Concurrency, Session State, Distribution

Large Scale
Software Engineering
Mike Helmick
University of Cincinnati
Spring 2014

Patterns of Enterprise Application Architecture

- Now, we shift back to software architecture and specific architecture patterns
- Starting with Chapters 5, 6, and 7 (this presentation)

Concurrency

Concurrency

- A very difficult subject
- Pervasive in enterprise applications
- Many users, many processes, many threads

Concurrency

Concurrency

- Something that is going to happen
- Something that is often ignored (wrongly)

Why Ignored?

Why Ignored?

- Because we have transaction managers
  - Pieces of software that can manage aspects of concurrency for us
  - i.e. Do everything in a managed transaction, and most things work out

But...

But...

- This doesn't always work well with items that span multiple database transactions
- This is called offline concurrency

In the server

In the server

- We also have concurrency in the application server
- Multiple threads of execution will be in the same code at the same time

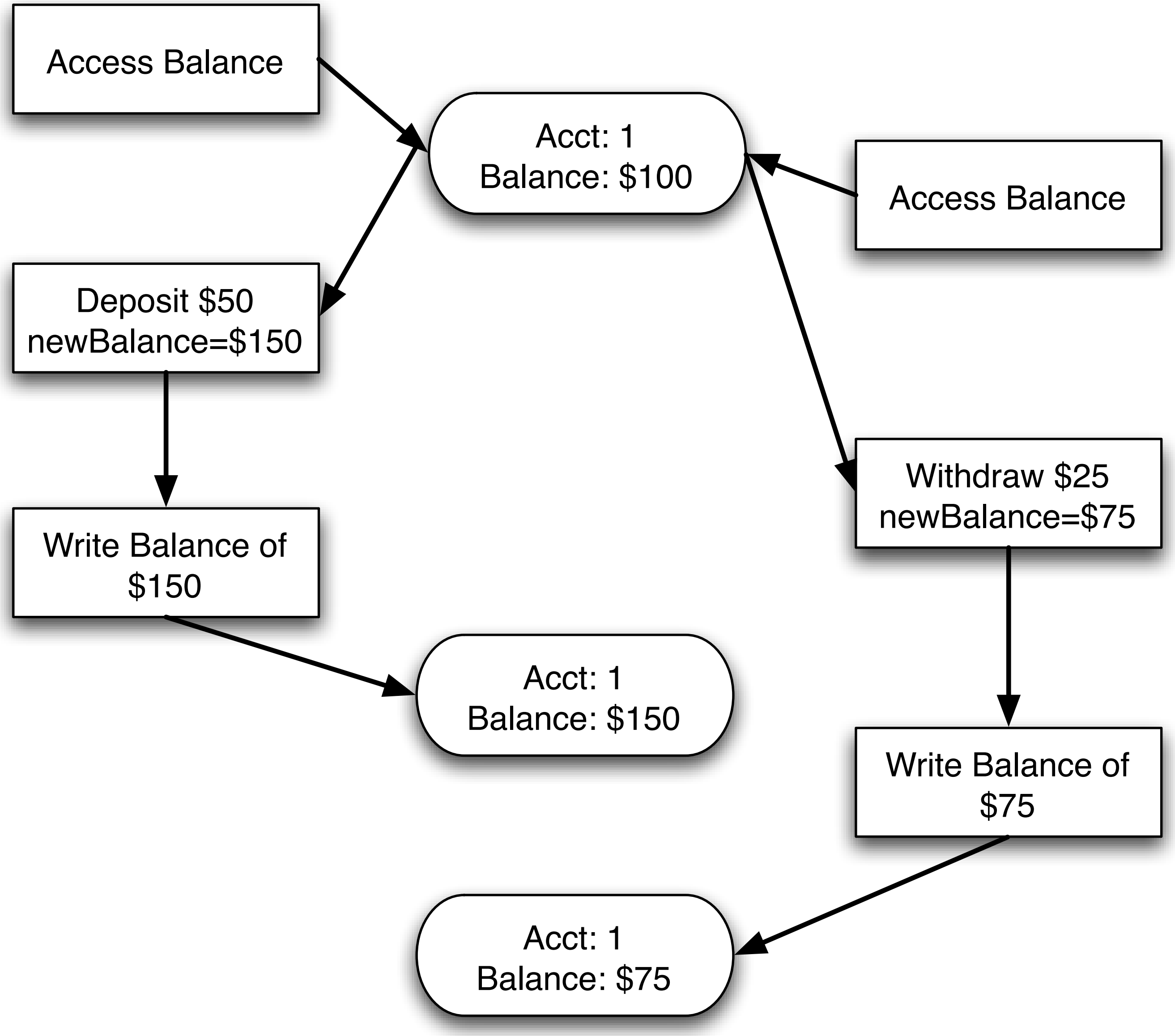
Concurrency Problems

Concurrency Problems

- Lost Updates

Concurrency Problems

- Lost Updates



Inconsistent Read

Inconsistent Read

- Occurs when you read two (or more) separate pieces of information
  - Read first
  - First changes
  - Read second
- If the two pieces of data go together - they are now inconsistent

Correctness

Correctness

- What we want is correctness
- Data that is always in a correct and consistent state

Liveness

Liveness

- How much can we do at the same time
- May have to sacrifice some correctness for liveness

Execution Contexts

Execution Contexts

