# Transaction chains: achieving serializability with low latency in geo-distributed storage systems

Chinmayee Mahendra Vaze
University of California, Irvine
Irvine, United States of America
cvaze@uci.edu

Foram Nilesh Bhanushali
University of California, Irvine
Irvine, United States of America
fbhanush@uci.edu

Tanuja Betha
University of California, Irvine
Irvine, United States of America
tbetha@uci.edu

## Abstract

In the realm of geo-distributed storage systems, users often face a difficult choice between high latency with serializable transactions and low latency with limited or no transactions. Inspired by the concepts of transaction chopping and SC-Cycles from existing research, this paper presents the development of an advanced e-commerce application that achieves both serializable transactions and low latency. By extending these ideas, we designed a system leveraging multiple MongoDB instances to ensure high availability, consistency, and scalability. Our application handles critical aspects of e-commerce, including user management, product catalog, order processing, and inventory management. By structuring transactions as chains of hops, each modifying data at a single server, we can perform static conflict analysis ahead of time. This approach allows clients to only wait for the first hop to complete, significantly reducing transaction latency. Through comprehensive evaluation, we demonstrate that our application efficiently manages distributed transactions, maintaining atomic operations and data integrity across nodes. This work highlights the practical benefits of adopting transaction chopping principles in a real-world e-commerce scenario. By showcasing the system's ability to balance high availability with robust transactional integrity, we provide a compelling solution to modern challenges in geo-distributed storage systems.

**ACM Reference Format:**
Chinmayee Mahendra Vaze, Foram Nilesh Bhanushali, and Tanuja Betha. 2018. Transaction chains: achieving serializability with low latency in geo-distributed storage systems. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation emai (Conference acronym 'XX).* ACM, New York, NY, USA, 9 pages. https://doi.org/XXXXXXX.XXXXXXX

## 1 Introduction

In the rapidly evolving landscape of online commerce, the ability to provide a seamless, efficient, and reliable shopping experience is paramount. Modern e-commerce applications must manage a myriad of complex operations, from user authentication and product catalog management to order processing and inventory tracking. These operations require robust data management solutions that ensure consistency, availability, and low latency, even as they scale to meet the demands of a global user base. Traditionally, geo-distributed storage systems have presented a challenging trade-off. Users had to choose between high latency associated with serializable transactions or lower latency with limited or no transactional guarantees. Serializable transactions ensure that all operations within a transaction appear as if they were executed in a strict sequence, which is critical for maintaining data integrity. However, achieving this in a distributed environment often incurs significant latency, which can degrade the user experience. The need for balancing serializable transactions and low latency in e-commerce applications is driven by several factors:

*1.0.1 Data Integrity and Consistency:* In an e-commerce setting, data integrity is crucial. For instance, if inventory levels are not accurately maintained due to inconsistencies, it can lead to situations where customers place orders for out-of-stock items, resulting in negative user experiences and potential revenue loss.

*1.0.2 User Experience:* High latency in transaction processing can lead to slow page loads, delayed updates, and a generally sluggish user experience, which can drive users away to competitors. Fast and responsive systems are key to retaining customers and ensuring a smooth shopping experience.

*1.0.3 Scalability:* As e-commerce platforms grow, the volume of transactions increases. Without an efficient way to manage distributed transactions, the system can become a bottleneck, leading to performance degradation and system outages during peak times.

In the context of e-commerce, where high availability and partition tolerance are often necessary due to the distributed nature of the system, achieving consistency can be challenging. Without a strategy to manage transactions effectively, such as transaction chopping and SC-Cycles, the system might face significant issues:

*1.0.4 Data Inconsistency:* Transactions might result in inconsistent states if not managed properly, causing errors like double booking of inventory, incorrect user account balances, or incomplete order records.

*1.0.5 High Latency:* Ensuring consistency without optimization can result in high latency, as each transaction would require coordination across multiple nodes, slowing down the entire system.

*1.0.6 Reduced Availability:* To maintain consistency, parts of the system might become unavailable during network partitions, leading to a poor user experience. This research paper addresses these challenges by extending the principles of transaction chopping and SC-Cycles, as introduced in existing literature, to the domain of e-commerce. The motivation behind this research stems from the

*Conference acronym 'XX, June 03–05, 2018, Woodstock, NY*
© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/XXXXXXX.XXXXXXX

need to improve the efficiency and scalability of e-commerce systems. It presents a novel application of transaction chopping and SC-Cycles to the field of e-commerce. By extending these principles, we provide a practical solution to the challenges of managing distributed transactions in a geo-distributed environment. Our work showcases the potential for achieving high performance and reliability in modern e-commerce applications, offering a valuable contribution to the ongoing evolution of online commerce systems.

## 2 Background

To fully grasp the concepts and implementations discussed in this research paper, it is essential to understand several foundational ideas[1]: transaction chopping, transaction chains, and SC-Cycles. These concepts are pivotal in managing distributed transactions efficiently, particularly in geo-distributed storage systems.

