Bookchop: A Geo-Distributed Library Management System with Transaction Chopping

Jared Macshane
Carol Hsu

ABSTRACT

This paper presents Bookchop, a novel geo-distributed library management system that leverages transaction chopping techniques to achieve high performance while maintaining strong consistency guarantees. The system employs a hierarchical architecture comprising a central library and multiple local libraries, each maintaining its own database instance. By decomposing complex library operations into smaller, independent hops, Bookchop enables concurrent execution of transactions across distributed locations while preserving ACID properties. The system implements four primary transaction types—book borrowing, returning, overdue checking, and hold placement—along with support for dynamic, customer-defined transactions. Our implementation utilizes a carefully designed partitioning strategy and conflict analysis framework to ensure safe transaction chopping while maximizing system throughput. Through this approach, Bookchop addresses the challenges of managing large-scale, geographically distributed library operations while maintaining data consistency and operational efficiency.

1 INTRODUCTION

The rapid growth of books in libraries and the increasing demand for efficient and reliable access to information have required the development of robust and scalable library management systems. Traditional centralized library systems, while effective for smaller-scale operations, struggle to meet the needs of large-scale, geographically distributed libraries. To address these challenges, we propose Bookchop, a novel geo-distributed library management system that leverages transaction chopping to achieve high performance and strong consistency guarantees.

Inspired by the concept of transaction chains[8], Bookchop introduces a distributed transaction protocol that decomposes complex transactions into smaller independent hops. These hops are executed consecutively across multiple data centers, significantly reducing latency and improving system throughput. By carefully coordinating these subtransactions and ensuring their atomicity, consistency, isolation, and durability (ACID) properties, Bookchop provides a reliable and efficient solution for managing library operations.

Our system architecture comprises a central library and multiple local libraries, each with its own database. The central library maintains a global view of the library's resources and state and handles complex operations that require coordination across multiple sites. Local libraries manage their respective collections and execute simple transactions autonomously. By carefully partitioning data and operations in transaction, Bookchop strikes a balance between centralization and decentralization, optimizing performance, scalability, and make data consistent.

In Bookchop, we implement four primary library transaction and dynamic transaction:

- Borrow a book: When a member requests to borrow a book, a transaction is initiated to check the book's availability and the member's eligibility. If the book is available and the member is not overdue, the transaction is chopped into two hops to create a new record in the central library and update book's status where the local library where the book is located.
- Return a book: Upon returning a book, a similar transaction chopping process is employed to update the book's status in both the central and local libraries.
- Check member's overdue books: To determine if a member has overdue books, the system checks the records and update member's status . The global check queries the central library's records to identify any overdue books based on their due dates. while the local check update the member's current status (Active, Inactive, Overdue)
- Hold a book: When a member places a hold on a book, a transaction is initiated to checks the member's status locally, and update the record in the central library.
- Dynamic transaction: Customer-defined transaction by the central database to orchestrate complex operations involving multiple local libraries. These transactions can be customized to suit specific library workflows and business rules.

2 BACKGROUND AND RELATED WORK

2.1 Transaction Chopping

Transaction chopping[7] is an advanced technique in database systems that aims to increase concurrency while maintaining the correctness and consistency of transactional executions. The key concept behind transaction chopping is to decompose large, monolithic transactions into smaller, independent units of execution, known as transaction hops. These hops represent atomic units of work that encapsulate specific database operations with defined consistency guarantees. By chopping transactions into these smaller hops, the system can execute different portions of the transaction in parallel, either across local or distributed nodes, significantly enhancing both inter-transaction and intra-transaction concurrency.

By original design in Transaction chopping[7] and implementation in Lynx[8] In the context of transaction chopping, what is "chopped" are the individual operations within a transaction. This can involve breaking the transaction into smaller units by isolating specific read or write operations, reordering instructions, or using techniques such as multiversion read consistency. Each of these units, or hops, is designed to function independently, ensuring they can be executed concurrently while still maintaining the overall consistency and serializability of the transaction. We will implement this process using a strategy in 4.1 that organizes and manages these hops effectively.

