Rails Application Optimization Techniques & Tools

Stefan Kaes

www.railsexpress.de

skaes@gmx.net

A tiny piece of history

Performance Tuning

- Trying to improve performance without measuring is foolish.
- If your app's design has an inherent performance problem, you will need a costly redesign.
- Planning ahead for an established performance goal will help you cut down on the cost of performance improvement work.
- There's no need to optimize every page of your web app.
- Focus on your app's hot spots (frequently visited pages), and measure continuously during development.




Performance Parameters

Latency

How fast can you answer a request?

Throughput

How many requests can you process per second?

Utilization

Are your servers/components idle most of the time?

Cost Efficiency

Performance per unit cost

Compute mean, min, max, standard deviation (if applicable)

Standard deviation will tell you how reliable your data is.

Benchmarking Tools

- Rails log files (*debug level* $\geq$ `Logger::DEBUG`)
- Rails Analyzer Tools(requires logging to syslog)
- Rails benchmarker script (script/benchmarker)
- Tools provided by DB vendor
- Apache Bench (`ab` or `ab2`)
- httperf
- railsbench
  - downloadable from http://rubyforge.org/projects/railsbench/




railsbench

measures raw performance of Rails request processing

configured through

- benchmark definitions
  `$RAILS_ROOT/config/benchmarks.yml`
- benchmark class configuration
  `$RAILS_ROOT/config/benchmarks.rb`

stores benchmark data in

`$RAILS_PERF_DATA`

indexed by date and benchmark name

uses additional Rails environment `benchmarking`

railsbench usage

`perf_run 100 "-bm=welcome` *options*`"` [*data file*]

run 100 iterations of benchmark with given options, print data

`perf_diff 100 "-bm=all` *opts*`"` `"`*opts1*`"` `"`*opts2*`"` [*file1*] [*file2*]

run 100 iterations of benchmark, print data comparing *opts1* and *opts2*

`perf_times` *data file 1* `...`

print performance data contained in files

`perf_comp [-narrow]` *data file 1* *data file 2*

print comparison of raw data files

railsbench options

-log[= *level*]
: turn on logging (defaults to no logging). optionally override log level.

-nocache
: turn off Rails caching

-path
: exit after printing `$:`

-svlPV
: run test using Ruby Performance Validator

-patched_gc
: use patched GC

Ruby Profiling Tools

- Ruby Profiler
- Zen Profiler
- rubyprof
- Rails profiler script
- Ruby Performance Validator (commercial, Windows only)

All but the last are pretty much useless for Rails performance work.

`railsbench` has builtin support for RPVL:

```
run_urls 100 -svlPV -bm=welcome ...
```

will start RPVL and run the named benchmark with given options

Top Rails Performance Problems

Depends on who you ask, but these are my favorites:

- slow helper methods
- complicated routes
- associations
- retrieving too much from DB
- slow session storage

Judging from my experience, DB performance is usually not a bottleneck.

Instantiating ActiveRecord objects is more expensive.

Available Session Containers

In Memory
Fastest, but you will lose all sessions on app server crash/restart. Restricted to 1 app server process. *Doesn't scale*.

File System
Easy setup. One file (below /tmp) for each session. Scales by using NFS or NAS (beware 10K active sessions!). *Slower than*

Database/ActiveRecordStore
Easy setup (comes with Rails distribution). *Much slower than*

Database/SQLSessionStore
Uses ARStore session table format. But does all processing using raw SQL queries. Needs tweaking if you want additional fields on session entries. *setup*

memcached
Slightly faster than SQLSessionStore. Presumably scales best. Very tunable. Automatic session cleaning. Harder to obtain statistics. *setup*

DrbStore
Can be used on platforms where `memcached` is not available. Slower than memcached. No automatic expiry (but could be added quickly).




Cachable Elements

Pages

Fastest. Complete pages are stored on file system. Web server bypasses app server for rendering. Scales through NFS or NAS. Problematic if your app requires login.