### 2.1 Transaction Chopping

Transaction chopping is a technique used to break down a complex transaction into smaller, more manageable pieces, known as "hops." Each hop represents an atomic operation that can be executed independently at a single server. The primary advantage of transaction chopping is that it reduces contention and improves parallelism in distributed systems. By decomposing transactions into smaller units, it becomes easier to execute these units concurrently, thereby enhancing system performance and scalability. For example, consider a complex transaction in an e-commerce system that involves checking inventory, updating stock levels, and recording order details. Instead of executing this as a single monolithic transaction, it can be chopped into multiple hops: Hop 1: Check inventory levels. Hop 2: Update stock levels. Hop 3: Record order details. Each hop can be executed independently, allowing the system to handle multiple such operations concurrently without waiting for the entire transaction to complete.

### 2.2 Transaction Chains

Transaction chains build upon the concept of transaction chopping by structuring these hops into a sequence. A transaction chain consists of a series of hops that are executed in a specific order to complete a transaction. This approach leverages static conflict analysis to ensure that each hop can be executed separately while preserving the overall serializability of the transaction. The key idea behind transaction chains is to allow clients to wait only for the first hop to complete, which significantly reduces perceived latency. Once the first hop is successfully executed, subsequent hops can be executed independently, and in some cases, they can even be retried if necessary without affecting the transaction's integrity.

### 2.3 SC-Cycles (Serializable Cycle-Cycles)

SC-Cycles are a mechanism used to detect and prevent potential conflicts in the execution order of chopped transactions. An SC-Cycle represents a cycle in the conflict graph of transactions, indicating that the transactions cannot be serialized in a way that preserves their execution order. Detecting SC-Cycles is crucial to ensuring that the system maintains serializability, which is the strongest form of consistency in transaction processing. When transactions are chopped into multiple hops, it is essential to analyze the conflicts between these hops statically. By performing this analysis ahead of time, the system can determine if the execution order of the hops will lead to SC-Cycles. If SC-Cycles are detected, the system can take corrective actions to re-order or re-execute the hops, ensuring that the overall transaction remains serializable.

## 3 Application Design

The design of our e-commerce application leverages the concepts of transaction chopping and SC-Cycles, structuring transactions into chains of hops to achieve both serializable transactions and low latency. The application is partitioned across multiple MongoDB instances to enhance scalability and performance. Below, we detail the table schemas and the partitioning strategy employed to optimize data management and transaction processing.

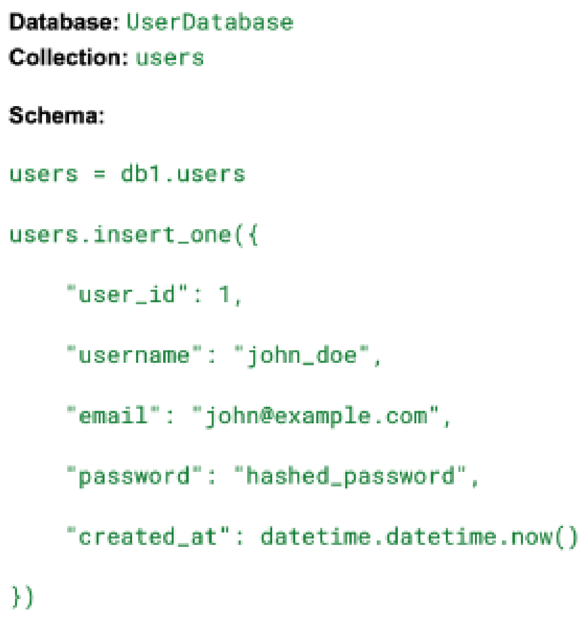
### 3.1 Partition Strategy - MongoDB Instances and Collections

The application is divided into three primary nodes each running on different ports, each responsible for different aspects of the e-commerce system:
Node 1: User and seller Management
Node 2: Product Catalog
Node 3: Order Processing and Inventory Management

We employed this strategy as we could utilize the following benefits:
1. Targeted Scaling: Each node can be scaled independently based on its specific load requirements. For example, if the product catalog grows faster than user data, only the product node needs to be scaled.
2. Fault Isolation: If one node fails (e.g., the product node), the failure is isolated and doesn't directly affect the users/sellers or orders nodes. This enhances the system's overall availability and reliability.
3. Optimized Storage: Different data types have different storage and access patterns. Partitioning allows you to optimize storage configurations and query optimizations tailored to each node's data type, such as read-heavy user profiles or write-heavy order records.
4. Reduced Contention: Minimizes data contention and locking issues. Operations on products don't interfere with user data operations, leading to more efficient processing.
5. Data Segmentation: Sensitive user data can be isolated from less sensitive data (like products), allowing you to apply stringent security measures and compliance protocols to specific nodes as required.
Let us understand what data does each node correspond to:
Node 1: User Management: This node handles user registration, authentication, profile management, and seller data management ensuring that user, seller data is managed independently to reduce contention and improve performance.

Node 2: Product Catalog : This node is responsible for managing product details, including adding, updating, and deleting products. By isolating product data, we can optimize read-heavy operations and streamline updates.

