Mock Assignment #5

Question 1.

Based on the given database scenario and concurrent transactions, determine the appropriate isolation levels.

Consider an online banking system with the following database schema:

- Account(account_id, customer_id, balance, account_type)
- Transaction_Log(log_id, account_id, transaction_type, amount, timestamp)

For each of the following transaction scenarios, determine the **lowest appropriate isolation level** and explain your reasoning:

(a) Transaction T1 performs a single account balance update:

```
-- T1:
UPDATE Account
SET balance = balance + 500
WHERE account_id = 12345;
COMMIT;
```

Consider other possible concurrent transactions that might read or modify account data.

Solution: READ COMMITTED. Since T1 only performs a single write operation to the Account table, we need to prevent other transactions from reading dirty (uncommitted) data. READ COMMITTED ensures that:

- No dirty reads occur (other transactions cannot see the balance update until T1 commits)
- The transaction completes quickly, minimizing lock duration
- No unrepeatable reads or phantom issues since T1 doesn't perform multiple reads

READ UNCOMMITTED cannot be used because it only supports read-only transactions, and T1 contains an UPDATE operation.

(b) Transaction T2 generates a monthly account statement:

```
-- T2:
SELECT customer_id, balance, account_type
FROM Account
WHERE customer_id = 67890;

SELECT log_id, transaction_type, amount, timestamp
FROM Transaction_Log
WHERE account_id IN (
    SELECT account_id FROM Account WHERE customer_id = 67890
);
COMMIT;
```

Solution: REPEATABLE READ. Transaction T2 performs multiple read operations across two tables (Account and Transaction_Log) that must be consistent with each other. The isolation level requirements are:

- No dirty reads: Prevents reading uncommitted balance changes
- No unrepeatable reads: Ensures account balances and types don't change between the two SELECT statements
- Allows phantoms: New transaction log entries could appear, but this doesn't affect the consistency of the statement generation

READ COMMITTED would be insufficient because account balances could change between the two queries, leading to inconsistent statement data.

(c) Transaction T3 calculates and updates interest for all savings accounts:

```
-- T3:
SELECT account_id, balance
FROM Account
WHERE account_type = 'SAVINGS';

-- For each account found, calculate 2% interest
UPDATE Account
SET balance = balance * 1.02
WHERE account_type = 'SAVINGS';

-- Log the interest transactions
INSERT INTO Transaction_Log
SELECT NEXTVAL('log_seq'), account_id, 'INTEREST', balance * 0.02, NOW()
FROM Account
WHERE account_type = 'SAVINGS';
COMMIT;
```

Solution: SERIALIZABLE. Transaction T3 requires the highest isolation level because:

- It performs multiple operations (SELECT, UPDATE, INSERT) that must be consistent
- The UPDATE affects multiple rows based on the initial SELECT results
- New savings accounts could be added by concurrent transactions (phantom reads), which would lead to inconsistent interest calculations
- The transaction must appear to execute in complete isolation to ensure all savings accounts are processed exactly once

Any lower isolation level could result in:

- Missing newly created savings accounts (phantom problem)
- Inconsistent balance calculations if accounts are modified concurrently
- Incorrect transaction logging

Question 2.

Analyze the performance implications of different file organizations and indexing strategies.

Consider a customer database with 1,000,000 customer records. Each record contains: Customer(customer_id, name, email, city, registration_date, status)

(a) If customer records are stored in an unordered file, estimate the number of disk block accesses required for the following query in the worst case and average case:

```
SELECT * FROM Customer WHERE customer_id = 12345;
```

Assume each disk block can hold 100 customer records.

Solution: With 1,000,000 records and 100 records per block, there are 10,000 disk blocks total.

For an unordered file with sequential search:

- **Worst case:** 10,000 block accesses (record is in the last block or doesn't exist)
- **Average case:** 5,000 block accesses (record is found halfway through the file on average)

(b) If the same records are stored in a file ordered by customer_id, estimate the number of disk block accesses for the same query using binary search.

Solution: For an ordered file with binary search:

- Number of blocks to search: $\log_2(10,000) \approx 14$ block accesses
- Both worst case and average case are approximately the same: 14 block accesses

This represents a significant improvement: from 5,000 average accesses to 14 accesses.

(c) Suppose you create a single-level index on customer_id. The index has one entry per customer record. If each index entry is 8 bytes (4 bytes for customer_id + 4 bytes for record pointer) and each disk block can hold 500 index entries, calculate the number of disk block accesses required to find a customer record.

Solution: Index calculation:

- Total index entries: 1,000,000
- Index entries per block: 500
- Total index blocks: 1,000,000 $\div$ 500 = 2,000 blocks

Binary search on index blocks: $\log_2(2,000) \approx 11$ block accesses

Total access cost:

- Index search: 11 block accesses
- Data record access: 1 block access
- **Total: 12 block accesses**

(d) Based on your calculations, rank the three approaches (unordered file, ordered file, indexed file) from best to worst performance for equality searches. Explain one advantage and one disadvantage of the indexed approach compared to the ordered file approach.

