Data Integrity and Security

Road Map – Data Integrity and Security

- seen so far
  - database constraints
- today
  - making multi-step operations safe
- coming up
  - making concurrent operations safe
  - security on the database end
    - MySQL accounts and permissions
    - views as a security mechanism
  - protecting against bad input
  - auditing: who changed what

Example

To transfer $100 from account 123 to account 456:

- check that the balance of account 123 is at least $100
- deduct $100 from account 123
- add $100 to account 456

```
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
```

What can go wrong?

- simultaneous transfers from account 123 can result in a negative balance
- a crash or a failure of one of the `UPDATE`s can result in a loss of money

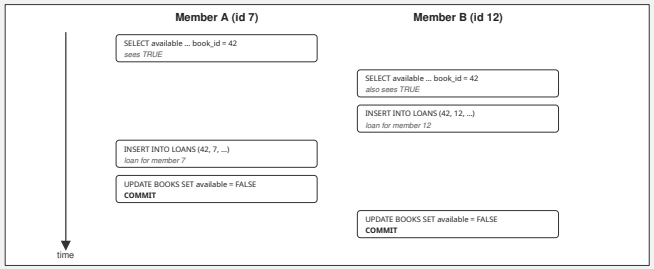
Example

- library checkout
  - read the book – does it exist and is it available?
  - `INSERT INTO LOANS …`
  - `UPDATE BOOKS SET available = FALSE …`

```
public Loan checkout(int memberId, int bookId) {
    Optional<Book> book = bookDao.findById(bookId);
    if (book.isEmpty()) {
        throw new NotFoundException("No book with id " + bookId);
    }
    if (!book.get().available()) {
        throw new ConflictException("Book " + bookId + " is already checked out");
    }
    LocalDate today = LocalDate.now();
    Loan loan = new Loan(null, bookId, memberId, today, today.plusDays(14), null);
    int loanId = loanDao.insert(loan);
    // crash here!
    bookDao.setAvailable(bookId, false);
    return new Loan(loanId,bookId,memberId,loan.checkedOutOn(),loan.dueOn(),null);
}
```

- what happens if there's a crash after the INSERT?
  - a loan exists (in the database) but the book is still shown as available so someone else could check it out

Example



- simultaneous checkouts can also result in problems

Transactions

Things can go wrong if tasks require multiple steps and those steps involve INSERT / DELETE / UPDATE.

- unfortunate interleaving of steps from simultaneous requests can lead to incorrect or inconsistent database state
- failure of a step (or a system crash) can lead to incorrect or inconsistent database state

A *transaction* is a group of operations treated as one logical unit of work.

- only two possible outcomes – *commit* (all changes kept) or *rollback* (none kept)
- guaranteed by the DBMS, not your code

Transactions in MySQL

- *autocommit* is on by default – every statement is its own transaction
- `START TRANSACTION` begins a multi-statement transaction
  - autocommit is suspended until the transaction ends, then the original setting is restored
- `COMMIT` ends the transaction, making the changes permanent
  - i.e. transaction has completed successfully
- `ROLLBACK` ends the transaction, discarding any changes
  - i.e. transaction failed in some way
  - ending the session without `COMMIT` is treated as a rollback

ACID

Every transaction gets four guarantees from the DBMS –

- atomicity – all or nothing
- consistency – valid state to valid state
- isolation – concurrent transactions don't interfere
- durability – committed means permanent

Each addresses a different failure point.

Atomicity

- *atomicity* guarantees all or nothing – either the whole transaction completed, or it is as if none of it did
- prevents partial updates resulting from a failure midway through
  - failure can be a crash, an error, or a deliberate `ROLLBACK` when a business rule fails

MySQL Notes

- a failed statement inside a transaction does *not* automatically roll back the transaction
  - only that statement is undone and the transaction stays open
  - application must call `ROLLBACK` explicitly on a failure

```
START TRANSACTION;
INSERT INTO LOANS (book_id, member_id, checked_out_on, due_on)
  VALUES (42, 999, CURDATE(), CURDATE() + INTERVAL 14 DAY);  -- ERROR 1452: no member 999
UPDATE BOOKS SET available = FALSE WHERE book_id = 42;       -- OK
COMMIT;
SELECT available FROM BOOKS WHERE book_id = 42;   -- FALSE
SELECT * FROM LOANS WHERE book_id = 42;           -- no loan
```