Node 3: Order Processing and Inventory Management : This node manages order creation, tracking, and inventory updates. Grouping these related operations minimizes cross-node communication and

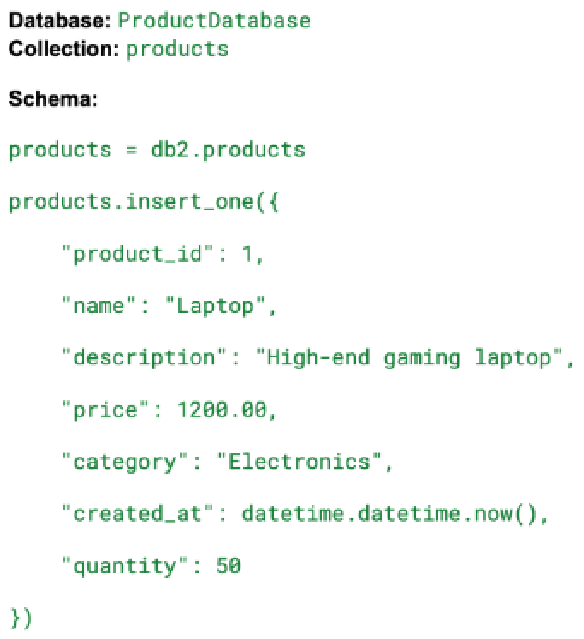


**Database:** UserDatabase
**Collection:** users

**Schema:**

```
users = db1.users

users.insert_one({

    "user_id": 1,

    "username": "john_doe",

    "email": "john@example.com",

    "password": "hashed_password",

    "created_at": datetime.datetime.now()

})
```

**Figure 1: User table Schema on Node 1**



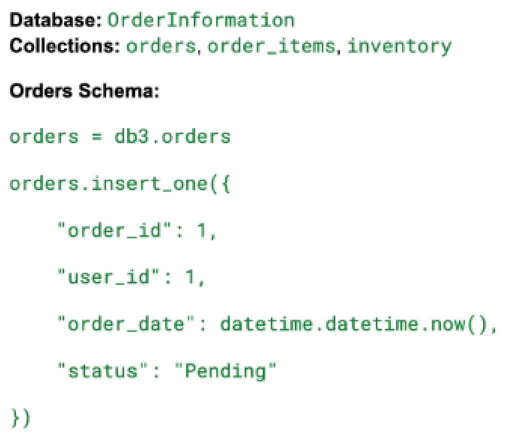
**Database:** ProductDatabase
**Collection:** products

**Schema:**

```
products = db2.products

products.insert_one({

    "product_id": 1,

    "name": "Laptop",

    "description": "High-end gaming laptop",

    "price": 1200.00,

    "category": "Electronics",

    "created_at": datetime.datetime.now(),

    "quantity": 50

})
```

**Figure 2: Product table Schema on Node 2**

ensures atomicity within order-related transactions.

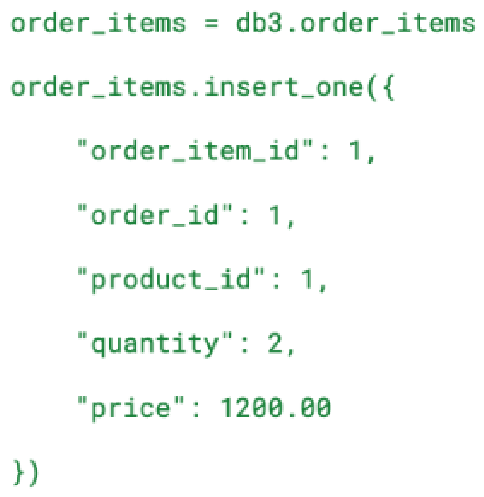


**Database:** OrderInformation
**Collections:** orders, order_items, inventory

**Orders Schema:**

```
orders = db3.orders

orders.insert_one({

    "order_id": 1,

    "user_id": 1,

    "order_date": datetime.datetime.now(),

    "status": "Pending"

})
```

**Figure 3: Order table schema on Node 3**



```
order_items = db3.order_items

order_items.insert_one({

    "order_item_id": 1,

    "order_id": 1,

    "product_id": 1,

    "quantity": 2,

    "price": 1200.00

})
```

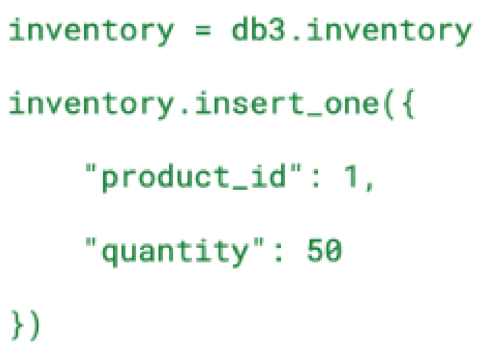
**Figure 4: OrderItems table Schema on Node 3**

## 3.2 Transaction Overview

Our e-commerce application leverages the principles of transaction chopping and SC-Cycles to efficiently manage seven key transactions, ensuring high performance and data integrity.

*3.2.1 Transaction 1.* : Registers new users in the UserDatabase. This involves a single-hop operation to verify if the user already exists and, if not, to add the new user, ensuring a seamless registration process.

*3.2.2 Transaction 2.* : Adds new products to the ProductDatabase and updates the inventory in the OrderInformation database. This transaction is executed in two hops: the first hop inserts product details, while the second updates inventory levels, ensuring product availability is accurately tracked.



```
inventory = db3.inventory

inventory.insert_one({

    "product_id": 1,

    "quantity": 50

})
```

**Figure 5: Inventory Schema on Node 3**

*3.2.3 Transaction 3.* : Manages the order placement process. The first hop checks inventory levels to confirm sufficient stock, then places the order, and the second hop updates the product quantity in the ProductDatabase, ensuring accurate stock management.

