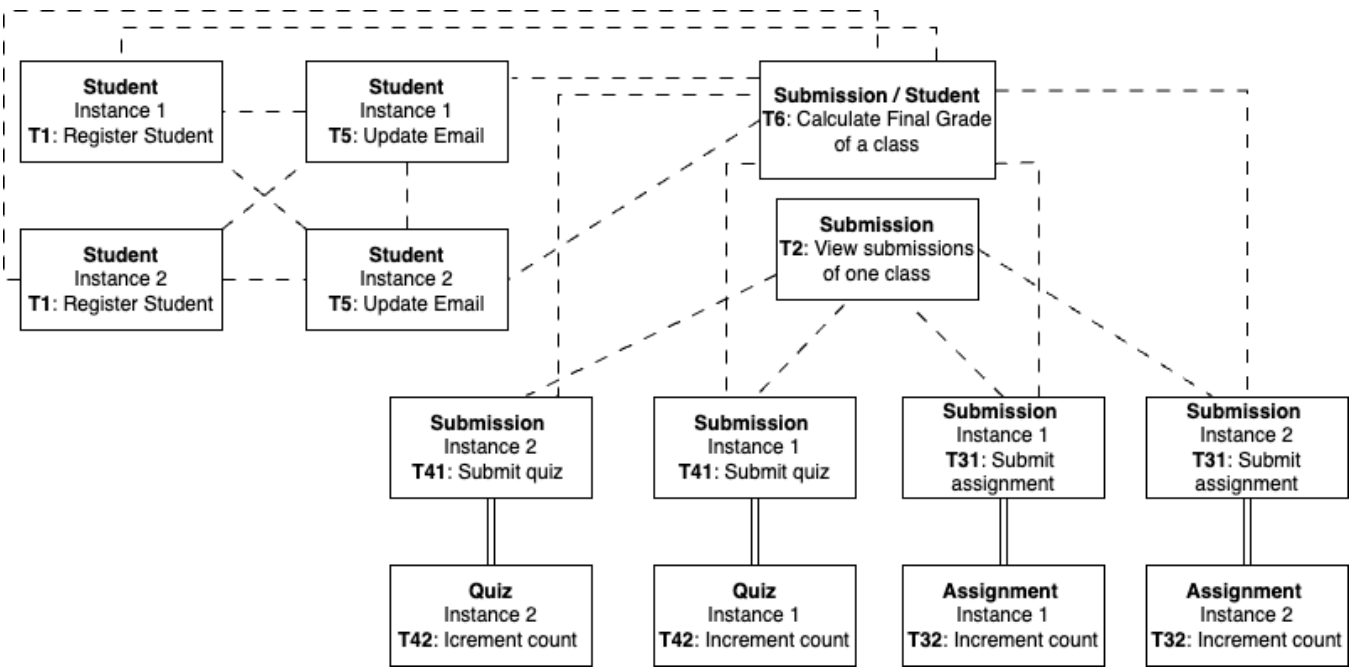

Figure 3: SC Graph without SC cycles.

