Transaction Manager

Big Data Energy

Rishabh Sukumaran (22CS10058)

Saishree Pradhan (22CS10089)

B Pranesh Vijay (22CS10013)

Byreddi Sri Chaitanya (22CS10018)

Siddharth Ambedkar CA (22CS10074)

Github Repo: https://github.com/praneshvijay/transaction-manager

1 Problem Statement

This project focuses on implementing a transaction management system that supports different isolation levels to ensure data consistency and integrity in concurrent database operations. The primary objectives are:

1. Develop a transaction manager that handles concurrent read and write operations while maintaining data consistency.
2. Implement four standard isolation levels: Read Uncommitted, Read Committed, Repeatable Read, and Serializable.
3. Demonstrate the behavior and differences of each isolation level through practical examples.
4. Implement rollback functionality and garbage collection to maintain system efficiency.

2 Methodology

2.1 System Architecture

The transaction management system consists of two main components:

1. **Database Class**: An in-memory database implementation that stores data and provides basic operations like read, write, commit, and rollback.
2. **Transaction Manager**: Manages transactions with different isolation levels and interacts with the database to perform operations.

2.2 Database Structure

The database is implemented as a collection of key-value pairs with additional metadata to track transaction history. Each database record consists of:

- **recent_commit**: Tracks the most recently committed transaction for a key
- **recent_write**: A deque tracking the history of write operations using live transactions
- **commit_value**: Maps transaction IDs to their respective values

This structure allows the system to maintain versioning information needed for different isolation levels.

2.3 Transaction Management

The `TransactionManager` class encapsulates a transaction's operations and manages the isolation level behavior. It provides the following core functionalities:

- **Transaction Initialization**: Assigns a unique transaction ID and sets the isolation level.
- **Read Operations**: Retrieves data based on the transaction's isolation level.
- **Write Operations**: Updates data in the database or in local change logs based on isolation level.
- **Commit**: Finalizes the transaction by persisting changes to the database.
- **Rollback**: Discards changes made by the transaction.

2.4 Snapshot-Based Implementation

A key component of our implementation is the snapshot-based approach for maintaining transaction isolation. This technique is particularly important for the Repeatable Read isolation level.

When a transaction reads data under the Repeatable Read isolation level, it uses the snapshot captured at the beginning of the transaction. The `db.read(key, last_commit_transaction)` method retrieves the value of the key as it existed at the time represented by `last_commit_transaction`. This ensures that the transaction sees a consistent view of the database throughout its lifetime, regardless of concurrent changes.

The database supports versioned reads through its implementation of the `read` method. This method implements a form of Multi-Version Concurrency Control by:

1. Tracking all versions of a value through the `commit_value` map
2. Using transaction IDs as timestamps to identify versions
3. Retrieving the appropriate version based on the requesting transaction's snapshot time

2.5 Isolation Levels Implementation

The implementation supports the following isolation levels:

2.5.1 Read Uncommitted

The lowest isolation level where transactions can read uncommitted changes made by other transactions, potentially causing "dirty reads." In this implementation:

- Read operations directly read the latest value, whether committed or not.
- Write operations are immediately visible to other transactions.

2.5.2 Read Committed

Ensures that a transaction only reads data that has been committed by other transactions, preventing dirty reads:

- Read operations return only committed values.
- The implementation tracks the last known state before a write to determine if conflicting updates occurred.
- A validation check during commit prevents lost updates by verifying that no other transaction modified the data since it was last read.

2.5.3 Repeatable Read

Guarantees that if a transaction reads a record multiple times, it will get the same value each time:

- The transaction creates a consistent snapshot of committed data at the start.
- All read operations use this snapshot throughout the transaction's lifetime.
- Changes made by other transactions after the snapshot was created are invisible to this transaction.

2.5.4 Serializable

The highest isolation level, providing complete isolation as if transactions were executed serially:

- The implementation uses a simple time-based approach to ensure serialization.
- A transaction waits until its predecessor completes before proceeding.
- This prevents anomalies like non-repeatable reads, and dirty reads.

2.6 Concurrency Control Mechanisms

The implementation uses several mechanisms to manage concurrency:

1. **Mutexes**: Used to protect critical sections of the database during operations.
2. **Transaction IDs**: Sequential identifiers assigned to each transaction for tracking.
3. **Change Logs**: Local copies of changes that can be committed or discarded.
4. **Version Control**: Through tracking of transaction IDs with values.

2.7 Garbage collection

A background thread periodically cleans up old transaction data to prevent memory bloat:

- Removes obsolete entries from recent_write deques
- Purges outdated values from commit_value maps
- Preserves data needed by active transactions

3 Results and Demonstration

3.1 READ UNCOMMITTED Demonstrations

Dirty Reads Demonstration

```
ishita@ishita-HP-240-G8-Notebook-PC:~/Desktop/transaction_manager$ make run
g++ mainfile.cpp -o tm
./tm

=== READ UNCOMMITTED: Dirty Reads Demonstration ===
T1 reads key 1: 0
T2 reads key 1: 0
T1 writes 100 to key 1 (uncommitted)
T1 reads key 1: 100
T2 reads key 1: 100 (DIRTY READ!)
T1 rolls back its changes
T2 reads key 1: 0
```

This demonstrates the classic dirty read problem. T2 can read T1's uncommitted changes (value 100). When T1 rolls back, T2 had been working with data that was never actually committed to the database.

