Domain Logic Patterns

09/149/2020

Overview

What Can Go Wrong?

- Correctness
  - Lost Updates - A overwrites B's changes because A read first and B wrote first
  - Inconsistent Read - 2 things that are correct, but not at the same time
- Liveness: How much concurrency can happen

Liveness sounds pretty vague

Google definitions of this to see if you can firm up this definition

Timeframe/Areas of Concern

- Request
  - a single call into the system
  - Largely the server's responsibility and usually can't be interrupted by user
- Session
  - long running interaction between client and a server
  - A logical sequence of requests
- Process
  - heavyweight execution process
  - Internal data is isolated
- Thread
  - lightweight execution process within a process
  - Shared memory -> concurrency problems
  - Isolated threads don't share memory

Make it real

Come up with an example of each of these

Can sessions be processes?

- Would like each session to be a process
  - Keep the data around for the whole session
  - Since internal data is isolated, no concurrency issues
- That isn't generally available
- Would take a lot of resources to create them and throw them away

Reducing Concurrency

Concurrency adds a LOT of complexity. How can we reduce it?

- Isolation - partition the data so only one "active agent" has access to it.
  - "Arrange things so that the program enters an isolation zone within which you don't have to worry about concurrency."
  - Good concurrency design = "Making such zones and ensuring that as much of the programming as possible is done in one of them"
  - What does that really mean?
- Immutability - data that can't change doesn't have concurrency issues

liveness?

How does increasing isolation affect liveness?

Immutability

How does all of this relate to Java streams and functional programming in general?

Transactions

- Bounded sequence of work
- Well-defined start and end points
- Resources are consistent at start and end
- Indivisible
- Can be at different levels of scope
  - System transaction
    - application to DB
    - This is the transaction you talked about in CSC371
  - Business transaction
    - User to application
    - One “operation” from a user’s perspective

ACID Characteristics

- Required for all transactions - ensures no concurrency problems
  - Atomicity - all or nothing
  - Consistency - at beginning and at end
  - Isolation - not visible to other resources until complete
  - Durability - committed effect must be permanent

Prove it

Create an example for each ACID characteristics to show how not having that characteristic can cause lost updates or inconsistent reads

Transaction Resources

- Length of a transaction is inversely proportional to the effect on throughput
- Want them to be short
- "Long transaction"
  - spans multiple requests
  - makes it hard to achieve ACID characteristics and maintain liveness
- "Late transaction"
  - reads are outside the transactions
  - Only updates are inside the transaction
  - Leaves the door open for inconsistent reads (really?)
  - Leaves the door open for lost updates (really?)

Prove it

Create an example for each of the risks of "late" transactions

Deadlocks

- Locks in a DB can cause deadlock just like locks in OS
- Can only deadlock if all of these:
  - Mutual exclusion
  - Hold and wait
  - No preemption
  - Circular wait

Dining Philosophers

Remind each other of the dining philosophers problem. What rule prevented deadlock?

Deadlock Recovery

- Victim
  - After detection
  - Pick one agent who has to throw everything away (rollback) and start over
- Lock Time Limit
  - If anyone holds a lock for too long, they have to throw everything away and start over
  - Similar to victim, but doesn't require detection

How far back?

If you victimize somebody, how far back do they have to go? How many locks do we take away from them? How would you restore their state to that point?

Deadlock Prevention

- Get all of the locks at the beginning of work
  - Does that really prevent anything?
  - Just makes it so you don't have to rollback when you get victimized
- Order the locks
  - Why does this work?
- Only one agent can hold locks at one time
  - How does that affect concurrency/liveness?

Lock Escalation

- If we lock too many rows in a table
- DB may force us to lock the table
- What would cause this?
- Imagine having a table for a Domain SuperType pattern . . .

Inheritance Patterns

Think about the patterns of access when you are persisting on object with each of the three inheritance mapping patterns. How many rows in how many tables? How many other kinds of objects could you be locking out?

Optimistic Locking

- Assume things will be ok
- Detect and handle collisions when they happen
- Good if collisions are rare and easy to detect
- Give multiple agents edit rights to the data
- First to finish can save without worry
- When rest save, have to make sure earlier writes aren't lost
- What if two wrote different values to the same thing? First wins!
- Locks only held on writes, but saves are more complex

Pessimistic Locking

- First to access locks the resource preventing access until finished
- Reduces concurrency

How do we choose

- If conflicts are rare and easy to detect -> optimistic
- If conflicts are common -> pessimistic

The price of optimism

The price of optimism is the need to resolve the conflict. What would resolution look like?

The price of pessimism

Since there are no conflicts, pessimism seems pretty easy to build. What's the price we are paying for that ease? (there is no free lunch . . .)

Preventing Inconsistent Reads

Only happens if we have to read multiple pieces in separate actions. How do we ensure they all are consistent?

- Pessimistic
  - Read locks - shared
  - Write locks
    - Only one and can't lock it if anyone has the read lock
    - Once it is locked, no one can get any locks
- Optimistic
  - Conflict detection depends on a version marker (timestamp or sequential counter)
  - Lost update prevention - version you are writing should match the existing and the gets updated on write
  - Inconsistent read - versions of each piece must match - Really???

Find the hole

Suppose you are using pessimistic locking. Find an example that shows why you need read locks

Temporal Reads

A strategy to eliminate need for read locks and/or provide versioning to prevent inconsistent reads

- Data gets timestamp or version number
- DB returns data as it was at that time
- DB is a full temporal history

Deadlock Prevention

Deadlock Prevention