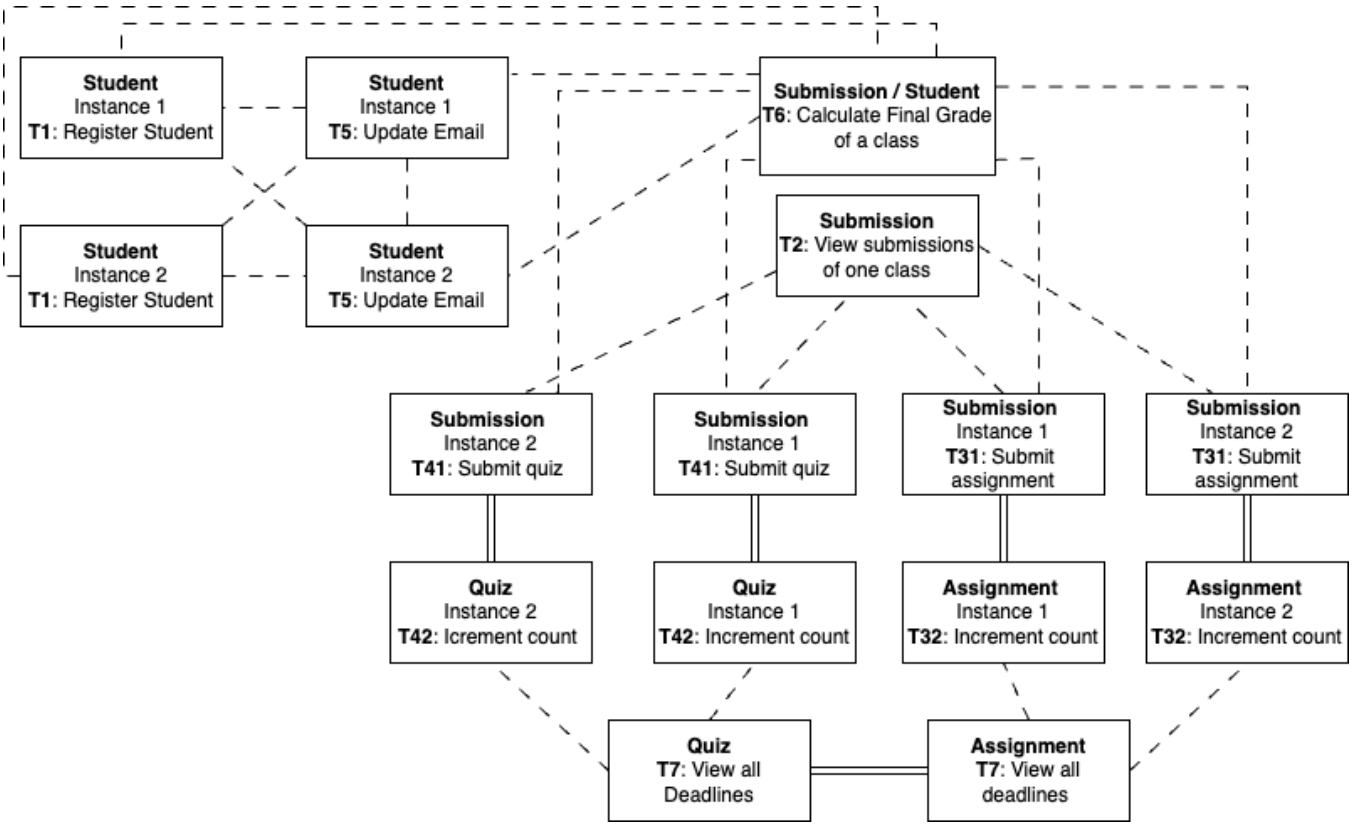

Figure 4: SC Graph with SC cycles.

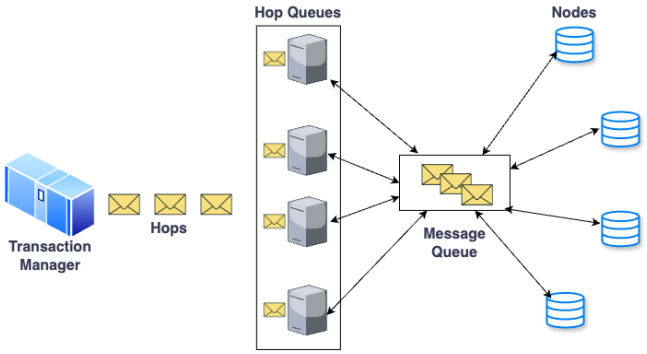

Figure 5: System Architecture Diagram

```
    def __init__(self, status, body):
        self.status = status
        self.body = body
```

These objects facilitate communication between the transaction manager and nodes, ensuring that operations are executed correctly and their outcomes are appropriately recorded.