2.2 Related Work

According to Transaction Chopping: Algorithms and Performance Studies [7] and its further study[1], numerous transaction chopping techniques have been developed since 1995. By breaking down large transactions into smaller sub-transactions, these techniques enable systems to achieve higher concurrency, improved performance, and greater fault tolerance. Furthermore, dynamic transaction chopping approaches, such as DTC: A Dynamic Transaction Chopping Technique for Geo-Replicated Storage Services [6], further enhance the flexibility and adaptability of transaction management across diverse environments.

2.2.1 Geo-distributed d storage systems. Cassandra, Megastore[2], and Spanner[4] are three powerful distributed storage systems designed to address the challenges of scalability and availability in modern applications. Cassandra excels in its high availability and scalability, making it ideal for handling large volumes of unstructured data. Megastore offers strong consistency guarantees, making it suitable for applications that require precise data accuracy, while also providing high performance and scalability. Spanner, developed by Google, stands out for its ability to provide strong consistency and high availability at a global scale, making it a compelling choice for mission-critical applications. While each system offers unique advantages, they all face challenges in providing strong consistency in a geo-distributed setting. Traditional transaction mechanisms can be expensive and introduce significant latency, limiting their effectiveness in these environments.

2.2.2 Concurrency control. : Concurrency control can enhance the throughput of transaction processing systems by breaking down transactions into smaller steps and allowing the steps of concurrent transactions to be interleaved. While some systems permit all interleavings, others restrict certain interleavings to maintain correctness. Lots of study in Concurrency control for step-decomposed transactions[3]

2.2.3 Transaction Decomposition Implementation in early years. Sagas [5] in 1987, a concept introduced to address the challenges posed by long-lived transactions (LLTs), which can hold onto database resources for extended periods and delay the completion of shorter, more frequent transactions. A saga is a type of LLT that can be decomposed into a sequence of smaller, independent transactions that can be interleaved with other transactions. This means that rather than locking resources for a prolonged period, a saga allows the system to continue processing other transactions concurrently.

The key feature of a saga is its ability to guarantee either complete success for all its transactions or the execution of compensating transactions to correct any partial execution. This guarantees that the saga as a whole either completes successfully or is fully rolled back to a consistent state, similar to the behavior of a traditional transaction, but without the need to hold resources for long periods.

The implementation of sagas is relatively straightforward, but it can significantly enhance performance by improving resource utilization and reducing wait times for other transactions. Although sagas can be executed on existing systems that do not natively support them, there are various techniques that can be used for their integration, such as designing the database and the LLTs in a way that makes it feasible to break them into smaller transactional units. Overall, the simplicity of the saga concept makes it a highly useful mechanism for improving the performance of long-lived transactions. It can be implemented with minimal effort, either as part of the database management system or as an add-on facility. Once in place, sagas can be leveraged by a large number of LLTs to significantly enhance system performance and scalability.

3 APPLICATION DESIGN

3.1 Database Schema

Our distributed library management system employs a hybrid architecture with both local and central database instances. The schema design reflects the requirements for managing book inventory, member information, and transaction records across distributed locations while maintaining consistency and enabling efficient transaction chopping.

3.1.1 Central Database Schema. The central database serves as the primary source of truth and maintains three core tables:

Table 1: Central Database Schema

Table	Field	Type	Constraints
books	book_id	INT	PK
	title	VARCHAR	NOT NULL
	status	VARCHAR	CHECK
members	id	INT	PK
	name	VARCHAR	NOT NULL
	status	VARCHAR	CHECK
records	trans_id	INT	PK, AUTO
	book_id	INT	FK
	member_id	INT	FK
	due	DATE	
	type	VARCHAR	CHECK

3.1.2 Local Database Schema. Local database instances maintain a subset of the central schema, focusing on frequently accessed data:

Table 2: Local Database Schema

Table	Field	Type	Constraints
books	book_id	INT	PK
	title	VARCHAR	NOT NULL
	status	VARCHAR	CHECK
members	id	INT	PK
	name	VARCHAR	NOT NULL
	status	VARCHAR	CHECK

3.2 Data Partitioning Strategy

Our system implements a hierarchical partitioning strategy that balances local autonomy with global consistency requirements. The partitioning approach is designed to minimize cross-partition transactions while enabling efficient transaction chopping.