Non-Repeatable Reads Demonstration

```
ishita@ishita-HP-240-G8-Notebook-PC:~/Desktop/transaction_manager$ make run
g++ mainfile.cpp -o tm
./tm

=== READ UNCOMMITTED: Non-Repeatable Reads Demonstration ===
T1 reads key 2: 0
T2 writes 200 to key 2
T2 commits
T1 reads key 2 again: 200 (NON-REPEATABLE READ!)
```

This shows that READ UNCOMMITTED also allows non-repeatable reads. T1 reads key 2 twice within the same transaction but gets different values because T2 modified and committed the data between those reads.

3.2 READ COMMITTED Demonstrations

Dirty Reads Prevention

```
ishita@ishita-HP-240-G8-Notebook-PC:~/Desktop/transaction_manager$ make run
g++ mainfile.cpp -o tm
./tm

=== READ COMMITTED: Dirty Reads Prevention ===
T1 reads key 3: 0
T2 reads key 3: 0
T1 writes 300 to key 3 (uncommitted)
T1 reads key 3: 300 (can see own uncommitted changes)
T2 reads key 3: 0 (cannot see T1's uncommitted changes)
T1 commits
T2 reads key 3 again: 300 (now sees committed changes)
```

This shows that READ COMMITTED prevents dirty reads. T2 cannot see T1's uncommitted changes to key 3. Only after T1 commits can T2 see the new value.

Non-Repeatable Reads Demonstration

```
ishita@ishita-HP-240-G8-Notebook-PC:~/Desktop/transaction_manager$ make run
g++ mainfile.cpp -o tm
./tm

=== READ COMMITTED: Non-Repeatable Reads Demonstration ===
T1 reads key 4: 0
T2 writes 400 to key 4
T2 commits
T1 reads key 4 again: 400 (NON-REPEATABLE READ!)
```

This demonstrates that READ COMMITTED still allows non-repeatable reads. T1 reads key 4 twice and gets different values because T2 committed changes between the reads.

3.3 REPEATABLE READ Demonstrations

Non-Repeatable Reads Prevention

```
ishita@ishita-HP-240-G8-Notebook-PC:~/Desktop/transaction_manager$ make run
g++ mainfile.cpp -o tm
./tm

=== REPEATABLE READ: Non-Repeatable Reads Prevention ===
T1 reads key 5: 0
T2 writes 500 to key 5
T2 commits
T1 reads key 5 again: 0 (still sees original value)
T1 commits
T3 (new transaction) reads key 5: 500
```

This shows that REPEATABLE READ prevents non-repeatable reads. T1 sees the same value for key 5 throughout its transaction, even though T2 has modified and committed changes to that key. A new transaction (T3) started after T2's commit will see the updated value.

3.4 SERIALIZABLE Demonstrations

Complete Isolation Demonstration

```
ishita@ishita-HP-240-G8-Notebook-PC:~/Desktop/transaction_manager$ make run
g++ mainfile.cpp -o tm
./tm

=== SERIALIZABLE: Complete Isolation Demonstration ===
T1 started
T1 reads key 6: 0
T1 writes 600 to key 6
T1 committed
T2 started
T2 reads key 6: 600
T2 writes 650 to key 6
T2 committed
T3 (new transaction) reads key 6: 650
```

This demonstrates serializable isolation where transactions appear to execute one after another. T2 waits for T1 to complete before proceeding, ensuring that transactions don't interfere with each other. The final value is 650 because T2 executed after T1 completed.

Analysis of Isolation Levels

Isolation Level	Dirty Reads	Non-repeatable Reads
Read Uncommitted	Possible	Possible
Read Committed	Prevented	Possible
Repeatable Read	Prevented	Prevented
Serializable	Prevented	Prevented

Our implementation reveals the performance trade-offs between isolation levels:

1. **Read Uncommitted**: Highest concurrency but lowest consistency.
2. **Read Committed**: Balance between concurrency and consistency, with additional overhead due to verification during commit.
3. **Repeatable Read**: Reduced concurrency due to snapshot maintenance, but strong consistency guarantees.
4. **Serializable**: Lowest concurrency but highest consistency, with potential for significant wait times.

4 Conclusion and Future Scope

This project successfully implemented a transaction management system supporting the four standard isolation levels. The implementation demonstrates the different behaviors and consistency guarantees provided by each level.

The results confirm the theoretical trade-offs between consistency and concurrency in database transactions, with higher isolation levels providing stronger consistency guarantees at the cost of reduced concurrency.

Future Enhancements:

- Integration of deadlock detection and resolution
- Support for phantom reads and range-based queries
- Visualization tools for transaction timelines and conflicts
- Performance benchmarking across isolation levels

5 References

- Abraham Silberschatz, Henry Korth, and S. Sudarshan, “Database System Concepts”, McGrawHill
- PostgreSQL Documentation - https://www.postgresql.org/docs/current/transaction-iso.html