- Generally we talk about where something executes in some context
  - request
  - session

Request

Request

- A single call into the system
- Work is done and a response is sent back
- Best if the client must wait for a response
  - not the case on the web
  - and the request appear linked to the user - but not to the server

Session

Session

- A series of requests over time between a client and a server
- Could be one or more requests
- Often login through logout

Session

Session

- With the web
- We use a session cookie to identify a user’s activities over time
- The Servlet API has methods for this

Process / Thread

Process / Thread

- process
  - heavyweight execution context
- thread
  - lightweight execution context

Requests

Requests

- Threads give us multiple requests in a single process
  - with a shared memory space
  - shared memory can be a concurrency issue

isolated threads

isolated threads

- Some environments allow you to specify memory on a per-thread basis
- In Java
  - Thread local storage

Contexts

Contexts

- We have a problem in that execution contexts don’t always line up correctly
- The same client might not talk to the same server for each of the requests in its session
- We can solve this in several ways

Isolation

Isolation

- A solution to multiple threads accessing the same data at the same time
- Partition data so that only one process can access a piece of data at a time
    - locks

Isolation

Isolation

- Reduces the change of data errors
- Create isolation zones within a program where operations are done safely

Immutable

Immutable

- Some data can be declared immutable
- This is a good idea if we're never going to change some reference data
    - A table containing dates

