What to measure?

- *Stress testing* or *load testing*: how far can I push my system...
  - ...before performance becomes unacceptable?
  - ...before it gasps and dies?
- Usually, one component will be *bottleneck*
  - a particular view, action, query, …
- Load testers can be simple or sophisticated
  - bang on a single URI over and over
  - do a fixed sequence of URI's over and over
  - play back a log file

Longevity Bugs

- Resource leak (RAM, file buffers, sessions table) is classic example
- Some infrastructure software such as Apache already does *rejuvenation*
  - aka “rolling reboot”
- Related: running out of sessions
- Solution: store whole session[] in cookie (Rails 3 does this by default)

Which is probably *not* a metric of high interest to you, the app operator?

- Slowest queries
- Maximum CPU utilization
- 99%ile response time
- Rendering time of 3 slowest views

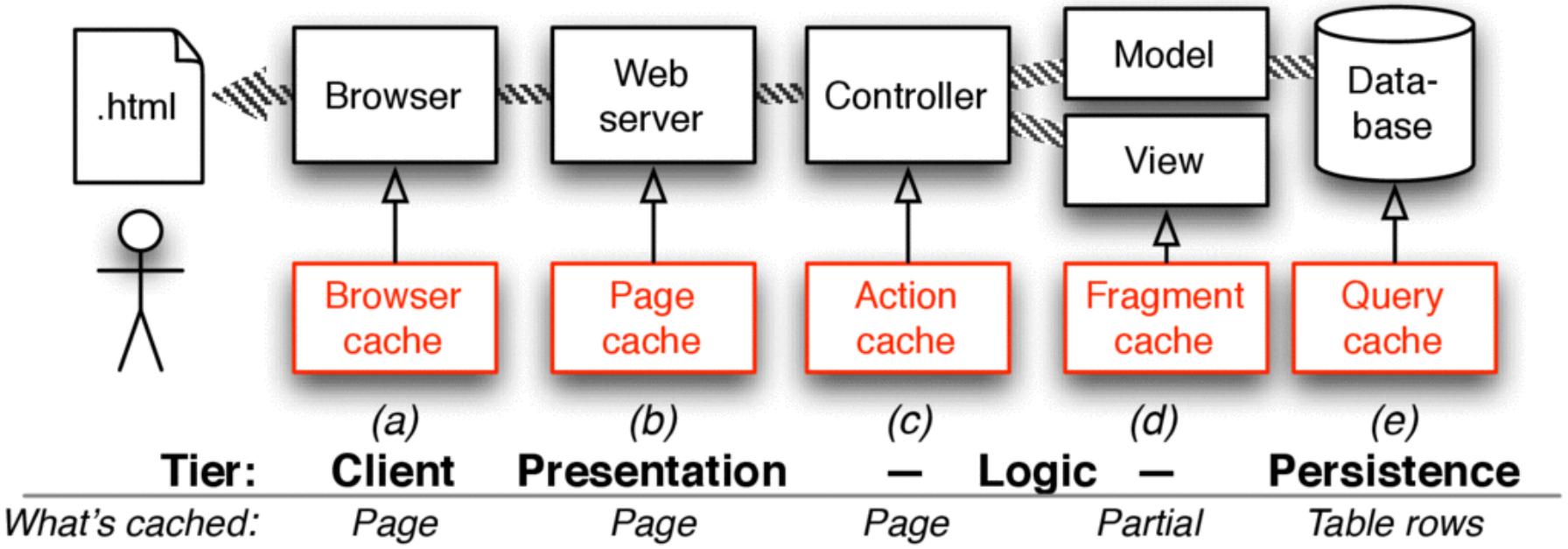
*

The fastest database is the one you don't use

- **Caching:** Avoid touching database if answer to a query hasn't changed

1. Identify what to cache
   - whole view: page & action caching
   - parts of view: fragment caching with partials
2. Invalidate (get rid of) *stale* cached versions when underlying DB changes

Cache flow


.html
Browser
Web server
Controller
Model
View
Data-base
Browser cache
Page cache
Action cache
Fragment cache
Query cache
(a)
(b)
(c)
(d)
(e)
Tier:
Client
Presentation
—
Logic
—
Persistence
What's cached:
Page
Page
Page
Partial
Table rows

Page & Action Caching

- When: output of *entire action* can be cached
- *Page caching* bypasses controller action
- *Action caching* runs filters first
- *Corollary:* don't mix filter & non-filter code paths in same action!
- *Caveat:* caching based on page URL *without* optional "?" parameters!

