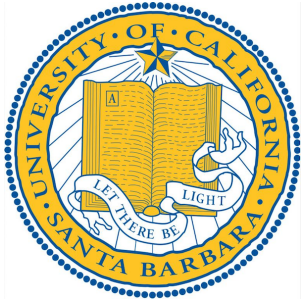
CS 290B

Scalable Internet Services

Andrew Mutz

October 28, 2014



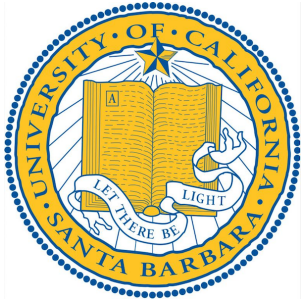
Motivation

Motivation

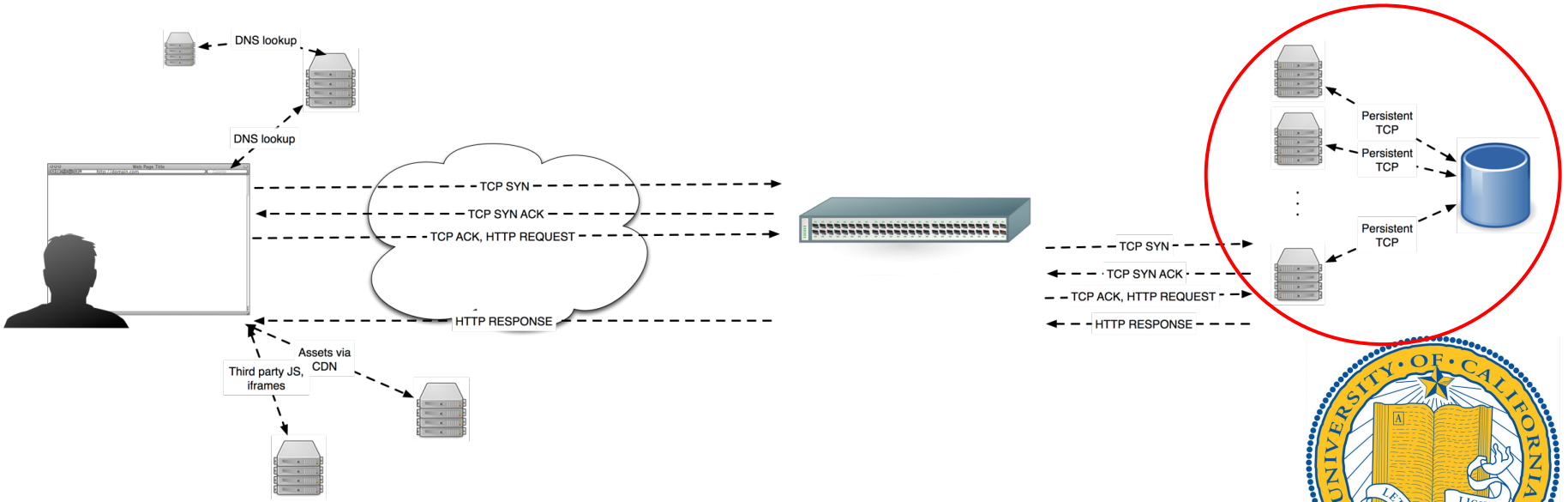
Concurrency Control in Rails

Query Analysis

For Next Time...



