

# CS 188/219

## Scalable Internet Services

Andrew Mutz  
October 20, 2015



# Today's Agenda

Motivation

A Stable Data Layer

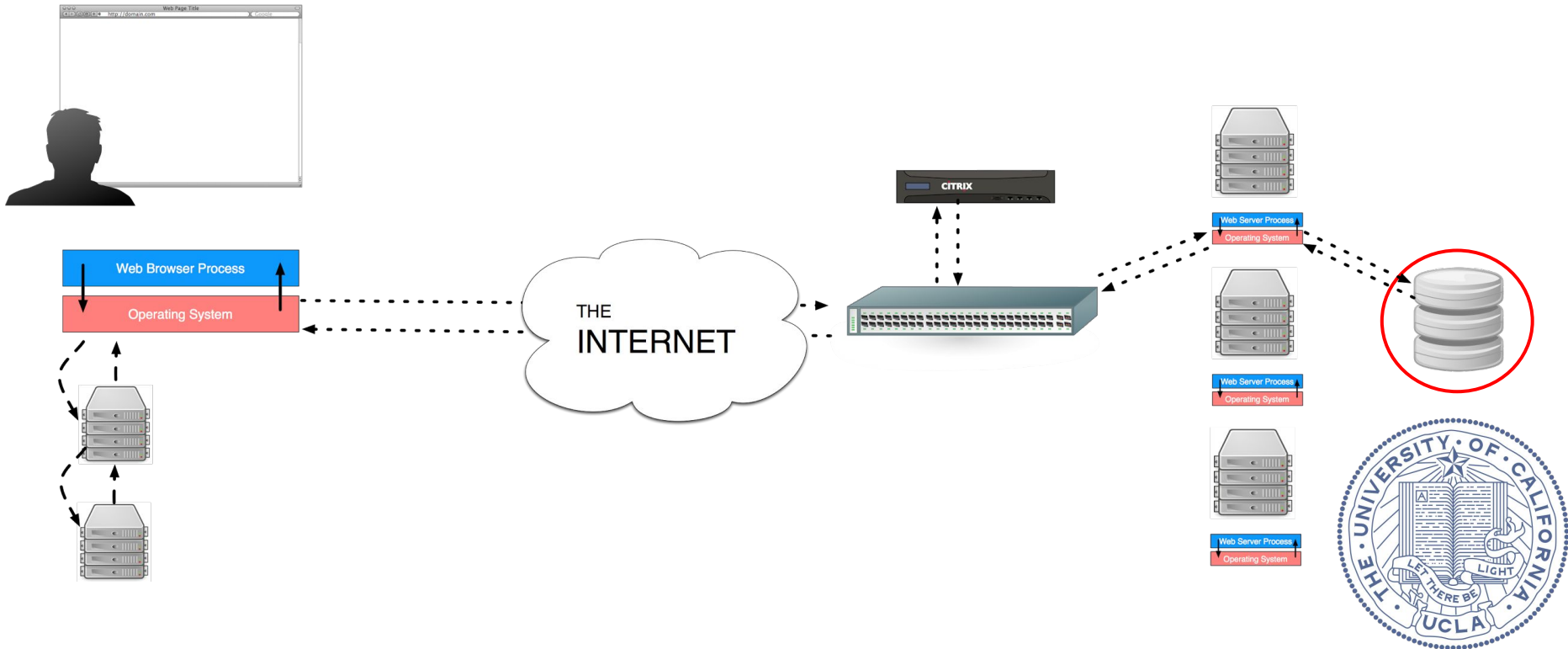
Database Concurrency Control

Query Analysis

For Next Time...



# Motivation



# Motivation

After today you should...

- Understand the importance of relational databases in architecting scalable internet systems.
- Understand how to design with concurrency in mind.
- Have a basic idea how to speed up your data layer
  - Where to start
  - Rails-specific techniques
  - Identify the source of slowness in a query