/movies/index?rating=PG = movies/index

/movies/index/rating/PG ≠ movies/index

Example

•Bad:

```
caches_page :index
def index
  if logged_in?
    ...
  else
   redirect_to login_path
  end
end
```

- Better:

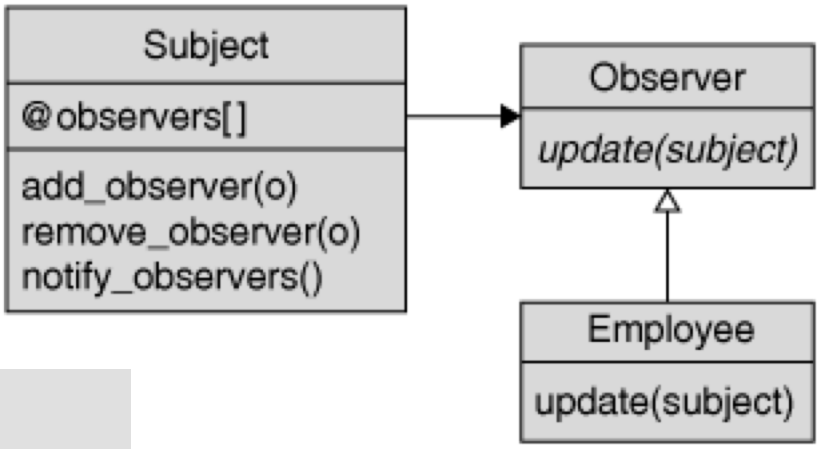
```
caches_page :public_index
caches_action :logged_in_index
before_filter :check_logged_in,
:only => 'logged_in_index'
def public_index
  ...
end

def logged_in_index
  ...
end
```

Fragment caching for views

- Caches HTML resulting from rendering part of a page (e.g. partial)
- Cuts back on *rendering time*, not database queries

```
- cache "movies_with_ratings" do
  = render :collection => @movies
```

- How do we detect when cached versions no longer match database?
- *Sweepers* use Observer design pattern to separate expiration logic from rest of app

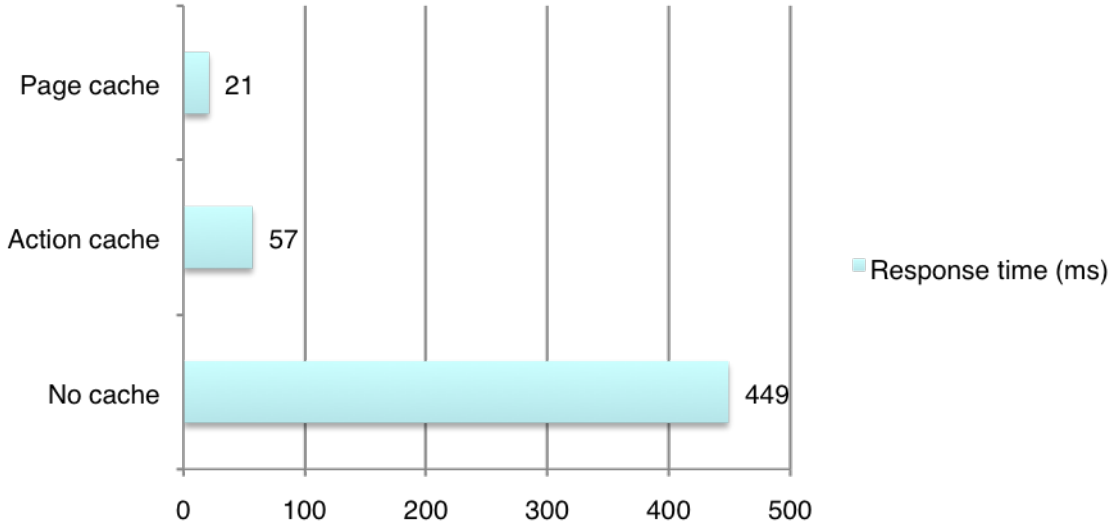


http://pastebin.com/fCZJSimS

*

How much does caching help?