Actions

Second fastest option. Caches the result of invoking actions on controllers. User login id can be used as part of the storage key.

Fragments

Very useful for caching small fragments (hence the name) of HTML produced during request processing. Can be made user aware.

Action caching is just a special case of fragment caching.

Several storage containers are available for fragment caching.

Storage Options for Fragment Caching

In Memory

Blazing speed! If your app is running fast enough with 1 app server process, go for it!

File System

Reasonably fast. Expiring fragments using regular expressions for keys is slow.

DrbStore

Comparable to FileStore. Expiring fragments is faster.

memcached

Faster and more scalable than DrbStore. Doesn't support expiring by regular expression.

The size of the actual code in Rails to handle caching is small.

It would be easy to extend so that all of the above options can be used concurrently.

ActionController Issues

Components

I suggest to avoid components. I haven't found any good use for them, yet.

Each embedded component will be handled using a fresh request cycle.

Can always be replaced by helper methods and partials.

Filters

If you are using components, make sure you don't rerun your filters n times. Better pass along context information explicitely.

You can use the `skip_filter` method for this. It will be evaluated at class load time, so no runtime overhead during request processing.

ActionView Issues

Instance variables

For each request, one controller instance and one view instance will be instantiated.

Instance variables created during controller processing will be transfered to the view instance (using `instance_variable_get` and `instance_variable_set`)

So: avoid creating instance variables in controller actions, which will not be used in the view (not always possible, see filters).

Slow Helper Methods

Consider:

```
1 pluralize(n, 'post')
```

This will create a new Inflector instance, and try to derive the correct plural for 'post'. This is expensive. Just do

```
1 pluralize(n, 'post', 'posts')
```

link_to and url_for

Due to the route generation involved, `url_for` and `link_to` are among the slowest helper methods around.

```
1 <%= link_to "look here for something interesting",
2         { :controller => "recipe", :action => edit, :id => @recipe.id },
3         { :class => " edit_link " } %>
```

- both hash arguments undergo heavy massage in the Rails bowel: symbolizing, html escaping, sorting, validation
- the routing module determines the shortest possible route to the controller, action and id

A much more efficient way to write this is:

```
<a href="/recipe/edit/<%=#{recipe.id}%>" class="edit_link">
look here for something interesting
</a>
```

ActiveRecord Issues

Accessing AR objects via association proxies is (comparetively) slow.

You can prefetch associated objects using `:include`

```
1  class Article
2    belongs_to :author
3  end
4  Article.find(:all, :include => :author)
```

Use piggy backing for `has_one` or `belongs_to` relations.

```
1  class Article
2    piggy_back :author_name, :from => :author, :attributes => [:name]
3  end
4  article = Article.find(:all, :piggy => :author)
5  puts article.author_name
```

Field values are retrieved from the DB as strings (mostly).

Type conversion happens on each access.

$\Longrightarrow$ cache converted values, if you use them several times during one request.

Caching Column Formatting

Computationally expensive transformations on AR fields can be cached (in the DB, using memcached, a DRb process)

Example: `textilize`

- I've analyzed an application, where 30% CPU was spent on textilizing. Another 30% were spent on GC. And about 10% on URL recognition.
- Caching the formatted fields in the DB eliminated the textilize overhead completely.

At one point in time a plugin existed for this task.




Ruby's Interpreter is Slow

- no byte code, no JIT, interprets ASTs directly
- doesn't perform any code optimization at compile time:
  - method inlining
  - strength reduction
  - constant propagation
  - common subexpression elimination
  - loop invariant detection
  - loop unrolling

$\Longrightarrow$

Performance aware Ruby programming can increase performance significantly!

