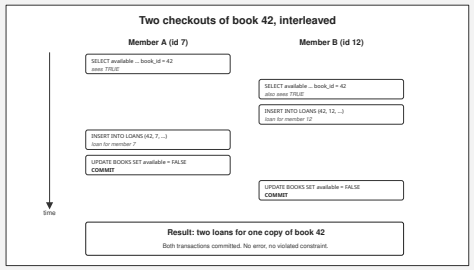
Isolation Recap

- *isolation* means that concurrent transactions should behave as if they ran one at a time



not ensuring isolation can result in invalid state

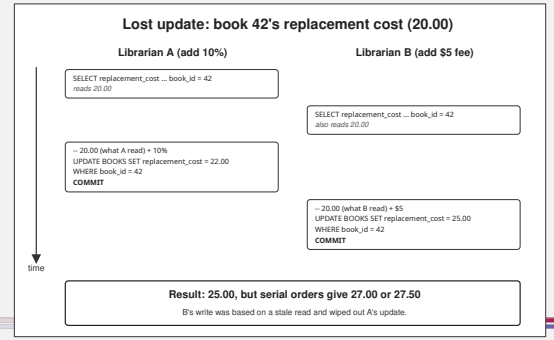
- full isolation carries a cost in overhead (providing for isolation) and lost concurrency

Serializability

- a *schedule* is an interleaving of the operations of concurrent transactions
- a schedule is *serializable* if its result matches some serial order of the same transactions
  - which order is not prescribed – A then B or B then A are both fine
- serializability guarantees correctness
- weaker isolation levels trade the serializability guarantee for greater efficiency, allowing some anomalies

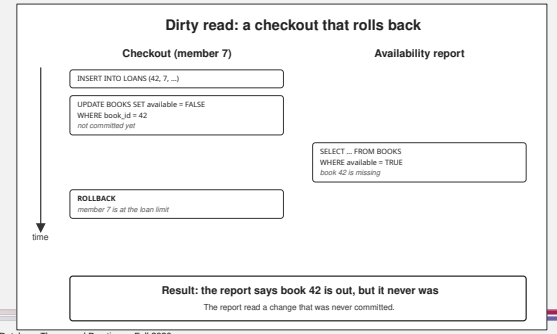
Anomalies – Lost Update

- two read-modify-write cycles operate on the same row
- the second write is based on a stale read and wipes out the first



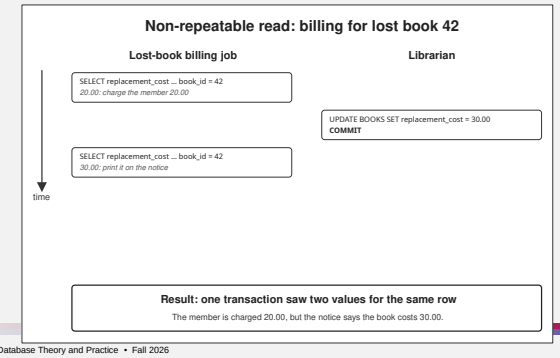
Anomalies – Dirty Read

- a change that was never committed was read and acted on – half-done work is seen
  - breaks consistency – invalid intermediate states inside transactions were allowed because no one could see them



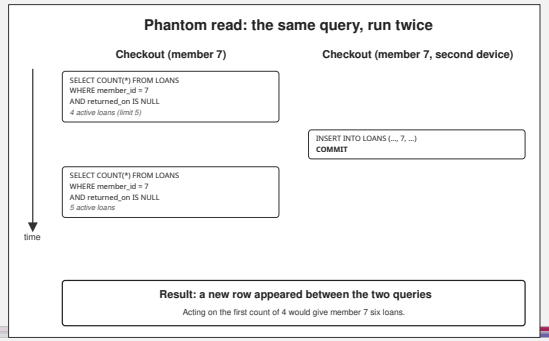
Anomalies – Non-Repeatable Read

- same row, read twice in the same transaction with different results
  - not a dirty read – the change was committed



Anomalies – Phantom Read

- same query, run twice by the same transaction with different results



Isolation Levels in Practice

- the SQL standard defines four levels according to which anomalies each must prevent

isolation level	dirty read	lost update	non-repeatable read	phantom read
read uncommitted	possible	possible	possible	possible
read committed	prevented	possible	possible	possible
repeatable read	prevented	prevented	prevented	possible
serializable	prevented	prevented	prevented	prevented