Title

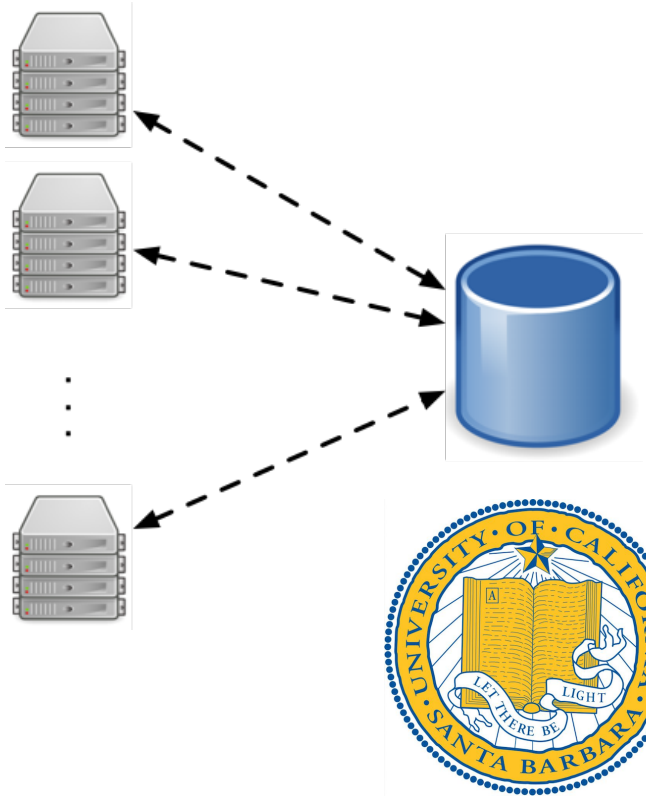

DNS lookup
DNS lookup
TCP SYN
TCP SYN ACK
TCP ACK, HTTP REQUEST
HTTP RESPONSE
Assets via CDN
Third party JS, iframes
TCP SYN
TCP SYN ACK
TCP ACK, HTTP REQUEST
HTTP RESPONSE
Persistent TCP
Persistent TCP
Persistent TCP

Motivation

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.

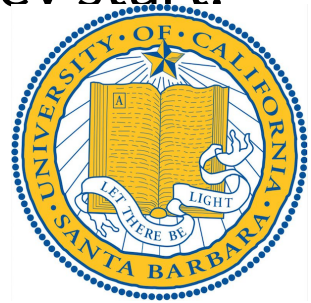
What does this look like in practice?



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

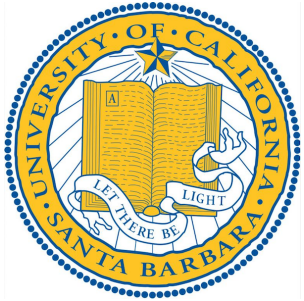
Optimistic locking in Rails

- 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

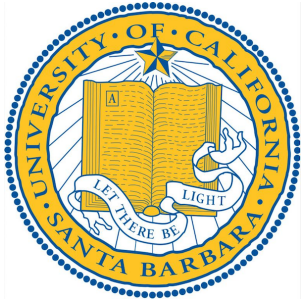
Optimistic locking

Strengths:

- Predictable performance
- Lightweight

Weaknesses:

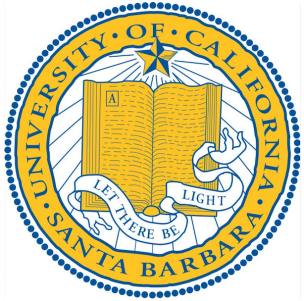
- Sometimes your users will see errors
  - Or you will engineer re-tries



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.



Concurrency Control in Rails

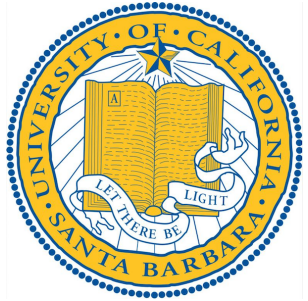
Pessimistic locking example:

```
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 its not commonly used.

- **What could possibly go wrong?**

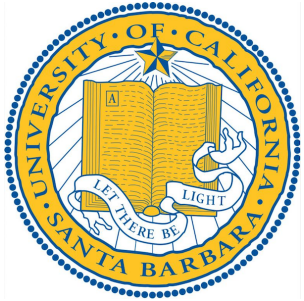


Concurrency Control in Rails

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

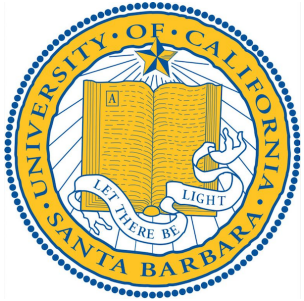
Pessimistic locking

Strengths:

- Failed transactions are more rare

Weaknesses:

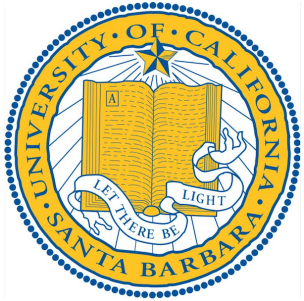
- Need to deal with deadlocks
- Performance is less predictable



Concurrency Control in Rails

Which mode would you choose?

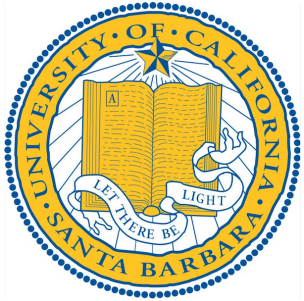
```
transaction do
  determine_auction_winner()
  send_email_to_winner()
  save_auction_outcome()
end
```



Concurrency Control in Rails

Which mode would you choose?

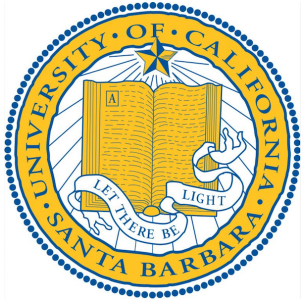
```
transaction do
  record_facebook_like()
  update_global_counter_of_all_likes_ever()
end
```



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

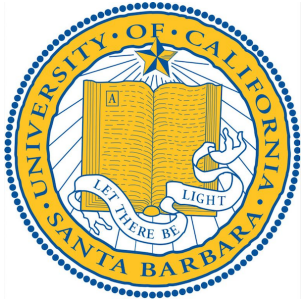
First step: find out what Rails is doing.

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`



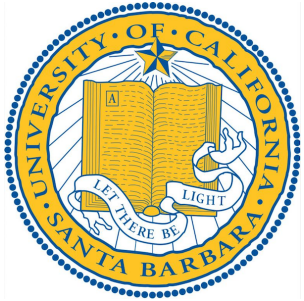
Query Analysis