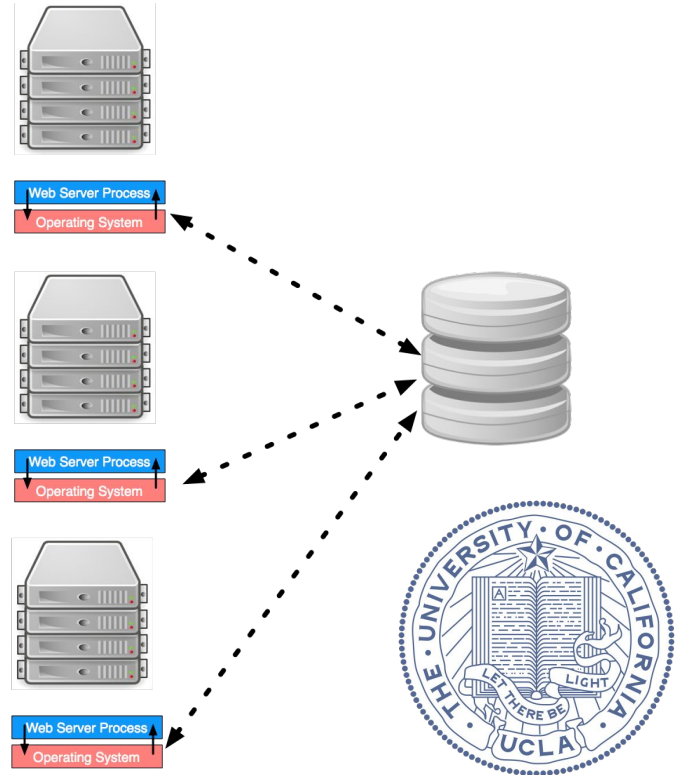


# Motivation

We have many stateless application servers responding to requests.

Clients can be served by any application server.

Client requests regularly include updates.

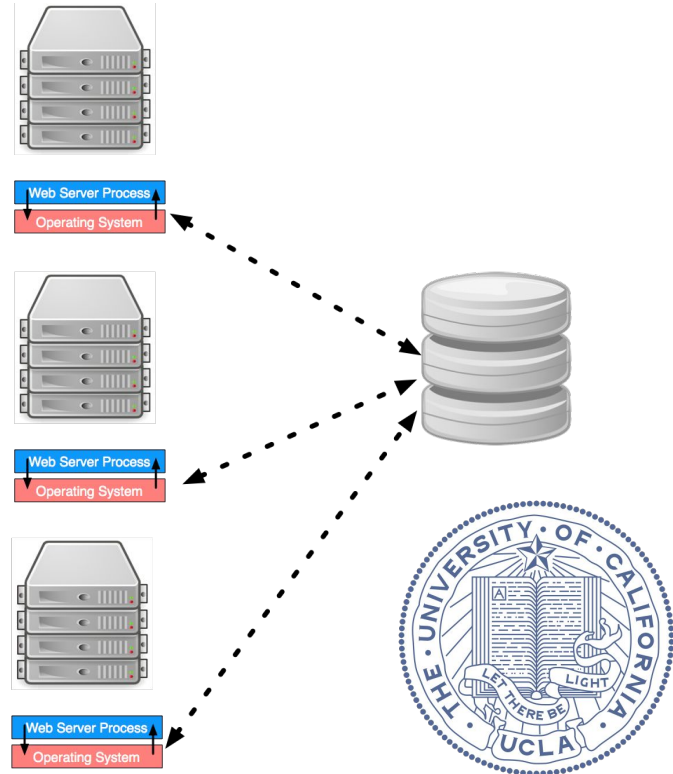


# A Stable Data Layer - Transactions

These application servers have needs

- Data needs to be seen by other requests/servers
- Access shouldn't be slow
- High Availability
- Data layer should make sense:

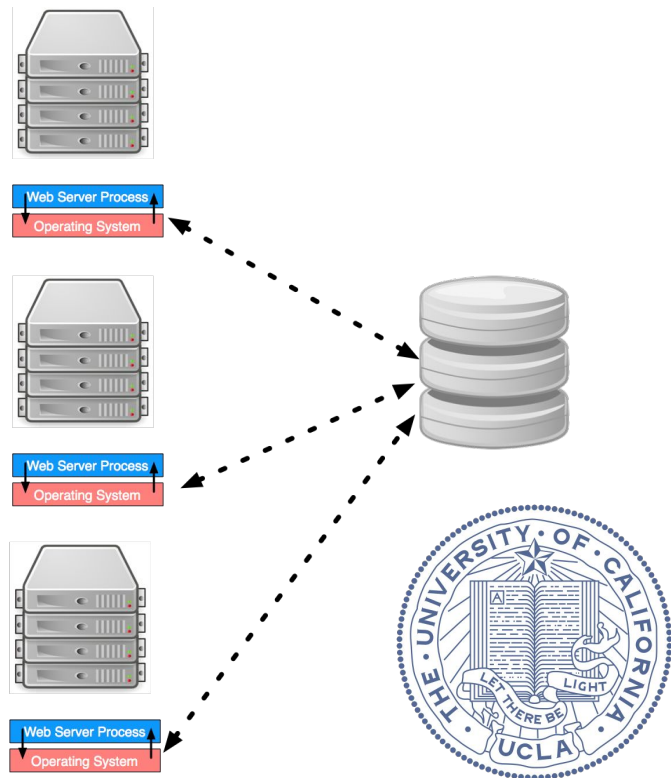
```
if david.balance > 100
  david.withdrawal(100)
  mary.deposit(100)
end
```



# Motivation

There are many options for a stable data layer.