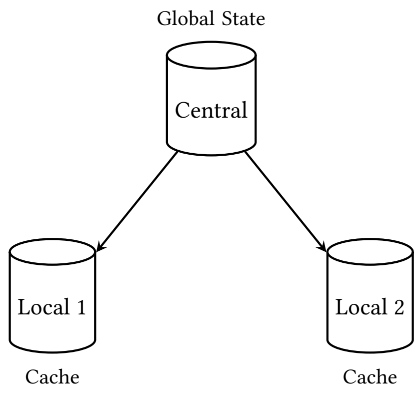

Figure 1: Hierarchical Data Partitioning Architecture

3.2.1 Partitioning Principles.

- **Vertical Partitioning:** The records table is maintained exclusively in the central database to ensure transaction consistency for borrowing and returning operations.
- **Horizontal Partitioning:** Book and member data are replicated to the central database instance
- **Consistency Boundaries:** Local instances maintain eventually consistent copies of books and members tables, while the central records table ensures strong consistency for critical operations.

3.3 Transaction Analysis

Our system implements three primary transactions: book borrowing, book returning, and overdue checking. We analyze these transactions for potential SC-cycles to ensure safe transaction chopping.

3.3.1 Transaction Decomposition.

3.3.2 Conflict Analysis. We identify three critical types of conflicts between the transactions:

1. **Borrow-Return Conflicts:**
   - RW conflict between book status read in borrow and update in return
   - WW conflict between book status updates
2. **Borrow-Overdue Conflicts:**
   - RW conflict between record insertion and overdue record reading
   - Affects member status determination
3. **Transaction Dependencies:**
   - Overdue check depends on accurate record history
   - Member status updates affect future borrow operations

3.3.3 Transaction Chopping Strategy. To maintain consistency while enabling concurrent execution, we implement the following chopping strategy:

This chopping strategy ensures that:

- Resource and record pieces maintain strict serializability
- Status updates are properly isolated
- No SC-cycles can occur between chopped pieces

Table 3: Transaction Pieces and Isolation Levels

Piece	Operations	Isolation Level
Resource	Book read, Book update	Serializable
Record	Record operations	Serializable
Status	Member status update	Serializable

4 SYSTEM DESIGN

4.1 Transaction Chopping Protocol

The transaction chopping protocol in Bookchop implements a two-phase execution strategy that ensures consistency across distributed nodes while maximizing concurrency. Each transaction is decomposed into independent hops that can be executed across local and central nodes.

4.1.1 Protocol Components.

- **Transaction Coordinator:** Manages the distributed transaction execution between local and central nodes, maintaining a registry of pending transactions and handling responses.
- **Transaction Hops:** Atomic units of execution that encapsulate specific database operations with defined consistency guarantees.
- **Message-based Communication:** Asynchronous message passing between nodes for coordination and status updates.

4.1.2 Protocol Phases.

1. **Local Execution Phase:**
   - Validate local preconditions (e.g., book availability, member eligibility)
   - Execute local state changes
   - Maintain transaction state
   - Send transaction request to central node
2. **Central Coordination Phase:**
   - Central node validates global constraints
   - Updates global state if validation succeeds
   - Sends completion or failure notification to local node
   - Ensures system-wide consistency

4.2 Dynamic Transaction Handling

We propose a dynamic transaction chopping algorithm that parses as input a set of database operations and outputs a set of transaction hops. The algorithm will scan the set of operations and group together operations that operate on the same data object. Placing the operations in a single hop will reduce the number of conflict edges of the SC graph.