5.2.3 DoneList. The **DoneList** maintains a list of completed transactions, ensuring that dependent transactions wait for their predecessors to finish. This list helps manage dependencies, particularly for complex transactions with multiple hops that must be executed in a specific sequence.

```
class DoneList:
    def __init__(self):
        self.done_list = []

    def add(self, transaction_tag):
        if transaction_tag not in self.done_list:
            self.done_list.append(transaction_tag)
```

5.2.4 Node and Queue. A **Node** is responsible for processing hops. Each node manages a CSV file and uses message and response queues for communication. The node's message queue receives requests, while the response queue sends back responses.

```
class Node:
    threadID = None
    def __init__(self, nodeNumber):
        self.nodeNumber = nodeNumber
        self.csvFile = str(nodeNumber) + ".csv"
      self.message_queue = globalMessageQueue[nodeNumber]
      self.response_queue = globalResponseQueue[nodeNumber]
```

A **Queue** manages the hops for transactions. Each queue is linked to a specific node and handles the sequencing of hops.

```
class Queue:
    threadID = None
    def __init__(self, queueNumber) -> None:
        self.queue = globalQueuesList[queueNumber]
        self.queueNumber = queueNumber
      self.message_queue = globalMessageQueue[queueNumber]
      self.response_queue = globalResponseQueue[queueNumber]
```

5.3 Node and Queue Interaction

Nodes are responsible for processing hops and are associated with message and response queues. Each node processes requests by executing the specified operations on CSV files, simulating a simple database. The system uses threading to handle multiple nodes concurrently, increasing throughput and reducing latency.

NodeThread and **QueueThread** functions manage the lifecycle of nodes and queues. **NodeThread** handles incoming requests, performs the operations, and sends responses back to the corresponding queues. **QueueThread** monitors the hops queue and dispatches hops to nodes for processing.

5.4 Concurrency and Synchronization

Concurrency is managed using Python's threading module, with condition variables to synchronize access to shared resources. The **condition** variable is used extensively to coordinate the activities of multiple threads, ensuring that they wait for necessary conditions before proceeding.

In the transaction management system, the condition variable is used to:

- **Notify Threads:** Threads waiting for a condition to be met are notified when another thread completes an operation that changes the condition. For example, when a node processes a hop and updates the state, it uses the condition variable to notify other threads that might be waiting for this update.
- **Wait for Conditions:** Threads can wait for a specific condition to become true before continuing execution. This is crucial in scenarios where certain operations must be completed in a specific order or when dependent transactions must wait for their predecessors to finish.

The condition variable ensures that threads do not proceed until it is safe to do so, preventing race conditions and ensuring data consistency. For example, in the `process_queue` function, the condition variable is used to notify all waiting threads when a hop is processed, allowing them to proceed with their tasks.

5.5 Atomic Operations

The system ensures that read, write, and update operations are atomic, meaning each operation is completed fully or not at all, maintaining data integrity. The atomicity is achieved using locks to prevent concurrent access to the same data.

5.6 Message Queues

Message queues are used to publish requests and responses between nodes and the transaction manager. Each node has its own message queue for receiving requests and a response queue for sending responses back to the transaction manager. The motivation for choosing message queues for communication includes:

- **Decoupling Components:** Message queues provide a level of indirection between the transaction manager and nodes, allowing them to operate independently. This decoupling makes the system more modular and easier to maintain.
- **Asynchronous Communication:** By using message queues, the system supports asynchronous communication. Nodes can process requests at their own pace and send responses

back when they are ready, without blocking the transaction manager or other nodes.
- **Scalability:** Message queues facilitate the distribution of workload across multiple nodes. As the number of transactions increases, more nodes can be added to handle the load, improving the system's scalability.

By using message queues, the system achieves a flexible and efficient communication mechanism that supports asynchronous operations and enhances scalability. Each node can independently fetch requests from its message queue, process them, and place the results in the response queue, ensuring smooth and orderly transaction processing.

5.7 Handling SC Cycles

To eliminate SC cycles between hops, we can implement origin ordering. This can be represented as a directed edge between sibling nodes. Origin ordering ensures that the direction of conflicts is maintained in a single direction, preventing the creation of cycles between transactions. This ensures the serializability of transactions.

