Scaling Rails with memcached

One Rubyist's Guide to Hazardous Hits

memcached is your best friend and your worst enemy.

Who is this kid?

CNET Networks

PHP Developer

- GameSpot
- TV.com
- MP3.com

Rails Developer

- Chowhound
- Chow.com

- I work for CNET Networks in San Francisco.
- I used to do PHP for www.gamespot.com, www.tv.com, and www.mp3.com.
- GameSpot is one of the two biggest gaming sites on the web, the other being IGN. They are pretty evenly matched.
- I now work full time as a Rails developer on www.chowhound.com and www.chow.com (launching September 15th, 2006).

GameSpot

- 30 million page views per day
- 50 million++ during E3 (mid-May)
  - [3000 req/s]
- ~70 app servers
- Netscalers
- ~15 database servers
- php4, apache 2, mysql 5

E3 was a press event held yearly, 2006 being the final year. During this year's E3 the Sony, Microsoft, and Nintendo press conferences were streamed live by GameSpot. We had the exclusive on the Sony and Nintendo conferences, with both companies' domains pointing to our webcast as it was happening. Massive traffic, everything went crazy. We had about 100 app servers for E3, all in the red during peak times.

Imagine millions of teenage boys constantly hitting refresh.

Typical setup: request hits Netscaler (load balancer), Netscaler decides which app server gets the request. Use gzip compression to save bandwidth (but had to turn it off due to load for peaks at E3).

memcached.

memcached helps.

memcached

- Distributed, in-memory data store
- Created by and for LiveJournal
- Scales
- Fast. C. Non-blocking IO. O(1)
- Digg, Facebook, Wikipedia, GameSpot
- 43things, Mog, Chowhound

- Distributed: multiple servers storing data in RAM, transparent to your app.
- Data is spread amongst servers, not duplicated.
- LiveJournal had bad growing pains -- needed a solution.
- Problem with databases: you can distribute reads but everyone needs to write (which blocks)
- Scales -- drop in a new memcached daemon and you're good to go.
- Lots of big sites use it successfully

Like a DRb'd {}

- CACHE = MemCache.new(options)
- CACHE.get(:key)
- CACHE.set(:key, data, ttl)
- CACHE.delete(:key)

- Very simple to use
- Conceptually a hash
- Time to live is expiration of key. You can set to 30 days in the future or pick a definitive time

Like a DRb'd {}

- memcached 'bucket' (server) is decided based on requested key
- memcache-client uses Marshal to store data
- Can't cache metaclassy things or procs
- Run daemons on your app servers. Low CPU overhead.

- memcache-client Ruby API decides which memcached server stores your key

Like a DRb'd {}

(sort of)

- Not enumerable
- Not a database

- Hash keys are not maintained
- This makes it faster
- Not a database: things only kept in RAM, not persistent.
- Restart your server, lose your data. That's ok.

The Pattern

- Check cache for key
- If key exists, return the stored bytes
- If key doesn't exist, find bytes
- Save bytes
- Return bytes

- We don't really have to worry about this if we use a good library.
- "find bytes" can be a database query or any sort of intensive data lookup -- something we don't want to do on every request and therefor cache.

In other words...

```
def get_cache(key)
  data = CACHE.get(key)

  return data unless data.nil?

  data = find(key)
  CACHE.set(key, data, self.ttl)
  return data
end
```


Simple example. Assume `find' will return what we want to cache. Can be anything.

In other words...

```
def get_cache(key)
  data = CACHE.get(key)

  return data unless data.nil?

  data = find(key)
  CACHE.set(key, data, self.ttl)
  return data
end
```


- What if we want to cache nil? We will query the database every time the key is requested.
- If you have 4 million rows that's not something you want.
- Cache false instead of nil.
- This and other things you need to watch for.

What about MySQL query cache?

- memcached is for data.
- You can cache anything.
- Generated PNG's, HTML, processed objects, etc.

Forget the query cache.

Are you sure you need it?

You might not.

It's Not "Better" Than SQL

- memcached is slower than SELECT on localhost for small sites / datasets
- memcached will not help if you can't keep up with the requests
- None of the big guys started with memcached.

- Not going to speedup your blog. Going to slow it down.
- You need good hardware and a good setup before you can think about memcached
  - multi server setup, load balancing, master / slave databases
- Facebook, Slashdot, Digg didn't start with memcached. They added it when they needed it.