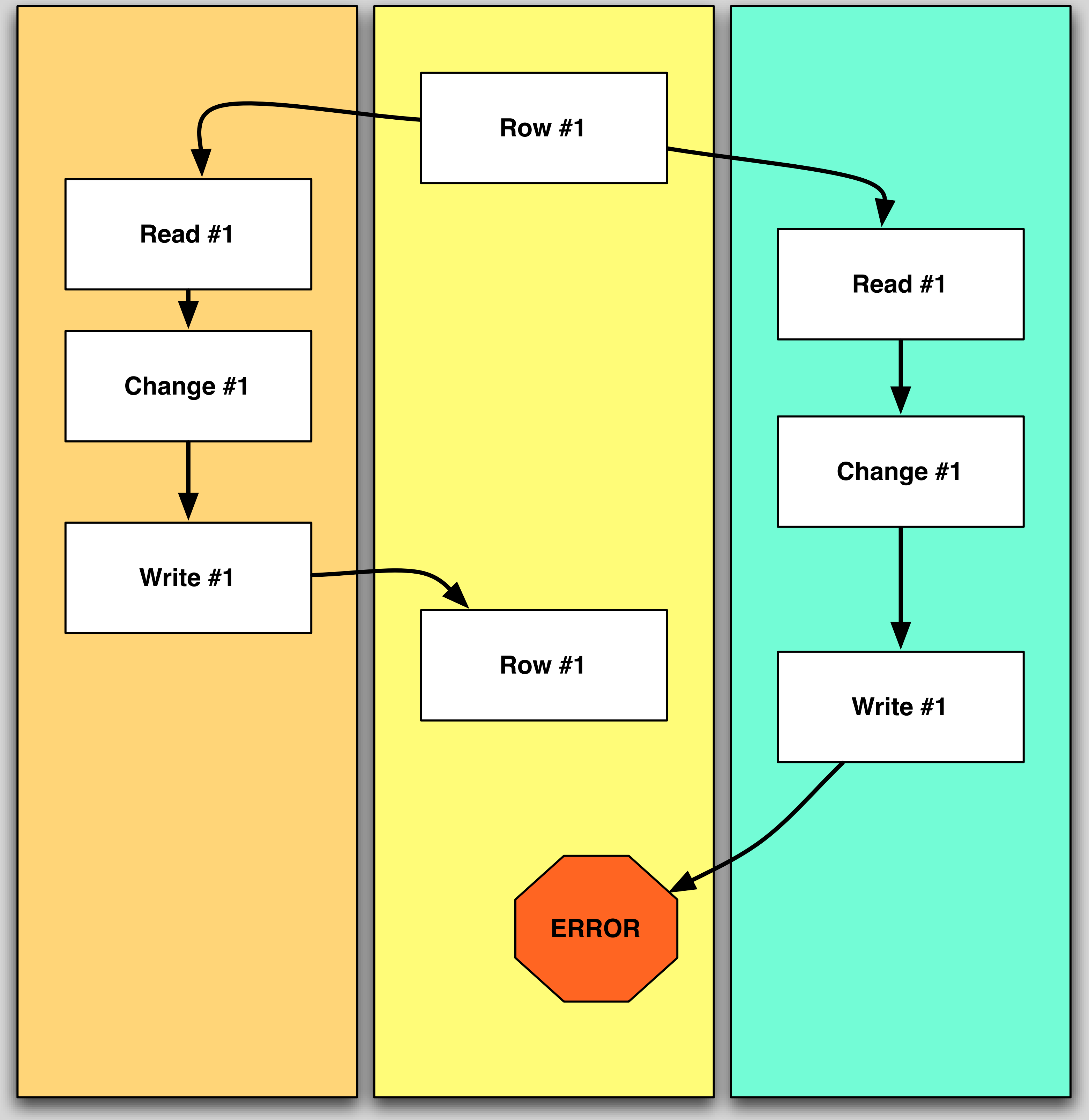
Control

Control

- In general - there are two types of concurrency control we use
  - optimistic locking
  - pessimistic locking