*3.2.4 Transaction 4.* : Increases product stock in both the ProductDatabase and the OrderInformation databases. This is achieved in two hops: the first updates the product quantity in the product catalog, followed by updating the inventory levels, ensuring consistency across the system.

*3.2.5 Transaction 5.* : Updates a user's email address in the UserDatabase. This single-hop operation finds the user by their userid and updates their email, facilitating smooth user profile management.

*3.2.6 Transaction 6.* : Adjusts the price of products in the ProductDatabase. This involves a single-hop operation that locates the product by its productid and updates its price, allowing for efficient price management.

*3.2.7 Transaction 7.* : Updates both the quantity and price of products in the ProductDatabase and synchronizes the inventory in the OrderInformation database. This transaction first updates the product details, followed by repeated attempts to update the inventory until successful, ensuring data consistency and reliability.

## 3.3 Nodes and Transactions

Each node (n1, n2, n3) represents a database instance handling specific parts of the application: User Management (n1), Product Catalog (n2), and Order Processing and Inventory Management (n3). Transactions (T1, T2, T3, T4, T5, T6, T7) represent the various operations performed within the application, such as adding users, adding products, placing orders, updating inventory, and updating user or product information. Transaction T1 involves a single-hop operation with no conflicts. Transaction T2, responsible for adding a product, conflicts with transactions T3, T4, and T7 due to operations on the same nodes. However, these conflicts are effectively managed through careful ordering of operations. Transaction T3, which handles placing an order, also conflicts with transaction T4, but this conflict is similarly resolved. Transaction T4, which increments product stock, creates a spurious cycle that is manageable and also conflicts with transaction T7. Transaction T5, which updates the user email, and transaction T6, which updates the product price, are both conflict-free. Lastly, transaction T7, which updates product information, is self-conflicting.

## 4 System Design and implementation

Our system is designed to execute transactions concurrently while maintaining origin ordering. This ensures "`serializability`" with "`low latency`", crucial for web applications with geo-distributed storage. The system leverages a coordinator and static analysis to decompose transactions into smaller, manageable pieces called hops, which are executed across different servers.

## 4.1 Coordinator and Concurrency

The core of our system is a coordinator responsible for orchestrating the execution of transaction chains. Coordinator decomposes a transaction into smaller pieces (hops) that can be executed independently while preserving serializability, each modifying data at one node. The coordinator assigns these hops to nodes and ensures they are executed in the correct order, that is maintaining inner ordering and origin ordering. Our implementation allows each client to perform multiple transactions concurrently. This design enables high concurrency, as different chains can be processed in parallel, minimizing wait times and improving overall system efficiency. The data is distributed across 3 nodes (each is a MongoDB server) working on different ports.

## 4.2 Static Analysis

A static analysis is employed to identify potential conflicts and determine the optimal execution order. The global static analysis is performed on the transaction chains. This analysis uses the Serializability Conflict graph (SC-graph), which identifies conflicts between different hops. If the SC-graph contains cycles, it indicates potential conflicts, and the system adjusts the execution plan to avoid these conflicts.

*4.2.1 SC-Graph Construction.* The SC-graph is a directed graph where nodes represent transaction hops, and C-edges represent potential conflicts between hops. A cycle in the SC-graph indicates a potential conflict that could violate serializability. The coordinator uses this graph to adjust the execution order of hops to ensure that all transactions can be executed without conflicts.

*4.2.2 Conflict Resolution.* When a cycle is detected in the SC-graph, the coordinator resolves the conflict by adjusting the execution order of the conflicting hops. This may involve delaying certain hops until other dependent hops have been completed. By doing so, the coordinator ensures that all transactions are executed in a serializable manner.

## 4.3 Origin Ordering

To maintain consistency, our system enforces origin ordering. If two chains start at the same server and one starts before the other,