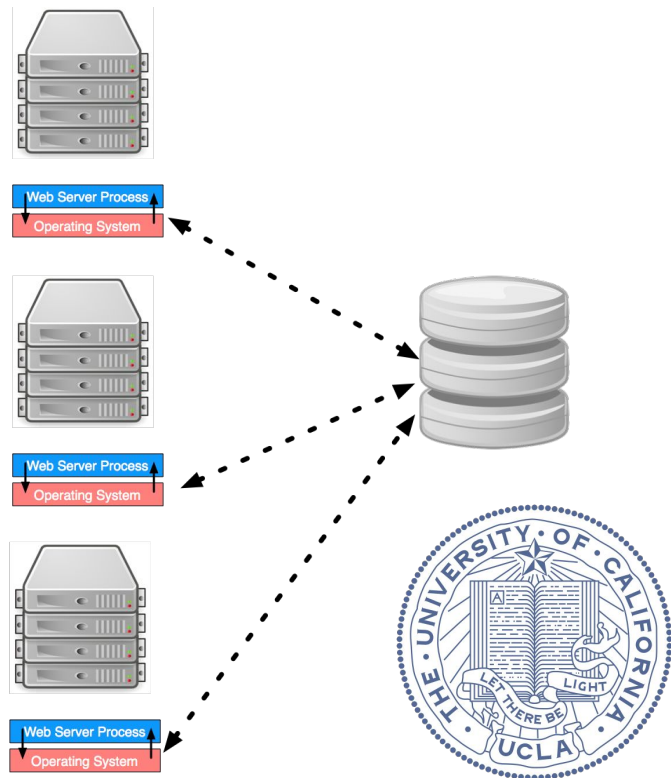
- Relational (SQL):
  - MySQL
  - Postgresql
  - Oracle
  - MS SQL
- Non-relational (NoSQL):
  - Cassandra
  - MongoDB
  - Redis



# Motivation

- Relational Databases...
  - Are a general-purpose persistence layer
  - Have more features
  - Limited ability to scale horizontally
- Non-relational Databases...
  - Tend to be more specialized
  - Require more from the application layer
  - Are better at scaling horizontally

If your needs fit within a RDBMS ability to scale, they tend to be best. If your scaling needs exceed RDBMS, go non-relational.

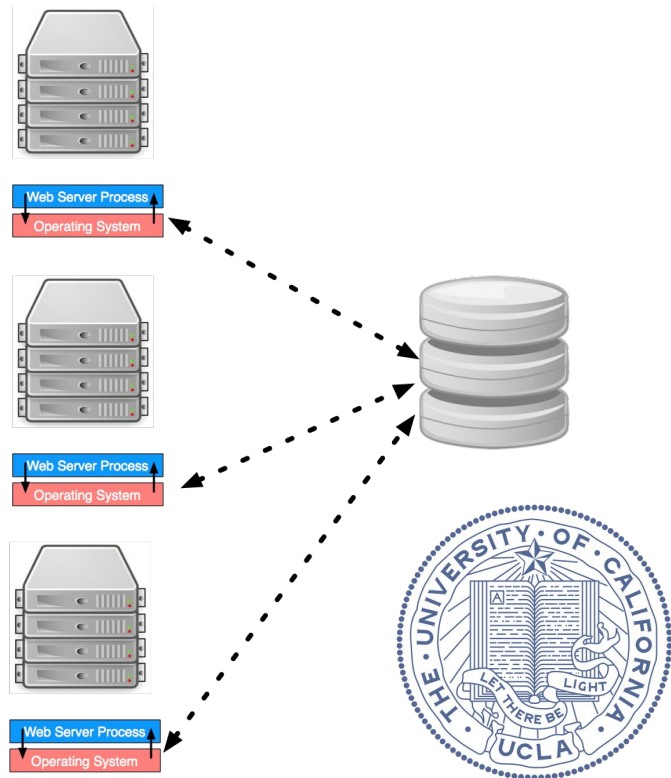




# Motivation

How we will approach these topics:

- All about relational databases
- Concurrency control
- SQL query analysis
- Scaling options for relational databases:
  - Sharding
  - Service Oriented Architectures
  - Distinguishing Reads from Writes
- A survey of NoSQL options



# A Stable Data Layer

## Database Transactions

- Background
  - Concept that allows a system to guarantee certain semantic properties. Gives control over concurrency.
  - Rigorously defined guarantees mean we can build correct systems on top of them.



