INSTITUT D'ENSEIGNEMENT SUPÉRIEUR DE RUHENGERI

FACULTY OF APPLIED FUNDAMENTAL SCIENCES

DEPARTMENT OF COMPUTER SCIENCE

OPTION OF SOFTWARE ENGINEERING

Scientia et Lux

GROUP 4 ASSIGNMENT 5

Year III SWE Group A

Members

MUKUNZI Fabrice 23/21864

NSANZAMAHORO Fabrice 23/23413

HABUMUGISHA Egide 23/21873

Musanze, October 2024

B.P. 155
Ruhengeri
Rwanda

T : +250 788 90 30 30
: +250 788 90 30 32
W : www.ines.ac.rw
E : inesruhengeri@yahoo.fr

implementing isolation in databases

using a Student Management System.

We explained some concepts

Isolation in Databases

Isolation ensures that transactions operate independently without interference from one another. In a Student Management System, various operations may occur simultaneously, such as enrolling students, updating grades, or processing payments.

Ex: Isolation Levels in a Student Management System:

- Read Uncommitted:

- Example: A teacher queries a student's grades while a transaction is in progress to update those grades. The teacher might see outdated information, leading to confusion.

Query ex:

```
-- Querying grades without considering uncommitted changes
SELECT * FROM Grades WHERE student_id = 1;
```

- Read Committed:

- Example: A student checks their grades. If an instructor updates a grade and commits the transaction, only then will the student see the updated grade. If the instructor is still in the process of updating and hasn't committed yet, the student will see the previous grade.

Query ex:

```
-- Update a grade and commit the transaction
BEGIN TRANSACTION;
UPDATE Grades SET grade = 'A' WHERE student_id = 1;
```

COMMIT;

- Repeatable Read:

  - Example: A student views their grades multiple times during an enrollment period. They should see the same grades each time they check, regardless of other transactions that may be happening, like other students changing their grades or enrolling.

Query ex:

```
-- Ensuring repeatable read for student grades

BEGIN TRANSACTION;

SELECT * FROM Grades WHERE student_id = 1;

-- This grade will be consistent for the duration of the transaction

COMMIT;
```

- Serializable:

  - Example: Two students trying to enroll in the same course at the same time. One student's transaction should block the other until it is complete to ensure that course capacity is not exceeded.

Query ex:

```
-- Locking the course for enrollment to prevent race conditions

BEGIN TRANSACTION;

SELECT * FROM Courses WHERE course_id = 1 FOR UPDATE;

UPDATE Courses SET enrollment_count = enrollment_count + 1 WHERE course_id = 1;

COMMIT;
```

2. Termination

   Termination in a transaction context refers to successfully completing or aborting a transaction.

Examples:

- Commit:

  - When a student enrolls in a course, the transaction commits if the student is successfully added to the class roster, updating both the student and course records.

  Query ex:

```
-- Committing an enrollment transaction
BEGIN TRANSACTION;
INSERT INTO Enrollments (student_id, course_id) VALUES (1, 1);
COMMIT;
```

- Rollback:

  - If a student attempts to enroll in a full course and the system detects that the enrollment would exceed the limit, the transaction rolls back, and the student is informed that enrollment was unsuccessful.

Query ex:

```
-- Rolling back an enrollment attempt due to full capacity
BEGIN TRANSACTION;
DECLARE @enrollment_limit INT;
SELECT @enrollment_limit = max_enrollment FROM Courses WHERE course_id = 1;

IF (SELECT COUNT(*) FROM Enrollments WHERE course_id = 1) >= @enrollment_limit
BEGIN
   ROLLBACK; -- Enrollment failed
   PRINT 'Enrollment unsuccessful: Course is full.';
END
ELSE
```

```
BEGIN
    INSERT INTO Enrollments (student_id, course_id) VALUES (1, 1);
    COMMIT; -- Successful enrollment
END
```

3. Distributed Deadlock

Distributed Deadlock occurs when multiple transactions are waiting on each other to release locks, preventing any of them from proceeding.

Example:

- Scenario:

  - Transaction A is trying to update the student information of Student 1, which requires a lock on Resource A(Student table).

  - Transaction B is trying to update the enrollment status of Course 1, which requires a lock on Resource B (Course table).

  - If Transaction A locks Resource A and then tries to lock Resource B while Transaction B has locked Resource B and is trying to lock Resource A, a deadlock occurs.

Resolution Techniques:

- Deadlock Detection: The system can periodically check for deadlocks by analyzing resource allocation graphs and abort one of the transactions to allow the other to proceed.

- Deadlock Prevention: Use a timeout for locks. If a transaction cannot obtain all required locks within a certain time frame, it rolls back, allowing others to proceed.

4. Global Serialization

Global Serialization ensures that transactions across different nodes or databases are executed in a serial order.

Example:

- Two-Phase Commit Protocol (2PC):

  - When a student enrolls in a course, this might involve multiple databases (e.g., one for student data and one for course data).

  - Prepare Phase: The system sends a "prepare" request to both databases to ensure they are ready to commit changes.

  - Commit Phase: If both databases respond affirmatively, the system sends a commit command to finalize the enrollment.

This ensures that either both databases commit the changes or neither does, maintaining consistency.

5. Replicated Databases

Replicated Databases involve maintaining copies of the same data across multiple nodes to enhance reliability and availability.

Example:

- Scenario: A Student Management System could replicate its databases across different campuses.

  - Consistency Challenges: If a student updates their contact information at one campus, that change must be replicated to all other campuses.

  - Conflict Resolution: If two campuses attempt to update the same student record simultaneously, the system must resolve which update to keep. This could involve using timestamps or a last-write-wins strategy.

6. Distributed Transactions in the Real World

In a Student Management System, distributed transactions may involve multiple services or databases. For example:

- Scenario: A student registers for a new semester, which requires updating several entities:
  - Student Record: Update the student's enrollment status.
  - Course Record: Update the course's enrollment count.
  - Payment System: Process any tuition fees associated with enrollment.

Distributed Transaction Flow:

1. Initiation: The student submits a registration form.

2. Transaction Creation: A distributed transaction is initiated that spans multiple services (student management, course management, and payment processing).

3. Two-Phase Commit:
   - Prepare Phase: Each service prepares for the transaction and ensures it can commit.
   - Commit Phase: If all services agree, the transaction commits across all services.

4. Compensation Transactions: If any part of the transaction fails (e.g., payment processing fails), compensation transactions may be invoked to roll back any successful updates (e.g., changing the enrollment status back).

Therefore we saw that

While implementing a Student Management System, the implementation of isolation, termination, handling of distributed deadlocks, ensuring global serialization, managing replicated databases, and coordinating distributed transactions are critical for maintaining data integrity and consistency. Each of these aspects plays a vital role in ensuring that the system functions smoothly, especially when multiple users and processes are involved simultaneously.

Understanding these concepts helps in designing robust systems that can handle various transaction scenarios effectively while maintaining a high level of data accuracy and user satisfaction.