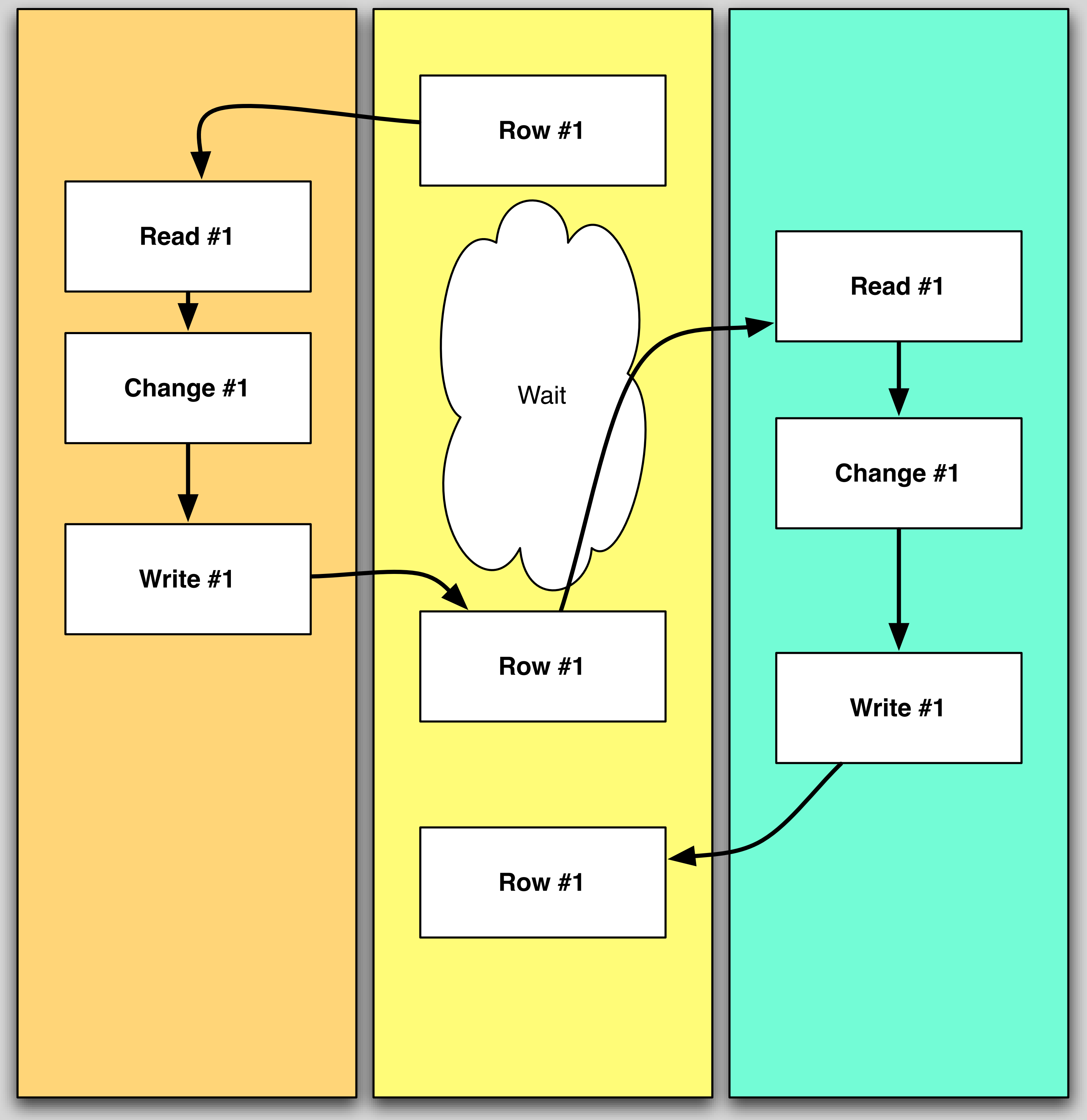
Optimistic

- Multiple copies of a record can be made
- but the first one to commit wins
    - Everyone else has to figure out what to do



Pessimistic locking

- Who ever asks for the data first gets it
- All other requests are queued
  - They get the data when
    - it is committed
    - the transaction is aborted



both

both

- Optimistic locking is used for conflict detection
- Pessimistic locking is about conflict prevention

either

either

- Real systems can use either one
- For source code control systems
  - we tend to prefer optimistic locking
  - i.e. software development would halt if only one person could edit a file at a time

choosing

choosing

- pessimistic lock reduces concurrency
    - prevents data from being read while it is being edited
    - supported natively by most databases

choosing

choosing

- optimistic lock
    - allows for greater concurrency
    - lock only happens during commit
    - causes merging problems - or transaction failures
    - not supported natively by most databases