# A Stable Data Layer - Transactions

## History of Database Transactions

- Mid 1970's, IBM System R's RSS
  - First system to implement SQL
  - Introduced formal notions of transactions and serializability
  - Led by Jim Gray
    - Result: 1998 Turing Award.



# A Stable Data Layer - Transactions

## ACID properties in a database:

- Atomicity
  - All or nothing.
  - No partial application of a transaction.
- Consistency
  - At the beginning and at the end of the transaction, the database should be consistent.
  - Consistency is defined by the integrity constraints
    - Foreign keys, NOT NULL, etc.



# A Stable Data Layer - Transactions

## ACID properties in a database:

- Isolation
  - A transaction should not see the effects of other uncommitted transactions.
- Durability
  - Once committed, the transaction's effects should not disappear. (being overwritten by later transactions is fine)



# A Stable Data Layer - Transactions

These have overlapping concerns

- Atomicity and Durability are related and are generally provided by journalling
- Consistency and Isolation are provided by concurrency control (usually implemented via locking)

No help with side-effects

- Actions that are visible outside of the system
- Transfer money, communicate with web service, etc.



# A Stable Data Layer - Transactions

Schedule (or “history”):

- Abstract model used to describe execution of transactions running in the system.

T1	R(X), W(X), Com.		
T2		R(Y), W(Y), Com.	
T3			R(Z), W(Z), Com.



# A Stable Data Layer - Transactions

## Conflicting Actions:

- Two actions are said to be in conflict if
  - The actions belong to different transactions
  - At least one of the actions is a write operation
  - The actions access the same object (read or write)
- Example of conflicting actions:
  - T1: R(X), T2: W(X), T3: W(X)
- And these are not conflicting:
  - T1: R(X), T2: R(X), T3: R(X)
  - T1: R(X), T2: W(Y), T3: R(X)

**Conflict** => we can't blindly execute them in parallel.





# A Stable Data Layer - Transactions

Why can't we blindly execute them in parallel? Example:

T1		R(X)	W(Y), Com.
T2	W(X)		Abort



# A Stable Data Layer - Transactions

Why can't we blindly execute them in parallel? Example:

## Dirty Read Problem

Transactions read a value written by a transaction that is later aborted and removed from the database. Reading transactions will have incorrect results.

T1		R(X)	W(Y), Com.
T2	W(X)		Abort



# A Stable Data Layer - Transactions

Why can't we blindly execute them in parallel? Example:

T1	R(All X), AVG	W(Y)	Com.
T2	W(Some X), Com.		



# A Stable Data Layer - Transactions

Why can't we blindly execute them in parallel? Example:

## Incorrect Summary Problem

1st transaction takes a summary over the values of all the instances of a repeated data item. While a 2nd transaction updates some instances of the data item. Resulting summary will not reflect a correct result for any deterministic order of the transactions. Result will be random depending on the timing of the updates.

T1	R(All X), AVG	W(Y)	Com.
T2	W(Some X), Com.		



# A Stable Data Layer - Transactions

A schedule is **serial** if

- The transactions are executed non-interleaved

Two schedules are **conflict equivalent** if

- They involve the same actions of the same transactions
- Every pair of conflicting actions is ordered in the same way

Schedule S is **conflict serializable** if

- S is **conflict equivalent** to some serial schedule

A schedule is **recoverable** if

- Transactions commit only after all transactions whose changes they read, commit.



# A Stable Data Layer - Transactions

Example of a schedule that is not **conflict serializable**:

T1	R(A), W(A)		R(B), W(B)
T2		R(A), W(A), R(B), W(B)	

Because it is not **conflict equivalent** to this:

T1	R(A), W(A), R(B), W(B)	
T2		R(A), W(A), R(B), W(B)

or this:

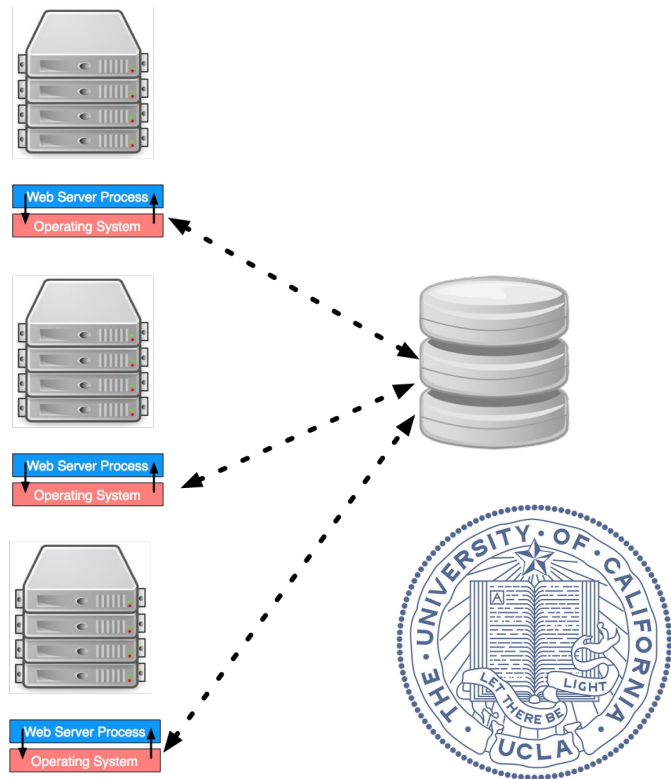
T1		R(A), W(A), R(B), W(B)
T2	R(A), W(A), R(B), W(B)	



# A Stable Data Layer - Transactions

Why is it important that we get a serializable schedule?

Why not just execute serially?



# A Stable Data Layer - Transactions

A serial schedule is important for consistent results - good when you are keeping track of your bank balance in the database

A serial execution of transactions is safe but slow

Most general purpose relational databases default to employing conflict-serializable and recoverable schedules

If you don't want to do a serial execution, what else can you do?





# A Stable Data Layer - Transactions

How do we implement a database that schedules that are conflict serializable and recoverable?

## Locks

- A lock is a system object associated with a shared resource such as a data item, a row, or a page in memory
- Prevent undesired, incorrect, or inconsistent operations on shared resources by concurrent transactions
- A database lock may need to be acquired by a transaction before accessing the object



# A Stable Data Layer - Transactions

Two types of database locks:

- Write-lock
  - Blocks writes and reads
  - Also called “exclusive lock”
- Read-lock
  - Blocks writes
  - Also called “shared lock”



# A Stable Data Layer - Transactions

## Two-Phase Locking

- 2PL is a concurrency control method that guarantees serializability
- Two-Phase Locking Protocol
  - Each Transaction must obtain a S (shared) lock on object before reading, and an X (exclusive) lock on object before writing
    - If a Transaction holds an X lock on an object, no other Transaction can get a lock (S or X) on that object
  - A transaction cannot request additional locks once it releases any locks
  - Two phases: acquire locks, release locks
- Issue: can result in “cascading aborts”

T1: R(A) W(A) unlock(A) ..... abort

T2: R(A) ..... abort



# A Stable Data Layer - Transactions

## Strong Strict Two-Phase Locking

- SS2PL allows only conflict serializable schedules
- SS2PL Protocol
  - Each Transaction must obtain a S (shared) lock on object before reading, and an X (exclusive) lock on object before writing
    - If a Transaction holds an X lock on an object, no other Transaction can get a lock (S or X) on that object
  - All locks held by a transaction are released when the transaction completes
- Avoids cascading aborts
- What's the downside?

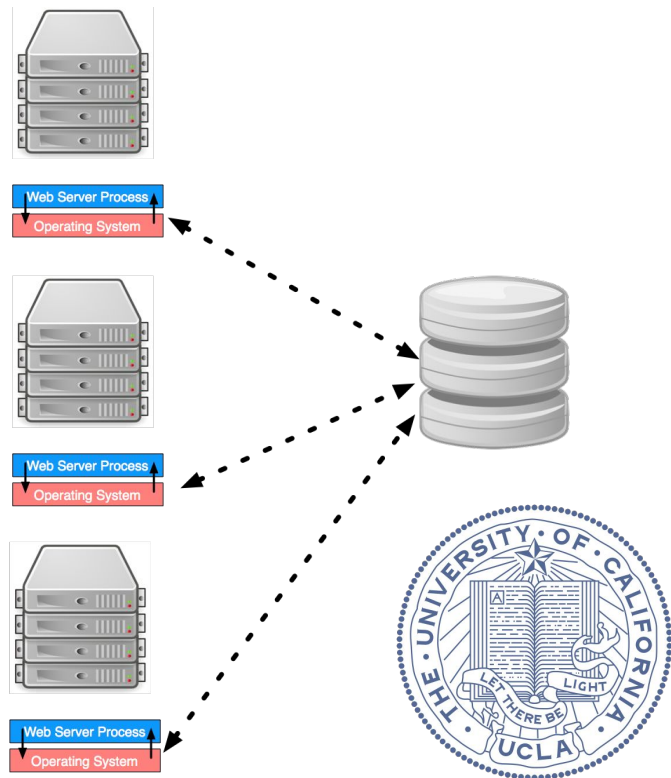


# Concurrency Control in Rails

We've got many application servers running our application.

