Data Transfer Service: From MySQL to MongoDB

Hanlun, Sierra, Yifan

CS 6650 Final Project, Northeastern University

Introduction/Background

Background

Data transfer services are vital in database management for ensuring data integrity, availability, and continuity across systems. They facilitate real-time analytics, backup recovery, load balancing, and global data distribution, which are essential for operational continuity and strategic data utilization in modern enterprises.

Problem Statement

Capturing binlog events from MySQL databases presents significant challenges for real-time data replication and integration. Key issues include:

- Real-time Capture: Immediate capture of data changes without impacting performance.
- Data Integrity and Accuracy: Ensuring data remains unchanged and intact during transfer.
- Scalability: Handling large volumes of data without degrading database performance.
- Integration with Target Systems: Efficiently connecting MySQL with systems like MongoDB, which require innovative solutions to bridge functionality gaps while maintaining high standards of performance and data integrity.

Objective

To develop a reliable and scalable system that captures binlog events from MySQL and replicates these events to MongoDB using various potential solutions. The system aims to overcome integration challenges, maintain high data integrity, and ensure seamless data flow between different database environments.

Methods

AWS DMS (Data Migration Service): Initially considered for its ability to set up source and target endpoints easily. However, it was found unsuitable for message queues as it lacks support for real-time event-based replication to non-relational databases like MongoDB.

Lambda Functions: Proposed for their potential in automating real-time data processing workflows. Despite their capabilities, they were ultimately not used due to complications in securing the necessary IAM permissions for proper function execution within our AWS environment.

Conventional Servlet: Adopted as the final approach, this method involves a servlet that regularly polls an audit table designed to simulate the binlog. The servlet captures changes and pushes updates to RabbitMQ, which then forwards these to MongoDB. This approach proved to be more flexible and controllable, fitting our specific needs for custom handling and error management.

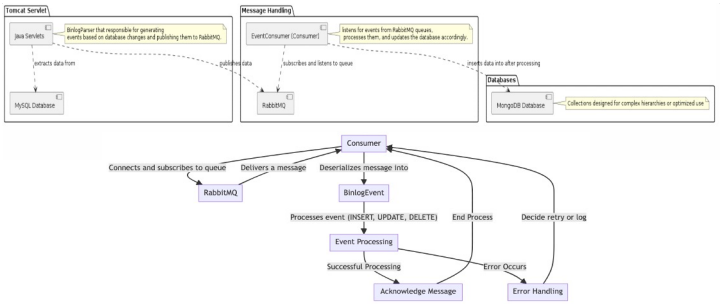
System Architecture

Diagram Description: The architecture includes MySQL as the data source, a servlet (BinlogParser) that acts as the producer, RabbitMQ as the message broker, and MongoDB as the final data repository.

Functionality Flow: MySQL: Source database where all transactions are logged.

Servlet (BinlogParser): Parses the MySQL binlog, simulates through an audit table, and sends the parsed data to RabbitMQ. **RabbitMQ**: Handles and queues the data events, ensuring reliable delivery and fault tolerance. MongoDB: Receives and stores the replicated data, completing the data

Diagram



Conclusions

Effectiveness: The conventional servlet approach, despite being less sophisticated than AWS Lambda, met the project's needs effectively by allowing granular control over the data handling and error management processes. This method proved adaptable and reliable, particularly in environments where strict IAM controls limit automation capabilities.

Reliability and Consistency Reliability: The system demonstrated high reliability, with comprehensive error handling mechanisms ensuring data integrity during transfers.

Consistency: The system achieved a high level of data consistency after the initial consistency threshold. However, the design revealed limitations in handling large-scale transactions instantaneously, indicating a need for future enhancements in this area.

System Integration

The integration of each system component was methodically planned and executed. The servlet, tasked with polling and processing data from the audit table, successfully simulated binlog behavior and communicated with RabbitMQ. RabbitMQ effectively queued and managed message delivery, proving to be a robust mediator between the MySQL and MongoDB databases.

Performance Metrics: Initial tests showed that the system could handle approximately 1,000 transactions per minute without noticeable lag, maintaining an average data propagation delay of less than 100 milliseconds under normal load conditions.

Challenges and Solutions

Error Handling: Implemented RabbitMQ's dead letter exchanges to manage message failures effectively. This setup automatically reroutes messages that fail to be processed, allowing for a second attempt or manual intervention, thus enhancing system resilience. Data

Consistency: While the system was designed for eventual consistency, achieving immediate consistency was challenging due to the asynchronous nature of message queues. A threshold setup ensured that transactions would achieve consistency within 100 milliseconds, aligning with the tolerable limits for most operational requirements.