choosing

choosing

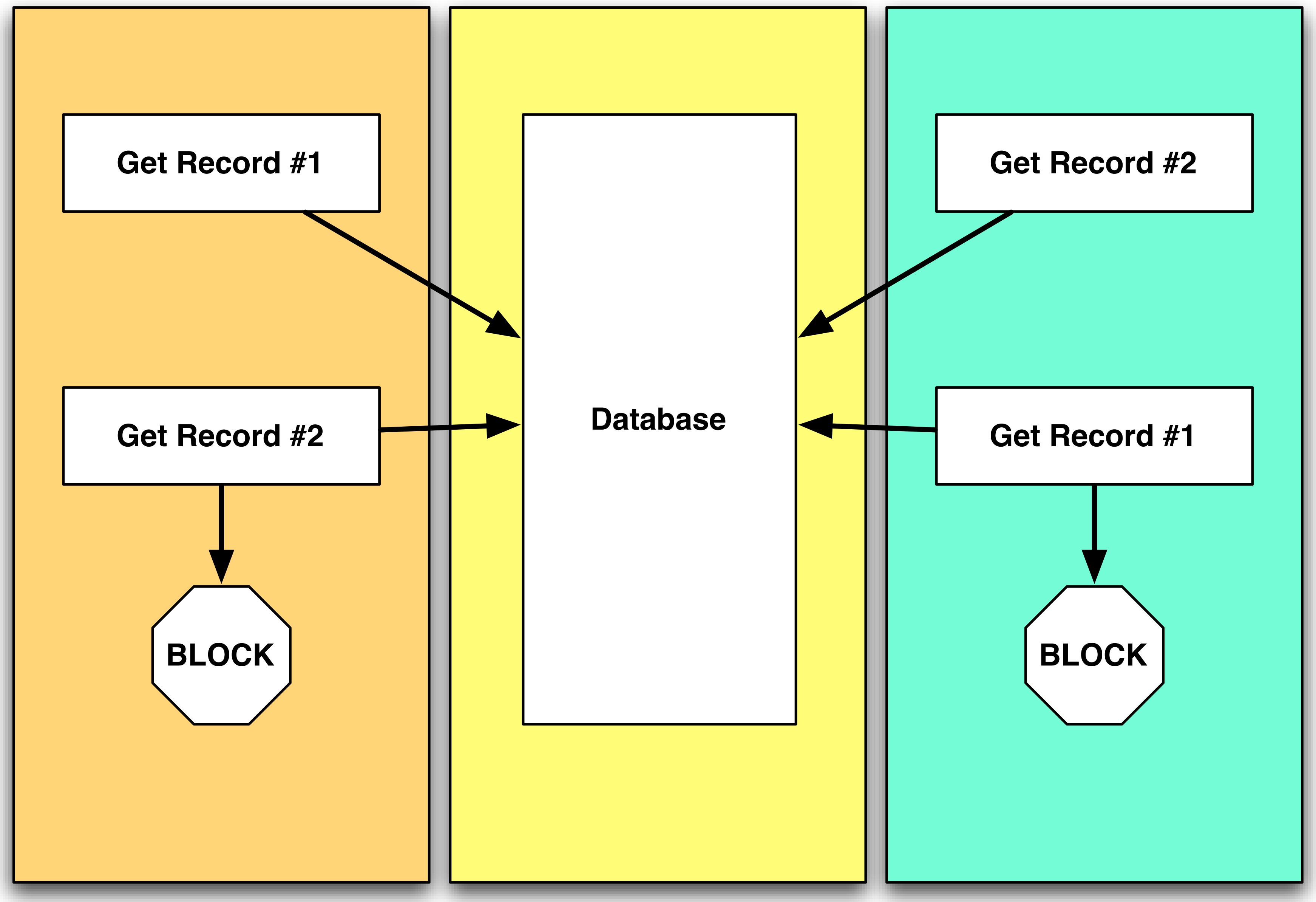
- optimistic locking is usually the best choice
- but if data is really sensitive or can not be easily edited by multiple people at the same time - then pessimistic locking