```
Community Load (1.9ms)  SELECT  `communities`.* FROM `communities`  WHERE `communities`.`id` = 4 LIMIT 1
```

```
Submission Load (12.7ms)  SELECT `submissions`.* FROM `submissions` WHERE `submissions`.`community_id` IN (4)
```

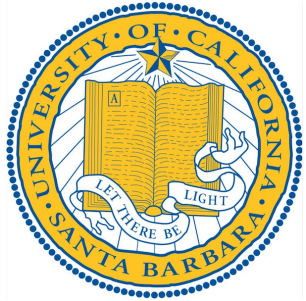
```
Comment Load (39.4ms)  SELECT `comments`.* FROM `comments`  WHERE `comments`.`submission_id` IN (4, … 10104)
```

```
User Load (2.1ms)  SELECT `users`.* FROM `users`  WHERE `users`.`id` IN (1)
```



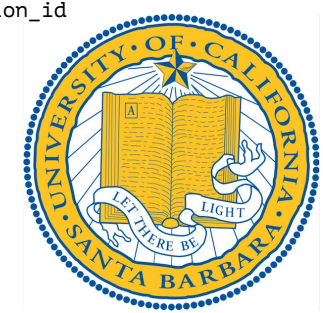
Query Analysis

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


+-------------------------------------+
| COUNT(DISTINCT `submissions`.`id`) |
+-------------------------------------+
|                               10200 |
+-------------------------------------+
1 row in set (0.11 sec)
```