Unfortunately, due to time constraints and implementation difficulties, we were not able to fully implement and evaluate this algorithm in time for this report. Part of implementation design detail in section 4.3.6

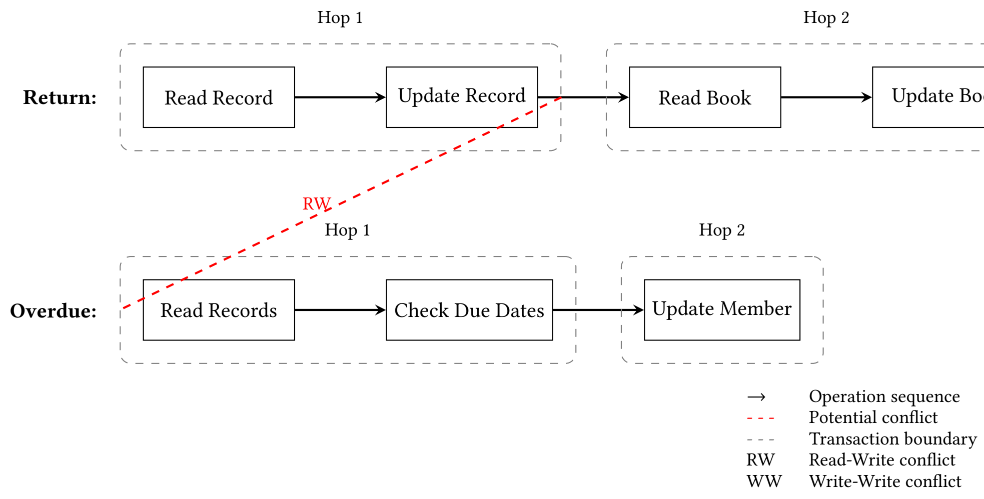

Figure 2: SC-cycle Analysis of Library Transactions

Algorithm 1 Dynamic Transaction Chopping

```
Require: Set of operations $O = \{o_1, o_2, ..., o_n\}$
Ensure: Set of transaction hops $H$
 1: $H \leftarrow \emptyset$                                  ▷ Initialize empty set of hops
 2: $currentHop \leftarrow \emptyset$
 3: $currentObject \leftarrow null$
 4: for $op \in O$ do
 5:     if $currentObject = null$ then
 6:         $currentObject \leftarrow getDataObject(op)$
 7:         $currentHop \leftarrow currentHop \cup \{op\}$
 8:     else if $getDataObject(op) = currentObject$ then
 9:         $currentHop \leftarrow currentHop \cup \{op\}$
10:     else
11:         $H \leftarrow H \cup \{currentHop\}$              ▷ Add completed hop
12:         $currentHop \leftarrow \{op\}$
13:         $currentObject \leftarrow getDataObject(op)$
14:     end if
15: end for
16: if $currentHop \neq \emptyset$ then
17:     $H \leftarrow H \cup \{currentHop\}$                  ▷ Add final hop
18: end if
        return $H$
```

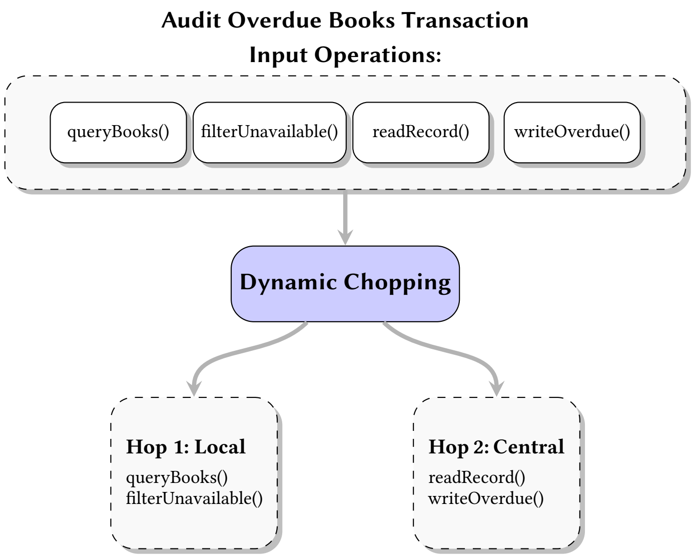

Figure 3: Example of dynamic transaction chopping for an overdue books audit operation