Deadlocks

Deadlocks

- A particular problem with pessimistic locking techniques
- Also called circular lock

Deadlock


Get Record #1
Get Record #2
BLOCK
Database
Get Record #2
Get Record #1
BLOCK

Deadlocks

Deadlocks

- Can be detected (see an Operating Systems book)
- We can attempt to prevent them

Deadlock

Deadlock

- occurs when
  - A user who already has locks - asks for more & won't release the ones they have

Preventing Deadlock

Preventing Deadlock

- We can force an order on how locks are requested

- This can avoid it

Transactions

Transactions

- A bounded sequence of work
  - defined starting point
  - defined ending point

Transactions

Transactions

- all participating resources are in a consistent state
  - when the transaction begins
  - when the transaction ends

Transactions

Transactions

- Are an “all-or-nothing”
- An ATM will not give you money without being certain that the balance of your account has been adjusted

ACID

ACID

- Atomicity
- Consistency
- Isolation
- Durability

Atomicity

Atomicity

- Each step in the transaction must complete successfully or the entire transaction will roll back

Consistency

Consistency

- “A system’s resources must be in a consistent, non-corrupt state at both the start and the completion of a transaction.”

Isolation

Isolation

- “The result of an individual transaction must not be visible to any other open transactions until that transaction commits successfully.”

Durability

Durability

- “Any result of a committed transaction must be made premanent. This translates to `must survive a crash of any sort.`”

Transactional Resources

Transactional Resources

- A transactional resource is anything that can use transactions to control concurrency

Duration

Duration

- We generally want transactions to be short
- If transactions span multiple requests, we call this a long transaction
- The common approach is to have one transaction for each request

Duration

Duration

- It is possible to read everything
- then open the transaction for updates
- this is probably the shortest possible transaction
    - but opens you up to inconsistent reads

App Server

App Server

- Process concurrency occurs in an application server
- Explicit concurrency control - threads, synchronization, etc...
  - We almost never have to deal with this

Solving

Solving

- We can launch a new process for each session
  - impractical
  - too slow

Process-per-request

Process-per-request

- Processes / threads are pooled
    - Each can handle multiple requests
    - but only one at a time
- Good concurrency control, good isolation

Thread-per-request

Thread-per-request

- Multiple threads in the same process
- Less resource intensive
  - more requests with the same hardware
- No isolation

Choosing

Choosing

- process-per-request is probably the easiest
  - good for a less experienced team
- I prefer thread-per-request
  - just requires a little more thought up front

Session State

Statelessness

Statelessness

- Statelessness doesn't refer to an object without state
  - What it means is
    - an object does not retain state between requests

Stateless

Stateless

- The same Java object can be reused over and over again
- The same service method can be used over and over again

Stateful

Stateful

- Objects that maintain their state between user requests
- Can be a drain on memory resources

but…

but...

- we can’t avoid state
- because client interactions are “inherently stateful”
    - A users shopping cart is state, and state that they certainly want you to remember

Session State

Session State

- Data that is relevant only in the context of a particular user's session is considered session state
  - Lives only for the duration of the users session (or slightly beyond)

Consistency

Consistency

- Data in the users session might not be legal
  - They have removed their zip code from their address, but we haven't persisted (or validated) yet
  - This is common in a users session

Storing

Storing

- There are many ways to store a users session
  - Client Session State
  - Server Session State
  - Database Session State

Client Session

Client Session

- Stores all session state with the client
  - URL encoded, cookies, serialized data
  - Just storing in memory (rich client)

Server Session

Server Session

- The server holds the data between requests
  - in memory
  - in a file
  - in a database table

Database Session

Database Session

- Slightly different from server session state stored in the database
- Instead of serializing the user's session you use (what I'm calling) shadow tables
    - These tables are like the master tables but only temporarily store data

Client Session

Client Session

- Totally impractical for the web when you have any significant amount of session information
  - you can make it work for a few fields
- This is because you have to transfer all of the information back and forth on every request
- Potential security problem - maybe you don't want to he client to see session data

Isolation

Isolation

- Maintaining isolation with database session state is difficult (according to our book)
- I think it is actually relatively easy do to - but with performance consequences

Server State

Server State

- So server state is probably our best option
- This is great if we have one application server
  - what about multiple servers?