Don’t Worry About It

- Write your app and tests first
- Get everything working smoothly
- Keep a few days open to spend on caching
- ^^ Near The End

- Don’t let memcached get in the way
- Don’t do premature optimization
- Write good code, but dont GUESS where the problems are going to be. Test and see.
- Your app needs to be able to run well without caching. Caching is an aid, not a crutch.
- If you want to test your caching code use a hash-based mock.
- acts_as_cached has one in test/fixtures/

Don’t Worry About It

- Identify slow queries
  - MySQL slow query log
- Hardware is the real secret sauce
- memcached is not a magic wand

- Find out what’s really holding you back, the big bottlenecks
- Use the QueryTrace plugin to see where your queries are coming from

Where it Shines

- Huge database, millions of rows
- Huge traffic, millions of hits
- Huge both, millions of everything

- Even a few row select can be slow on a massive database
- If your query does a filesort on a big dataset, forget it
- Set your cache based on a unique id
- Maybe your database isn't massive but is getting completely hammered. Cache.

Yeah, but I'm about to launch Web 9.0

– COUNT ME IN!!!

Hook Me Up

- Download and compile from danga.com
- OSXers: libevent and Hodel's hacks
- $ sudo gem install memcache-client
- $ memcached -vv

- Danga are the LiveJournal guys
- Won't work well on osx without Hodel's hacks
- Install the Robot Coop's memcache-client
- Start daemon in interactive mode, watch what's going on.
- Setup a MemCache.new instance, hit .set and .get, watch what happens in your memcached daemon

Options

- Fragment Cache Store
  - http://rubyforge.org/projects/mcache-fragment/
- Session Store
  - config.action_controller.session_store = :mem_cache_store
- Models (DB Queries)
  - CachedModel

- Fragment cache store plugin supports time based expirations
- Lots of stuff written about setting up session store
- CachedModel is what 43things wrote and uses.

CachedModel

- Overwrites find() to pull from cache
- Works on single objects only
- Doesn't play well with multi-object queries on gets or updates
- Makes you inherit
- Nice and simple
- Clears cache keys on update()

- You don't have to worry about caching: does it for you
- Doesn't work on find(:all) or complex queries (:include, etc)

acts_as_cached

- Rails plugin
- Allows you to store any Ruby object in memcached
- Gives you more than enough rope...

- Used on Chowhound and Chow
- Gives us flexibility
- Still have to do things by hand, but helps ease the pain
- Puts the caching in your face. Think about what you're doing.

What To Cache

- Slow Queries
- Page fragments
  - (We don't cache fragments on Chowhound)
- Processor intensive computations

What To Cache

```
class StoriesController < ApplicationController

  def show
    @story = Story.get_cache(params[:id])
  end

  def show
```


- Simple single row get
- We like get_cache because you are always aware of what is and isn't cached

Issues

- Keeping cache data fresh
  - Editors want to see their changes now!
- Caching associations
- Pagination / Big Collections

- Chowhound is a forum site, Chow is an editorial content site
- Everyone wants to see their changes immediately
- Mix that with lots of pagination and big collections
- How the hell?
- We want to cache associations because we want to use Rails' niceness
- But we don't have to keep reimplementing the same patterns

Keep Cache Fresh

```
class Story < ActiveRecord::Base
  acts_as_cached

  def after_save
    expire_cache(id)
  end
end
```


- Simplest example.
- acts_as_cached doesn't handle auto-deletion of keys for you like CachedModel
- Don't forget to after_destroy, too!

Keep Cache Fresh

```
class Story < ActiveRecord::Base
  acts_as_cached :include => :author
  belongs_to :author

  def after_save
    expire_cache(id)
  end
end
```


- acts_as_cache can be passed a hash which is passed straight to find()
- In this case, Story.get_cache(20) calls Story.find(20, :include => :author)
- Caches the author along with the story
- If the author is updated, the story's associated cached author is STALE
- If you have 20 stories all with 1 author, you have 20 redundant copies of that same author in the cache
- RAM is cheap but you need to think this sort of thing through

Keep Cache Fresh

```
class Author < ActiveRecord::Base
  acts_as_cached
  has_many :stories

  def after_save
    expire_cache(id)
    stories.each do |story|
      story.expire_cache
    end
  end
end
```


- Stories' author associations are now always up to date
- Pattern can be powerful when dealing with complex associations

Why?