Complexity of Ruby Language Elements

Local Variable access: $O(1)$
index into array, computed at parse time

Instance Variable Access: expected $O(1)$
hash access by literal

Method Call: expected $O(1)$

- hash access to determine literal value (`"f"` $\Rightarrow$ `:f`)
- method search along inheritance chain
- hash accesses to get the method
- there's also a method cache (which helps a bit)
- construction of argument array on heap (unless attr_reader)

Recommendation:

- don't add method abstractions needlessly
- use attr_accessors as external interfaces only
- use local variables to short circuit repeated hash accesses

Avoiding Repeated Hash Access


```ruby
def submit_to_remote(name, value, options = {})
  options[:with] ||= 'Form.serialize(this.form)'
  options[:html] ||= {}
  options[:html][:type] = 'button'
  options[:html][:onclick] = "#{remote_function(options)}; return false;"
  options[:html][:name] = name
  options[:html][:value] = value
  tag("input", options[:html], false)
end
```

This code is both simpler and faster:

```ruby
def submit_to_remote(name, value, options = {})
  options[:with] ||= 'Form.serialize(this.form)'
  html = (options[:html] ||= {})
  html[:type] = 'button'
  html[:onclick] = "#{remote_function(options)}; return false;"
  html[:name] = name
  html[:value] = value
  tag("input", html, false)
end
```

Caching Data in Instance Variables

If you need the same data structure repeatedly during request processing, consider caching on controller (or view) instance level.

Turn

```
1 def capital_letters
2   ("A" .. "Z").to_a
3 end
```

into

```
1 def capital_letters
2   @capital_letters ||= ("A" .. "Z").to_a
3 end
```

Caching Data in Class Variables

If your data has a reasonable size to keep around permanently and is used on a hot application path, consider caching on class level.

Turn

```
1 def capital_letters
2   ("A" .. "Z").to_a
3 end
```

into

```
1 @@capital_letters = ("A" .. "Z").to_a
2
3 def capital_letters
4   @@capital_letters
5 end
```

the cached value could be a query from the database, e.g. guest user account.

Coding Variable Caching Efficiently

Turn

```
1  def actions
2    unless @actions
3      # do something complicated and costly to determine action's value
4      @actions = expr
5    end
6    @actions
7  end
```

into

```
1  def actions
2    @actions ||=
3      begin
4        # do something complicated and costly to determine action's value
5        expr
6      end
7  end
```

Defining Constants vs. Inlining

Less than optimal:

```
def validate_find_options (options)
  options.assert_valid_keys(:conditions, :include, :joins, :limit, :offset,
                            :order, :select, :readonly, :group, :from )
end
```

Better:

```
VALID_FIND_OPTIONS = [
    :conditions, :include, :joins, :limit,
    :offset, :order, :select, :readonly, :group, :from ]

def validate_find_options (options)
  options.assert_valid_keys(VALID_FIND_OPTIONS)
end
```

Faster and much easier to customize.

Local Variables are Cheap

Consider:

```
1  sql << " GROUP BY #{options[:group]} " if options[:group]
```

vs.

```
1  if opts = options[:group]
2    sql << " GROUP BY #{opts} "
3  end
```

or

```
1  opts = options[:group] and sql << " GROUP BY #{opts} "
```

Alas,

```
1  sql << " GROUP BY #{opts} " if opts = options[:group]
```

won't work, because matz refused to implement it (at least last time I asked for it).

Beware Variable Capture When Defining Methods

Defining a new method passing a block, captures the defining environment.

This can cause memory leaks.

```
1  def define_attr_method(name, value=nil, &block)
2    sing = class << self; self; end
3    sing.send :alias_method, "original_#{name}", name
4    if block_given?
5      sing.send :define_method, name, &block
6    else
7      # use eval instead of a block to work around a memory leak in dev
8      # mode in fcgi
9      sing.class_eval "def #{name}; #{value.to_s.inspect}; end"
10   end
11 end
```

It's usually preferable to use `eval` instead of `define_method`, unless you need the variable capture.

Be Careful w.r.t. Logging

- set the production log level to something other than `DEBUG`.
- don't log to log level `INFO` what should be logged to `DEBUG`.

This is a bad idiom:

```
1 logger.debug "args: #{hash.keys.sort.join(' ')}" if logger
```

`hash.keys.sort.join(' ')` will be evaluated and the arg string will be constructed, even if logger.level == ERROR.