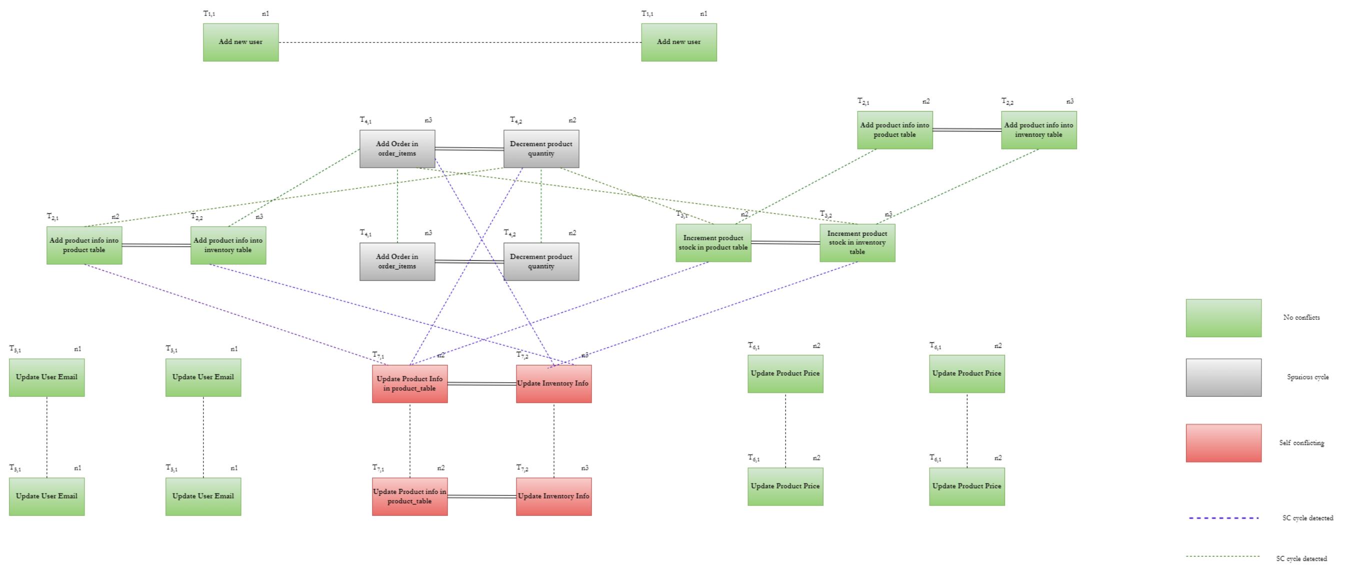

T1,1 n1 Add new user
T1,1 n1 Add new user
T4,1 n3 Add Order in order_items
T4,2 n2 Decrement product quantity
T2,1 n2 Add product info into product table
T2,2 n3 Add product info into inventory table
T3,1 n2 Increment product stock in product table
T3,2 n3 Increment product stock in inventory table
T5,1 n1 Update User Email
T7,1 n2 Update Product Info in product_table
T7,2 n3 Update Inventory Info
T6,1 n2 Update Product Price
No conflicts
Spurious cycle
Self conflicting
SC cycle detected
SC cycle detected

**Figure 6: Illustrating SC Graph for the transactions**

the coordinator ensures the execution order is preserved at all subsequent servers. This is achieved using node transaction queue at each node, which maintain the order of transactions across servers.

## 4.4 Low latency and consistency

Once the first hop of the transaction is executed, the coordinator sends a status message to the client indicating whether the transaction is likely to be successfully committed or aborted. Upon receiving this confirmation, the client can decide to initiate another transaction immediately, optimizing for low latency.

Assuming the first hop of the transaction is successful and the client is informed that the transaction is proceeding smoothly, but subsequent hops encounter failures, the coordinator will automatically re-execute these failing hops. This process continues until all hops are successfully executed, ensuring transaction completion while maintaining low latency and system consistency.

## 4.5 Implementation overview

All the client requests are forwarded to the coordinator and the coordinator is responsible for the concurrent execution of transactions and dividing the transactions into hops. The coordinator is implemented in such a way that it can determine which node each hop should execute on. The coordinator also manages the inner ordering of the hops. The transactions are then forwarded by the coordinator to their nodes. Each node has its own transaction queue-like data structure which will help in determining whether the transaction can be executed right away or has to wait until any other transactions are being addressed. Let us now understand how the transactions discussed in the previous sections are implemented.

*4.5.1 Transaction 1 : Adding the user.* - This transaction involves adding a new user to the database. It consists of a single hop executed on Node 1. The coordinator first checks if the user already

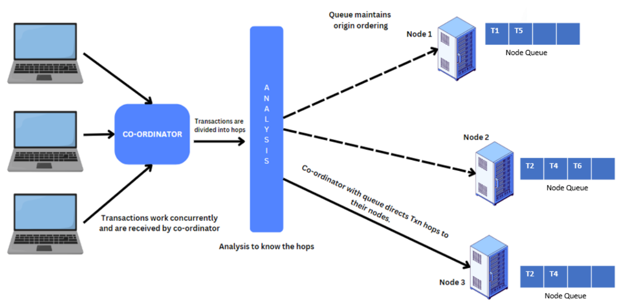

Queue maintains origin ordering
Node 1
T1 T5
Node Queue
CO-ORDINATOR
Transactions are divided into hops
ANALYSIS
Node 2
T2 T4 T6
Node Queue
Co-ordinator with queue directs Txn hops to their nodes.
Transactions work concurrently and are received by co-ordinator
Analysis to know the hops
Node 3
T2 T4
Node Queue

**Figure 7: System Design.**

exists in the database. If the user is found, the transaction is aborted to prevent duplication. If the user does not exist, the coordinator proceeds to add the new user to the database. Upon completion of this hop, the coordinator immediately sends a message to the client indicating the successful addition of the user.

*4.5.2 Transaction 2 : Adding a new product.* - This transaction requires updates to both the Product table (Node 2) and the Inventory table (Node 3). This transaction is divided into two hops. The first hop checks if the product already exists in the Product table and adds it if it does not exist. If the product is already present, the client is notified that the product already exists, and no further hops are executed. Upon completion of this hop, the coordinator sends a message to the client confirming whether the product has been successfully added, thus giving the client the impression that the transaction is complete. Internally, the coordinator continues to execute the remaining hops, specifically adding the new product to the Inventory table at Node 3. This approach enables clients to receive faster responses and handle multiple transactions efficiently, thereby providing low latency and improved throughput. Although Transaction 2 is self-conflicting, this can be considered spurious as it adds a unique ID for the product each time it executes.