- Keeping track of keys is hard
- Sweepers / Observers get complicated fast
- Maintains integrity with script/console, rake, and migrations

- CachedModel does it for you, which is nice
- We want to cache associations

Alternative

```
class Story < ActiveRecord::Base
  acts_as_cached
  belongs_to :author

    def author
      Author.get_cache(author_id)
    end

    def after_save
      expire_cache(id)
    end
  end
```


- Lots of cache hits -- unless you use an in-process cache
- Everything always up to date
- Best for simple associations
- Great if your authors table is in another database -- you can't do a join anyway

Versioning!

```
class Story < ActiveRecord::Base
  acts_as_cached :version => 5

  def after_save
    expire_cache(id)
  end
end
```


- What if you run a migration and update your Story model, then need to deploy?
- All stories in cached are not only stale but broken
- Mismatch between cached attributes and new attributes
- You may have run into this with sessions
- Update the version, acts_as_cached helps you out here
- Keys are normally stored as ENVIRONMENT:CLASS:ID
- So app-development:Story:20
- :version => 5 makes Story's keys: app-development:Story:v5:20
- Instantly invalidate all currently cached, older versions.

List Objects

- Pagination is tricky
- Who controls it? Controller? Model?
- How do you keep pagination fresh?

- Pagination helpers are way out the window
- It makes sense to think of list objects as data structures (models), not active record objects
- They’re just simple data structures we want to arbitrarily cache

```
class BlogPostList
  acts_as_cached

  PAGES = 5
  PER_PAGE = 30

  def self.find(page = 0)
    BlogPost.find(:all, :limit => PER_PAGE, :offset => page * PER_PAGE)
  end

  def self.get_cache(page = 0)
    if page < PAGES
      super(page)
    else
      find(page)
    end
  end
end
```


- Create class, give it a find, overwrite its get_cache
- Pages 6 - infinity are not cached. Do they get any traffic? Probably not. Check your logs.
- Pages 1 - 5 are cached.

```
class BlogPostList
  acts_as_cached

  PAGES = 5
  PER_PAGE = 30

  def self.expire_list_cache
    0.upto(PAGES) do |page|
      expire_cache(page)
    end
  end
end
```


- Now it's easy to expire
- Put this in BlogPost.after_save:

```
def after_save
  BlogPostList.expire_list_cache
end
```

List Objects

- Not just for pagination
- Use it in your sidebar, footer, RSS, API, etc...
- Reuse the caches you have

- List objects can be reused
- Again, just data structures
- Don't bloat models with tons of caching code
- Do post-processing if you need to

Clear All

```
class ApplicationController < ActionController::Base
  before_filter :check_cache_skip

  def check_cache_skip
    returning true do
      Object.skip_cache_gets = params[:skip_cache] ? true : false
    end
  end
end
```

http://www.bigsite.com/index?skip_cache=1

- Resets every cache key on that page load
- Skips the get() so find() is called and result is saved to cache then returned
- Good for debugging caching code
- Is what's in my cache what I really want? Is all the clearing working?

Caveats

- 1 meg per key limit
- Disable caching in unit tests, or use a mock
- No whitespace in key names!
- Don't save 'nil'
- Monitor your cache. memcached is fickle
- Weird behavior? Restart.

- Dont go overboard caching - 1 meg limit
- acts_as_cached handles some of this: no whitespace in names, nils are converted to false
- memcached has a tendency to return corrupt data after tons of use. Restart liberally.
- Don't try to keep memcached up forever. Bad mojo.

Monitoring

```
$ sudo gem install memcache-client-stats
$ ruby script/console
>> require ‘memcache_client_stats’
=> true
>> CACHE.stats(:key => 'bytes')
=> {"localhost:11211"=>"271349",
    "localhost:11212"=>"62729"}
>> CACHE.stats(:type => 'items')
{"localhost:11211"=>
  {"items:18:age"=>"6497", "items:18:number"=>"1", ... }}
```


- Great gem
- Spend some time learning what everything means
- Watch all your servers in one place -- build a page which aggregates and displays this information
- It’d be nice if this was integrated into capistrano somehow...
  (I’ll get back to you on that)

Monitoring

- Cacti
  - http://cacti.net/
- memcached template
  - http://dealnews.com/developers/cacti/memcached.html

- Haven't used it but it gets recommended on the mailing list.

Resources

- memcached site
  - http://www.danga.com/memcached/
- memcached mailing list
  - http://lists.danga.com/pipermail/memcached/
- topfunky's post
  - http://rubyurl.com/sbE
- Robot Coop
  - http://dev.robotcoop.com/
- The Adventures of Scaling
  - http://rubyurl.com/vK7
- Sessions N Such (memcached sessions)
  - http://errtheblog.com/post/24

Questions? Thanks!

Chris Wanstrath
chris@ozmm.org

Err the Blog
http://errtheblog.com

www.chow.com
www.chowhound.com

2006-09-12