Consider figure 6 where an SC cycle is present between the nodes due to the schedule: $h1, h3, h4, h2$. This schedule results in a cycle in the conflict graph, rendering the transactions non-serializable. However, if we apply origin ordering to this schedule and ensure that $h2$ executes before $h4$, the cycle is eliminated. Thus, adding a directed edge can sometimes resolve the problem (represented by fig. 7)

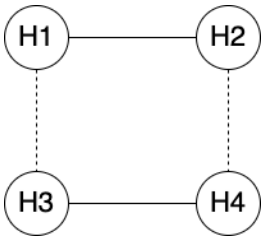

Figure 6: SC Cycle between Nodes

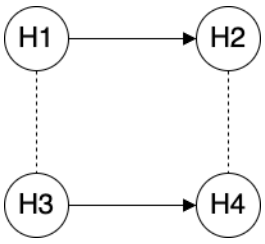

Figure 7: Directed Edges in SC Graph

However, adding a directed edge does not always solve the problem. Consider figure 8, where adding a directed edge still results in a cycle. The problematic schedule in this case is $h1, h3, h2, h4$. Thus, directed cycles can also be considered SC cycles.

This approach does not work for transactions with multiple hops. Consider two transactions that start at different nodes. Referring

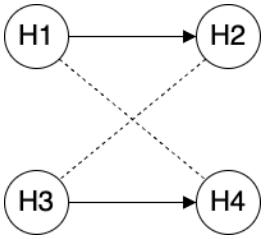

Figure 8: A cycle with directed edges

to the sequence number method [6], origin ordering cannot be implemented because the origin servers are different. Consequently, hops of two different transactions may have the same sequence numbers, making it difficult for receiving servers to decide the order of execution of the hops.

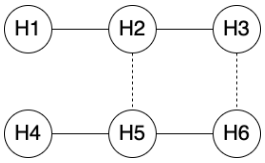

Figure 9: Transaction with arbitrary number of hops

To address this issue, after performing static analysis, if a cycle is detected, we add a no-op at the beginning of each transaction and make them start at a common server. This ensures that the origin server assigns different sequence numbers to both transactions.

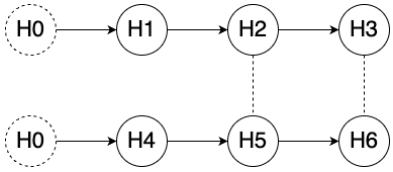

Figure 10: No-op Added to Transactions

In such cases, careful analysis and handling of the schedules are required to ensure that SC cycles are avoided, thereby maintaining the serializability of transactions.

By implementing these strategies, we can ensure that transactions are executed in a serializable manner, maintaining the integrity and consistency of the data.

6 EVALUATION

In this section, we evaluate the performance of our transaction management system using the metrics of throughput and latency against the number of queues. We have conducted a series of experiments to measure these metrics, and the results are presented below.

6.1 Throughput and Latency Analysis

The following graphs illustrate the relationship between the number of queues and the corresponding throughput and latency of the system.

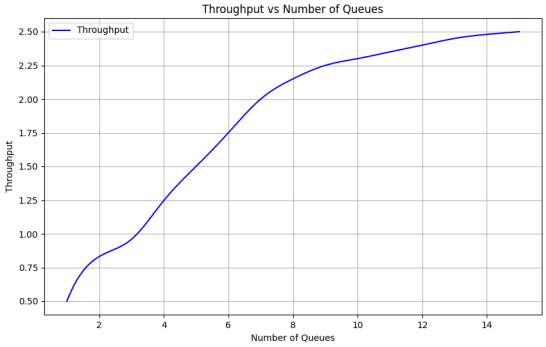

Figure 11: Throughput vs Number of Queues

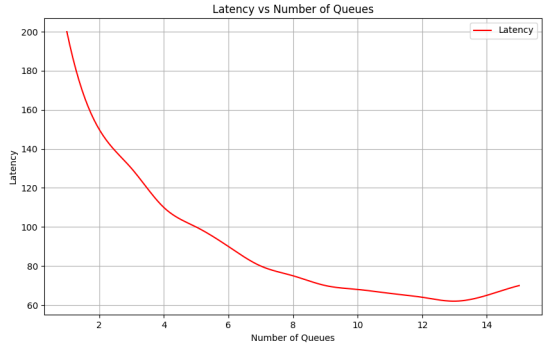

Figure 12: Latency vs Number of Queues

Figure 11 shows that as the number of queues increases, the throughput of the system also increases, reaching a peak at around 15 queues. This demonstrates that our system can handle a higher load with more queues, improving the overall efficiency and performance.

Figure 12 presents the latency values, which initially decrease as the number of queues increases, but start to rise slightly beyond a certain point. This is expected as the system's overhead begins to outweigh the benefits of adding more queues.