Session Migration

Session Migration

- Allows a session to flow from one application server to another
  - works well when your session data is stored in the database
  - Can be done with in memory/file storage - but this is additional bandwidth between servers

Server Affinity

Server Affinity

- Force a user to use the same application server for their entire session
- This works but
  - bad for load balancing
  - horrible for redundancy (server crash and the users session is toast)

What I think

What I think

- User Server Session State with files (1 server) or database (1 or more servers) backing
  - Database here can be interpreted many ways:
    - Actual database
    - memcached
    - Not keeping session state, but re-reading current user state on each operation

Real world

Real world

- amazon.com uses some form of Database Session State

- Items are remembered between sessions

Timeout

Timeout

- What happens when the users closes their browser and doesn’t shop anymore
- This can lead to memory bloat if we just wait for them to come back
  - so, sessions typically expire after some period of inactivity

Client State

Client State

- In the web - this has a place as well
- We typically shuffle a token back and forth (128, 160, or more unique bits)
- This is refereed to as a session ID

Distribution Strategies

Distributed Objects

Distributed Objects

- In general
    - These are bad
    - Overused
    - Provide poor performance
    - Are not scalable architectures

Remote / Local

Remote / Local

- computers are the reason distributed objects don’t work so well
    - A method call within a process if very fast
    - A method call between processes is markedly slower
    - A method call between processes on different machines, even slower

and...

and...

- As a result how we deal with a remote object versus a local object is different
    - local: fine-grained interface. Individual getter methods for each field
    - remote: this model breaks down - I can't retrieve individual fields, but rather need to get as much as I can at once

SO…

SO…

- the programming model fundamentally changes

Tools

Tools

- The tools provide transparency so it is not known to the caller that an object is local or remote
- but...
  - the programming model should be different, so we have to know

Classes

Classes

- A distribution strategy based around classes (as we know them) doesn't work

First Law of Distributed Objects Design

First Law of Distributed Objects Design

- “First law of Distributed Object Design: Don’t distribute your objects”

Multiple CPUs

Multiple CPUs

- Of course we want to use multiple processes, across multiple machines in order to scale
- Clustering is usually the answer
  - Run the full application on many machines
  - Everyone makes local calls, goes faster
  - Scalability is still present

When you have to

When you have to

- There are several places where you must distribute your processes
- The goal is to then minimize the distribute boundaries

Client / Server

Client / Server

- In traditional client / sever programming there is a clear distribution requirement

Server / Database

Server / Database

- The server and the database typically run on different machines
- You can go to extremes of running the database in the same process as the application (I’ve tried this)
- but… database servers are designed to be remote from the application and are usually optimized to that effect

Web / Application

Web / Application

- Sometimes these two jobs are split
- Don't split them if you don't have to

3rd Party

3rd Party

- Some 3rd party code can be compiled into your process, some cannot
- Usually can’t do anything about this
  - hopefully they are coarse grained calls

Just have to

Just have to

- Maybe your higher ups have decided to force a distributed architecture
- Make your interfaces coarse grained

Distribution Boundary

Distribution Boundary

- We would still like to code with fine-grained objects
- The key is to use a Remote Facade to handle the coarse grained behavior
  - this is sort of a gateway to the remote process
  - I’ve found it useful even in local circumstances

Interfaces

Interfaces

- XML being sent over HTTP is the most common distribution technique at the moment
- Good interoperability

Summary

Summary

- We are going to build a framework and application side by side
- The best frameworks are extracted from real world applications