Solution: Performance ranking (best to worst): 1. Indexed file: 12 block accesses 2. Ordered file: 14 block accesses 3. Unordered file: 5,000 block accesses (average)

Indexed approach vs. Ordered file:

- **Advantage:** Slightly better search performance (12 vs 14 accesses) and easier insertion/deletion operations (no need to maintain physical order)
- **Disadvantage:** Additional storage overhead for the index (2,000 extra blocks) and increased complexity in maintaining the index during updates

Question 3.

Design and analyze a database system that handles both transaction isolation and indexing requirements.

You are designing a library management system with the following tables:

- Book(isbn, title, author, category, copies_available)
- Member(member_id, name, email, membership_type)
- Checkout(checkout_id, member_id, isbn, checkout_date, due_date, returned_date)

(a) The system needs to handle the following concurrent transaction. Determine the minimum isolation level required and explain your reasoning:

Transaction T_CHECKOUT: A member checks out a book

```
-- T_CHECKOUT:
SELECT copies_available FROM Book WHERE isbn = '978-0123456789';
-- If copies_available > 0:
UPDATE Book SET copies_available = copies_available - 1
WHERE isbn = '978-0123456789';
INSERT INTO Checkout
VALUES (NEXTVAL('checkout_seq'), 12345, '978-0123456789',
        CURRENT_DATE, CURRENT_DATE + 14, NULL);
COMMIT;
```

Consider that multiple members might try to check out the last copy of the same book simultaneously.

Solution: Minimum isolation level: SERIALIZABLE

Reasoning:

- The transaction performs a **read-then-write** pattern that is susceptible to race conditions
- If two members simultaneously try to check out the last copy, both might see `copies_available = 1` in their SELECT statements
- Without SERIALIZABLE isolation, both transactions could proceed to update and insert, resulting in `copies_available = -1` (invalid state)
- SERIALIZABLE ensures that concurrent executions appear as if they were run sequentially, preventing the lost update problem
- Lower isolation levels would allow phantom reads or unrepeatable reads that could lead to data inconsistency in the book inventory

This is a classic example where business logic requires the highest isolation level to maintain data integrity.

(b) Design an indexing strategy for this library system. For each table, specify what indexes should be created and justify your choices based on expected query patterns:

Expected queries:

- Find books by ISBN (frequent)
- Find books by author (frequent)
- Find books by category (moderate)
- Find member information by member_id (frequent)
- Find all checkouts for a member (frequent)
- Find all current checkouts (books not yet returned) (frequent)
- Find overdue books (daily batch job)

Solution: Recommended indexes:
Book table:

- `CREATE UNIQUE INDEX idx_book_isbn ON Book(isbn);` - Primary access pattern, ensures uniqueness
- `CREATE INDEX idx_book_author ON Book(author);` - Frequent searches by author
- `CREATE INDEX idx_book_category ON Book(category);` - Moderate frequency, supports browsing

Member table:

- `CREATE UNIQUE INDEX idx_member_id ON Member(member_id);` - Primary key access
- `CREATE INDEX idx_member_email ON Member(email);` - Login functionality (assuming email-based login)

Checkout table:

- `CREATE UNIQUE INDEX idx_checkout_id ON Checkout(checkout_id);` - Primary key
- `CREATE INDEX idx_checkout_member ON Checkout(member_id);` - Find member's checkout history
- `CREATE INDEX idx_checkout_isbn ON Checkout(isbn);` - Find checkout history for specific books
- `CREATE INDEX idx_checkout_returned ON Checkout(returned_date);` - Find current checkouts (WHERE returned_date IS NULL)
- `CREATE INDEX idx_checkout_due ON Checkout(due`$_{d}ate); -Findoverduebooksefficiently$

(c) The library runs a nightly batch job to identify overdue books and send reminder emails. This job must run consistently even with concurrent checkout/return transactions. Design the batch transaction and specify:

1. The appropriate isolation level
2. The complete SQL query
3. How to handle the scenario where books are returned while the batch job is running

Solution: 1. Isolation level: REPEATABLE READ
The batch job needs consistent data throughout its execution but can tolerate new checkouts (phantoms) that occur during processing. REPEATABLE READ prevents the overdue status from changing during the job execution.
2. Complete SQL query:

```
-- Batch transaction for overdue books
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;

SELECT c.checkout_id, c.member_id, b.title, b.author,
       m.name, m.email, c.checkout_date, c.due_date,
       CURRENT_DATE - c.due_date as days_overdue
FROM Checkout c
JOIN Book b ON c.isbn = b.isbn
JOIN Member m ON c.member_id = m.member_id
WHERE c.returned_date IS NULL
  AND c.due_date < CURRENT_DATE
ORDER BY c.due_date ASC;

COMMIT;
```

3. Handling concurrent returns:

- The REPEATABLE READ isolation level ensures that once the batch job starts, the set of overdue books remains consistent throughout the transaction
- Books returned after the batch job starts will still appear as overdue in this run, but will be excluded from the next day's batch
- This is acceptable behavior - it's better to send one extra reminder than to miss overdue books
- The `returned_date IS NULL` condition, combined with REPEATABLE READ, ensures consistent results
- Alternative: Use a timestamp-based approach where the batch job processes books overdue as of a specific point in time