4.3 Implementation Details

The implementation utilizes a modular architecture with clear separation of concerns:

4.3.1 Technical Stack.

- **Programming Language:** Python 3, chosen for its robust standard library and expressive object-oriented features
- **Database:** SQLite3, selected for its lightweight nature and ACID compliance without external dependencies
- **Communication:** Custom message-passing system implemented using Python's socket programming

4.3.2 Code Organization. The codebase is structured into five primary modules:

- **local.py:** Implements local node functionality including database operations and transaction coordination
- **central.py:** Contains central node logic for global state management
- **transaction.py:** Defines the core transaction classes and hop abstractions
- **communication.py:** Handles inter-node messaging and network protocols
- **test_transactions.py:** Contains comprehensive test suites

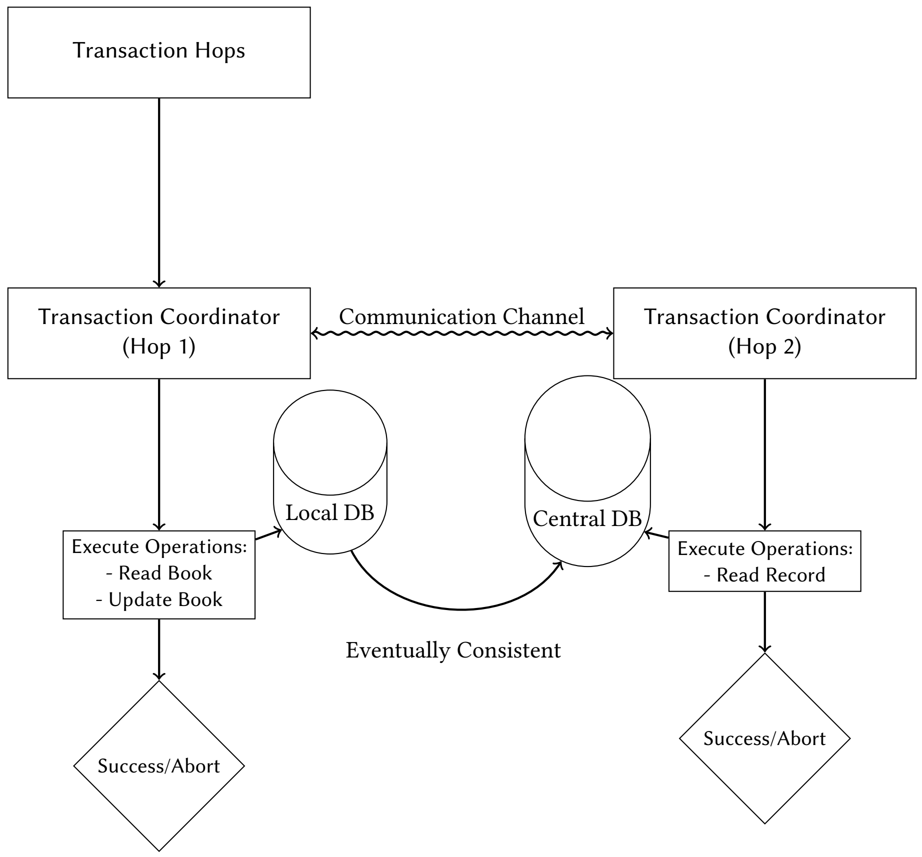
Transaction Hops
Transaction Coordinator (Hop 1)
Communication Channel
Transaction Coordinator (Hop 2)
Local DB
Central DB
Execute Operations: - Read Book - Update Book
Execute Operations: - Read Record
Eventually Consistent
Success/Abort
Success/Abort

Figure 4: Transaction Processing Layer

4.3.3 *Class Hierarchy.* The system employs a hierarchical class structure:

- **Database Classes:**
  - `LocalDatabase`: Manages SQLite connections and local operations
  - `CentralDatabase`: Manages SQLite connections and global state
  - `Book` and `Member`: Data classes for entity representation
- **Transaction Classes:**
  - `TransactionCoordinator`: Orchestrates distributed transactions
  - `TransactionHop`: Base class for atomic operations
  - Specialized hop classes (e.g., `BookBorrowHop`)

4.3.4 *Error Handling and Recovery.* The implementation includes robust error handling mechanisms:

- **State Management:** Consistent state maintenance across distributed operations
- **Message Retry:** Reliable communication between nodes
- **State Verification:** Consistency checks before and after each operation

4.3.5 *Optimization Techniques.* Several optimizations enhance system performance:

- **Minimal Locking:** Transaction chopping reduces lock duration
- **Asynchronous Communication:** Non-blocking message passing between nodes
- **Local Caching:** Frequently accessed data maintained in local nodes

4.3.6 *Dynamic chopping handler.* To support customized transactions, it must separate operation in transaction:

- **Operation.py:** Operation Read or Write to specify data object(Tables) in different location
- **Transaction.py:** For customers to enter its customized operations to its transaction.
- **transaction_process_layer.py:** Handle chopping logic

5 EXPERIMENTAL EVALUATION

5.1 Experimental Setup

The experimental evaluation was conducted on a distributed library management system implementing a two-phase commit protocol. The system consists of a central coordinator and local nodes, with the following configuration:

- Database: SQLite for both central and local nodes
- Network: Local TCP/IP communication
- Test Duration: 200 transactions across three operation types

- Member Distribution: 40 members across three status categories

5.2 Transaction Performance

We've constructed a number of tables to present the results of the experiment. We list the number and types of transactions Table 4, the member status distribution Table 5, the operation execution times Table 6, and the network communication latency Table 7.

Metric	Value
Total Transactions	200
Successful Transactions	140
Failed Transactions	60
Success Rate	70.0%

Table 4: Transaction Performance Summary

Member Status	Count	ID Range
Active	20	4–23
Overdue	10	24–33
Inactive	10	34–43

Table 5: Member Status Distribution

We constructed a variety of member status to test and verify the transactions under different conditions. We can see that the system is able to reject on the first hop and save time by not sending the transaction to the central library if it fails the first hop such as a member not being active or book not being available.

Operation Type	Count	Avg (s)	Min (s)	Max (s)
BookBorrowHop	50	0.004	0.001	0.008
ReturnBookHop	50	0.004	0.001	0.007
PlaceHoldHop	40	0.002	0.001	0.004

Table 6: Operation Execution Times

Component	Messages	Avg (s)	Min (s)	Max (s)
NodeServer	140	0.204	0.200	0.205

Table 7: Network Communication Latency

5.3 Performance Analysis

Note that the local library is able to service a borrow and return transaction in 0.004 seconds. Whereas if the transaction had to operate with the central library before servicing the request, the transaction would take 0.204 seconds. While this may seem like a small difference, it is important to note that other factors such as network latency and load could significantly affect this round trip time. If we were to simulate dropped packets then the round trip time would be significantly higher.

The experimental results demonstrate a clear performance advantage of local transaction processing over centralized operations. The 50x difference in execution time (0.004s vs 0.204s) highlights the effectiveness of our transaction chopping strategy. This performance gap becomes particularly significant in high-traffic scenarios, where the cumulative effect of transaction latency can substantially impact user experience and system throughput. Furthermore, the success rate of 70% indicates that our local validation approach effectively filters invalid transactions before they reach the central node, preventing unnecessary network communication and reducing system load.

The operation execution times shown in Table 6 reveal that PlaceHoldHop operations are notably faster than both BookBorrowHop and ReturnBookHop operations, with average execution times of 0.002 seconds compared to 0.004 seconds. This efficiency can be attributed to the simpler nature of hold operations, which require fewer state changes and consistency checks. The consistent minimum execution times across all operation types (0.001s) suggest that our implementation maintains stable baseline performance regardless of operation complexity.