Query Analysis

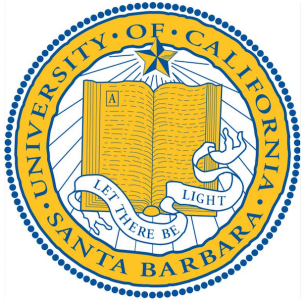
```
mysql> EXPLAIN SELECT COUNT(DISTINCT `submissions`.`id`) FROM
`submissions`  JOIN `comments` WHERE `comments`.`submission_id` =
`submissions`.`id` AND `comments`.`message` = 'This is not a
test!'\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        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

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

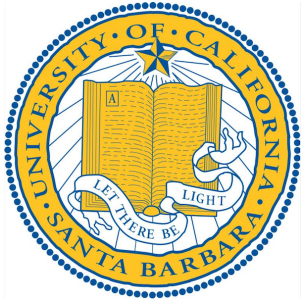


Query Analysis

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

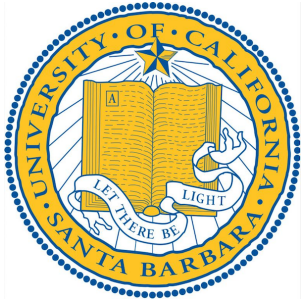


Query Analysis

`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

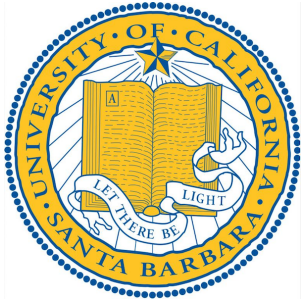


Query Analysis

`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

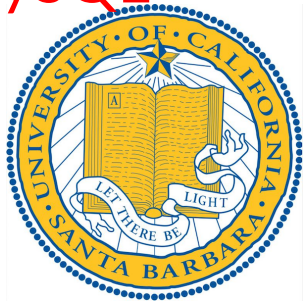


Query Analysis

`possible_keys & key`

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

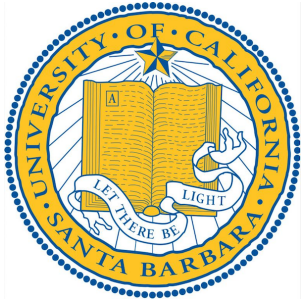
- If you don't like the index that MySQL is using, you can tell it to ignore indices using the IGNORE INDEX
- If possible_keys is null, you have no indices that MySQL can use and should consider adding some.



Query Analysis

`rows`

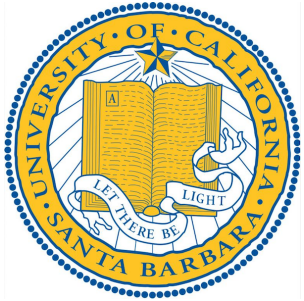
MySQL's estimate of how many rows need to be read. If this number is really big, that can indicate a problem.



Query Analysis

Optimizations take three main forms:

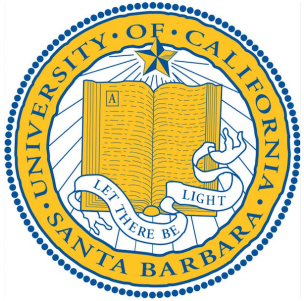
- Add or modify indices
- Query optimizations
- Modify table structure
  - Denormalization, for example



Query Analysis - Indices

What is an index?

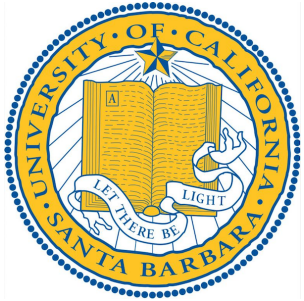
- Fast, compact structure for identifying row locations
- Chop down your result set as quickly as possible
- MySQL will only use one index per table per query
    - It cannot combine two separate indexes to make one more useful index.



Query Analysis - Indices

Adding indices in Rails:

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

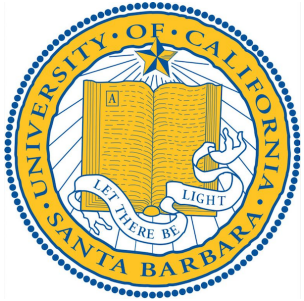


Query Analysis - Indices

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:

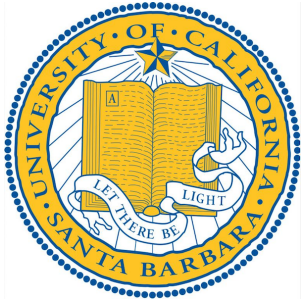
```
class AddForeignKeyToOrders < ActiveRecord::Migration
  def change
    add_index :orders, :products
  end
end
```



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?

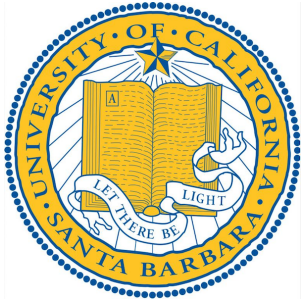


Query Analysis - Query Optimization

Another way to improve performance is to modify your query.

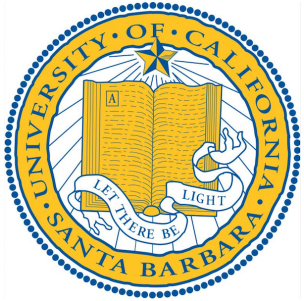
Example at the Rails level:

```
c = Community.find(4)
c.submissions.each do |s|
  puts "Submission is #{s.title}"
  puts "Number of comments is #{s.comments.size}"
  puts "First commenter is #{s.comments.first.user.email}"
end
# 2963ms
```



Query Analysis - Query Optimization

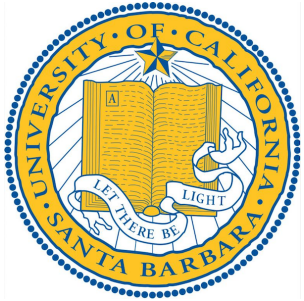
```
# SELECT  communities.* FROM communities  WHERE communities.id = 4 LIMIT 1
c = Community.find(4)