We are using a relational database to ensure that each request sees a consistent view of the database.

What does this look like in practice?



# Concurrency Control in Rails

Rails uses two types of concurrency control:

- Optimistic
  - Let's not prevent concurrency problems, but instead detect them. And blow up when we detect them.
- Pessimistic
  - Let's prevent concurrency problems before they start.



# Concurrency Control in Rails

## Optimistic locking in Rails

- Easy to setup: just add an integer `lock_version` column to the table in question.
- Whenever an ActiveRecord object is read from the database, the `lock_version` is remembered.
- When the programmer tries to persist this object back to the database, it compares the `lock_version` it saw with the current lock version.
  - If they are different, it throws a `StaleObjectException`
  - If they are the same, it writes to the database and increments the `lock_version`
- This locking is an application level construct, the database knows nothing about it



# Concurrency Control in Rails

## Optimistic Locking Example:

```
p1 = Product.find(5)  
p1.name = "Daipers"
```

```
p2 = Product.find(5)  
p2.name = "Sheets"
```

```
p1.save!    # works fine  
p2.save!    # throws StaleObjectException
```





# Concurrency Control in Rails

## Optimistic locking

### Strengths:

- Predictable performance
- Lightweight

### Weaknesses:

- Sometimes your users will see errors
  - Or you will engineer re-tries



# Concurrency Control in Rails

## Pessimistic locking in Rails

- Easy to use: just add a `lock : true` option to ActiveRecord find.
- Whenever an ActiveRecord object is read from the database with that option, an exclusive lock is acquired.
- While this lock is held, others are prevented from acquiring the lock or reading/writing the value.
  - Others block until lock is released.
- This locking is database-level locking.
  - Implemented using `SELECT FOR UPDATE`



# Concurrency Control in Rails

## Pessimistic locking example:

```
transaction do
  p1 = Product.find(5).lock(true)
  p1.name = "Daipers"

  p1.save! # works fine
end
```

```
transaction do
  p2 = Product.find(5).lock(true)
  p2.name = "Sheets"

  p2.save! # works fine
end
```

This works great, yet it's not commonly used.

- **What could go wrong?**



# Concurrency Control in Rails

## What could possibly go wrong?

```
transaction do
```

```
  p1 = Product.find(5).lock(true)
```

```
  p1.name = "Daipers"
```

```
  ...
```

```
  my_long_procedure()
```

```
  ...
```

```
  p1.save!
```

```
end
```

```
transaction do
```

```
  p1 = Product.find(5).lock(true)
```

```
  p1.name = "Daipers"
```

```
  p1.save!
```

```
end
```



# Concurrency Control in Rails

## What could possibly go wrong?

```
transaction do
  o1 = Order.find(7).lock(true)
  ...
  p1 = Product.find(5).lock(true)
  p1.name = "Daipers"
  p1.save!
  ...
  o1.amount = 4
  o1.save!
end
```

```
transaction do
  p1 = Product.find(5).lock(true)
  p1.name = "Daipers"
  p1.save!
  ...
  o1 = Order.find(7).lock(true)
  ...
  o1.amount = 4
  o1.save!
end
```



# Concurrency Control in Rails

Pessimistic locking

Strengths:

- Failed transactions are more rare

Weaknesses:

- Need to deal with deadlocks
- Performance is less predictable



# Concurrency Control in Rails

**Which mode would you choose?**

```
transaction do
  determine_auction_winner()
  send_email_to_winner()
  save_auction_outcome() # db write
end
```



# Concurrency Control in Rails

**Which mode would you choose?**

```
transaction do
  record_facebook_like()
  update_global_counter_of_all_likes_ever() # db write
end
```





# Query Analysis

Ok, so you've hooked up MySQL to your Rails app and it's slower than you'd like.

You think it might be the database. How do we find out?



# Query Analysis

Let's use an example from the Demo app!

By changing the way you interact with the Rails ORM (ActiveRecord), you can significantly improve performance.

Example from the Demo app. You can see these in the “database\_optimizations” branch on github.

Demo App

## Submissions

Title	Url	Community
In quia ut esse quia.	<a href="http://reilly.biz/francisca_green">http://reilly.biz/francisca_green</a>	Ex magnam sequi neque cupiditate ipsam. <a href="#">20 comments</a>
Eveniet cumque quia quia sed dignissimos consectetur.	<a href="http://kleindayle.org/cloyd_howell">http://kleindayle.org/cloyd_howell</a>	Ex magnam sequi neque cupiditate ipsam. <a href="#">20 comments</a>
Molestiae velit eaque perferendis et sit qui expedita at.	<a href="http://gerhold.biz/luz">http://gerhold.biz/luz</a>	Ex magnam sequi neque cupiditate ipsam. <a href="#">20 comments</a>



# Query Analysis

I deployed this on a m3.medium instance and requested “/”.