5.4 Scalability Assessment

Because of the asynchronous communication design, each independent local library can service a transaction in parallel. This allows the system to scale to a large number of local libraries. The central library is not a bottleneck or limiting factor since the transactions are only sent to the central library once the local library has processed enough to ensure the transaction will commit.

6 CONCLUSION

The experiment demonstrates the system's effectiveness in efficiently handling transactions with varying member statuses. Results show that local libraries process borrow and return transactions faster than when involving the central library, which is crucial for improving performance, particularly when network latency or load affects round-trip times.

The system's design, with hierarchical data partitioning and the transaction chopping protocol, ensures concurrent and efficient transaction processing across distributed nodes. The asynchronous communication model allows local libraries to operate independently, scaling seamlessly as more libraries are added. This design prevents the central library from becoming a bottleneck, as it only participates once the local library has completed sufficient processing. Overall, Bookchop offers a robust solution balancing efficiency, consistency, and scalability across the network.

7 FUTURE WORK

7.1 Real-Environment Testing

Should include testing the system in real-world environments with variable network conditions, including network latency, packet loss, and fluctuating bandwidth. Simulating these real-world network factors will help assess how the system performs under conditions that differ from controlled test environments. Evaluating the impact of these factors on transaction times and overall system reliability will provide valuable insights for optimizing communication protocols and improving system resilience.

7.2 Handling More Complex Transactions

Involve more complex transactions that include a broader range of operations and longer processing times. These complex transactions, such as multi-step borrowing, returns, reservations, and inter-library requests, will push the limits of the current design. The goal is to examine how the system handles dependencies between transactions, maintain consistency, and ensure high throughput when faced with more intricate workflows.

8 CODE AVAILABILITY

The complete implementation of Bookchop, including the transaction chopping protocol, database schemas, and test suites, is available in our public repository at https://github.com/Jared-Mac/cs223-project/.

REFERENCES

[1] 2003. Appendix B - Transaction chopping. In *Database Tuning*, Dennis Shasha and Philippe Bonnet (Eds.). Morgan Kaufmann, San Francisco, 305–324. https://doi.org/10.1016/B978-155860753-8/50013-5

[2] Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. 2011. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *Proceedings of the Conference on Innovative Data system Research (CIDR)*. 223–234. http://www.cidrdb.org/cidr2011/Papers/CIDR11_Paper32.pdf

[3] Arthur J. Bernstein, David S. Gerstl, and Philip M. Lewis. 1999. Concurrency control for step-decomposed transactions. *Information Systems* 24, 8 (1999), 673–698. https://doi.org/10.1016/S0306-4379(00)00004-1

[4] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2013. Spanner: Google's Globally Distributed Database. *ACM Trans. Comput. Syst.* 31, 3, Article 8 (Aug. 2013), 22 pages. https://doi.org/10.1145/2491245

[5] Hector Garcia-Molina and Kenneth Salem. 1987. Sagas. *SIGMOD Rec.* 16, 3 (Dec. 1987), 249–259. https://doi.org/10.1145/38714.38742

[6] Ning Huang, Lihui Wu, Weigang Wu, and Sajal K. Das. 2022. DTC: A Dynamic Transaction Chopping Technique for Geo-Replicated Storage Services. *IEEE Transactions on Services Computing* 15, 6 (2022), 3210–3223. https://doi.org/10.1109/TSC.2021.3089819

[7] Dennis Shasha, François Llirbat, Eric Simon, and Patrick Valduriez. 1995. Transaction Chopping: Algorithms and Performance Studies. *ACM Transactions on Database Systems* 20 (03 1995), 325–. https://doi.org/10.1145/211414.211427

[8] Yang Zhang, Russell Power, Siyuan Zhou, Yair Sovran, Marcos K. Aguilera, and Jinyang Li. 2013. Transaction Chains: Achieving Serializability with Low Latency in Geo-distributed Storage Systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 276–291. https://doi.org/10.1145/2517349.2522729