Instead do this:

```
1 logger.debug "args: #{hash.keys.sort.join(' ')}" if logger && logger.debug?
```




ObjectSpace.each_object

Contrary to popular belief

```
1 ObjectSpace.each_object(Class) {|c| f(c) }
```

is just as slow as

```
1 ObjectSpace.each_object {|o| o.is_a?(Class) && f(o) }
```

In both cases, *every* object on the heap is inspected!

Don't call it in production mode on a per request basis.

BTW: `ObjectSpace.each_object` has dubious semantics

Ruby's Memory Management

- designed for batch scripts, not long running server apps
- tries to minimize memory usage
- simple mark and sweep algorithm
- uses malloc to manage contiguous blocks of Ruby objects (Ruby heap)
- complex data structures:
  - only references to C structs are stored on Ruby heap
  - comprises strings, arrays, hashes, local variable maps, scopes, etc.
- eases writing C extensions

Current C interface makes it hard to implement generational GC

$\Longrightarrow$ unlikely to get generational GC in the near future

Maybe Ruby2 will have it (but Ruby2 is a bit like Perl6)

Why Ruby GC is suboptimal for Rails

ASTs are stored on the Ruby heap and will be processed on each collection

usually the biggest part of non garbage for Rails apps

Sweep phase depends on size of heap, not size of non garbage

can't increase the heap size above certain limits

More heap gets added, if

size of freelist after collection $<$ `FREE_MIN`

a constant defined in `gc.c` as 4096

200.000 heap slots are a good lower bound for live data

for typical Rails heaps, 4096 is way too small!

Improving GC Performance

Control GC from the Rails dispatcher:

```
1 # excerpt from dispatch.fcgi
2 RailsFCGIHandler.process! nil, 50
```

Will disable Ruby GC and call GC.start after 50 requests have been processed

However, small requests and large requests are treated equally

- heap could grow too large
- performance for small pages suffers
- Ruby will still deallocate heap blocks if empty after GC

Patching Ruby's Garbage Collector

Download latest `railsbench` package. Patch Ruby using file `rubygc.patch`, recompile and reinstall binaries and docs.

Tune GC using environment variables

RUBY_HEAP_MIN_SLOTS
: initial heap size in number of slots used (default 10000)

RUBY_HEAP_FREE_MIN
: number of free heap slots that should be available after GC (default 4096)

RUBY_GC_MALLOC_LIMIT
: amount of C data structures (in bytes) which can be allocated without triggering GC (default 8000000)

Recommended values to start with:

```
RUBY_HEAP_MIN_SLOTS   =    600000
RUBY_GC_MALLOC_LIMIT = 60000000
RUBY_HEAP_FREE_MIN    =    100000
```

Running the previous benchmark again, gives much nicer GC stats

Compile Time Template Optimization

Many helper calls in Erb templates can be evaluated at template compile time.

```
<%= end_form_tag %>  ===>  </form>
```

It's a *complete waste* to do it over and over again on a per request basis

For some calls, we know what the output should be like, even if we don't have all arguments available

```
<%= link_to "Edit",
     {:controller => "recipe", :action => edit, :id => @record},
     {:class => "edit_link"} %>
```

could be replaced by

```
<a href="/recipe/edit/<%= @record.to_param %>"
   class="edit_link">Edit</a>
```

Rails Template Optimizer

Uses Ryan Davis' ParseTree package and ruby2ruby class

Retrieves AST of ActionView render method after initial compilation

Transforms AST using

- helper method inlining
- dead code removal
- unused variable removal (from partials)
- hash merging
- constant evaluation
- strength reduction
- constant call evaluation
- symbolic evaluation

until AST cannot be simplified further

Compiles new AST into optimized render method using *eval*

Optimizer Customization

```
TemplateOptimizer::INLINE_CALLS.merge( ... )

TemplateOptimizer::EVALUATE_CALLS.merge( ... )

TemplateOptimizer::EVALUATE_CONSTANTS.merge( ... )

TemplateOptimizer::IGNORED_METHODS.merge( ... )

TemplateOptimizer::CALLS_RETURNING_STRINGS.merge( ... )
```