# SELECT submissions.* FROM submissions  WHERE submissions.community_id = 4
c.submissions.each do |s|
  puts "Submission is #{s.title}"
  # SELECT COUNT(*) FROM `comments`  WHERE `comments`.`submission_id` = X
  puts "Number of comments is #{s.comments.size}"
  # SELECT comments.* FROM comments  WHERE comments.submission_id = X
  # SELECT users.* FROM users WHERE users.id = 1
  puts "First commenter is #{s.comments.first.user.email}"
end
# 2963ms
```



Query Analysis - Query Optimization

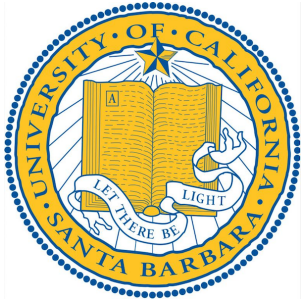
Why is this faster?

```
c = Community.includes({:submissions => {:comments => :user}}).find(4)
c.submissions.each do |s|
  puts "Submission is #{s.title}"
  puts "Number of comments is #{s.comments.size}"
  puts "First commenter is #{s.comments.first.user.email}"
end
# 519ms
```



Query Analysis - Query Optimization

```
c = Community.includes({:submissions => {:comments => :user}}).find(4)
# SELECT  `communities`.* FROM `communities`  WHERE `communities`.`id` = 4
# SELECT `submissions`.* FROM `submissions`  WHERE `submissions`.`community_id` IN (4)
# SELECT `comments`.* FROM `comments`  WHERE `comments`.`submission_id` IN (4... 10104)
# SELECT `users`.* FROM `users`  WHERE `users`.`id` IN (1...)
c.submissions.each do |s|
  puts "Submission is #{s.title}"
  puts "Number of comments is #{s.comments.size}"
  puts "First commenter is #{s.comments.first.user.email}"
end

# 519ms
```

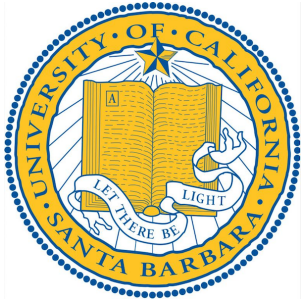


Query Analysis - Query Optimization

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
```

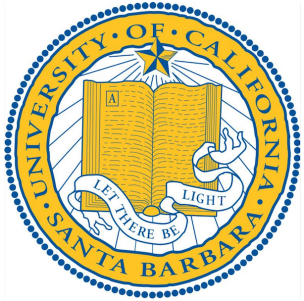


Query Analysis - Query Optimization

Modify your query.

Example at the SQL level:

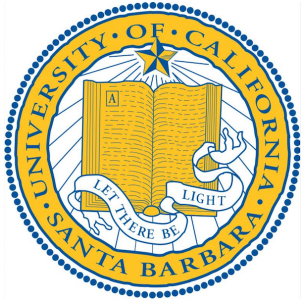
```
mysql> explain select count(*) from txns where parent_id = 16340 + 1600
  select_type: SIMPLE
  table: txns
  type: const
  key: index_txns_on_reverse_txn_id
  rows: 1
  Extra: Using index
```



Query Analysis

Intuition is often wrong…

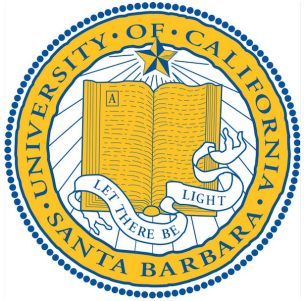
- A local company complained about a db performance problem
- They brought in a local database consultant to help them
- Looking at the problem every query was taking from 100ms to 1 second
- What was the problem?



Query Analysis

What was the problem?

- To establish a connection to the database can take 100ms+
- Both client and server need to authenticate and reserve resources such as threads and memory/cache
- If you don't reuse these connections this can be the bottleneck
- Connection pooling solved the problem
- Connection pooling is web app 101



For Next Time...

Be prepared to demo your first sprint's worth of work tomorrow at lab!