*4.5.3 Transaction 3 : Increments the quantity of an existing product.* - this transaction affects both the Product table (Node 2) and the Inventory table (Node 3). This transaction is divided into two hops. The first hop checks if the product is present in the Product table. If the product exists, its quantity is incremented by the specified amount, and the client is notified of the successful completion. Meanwhile, the second hop updates the Inventory table on Node 3. If the product is not found, the coordinator sends a message to the client indicating that no product with the given product ID exists, and no further hops are executed. Although Transaction 3 is self-conflicting, this conflict is considered spurious as the operation logically increments the product quantity by a specific number each time.

*4.5.4 Transaction 4 : Place Order.* - This transaction involves placing a new order, which updates the Inventory, Order, and OrderInformation tables (Node 3) and decrements the product quantity in the Product table (Node 2). This transaction is divided into two hops. The first hop adds a new unique orderId to the Orders table and updates the OrderInformation table with the new orderId and the product details. Simultaneously, it decrements the product quantity in the Inventory table. If the product quantity is zero, the user is notified that the product cannot be ordered. Once this hop executes successfully, the client is informed of the successful completion, creating the impression that the entire transaction is complete. The second hop then executes on Node 2, decrementing the product count. Although Transaction 4 is self-conflicting, this conflict is considered spurious because it involves adding a unique orderId and decrementing the product quantity by one, which are logical operations that do not affect the overall transaction integrity.

*4.5.5 Transaction 5 : Update user email.* - This transaction updates the email address of an existing user with a new email provided by the client. This transaction is a single hop operation executed on Node 1. The process begins with the coordinator checking if the userId is present in the User table. If the userId is found, the coordinator updates the email address associated with that userId and immediately notifies the client of the successful update. If the userId is not found, the coordinator aborts the transaction and informs the client that the specified userId does not exist, ensuring no further processing occurs.

*4.5.6 Transaction 6 : Updating Product Price.* - It is a single-hop operation that updates the price field in the Product table on Node 2. The coordinator first checks if the product is available. If the product exists, the coordinator updates its price and notifies the client of the successful update. If the product does not exist, the coordinator informs the client that the product is not found, and no further actions are taken.

*4.5.7 Transaction 7 : Update Product Information.* - The transaction involves updating both the price and quantity in the Product table on Node 2, followed by updating the quantity in the Inventory table on Node 3. This transaction is a two-hop operation. In the first hop, the coordinator checks if the product exists. If the product is found, the coordinator updates the price and quantity, and notifies the client of the successful update. If the product does not exist, the coordinator aborts the transaction, and no further hops are executed. This operation is self-conflicting and cannot be resolved due to its nature. Additionally, Transaction 7 may form conflicts with Transactions 2, 3, 4, and 6, creating cycle conflicts that cannot be resolved.

## 4.6 Cycle Handling and Data Structures

To illustrate cycle handling, consider the scenario where transactions T2, T3, and T4 access both Node 2 and Node 3, leading to a potential cycle. However, the cycles involving T2, T3, and T4 are considered spurious since they add unique IDs and perform increments and do not work on same productId, we are assuming that those transactions do not access the same data item, respectively. [1] Mentions that C-edges connect vertices of different transactions if they access the same item and an access is a write. Origin ordering ensures that T2's hops are executed before T4's and T3's hops if T2 starts first. Although T3 and T4 might begin execution simultaneously with T2, they must wait until the transactions ahead in the queue are completed and committed. This approach ensures efficient execution and avoids conflicts.

When a transaction begins, the coordinator, aided by static analysis, determines the appropriate nodes for each hop of the transaction and places the hops into the respective node transaction queues. This ensures that if T2 is the first to reach Node 2, then its corresponding hop is also the first to be executed on Node 3. Even if T3 and T4 start at the same time as T2, they will wait for the completion and commitment of all preceding transactions in the queue.

For handling cycles that cannot be resolved through origin ordering, such as the cycles formed by transactions T4 and T7, T2 and T7, or T3 and T7, the coordinator ensures that T7 waits until all other transactions are executed. A dedicated data structure tracks the currently running transactions, ensuring that T7 waits until this structure is empty. This mechanism prevents conflicts and maintains consistency. The data structure is essential for managing the concurrent execution of transactions, ensuring that any transaction causing a cycle waits until it can be safely executed.

## 4.7 Directed Sibling edges

Even with directed edges, the directed cycles formed can still be considered as SC cycles. Below, we address the implications and propose solutions for handling these cycles under two specific conditions: (a) transactions with at most two hops, and (b) arbitrary transactions with multiple hops.

*4.7.1 a. Transactions with at Most Two Hops.* For transactions with at most two hops and with directed edges, cycles can indeed be formed. Hence, these directed cycles are considered SC cycles. Consider an example where one transaction updates the product table (Transaction 7) and another places an order (Transaction 4). Both transactions involve Node 2 and Node 3. The cycle created by these transactions, with or without directed edges, cannot be resolved through origin ordering. To handle this issue, we ensure that Transaction 7 is executed only after the completion of Transaction 4. This approach guarantees efficient execution and prevents conflicts by avoiding simultaneous execution of conflicting transactions.