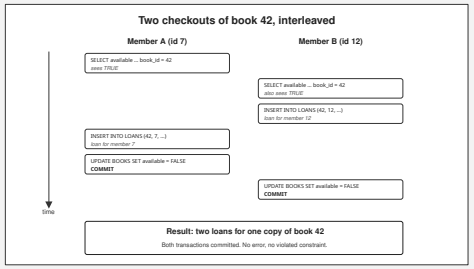
- DDL statements (`CREATE`, `ALTER`, `DROP`) commit implicitly
  - don't mix schema changes into other transactions

Consistency

- *consistency* means that a transaction takes the database from one valid state to another valid state
  - it is in a valid state before the transaction starts
  - it is in a valid state after the transaction ends
  - intermediate states within a transaction may not be valid
- *valid state* means that every rule holds
  - declared constraints (primary keys, foreign keys, `UNIQUE`, `NOT NULL`, `CHECK`, domain constraints, etc)
  - other rules enforced by the DBMS (e.g. triggers)
  - rules the DBMS doesn't know about (application-enforced rules)
- consistency is not 100% DBMS
  - the application must write transactions that when run alone on a valid database result in a valid database

Isolation

- *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
  - different *isolation levels* allow relaxation for greater efficiency with the cost of some anomalies

Durability

- *durability* survives a crash
  - once `COMMIT` returns, the change will still be there even after a restart
- prevents losing work already reported as done
- DBMS implements durability by writing each change to a log on disk before confirming the commit
  - the log enables recovery after a crash – restored committed changes and rolling back uncommitted transactions

ACID Recap

Which property?

- the power fails after the loan INSERT but before the BOOKS update – after restart, the loan exists and the book is available
- the API reports a successful checkout, then the server crashes – after restart, the loan is gone
- two librarians update the same book's replacement cost at once – one change silently disappears
- a loan is saved for book 999, which doesn't exist

Transactions in JDBC

```
try (Connection conn = Db.getConnection()) {
    conn.setAutoCommit(false);
    try {
        // several DAO calls, all using conn
        conn.commit();
    } catch (SQLException | RuntimeException e) {
        conn.rollback();
        throw e;
    } finally {
        conn.setAutoCommit(true);
    }
} catch (SQLException e) {
    throw SqlErrors.translate(e);
}
```

- `setAutoCommit(false)` is JDBC's version of `START TRANSACTION`
  - unlike `START TRANSACTION`, the previous autocommit setting is not restored after `COMMIT`/`ROLLBACK`
    - restore with `setAutoCommit(true)` in `finally`
- catch must cover `RuntimeExceptions` as well because our service exceptions are `RuntimeExceptions`

"All Using `conn`..."

- so far each DAO method opens its own connection, but a whole transaction must use the same connection
  - closing the connection ends the transaction
- new pattern
  - DAO methods take the connection as a parameter
  - the service opens the connection and owns the transaction
    - DAOs don't know about transactions – no commit or rollback
    - DAOs don't close connections they are passed
  - DAOs let errors propagate

```
public boolean setAvailable(Connection conn, int bookId, boolean available) {
    String sql = "UPDATE BOOKS SET available = ? WHERE book_id = ?";
    try (PreparedStatement ps = conn.prepareStatement(sql)) {
        ps.setBoolean(1, available);
        ps.setInt(2, bookId);
        return ps.executeUpdate() == 1;
    } catch (SQLException e) {
        throw SqlErrors.translate(e);
    }
}
```

Checkout With Transactions

```
public Loan checkout(int memberId, int bookId) {
    try (Connection conn = Db.getConnection()) {
        conn.setAutoCommit(false);
        try {
            Optional<Book> book = bookDao.findById(conn, bookId);
            if (book.isEmpty()) {
                throw new NotFoundException("No book with id " + bookId);
            }
            if (!book.get().available()) {
                throw new ConflictException("Book " + bookId + " is already checked out");
            }
            LocalDate today = LocalDate.now();
            Loan loan = new Loan(null, bookId, memberId, today, today.plusDays(14), null);
            int loanId = loanDao.insert(conn, loan);
            bookDao.setAvailable(conn, bookId, false);
            conn.commit();
            return new Loan(loanId,bookId,memberId,loan.checkedOutOn(),loan.dueOn(),null);
        } catch (SQLException | RuntimeException e) {
            conn.rollback();
            throw e;
        } finally {
            conn.setAutoCommit(true);
        }
    } catch (SQLException e) {
        throw SqlErrors.translate(e);
    }
}
```


Where to Use Transactions

- guideline – use a transaction for a multi-step operation if
  - there's more than one write
  - there's a read the decides a write