CSE 512: Distributed Database Systems

Project Report: Distributed Transaction Management (Part 4)

Group Name: Data Dominators

1. Introduction

The objective of Part 4 of our project was to implement distributed transaction management, ensuring data consistency and concurrency control in our chosen topic. We accomplished this by adhering to the ACID properties in distributed databases, implementing concurrency control mechanisms, and utilizing distributed coordination for transactions.

2. Implementation

We selected PostgreSQL as our primary database system for this project. Additionally, we integrated Apache Ignite as our distributed caching solution to enhance performance and provide distributed coordination for transactions.

2.1 ACID-compliant Distributed Transactions

The foundation of our distributed transaction management lies in adhering to the ACID properties: Atomicity, Consistency, Isolation, and Durability. The begin_distributed_transaction, commit_distributed_transaction, and rollback_distributed_transaction functions collectively ensure Atomicity. These functions initiate, commit, and roll back transactions in a way that guarantees either the entire transaction succeeds, or none of its changes are persisted. This is vital to prevent partial updates and maintain the integrity of the data across the distributed databases.

```
def begin_distributed_transaction():
    # Begin distributed transaction
    conn_project.set_isolation_level(psycopg2.extensions.ISOLATION_LEVEL_AUTOCOMMIT)
    conn_child1.set_isolation_level(psycopg2.extensions.ISOLATION_LEVEL_AUTOCOMMIT)
    conn_child2.set_isolation_level(psycopg2.extensions.ISOLATION_LEVEL_AUTOCOMMIT)

    # Start Ignite transaction
    ignite_transaction = ignite.tx_start()

    return conn_project, conn_child1, conn_child2, ignite_transaction

def commit_distributed_transaction(ignite_transaction):
    # Commit PostgreSQL transactions
    conn_project.commit()
    conn_child1.commit()
    conn_child2.commit()

    # Commit Ignite transaction
    ignite_transaction.commit()

def rollback_distributed_transaction(ignite_transaction):
    # Rollback PostgreSQL transactions
    conn_project.rollback()
    conn_child1.rollback()
    conn_child2.rollback()

    # Rollback Ignite transaction
    ignite_transaction.rollback()
```

Consistency is addressed through the row-level lock acquired on the Comment table using the SQL statement SELECT * FROM Comment WHERE FALSE FOR UPDATE. This ensures that during the transaction, no other concurrent transactions can modify the data being processed, maintaining a consistent state. Isolation is achieved by setting the isolation level to ISOLATION_LEVEL_AUTOCOMMIT for each participating connection, preventing interference between transactions.

The commit and rollback functions handle the Durability aspect by persisting or discarding changes in both PostgreSQL and Ignite. If any error occurs during the transaction, the system ensures a rollback, preventing any partially applied changes from persisting and ensuring the durability of the data in its original state.

2.2 Concurrency Control Mechanisms

Concurrency control is crucial to prevent conflicts and maintain data consistency in a distributed environment. By utilizing a row-level lock on the Comment table, the system enforces a serialization of transactions. This means that each transaction is isolated from others, preventing interference during the critical section where data modifications occur. This approach aligns with the principles of Isolation in ACID, ensuring that transactions are executed independently without compromising data integrity.

```
with conn.cursor() as cursor:
    # Acquire a row-level lock on the Comment table
    cursor.execute("SELECT * FROM Comment WHERE FALSE FOR UPDATE")
```

The SELECT * FROM Comment WHERE FALSE FOR UPDATE statement is strategically placed within the add_comment function, acquiring a lock on the Comment table before insertion. This prevents multiple transactions from concurrently attempting to add a comment with the same comment_id, safeguarding against race conditions and maintaining the consistency of the Comment table across distributed databases.

2.3 Distributed Coordination

Our distributed transaction management leverages Apache Ignite for efficient coordination across distributed databases. Ignite acts as a distributed, in-memory cache, enhancing data retrieval speed and providing a synchronized environment for our transactions. Connections to Ignite are established, and a dedicated cache, 'my_cache,' is employed to store comments. Ignite transactions are seamlessly integrated into the workflow, ensuring that changes made in PostgreSQL databases are mirrored in the distributed cache.

```
# Retrieve the comment from Ignite cache
existing_comment = get_comment_from_cache(comment_id)

if existing_comment:
    # If comment is present in the cache then it has already been added to the database
    pass
```

The add_comment_distributed function exemplifies this coordination by checking the Ignite cache before inserting a comment into PostgreSQL databases. If the comment is present in the cache, redundant operations are skipped. Otherwise, the comment is added to the Ignite cache, maintaining a consistent state between primary databases and

the distributed cache. The commit and rollback functions guarantee coordinated data persistence or rollback across both PostgreSQL and Ignite, fulfilling the distributed coordination requirements of Part 4.

3. Code Structure and Usage

The code is organized into functions to facilitate modularity and readability. The add_comment_distributed function serves as an example of how distributed transactions can be utilized in practice. It accepts parameters for comment details, initiates a distributed transaction, adds the comment to the project database and child databases, and commits the transaction.

4. Testing and Error Handling

We have implemented a try-except block in the add_comment_distributed function to handle exceptions during the distributed transaction. If an error occurs, the system rolls back all changes, ensuring data integrity.

5. Conclusion

In conclusion, our implementation successfully meets the requirements of Part 4 by providing ACID-compliant distributed transactions, implementing concurrency control mechanisms, and utilizing Apache Ignite for distributed coordination. The code structure, coupled with effective error handling, ensures the reliability and consistency of distributed transactions in our project.

6. Future Enhancements

Potential enhancements include optimizing and fine-tuning the performance of distributed transactions, exploring additional features of Apache Ignite, and evaluating other distributed database systems for compatibility and performance improvements.

7. Deliverables

The deliverables for this part include the code/script provided, snapshots capturing the code execution, and this documentation, summarizing the implementation of distributed transaction management.