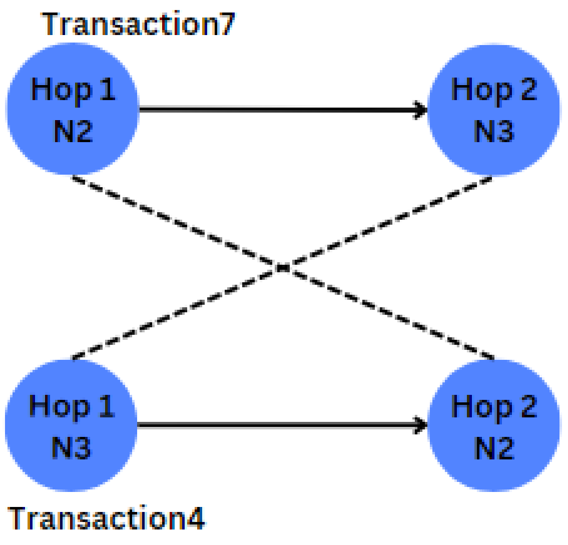


**Figure 8: Directed cycle with atmost 2 hops.**

*4.7.2 b. Any Arbitrary Transactions.* For transactions with an arbitrary number of hops, consider a transaction that deletes a seller. This transaction, a three-hop process, involves deleting the seller's details from the Seller table on Node 1, deleting related product details on Node 2, and removing the product from the Inventory table on Node 3. This operation is self-conflicting and also conflicts with Transaction 4. Since these two transactions cannot start simultaneously in parallel as origin ordering cannot be maintained, we execute them sequentially to avoid conflicts. By executing one transaction after the other, we ensure consistency and prevent the formation of unresolved cycles, thus maintaining the integrity of the transaction process.

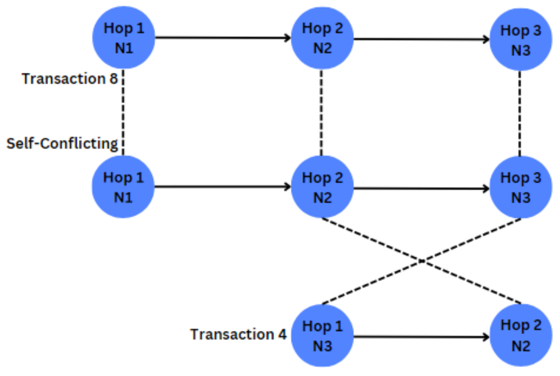


**Figure 9: Directed cycle with arbitrary transactions**

## 5 Performance Analysis

This section provides a detailed performance analysis of the concurrent transaction system, focusing on the execution times and throughput of seven distinct transaction types. All the 7 transactions are executed concurrently and each transaction was executed 100 times, and the average latencies for the first hop and the total execution times were calculated. The results were then analyzed to understand the performance characteristics and the system's overall efficiency.

### 5.1 Experimental Setup

The system was designed to handle concurrent transactions distributed across three nodes. The transactions include user addition, product addition, product quantity increment, order placement, user email update, product price update, and combined product quantity and price update. The MongoDB databases were used to simulate the transaction processing environment, with separate instances for different nodes to reflect a distributed system. To ensure that transactions were processed concurrently, multiple threads were used. Each thread was responsible for executing a specific transaction type. Synchronization mechanisms were implemented to manage the order of operations and ensure data consistency across the distributed system.

### 5.2 Results

The average first hop latency and the average total latency for each transaction type over 100 runs are summarized in Table 1.

### 5.3 Graphical Representation

Two bar graphs were created to represent the latencies and throughput.

Figure 1 shows the comparison between first Hop Latency i.e. the average time taken for the first hop of each transaction, highlighting the initial response time perceived by the system and Total Latency i.e. the average total execution time for each transaction, providing insight into the overall performance and efficiency of the transaction processing system.

Figure 2 shows Throughput which is was calculated as the inverse of the average total latency for each transaction type, representing the number of transactions processed per second.

### 5.4 Discussion

The performance analysis reveals several key insights into the transaction processing system:

*5.4.1 Transaction 1 and Transaction 5:* Transaction 1 adds new user and transaction 5 updates current user's email ID. Both transactions exhibit low first hop and total latencies, resulting in high throughput. This indicates that these transactions are efficient and can be processed quickly, likely due to their relatively simple operations and minimal contention for resources. Also transaction 1 and 5 are about updating user's table which is run on default port of mongodb. So the latencies are less as compared to the other transactions which update databases on different ports

Table 1: Average Latency and Throughput for Each Transaction Type

| Transaction | Avg 1st Hop Latency (seconds) | Avg Total Latency (seconds) | Throughput (transactions/second) |
|---|---|---|---|
| Transaction 1 | 0.0665 | 0.0668 | 14.97 |
| Transaction 2 | 0.6286 | 0.6988 | 1.43 |
| Transaction 3 | 0.8942 | 0.9342 | 1.07 |
| Transaction 4 | 1.3853 | 1.5123 | 0.66 |
| Transaction 5 | 0.0677 | 0.0677 | 14.76 |
| Transaction 6 | 2.0578 | 2.0676 | 0.48 |
| Transaction 7 | 0.0500 | 2.9844 | 0.34 |