There were:

- 20 Communities
- 400 Submissions
- 8000 Comments

Completed 200 OK

ActiveRecord: 220.6ms.

- Why so slow?

Demo App

## Submissions

Title	Url	Community
In quia ut esse quia.	<a href="http://reilly.biz/francisca_green">http://reilly.biz/francisca_green</a>	Ex magnam sequi neque cupiditate ipsam. <a href="#">20 comments</a>
Eveniet cumque quia quia sed dignissimos consectetur.	<a href="http://kleindayle.org/cloyd_howell">http://kleindayle.org/cloyd_howell</a>	Ex magnam sequi neque cupiditate ipsam. <a href="#">20 comments</a>
Molestiae velit eaque perferendis et sit qui expedita at.	<a href="http://gerhold.biz/luz">http://gerhold.biz/luz</a>	Ex magnam sequi neque cupiditate ipsam. <a href="#">20 comments</a>



# Query Analysis

First step: find out what Rails is doing.

In development mode, Rails will put the SQL it's generating in the log.

To (temporarily) enable this in production, change:

```
config.log_level = :debug
```

```
in config/environments/production.rb
```



# Query Analysis

```
class SubmissionsController < ApplicationController
  def index
    @submissions = Submission.all
  end
end
```

```
<h3>Submissions</h3>
<table class="table">
  <thead><tr><th>Title</th><th>Url</th><th>Community</th><th colspan="3"></th></tr></thead>
  <tbody>
    <% @submissions.each do |submission| %>
      <tr>
        <td><%= link_to(submission.title, submission.url) %></td>
        <td><%= submission.url %></td>
        <td><%= submission.community.name %></td>
        <td><%= link_to "#{submission.comments.size} comments", submission, class: 'btn btn-primary btn-xs'%></td>
      </tr>
    <% end %>
  </tbody>
</table>
```



# Query Analysis

Processing by SubmissionsController#index as HTML

Submission Load (0.5ms) `SELECT `submissions`.* FROM `submissions``

Community Load (0.3ms) `SELECT `communities`.* FROM `communities` WHERE `communities`.`id` = 1 LIMIT 1`

`SELECT COUNT(*) FROM `comments` WHERE `comments`.`submission_id` = 1`

`SELECT `communities`.* FROM `communities` WHERE `communities`.`id` = 1 LIMIT 1` `["id", 1]`

`SELECT COUNT(*) FROM `comments` WHERE `comments`.`submission_id` = 2`

`SELECT `communities`.* FROM `communities` WHERE `communities`.`id` = 1 LIMIT 1` `["id", 1]`

`SELECT COUNT(*) FROM `comments` WHERE `comments`.`submission_id` = 3`

`SELECT `communities`.* FROM `communities` WHERE `communities`.`id` = 1 LIMIT 1` `["id", 1]`

....

`SELECT `communities`.* FROM `communities` WHERE `communities`.`id` = 20 LIMIT 1` `["id", 20]`

`SELECT COUNT(*) FROM `comments` WHERE `comments`.`submission_id` = 400`



# Query Analysis

Because we are issuing so many queries, the overhead associated with each SQL statement is expensive.

**How should we fix this?**



# Query Analysis

Because we are issuing so many queries, the overhead associated with each SQL statement is expensive.

**How should we fix this?**

**Issue fewer queries.**

- Don't ask for the community each time (esp. repeats)
- Don't ask for the number of comments each time.





# Query Analysis

Change from this:

```
class SubmissionsController < ApplicationController
  def index
    @submissions = Submission.all
  end
end
```

To this:

```
class SubmissionsController < ApplicationController
  def index
    @submissions = Submission.includes(:comments).includes(:community).all
  end
end
```



# Query Analysis

**Result: ActiveRecord 39.6ms**

Submission Load (0.9ms) `SELECT `submissions`.* FROM `submissions``

Comment Load (38.3ms) `SELECT `comments`.* FROM `comments` WHERE `comments`.`submission_id` IN (1, 2,... 399, 400)`

Community Load (0.4ms) `SELECT `communities`.* FROM `communities` WHERE `communities`.`id` IN (1, 2, ...19, 20)`



# Query Analysis

With MySQL you can use EXPLAIN to analyze queries

- Won't actually execute the query.
- Helps us understand how and when MySQL will use indices.
- Returns a table of data from which you identify potential improvements