- DBMSes vary in –
  - which levels are supported
  - what each level actually guarantees
  - how each is enforced – waiting for locks vs aborting a conflicting transaction
- most variable is `REPEATABLE READ`

Isolation Levels in Practice

- isolation levels govern how statements within one transaction interact with other transactions
- with autocommit on, each statement is its own transaction
  - protection spanning several statements needs an explicit transaction

Isolation Levels in MySQL

- `SERIALIZABLE`
  - `SELECT` locks the rows it reads and the gaps between them so no other transaction can modify those rows or insert into those gaps until this transaction completes
    - applies only within a transaction – a plain `SELECT` takes no locks
  - blocks if another transaction has uncommitted changes to those rows
  - prevents lost updates
  - dirty reads are not permitted
  - reads are repeatable, phantoms prevented
  - when possible, instead use locking reads on just the rows involved at a lower isolation level
    - e.g. banking transactions – lock the accounts involved
    - e.g. enforcing 5-book checkout limit – lock the member's row
  - limit `SERIALIZABLE` to when it is difficult to identify the specific set of rows to protect
    - there is a significant performance cost
    - applications must handle retrying transactions that fail

Isolation Levels in MySQL

- `READ UNCOMMITTED`
  - transaction sees what has happened in other transactions, even if not committed
  - repeated `SELECT`s may return different data
  - lost updates are possible
  - dirty reads are permitted
  - reads may be non-repeatable, phantoms are possible
  - only appropriate in limited circumstances
    - e.g. approximate results where nobody acts on exact values, such as a dashboard showing roughly how many books are checked out right now
  - rarely useful in MySQL
    - offers little in the way of performance gains

Isolation Levels in MySQL

- `READ COMMITTED`
  - each `SELECT` sees data committed before it started
    - other transactions' uncommitted changes are never visible
  - repeated `SELECT`s may return different data
  - lost updates are possible
  - dirty reads are not permitted
  - reads may be non-repeatable, phantoms are possible
  - appropriate when each statement should see the latest committed data
    - e.g. single-query reports or inventory lists, or ones where small inconsistencies between queries don't matter
    - e.g. an overdue-notice job that works through loans one by one and should see returns made while the job is running
  - less blocking and fewer deadlocks than `REPEATABLE READ`
  - the default in some DBMSes (not MySQL)

Isolation Levels in MySQL

- `REPEATABLE READ`
  - all plain reads within a transaction are consistent – reads are repeatable
    - the first plain `SELECT` establishes a snapshot used by all subsequent plain `SELECT`s
    - rows another transaction inserts, deletes, or updates after that point aren't seen, even once committed
    - prevents phantom reads for plain reads
  - locking reads, `UPDATE`, and `DELETE` act on the latest committed rows, and the rows they change become visible
    - lost updates are possible
  - dirty reads are not permitted
  - deadlocks are still possible, so retries may be needed
  - appropriate when a calculation needs consistent results from multiple `SELECT`s
    - e.g. lost book billing job – compute the fine and print the notice with separate `SELECT`s
  - the default in MySQL

Setting Isolation Levels in MySQL

```
SET [ GLOBAL | SESSION ]
    TRANSACTION ISOLATION LEVEL level
```

- `GLOBAL` sets for all subsequent sessions (but not the current one)
  - requires admin privileges to use
- `SESSION` sets for all subsequent transactions in the current session
- no keyword sets for just the next transaction

See the current session isolation level with

```
SELECT @@transaction_isolation
```


Setting Isolation Levels in JDBC

```
try (Connection conn = DB.getConnection()) {
    // set before the transaction starts
    conn.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE);
    conn.setAutoCommit(false);
    ...
}
```

- levels are `int` constants on `Connection`
  - `TRANSACTION_READ_UNCOMMITTED, TRANSACTION_READ_COMMITTED, TRANSACTION_REPEATABLE_READ, TRANSACTION_SERIALIZABLE`
- the setting applies to the connection until changed
- set before the transaction starts
  - changing mid-transaction is driver-dependent

```
// the connection's current level
int level = conn.getTransactionIsolation();
if (level == Connection.TRANSACTION_REPEATABLE_READ) {
    ...
}

// the DBMS's default level
int defaultLevel = conn.getMetaData().getDefaultTransactionIsolation();
```