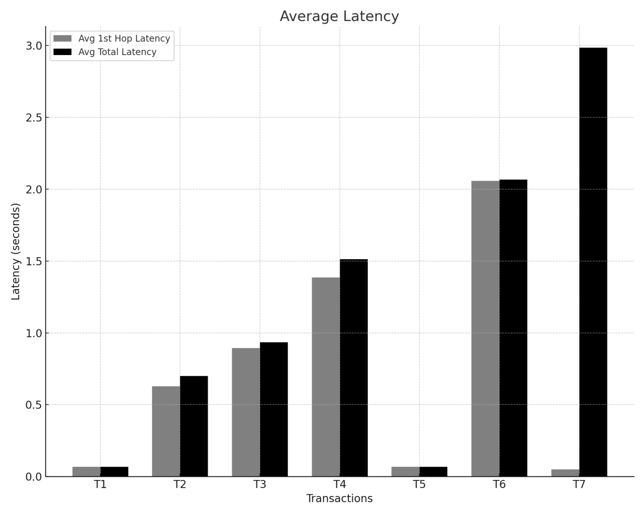


Figure 10: Average 1st hop and overall latencies.

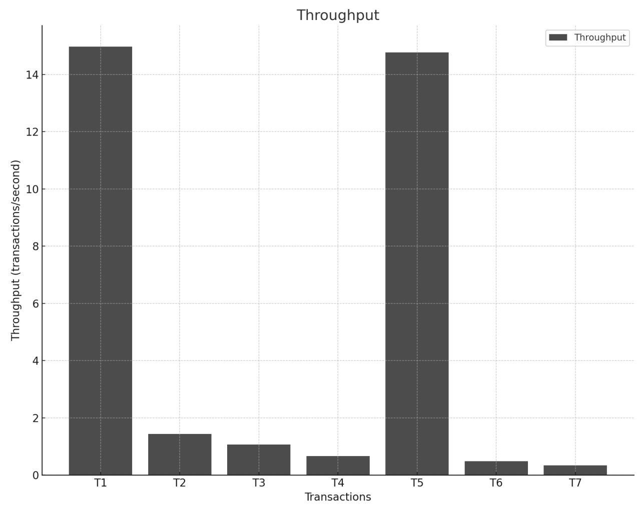


Figure 11: Throughput.

*5.4.2 Transaction 2 and Transaction 3:* These transactions are about adding product and incrementing product quantity. These transactions have moderate first hop and total latencies, leading to lower throughput compared to Transactions 1 and 5. The additional complexity and resource contention involved in product-related operations contribute to the increased latencies.

*5.4.3 Transaction 4:* It involves order placement, has higher latencies and lower throughput. This is expected due to the multiple steps involved in checking inventory, updating order information, and ensuring data consistency across nodes.

*5.4.4 Transaction 6:* The product price update transaction shows the highest latency and lowest throughput, highlighting the overhead associated with updating product information across distributed databases.

*5.4.5 Transaction 7:* The combined update transaction exhibits the lowest first hop latency but the highest total latency. This is because transaction 7 introduces an SC-cycle and hence has to wait for the executions of other transactions to to get completed.

The performance analysis demonstrates that the transaction processing system can efficiently handle concurrent transactions with varying complexities. The low-latency transactions (Transactions 1 and 5) benefit from quick execution times, whereas more complex transactions (Transactions 4, 6, and 7) face increased latencies due to additional processing and coordination overheads.

Future work will focus on optimizing the high-latency transactions to improve overall system throughput and efficiency, potentially through techniques such as enhanced indexing, more efficient data propagation methods, and further concurrency control optimizations. This performance evaluation provides a solid foundation for understanding and improving the transaction processing capabilities of the system.

## 5.5 Scalability Analysis with Increased Load

We extended this experiment to evaluate the system's performance under higher loads by increasing the number of requests to 500 and 1000 respectively. During these extended experiments, the database grew proportionally in size, accommodating the additional data generated by the increased number of transactions. Despite the substantial increase in the number of requests and the resulting growth in database size, we did not observe any significant differences in the average first hop latency, overall latency, or throughput for each transaction type. These results suggest that the system can efficiently handle higher loads without a noticeable degradation in performance, demonstrating its robustness and scalability.

## 6 Conclusion

In this paper, we presented a transaction processing system designed for a geo-distributed e-commerce application, focusing on achieving serializable transactions with low latency. By leveraging the concepts of transaction chopping and SC-Cycles, our system effectively manages distributed transactions across multiple nodes ensuring high availability, consistency, and scalability.

The performance analysis demonstrates that the system can efficiently handle concurrent transactions of varying complexities. Low-latency transactions, such as user addition and email updates, benefit from quick execution times and high throughput. In contrast, more complex transactions, including order placement and product updates, face increased latencies due to additional processing and coordination overheads.

The results indicate that our system provides a practical solution to the challenges of managing distributed transactions in a geo-distributed environment. It achieves a balance between high availability and robust transactional integrity, offering a compelling approach for modern e-commerce applications. Overall, this research highlights the practical benefits of adopting transaction chopping principles and SC-Cycles in real-world e-commerce scenarios, providing valuable insights into the ongoing evolution of online commerce systems.

## 7 References

[1] https://dl.acm.org/doi/pdf/10.1145/2517349.2522729