# Query Analysis

```
mysql> EXPLAIN SELECT COUNT(DISTINCT `submissions`.`id`) FROM
`submissions` JOIN `comments` WHERE `comments`.`submission_id` =
`submissions`.`id` AND `comments`.`message` = 'This is not a
test!'\G
```

```
***** 1. row *****
```

```
id: 1
select_type: SIMPLE
table: comments
type: ALL
possible_keys: NULL
key: NULL
key_len: NULL
ref: NULL
rows: 19188
Extra: Using where
```

```
***** 2. row *****
```

```
id: 1
select_type: SIMPLE
table: submissions
type: eq_ref
possible_keys: PRIMARY
key: PRIMARY
key_len: 4
ref: default_db_name.comments.submission_id
rows: 1
Extra: Using index
2 rows in set (0.00 sec)
```



# Query Analysis

How to read EXPLAIN output:

These results are the most important for performance analysis

select_type	The SELECT type
type	The join type
possible_keys	The indices available to be chosen
key	The index actually chosen
rows	Estimate of rows to be examined



# Query Analysis

`select_type`

The type of select statement being performed.

- Most are fine, but two indicate potential performance problems
  - **Dependent Subquery:** reevaluated for every different value of the outer query
  - **Uncacheable Subquery:** reevaluated for every value of the outer query



# Query Analysis

type

The type of JOIN being used. From best to worst:

- system - The table only has one row
- const - From uniqueness, we know only one row can match
- eq\_ref, ref - Only one row at most can match from the previous table
- fulltext - mysql fulltext index
- ref\_or\_null - like ref, but also null values
- index\_merge
- unique\_subquery
- index\_subquery
- range - Only rows in a given range are retrieved, but can use index
- index - Full table scan, but can scan index instead of actual table
- ALL - Full table scan

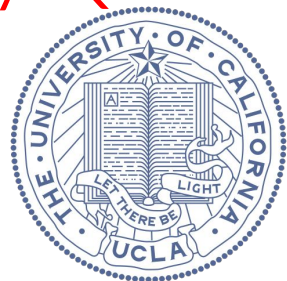


# Query Analysis

`possible_keys` & `key`

`Possible_keys` lists the indices that could possibly be used. `Key` indicates which was actually chosen.

- If you don't like the index that MySQL is using, you can tell it to ignore indices using the `IGNORE INDEX`
- If `possible_keys` is null, you have no indices that MySQL can use and should consider adding some.





# Query Analysis

rows

MySQL's estimate of how many rows need to be read. If this number is really big, that can indicate a problem.



# Query Analysis

Optimizations take three main forms:

- Add or modify indices
- Query optimizations
- Modify table structure
  - Denormalization, for example



# Query Analysis - Indices

## What is an index?

- Fast, compact structure for identifying row locations
- Chop down your result set as quickly as possible
- MySQL will only use one index per table per query
  - It cannot combine two separate indexes to make one more useful index.



# Query Analysis - Indices

Adding indices in Rails:

```
class AddNameIndexProducts < ActiveRecord::Migration
  def change
    add_index :products, :name
  end
end
```



# Query Analysis - Indices

## Adding foreign keys in Rails:

- The “Rails way” is to enforce these things at the application layer
- You may disagree, in which case you can use the “Foreigner” gem like so:

```
class AddForeignKeyToOrders < ActiveRecord::Migration
  def change
    add_index :orders, :products
  end
end
```



# Query Analysis - Indices

Indices work best when they can be kept in memory. Some ways to trim the fat:

- Can I reduce the characters in that VARCHAR index?
- Can I use a TINYINT instead of a BIGINT?
- Can I use an integer to describe a status instead of a text-based value?



# Query Analysis - Query Optimization

Modify your query.

Example at the SQL level:

```
mysql> explain select count(*) from txns where parent_id - 1600 = 16340
select_type: SIMPLE
table: txns
type: index
key: index_txns_on_reverse_txn_id
rows: 439186
Extra: Using where; Using index
```



# Query Analysis - Query Optimization

Modify your query.

Example at the SQL level:

```
mysql> explain select count(*) from txns where parent_id = 16340 + 1600
  select_type: SIMPLE
    table: txns
  type: const
  key: index_txns_on_reverse_txn_id
  rows: 1
Extra: Using index
```





# Motivation

After today you should...

- Understand the importance of relational databases in architecting scalable internet systems.
- Understand how to design with concurrency in mind.
- Have a basic idea how to speed up your data layer
  - Where to start
  - Rails-specific techniques
  - Identify the source of slowness in a query



# For Next Time...

Keep pushing forward on your project

- This week's demo should be on AWS
- The sooner you get application features up and deployed, the sooner we can scale your application!