Locking Reads in MySQL

```
IF ( SELECT balance FROM ACCOUNT WHERE acctnum=123 ) >= 100 THEN
  UPDATE ACCOUNT SET balance=balance-100 WHERE acctnum=123;
  UPDATE ACCOUNT SET balance=balance+100 WHERE acctnum=456;
END IF
```

- a negative balance can result if another transaction modifies account 123's balance between the `SELECT` and `UPDATE`
  - `SERIALIZABLE` prevents this by failure
    - both transactions lock the row for reading, both try to update, deadlock is detected, and transaction is rolled back and must be retried
- `SELECT … FOR SHARE` sets a shared lock
  - other transactions can read those rows but not modify them
  - if two transactions hold the lock and try to update, they deadlock; one is rolled back
- `SELECT … FOR UPDATE` sets an exclusive lock
  - blocks locking reads and modifications by other transactions
  - better choice for read-then-write scenarios
- locking reads apply only within a transaction and don't block plain reads

Deadlock

Deadlock occurs when two or more transactions are blocked waiting to acquire a lock held by one of the others.

- the DBMS detects the deadlock and terminates and rolls back at least one of the transactions involved
  - application retries failed transactions
  - MySQL lock wait timeout (error 1205) is different from deadlock (error 1213) – in that case the application must roll back the transaction itself

To reduce the likelihood of deadlock –

- minimize the use of locks – use higher isolation levels and locking reads only when required
- keep transactions small and short in duration – less likely to have two simultaneous ones
- take locks in a consistent order each time
  - e.g. for bank transfers, lock the lower account number first

MySQL Transactions

MySQL defaults –

- each statement is a transaction
- isolation level is `REPEATABLE READ`

Limitations –

- there are statements that can't be rolled back, and others that involve an implicit commit
  - e.g. data definition statements (`CREATE TABLE`, `DROP TABLE`, …)
  - check the MySQL manual for details
- triggers cannot contain `START TRANSACTION`, `COMMIT`, or `ROLLBACK`
  - check the MySQL manual for details on what happens if a trigger or the operation itself fails

Takeaways

- transactions should be considered when operations involve more than one SQL statement
  - locking reads provide targeted protection at lower isolation levels
- which isolation level to use?
  - short answer – `READ COMMITTED` and `REPEATABLE READ` (MySQL default) are most common
  - longer answer

> `InnoDB` supports each of the transaction isolation levels described here using different locking strategies. You can enforce a high degree of consistency with the default `REPEATABLE READ` level, for operations on crucial data where ACID compliance is important. Or you can relax the consistency rules with `READ COMMITTED` or even `READ UNCOMMITTED`, in situations such as bulk reporting where precise consistency and repeatable results are less important than minimizing the amount of overhead for locking. `SERIALIZABLE` enforces even stricter rules than `REPEATABLE READ`, and is used mainly in specialized situations, such as with XA transactions and for troubleshooting issues with concurrency and deadlocks.

  - but there are many subtleties and implementations vary between DBMSes – for serious use, consult your DBMSes documentation

https://dev.mysql.com/doc/refman/8.4/en/innodb-transaction-isolation-levels.html

Fixing Checkout

- use `SELECT … FOR UPDATE` to take an exclusive lock on the book being checked out
  - add a `findByIdForUpdate` DAO method with the locking read
- this is sufficient to fix the problem –
  - the second checkout's `SELECT … FOR UPDATE` will wait until the first commits
  - it then sees that the book is unavailable
    - locking read sees the latest committed value at `REPEATABLE READ`
- notes
  - every read-then-write path needs the locking version
    - anything else that reads availability and then acts on it must use `findByIdForUpdate` too
    - `GET /books/{id}` and other plain reads can still use `findById`
  - a 5-loan limit needs a separate fix
    - this change makes checkouts for the same book wait for each other
    - a limit needs a member-row lock to make checkouts for the same member wait for each other

Fixing Checkout

```
// Checkout's version: reads the book and locks its row until the caller's
// transaction ends, so a concurrent checkout of the same book waits here

public Optional<Book> findByIdForUpdate(Connection conn, int bookId) {
    String sql = "SELECT " + COLUMNS + " FROM BOOKS WHERE book_id = ? FOR UPDATE";
    try (PreparedStatement ps = conn.prepareStatement(sql)) {
        ps.setInt(1, bookId);
        try (ResultSet rs = ps.executeQuery()) {
            if (rs.next()) {
                return Optional.of(map(rs));
            }
            return Optional.empty();
        }
    } catch (SQLException e) {
        throw SqlErrors.translate(e);
    }
}
```