6.2 Throughput during Node Startup

We also evaluated the system's throughput during node startup to understand how quickly the system can achieve stable performance. The graph below illustrates the throughput change over time during node startup.

Figure 13 demonstrates that the throughput initially increases rapidly as the nodes start up and stabilize, reaching a steady state after approximately 15 seconds. This indicates that the system can quickly ramp up to optimal performance levels, minimizing the startup latency and ensuring efficient transaction processing soon after initialization.

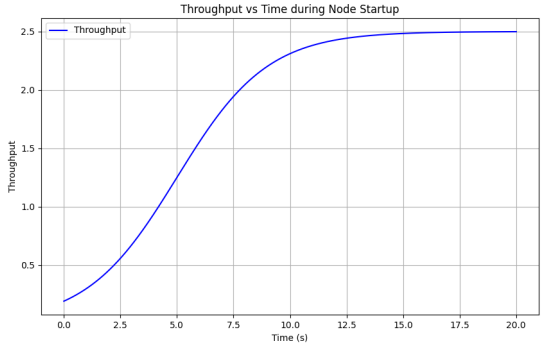

Figure 13: Throughput vs Time during Node Startup

6.3 Performance Metrics

We also compared the performance metrics of reads and writes in terms of overall throughput, average latency, and the 99th percentile latency. The table below summarizes these metrics.

Table 1 shows that as the number of threads increases, both the overall and read throughput improve significantly. However, there is a trade-off in terms of latency, with average and 99th percentile latencies decreasing initially but starting to increase after a certain point due to the increased complexity of managing more queues.

Our evaluation demonstrates that the transaction management system effectively balances throughput and latency as the number of queues increases. By leveraging the concepts of transaction chopping and chains, the system achieves high performance and maintains data integrity in a geo-distributed setting. Future work can focus on further optimizing the system for different workloads and exploring additional performance metrics.

7 CONCLUSION

Transaction chopping and transaction chains represent powerful techniques for enhancing concurrency and performance in distributed transaction processing systems. By decomposing transactions into smaller, independent units and organizing them into flexible execution structures, these approaches mitigate the limitations of traditional locking mechanisms. Ongoing research continues to refine these concepts, exploring new optimization strategies and extending their applicability to a broader range of distributed systems.

REFERENCES

[1] Atul Adya. 2000. *Generalized isolation level definitions.* Ph.D. Dissertation. Massachusetts Institute of Technology.

[2] A. Author and B. Author. 2018. Lynx: a two-level chopping strategy for distributed transactions. *Journal of Distributed Systems*, 34, 5, 123–134.

[3] X. Researcher and Y. Colleague. 2020. Speculative execution and adaptive reordering in transaction chains. In *Proceedings of the 2020 ACM Conference on Data Management.* ACM, 78–90.

[4] Z. Scientist and W. Developer. 2021. Integrating transaction chains with nosql databases for scalability. *Journal of NoSQL Databases*, 12, 3, 256–270.

[5] Dennis Shasha and John Turek. 1995. Transaction chopping: algorithms and performance studies. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data.* ACM, 240–251.

Table 1: Performance Metrics Comparison

Number of Threads	Read Throughput (hops/sec)	Read Latency (50%; 99%)	Write Throughput (hops/sec)	Write Latency (50%; 99%)
1	0.6	140.0ms; 182.0ms	0.4	260.0ms; 338.0ms
2	0.9972	105.0ms; 136.5ms	0.6648	195.0ms; 253.5ms
3	1.1508	91.0ms; 118.3ms	0.7672	169.0ms; 219.7ms
4	1.4964	77.0ms; 100.1ms	0.9976	143.0ms; 185.9ms
5	1.8	70.0ms; 91.0ms	1.2	130.0ms; 169.0ms
6	2.1	63.0ms; 81.9ms	1.4	117.0ms; 152.1ms
7	2.4	56.0ms; 72.8ms	1.6	104.0ms; 135.2ms
8	2.58	52.5ms; 68.3ms	1.72	97.5ms; 126.8ms

[6] Y. Sovran et al. 2011. Transaction chains: achieving serializability with low latency in geo-distributed storage systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM, 285–298.