- With ~1K movies and ~100 reviews/movie in RottenPotatoes on Heroku, `heroku logs` shows:



- Can serve 8x to 21x as many users with same number of servers if caching used

Under-17 visitors to RottenPotatoes shouldn't see NC-17 movies in any listing. What kinds of caching would be appropriate:
i) Page ii) Action iii) Fragment

- (i) & (iii)
- (ii) & (iii)
- (iii) only
- (i), (ii) and (iii)

Be kind to the database

- Outgrowing a single-machine database means a big investment: sharding, replication, etc.
- Alternative: find ways to relieve pressure on database so can stay in "PaaS-friendly" tier

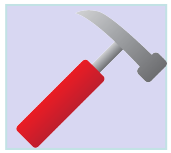
1. Use caching to reduce number of database accesses
2. Avoid "n+1 queries" problem in Associations
3. Use indices judiciously

n+1 queries problem

- **Problem:** you are doing n+1 queries to traverse an association, rather than 1 query

http://pastebin.com/QKxqcbhk

- **Solution:** bullet gem can help you find these



- **Lesson:** all abstractions eventually leak!

Eager loading

- Naive way:

```
@movie = movie.where( … )
reviews = @movie.reviews
```

- May be faster:

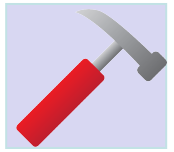
```
@movie = movie.where( … ).include(:reviews)@movie.reviews.each do |review|
  # reviews are already loaded!
```

Indices

- Speeds up access when searching DB table by column other than primary key
  - e.g. Movie.where("rating = 'PG'")
- Similar to using a hash table
  - alternative is *table scan*—bad!
  - even bigger win if attribute is unique-valued
- Why not index *every* column?
  - takes up space
  - all indices must be updated when table updated

What to index?

- Foreign key columns, eg movie_id field in Reviews table
- why?
- Columns that appear in where() clauses of ActiveRecord queries
- Columns on which you sort
- Use rails_indexes gem (on GitHub) to help identify missing indices (and unnecessary ones!)



How much does indexing help?

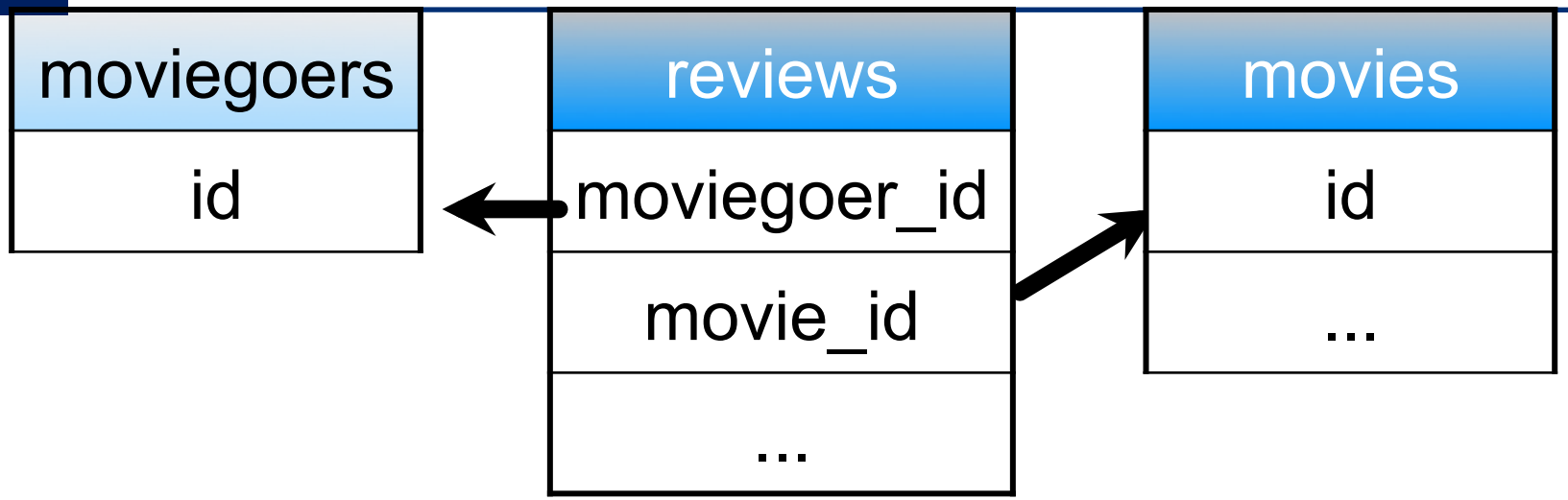
# of reviews:	2000	20,000	200,000
Read 100, no indices	0.94	1.33	5.28
Read 100, FK indices	0.57	0.63	0.65
Performance	166%	212%	808%