Optimizer Restrictions

url hashes cannot be optimized if the hash domain isn't constant

if your app hosts several domains, url hashes cannot be optimized if `:only_path => false` gets passed




Back
Close

Optimizer Performance Benchmark

Run this morning (1000 -bm=uncached -mysql_session . . .):

```
page    c1 real   c2 real c1 r/s c2 r/s c1 ms/r c2 ms/r  c1/c2
 1:   22.52100   6.80167   44.4  147.0   22.52    6.80   3.31
 2:   39.61467   6.86433   25.2  145.7   39.61    6.86   5.77
 3:   40.67167   6.43267   24.6  155.5   40.67    6.43   6.32
 4:   33.89600   5.80767   29.5  172.2   33.90    5.81   5.84

all: 136.70333 25.90633   29.3  154.4   34.18    6.48   5.28

GC:     c1 real   c2 real c1 #gc c2 #gc  c1 gc%  c2 gc%  c1/c2
       11.06067  2.91533   50.0   20.0    8.09   11.25   2.50
```

What do you say?

;-)

Project Status: α

Licensing: undecided

End

Thanks very much for your attention.

If you appreciated this session, you might consider buying my book, available early next year from Addison Wesley, as part of the ”Professional Ruby” series.

Questions?




ActiveRecordStore vs. SQLSessionStore

```
page   c1 real   c2 real c1 r/s c2 r/s c1 ms/r c2 ms/r c1/c2
 1:    2.80733  1.14600  356.2  872.6    2.81    1.15   2.45
 2:    3.91667  1.33867  255.3  747.0    3.92    1.34   2.93
 3:    5.21367  1.94300  191.8  514.7    5.21    1.94   2.68
 4:    5.65633  2.41167  176.8  414.7    5.66    2.41   2.35
 5:   11.64600  7.39600   85.9  135.2   11.65    7.40   1.57
 6:   16.83333 15.10933   59.4   66.2   16.83   15.11   1.11
 7:   17.09333 15.52067   58.5   64.4   17.09   15.52   1.10
 8:    8.19267  6.78133  122.1  147.5    8.19    6.78   1.21

GC:    c1 real  c2 real c1 #gc c2 #gc  c1 gc%  c2 gc% c1/c2
       3.83667  2.76133   25.0   20.0    5.38    5.35   1.25
```

Additional details regarding SQLSessionStore and memcached can be found here:

http://railsexpress.de/blog/articles/2005/12/19/roll-your-own-sql-session-store

http://railsexpress.de/blog/articles/2006/01/24/using-memcached-for-ruby-on-rails-session-storage




Configuring Rails to use SQLSessionStore with Mysql/Postgres

Download latest version from my web site

Put Ruby source under `lib` directory.

Adjust `environment.rb`:

```
require 'sql_session_store'
ActionController::CgiRequest::DEFAULT_SESSION_OPTIONS.update(
  :database_manager => SQLSessionStore)

require 'mysql_session'
SQLSessionStore.session_class = MysqlSession
```

For Postgres, use

```
require 'postgresql_session'
SQLSessionStore.session_class = PostgresqlSession
```

Note: requires Postgres 8.1!

memcached Session Storage Setup

Download memcache-client: http://rubyforge.org/frs/?group_id=1266

```
require 'memcache'
require 'memcache_util'

# memcache defaults, environments may override these settings
unless defined? MEMCACHE_OPTIONS then
  MEMCACHE_OPTIONS = {
    :debug => false,
    :namespace => 'my_name_space',
    :readonly => false
  }
end

# memcache configuration
unless defined? MEMCACHE_CONFIG then
  File.open "#{RAILS_ROOT}/config/memcache.yml" do |memcache|
    MEMCACHE_CONFIG = YAML::load memcache
  end
end
```

```ruby
# Connect to memcache
unless defined? CACHE then
  CACHE = MemCache.new MEMCACHE_OPTIONS
  CACHE.servers = MEMCACHE_CONFIG[RAILS_ENV]
end

# Configure the session manager to use memcache data store.
ActionController::CgiRequest::DEFAULT_SESSION_OPTIONS.update(
        :database_manager => CGI::Session::MemCacheStore,
        :cache => CACHE, :expires => 3600 * 12)
```