	200,000
Create 1K, no indices	9.69
Create 1K, all indices	11.30
Performance	–17%

Suppose Movie has many Moviegoers through Reviews. Which column index would MOST help speed up the queryfans = @movie.moviegoers

- movies.review_id
- reviews.movie_id
- reviews.moviegoer_id
- moviegoers.review_id

has_many :through



- moviegoer:

has_many :reviewshas_many :movies, :through => :reviews

- movie: has_many :reviews

has_many :moviegoers, :through => :reviews

- reviews: belongs_to :moviegoer

belongs_to :movie

Common Attacks on the App

1. Eavesdropping
2. Man-in-the-middle/Session hijack
3. SQL injection
4. Cross-site request forgery (CSRF)
5. Cross-site scripting (XSS)
6. Mass-assignment of sensitive attributes

…more in book

SSL (Secure Sockets Layer)

- Idea: *encrypt* HTTP traffic to foil eavesdroppers
- Problem: to create a *secure channel,* two parties need to *share a secret* first
- But on the Web, the two parties don't know each other
- Solution: *public key cryptography* (Rivest, Shamir, Adelman shared 2002 Turing Award)

What SSL Does, and Doesn't

- Each principal has a *key* of 2 matched parts
- public part: everyone can know it
- private part: principal keeps secret
- given one part, cannot deduce the other
- Key mechanism: *encryption* by one key requires *decryption* by the other
- If a message can be decrypted with Bob's public key, then Bob must have used created it
- If I use Bob's public key to create a message, only Bob can read it

How SSL works (simplified)

1. Bob.com proves identity to CA
2. CA uses its *private* key to create a "cert" tying this identity to domain name "bob.com"
3. Cert is installed on Bob.com's server
4. Browser visits http://bob.com
5. CA's public keys *built into browser*, so can check if cert matches hostname
6. *Diffie-Hellman key exchange* is used to bootstrap an encrypted channel for further communication

Use Rails force_ssl method to force some or all actions to use SSL

What it Does and Doesn't Do

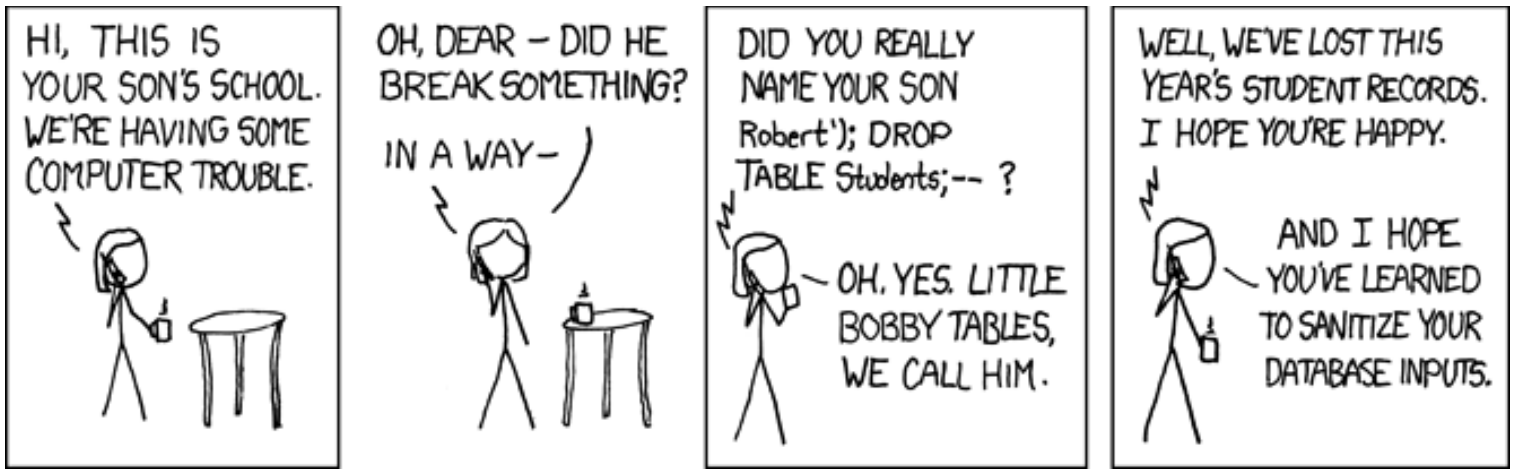
- Assures browser that bob.com is legit
- Prevents eavesdroppers from reading HTTP traffic between browser & bob.com
- Creates additional work for server!

DOES NOT:

- Assure server of who the user is
- Say anything about what happens to sensitive data *after* it reaches server
- Say anything about whether server is vulnerable to other server attacks
- Protect browser from malware if server is evil

SQL Injection

- View: = text_field_tag 'name'
- App: Moviegoer.where("name='#{params[:name]}'")
- Evil user fills in: BOB'); DROP TABLE moviegoers; --
- SELECT * FROM moviegoers WHERE (name='BOB'); DROP TABLE moviegoers; --'
- Solution: Moviegoer.where("name=?", params[:name])



xkcd.com/327

Cross-site Request Forgery

1. Alice logs into bank.com, now has cookie
2. Alice goes to blog.evil.com
3. Page contains: `<img src="http://bank.com/account_info"/>`
4. evil.com harvests Alice's personal info

Solutions:

1. (weak) check Referer field in HTTP header
2. (strong) include *session nonce* with every request
   - csrf_meta_tags in layouts/application.html.haml
   - protect_from_forgery in ApplicationController
   - Rails form helpers automatically include nonce in forms

If a site has a valid SSL certificate from a trusted CA, which of the following is true:

i) The site is probably not "masquerading" as an impostor of a real site

ii) CSRF + SQL injection are harder to mount against it

iii) Your data is secure once it reaches the site

- ☐ (i) only
- ☐ (i) & (ii) only
- ☐ (ii) & (iii) only
- ☐ (i), (ii) & (iii)

Optimizing Prematurely or Without Measurements

- Speed is a feature that users expect
- 99%ile (eg), not “average”
- *Horizontal scaling* >> per-machine performance, *but* lots of ways things can slow down
- Monitoring is your friend: measure twice, cut once
- More*:* *railslab.newrelic.com/scaling-rails*

*

"Mine is a 3-tier app on cloud computing, so it will scale"

- Database is particularly hard to scale
- Even if you do, still want to get "expensive" operations out of the way of your SLO
- One help: cache at many levels
- whole page, fragment, query
- Cache *expiration* is a crosscutting concern
- RoR support for crosscutting concerns allows you to specify it declaratively
- Use PaaS for as long as you can

*

"My small site isn't a target"

- Hackers may be after your users, not your data
- Like performance, security is a *crosscutting concern*—hard to add after the fact
- Stay current with best practices and tools—you're unlikely to do better by rolling your own
- Prepare for catastrophe: keep regular backups of site and database

Your users are sporadically complaining that your site is slow, yet New Relic reports low traffic levels and low CPU utilization. What is the likely cause?

- Not enough Heroku “dynos”, so requests occasionally get “backed up”
- Some queries are unusually slow because you’re sharing DB with other apps
- Some views take unusually long to render in certain browsers (eg, due to JavaScript)
- It could be any of these/Not enough information

*