YAML file:

```yaml
production:
  - localhost:11211

development:
  - localhost:11211

benchmarking:
  - localhost:11211
```

Don't forget to start the server: `memcached&`

Session Container Overview





Back
Close

GC Statistics (unpatched GC)


```
GC data file: c:/home/skaes/perfdata/xp/perf_runworld.gc.txt

collections             :        66
garbage total           :   1532476
gc time total (sec)     :         1.86
garbage per request     :      2554.13
requests per collection:         9.09

                      mean  stddev%          min          max
gc time(ms):         28.08     22.0        15.00        32.00
heap slots :    223696.00      0.0    223696.00    223696.00
live       :    200429.88      0.4    199298.00    201994.00
freed      :     23266.12      3.3     21702.00     24398.00
freelist   :         0.00      0.0         0.00         0.00
```

GC Statistics (patched GC)


```
GC data file: c:/home/skaes/perfdata/xp/perf_runworld.gc.txt

collections             :         5
garbage total           :   1639636
gc time total (sec)     :         0.64
garbage per request     :      2732.73
requests per collection:       120.00

                      mean  stddev%          min          max
gc time(ms):        148.75       6.0       141.00       157.00
heap slots :     600000.00       0.0    600000.00    600000.00
live       :     201288.00       0.2    200773.00    201669.00
freed      :     398712.00       0.1    398331.00    399227.00
freelist   :          0.00       0.0         0.00         0.00
```

`ObjectSpace.each_object` *riddle*

Can you explain the output of this script?

```
1  def f(o)
2    1000.times { Array.new }; 1
3  end
4
5  puts ObjectSpace.each_object { }
6
7  f(0)
8
9  puts ObjectSpace.each_object { }
10
11 100.times do
12   puts ObjectSpace.each_object { |o| f(o) }
13 end
```

config/benchmarks.rb

```
1  # create benchmarker instance
2
3  RAILS_BENCHMARKER = RailsBenchmarkWithActiveRecordStore.new
4
5  # RAILS_BENCHMARKER.relative_url_root = '/'
6
7  # if your session manager isn't ActiveRecordStore, or if you don't
8  # want sesssions to be cleaned after benchmarking, just use
9  # RAILS_BENCHMARKER = RailsBenchmark.new
10
11 # create session data required to run the benchmark
12 # customize this code if your benchmark needs session data
13
14 require 'user'
15 RAILS_BENCHMARKER.session_data = {:user_id => 23}
16
```





Back
Close

config/benchmarks.yml

```
default:
    uri: /

all: default, empty, welcome, cat, letter

empty:
    uri: /empty/index
    new_session: true

welcome:
    uri: /welcome/index
    new_session: true

letter:
    uri: /rezept/letter/G

cat:
    uri: /rezept/cat/Hauptspeise
    query_params: page=5
```

Coding `link_to` *manually*


```
page   c1 real   c2 real c1 r/s   c2 r/s c1 ms/r c2 ms/r   c1/c2
 1:    1.38033  1.36467  724.5    732.8    1.38    1.36    1.01
 2:    2.21867  2.32833  450.7    429.5    2.22    2.33    0.95
 3:    2.90067  2.92733  344.7    341.6    2.90    2.93    0.99
 4:    2.87467  2.77600  347.9    360.2    2.87    2.78    1.04
 5:   11.10467  7.63033   90.1    131.1   11.10    7.63    1.46
 6:   12.47900  6.38567   80.1    156.6   12.48    6.39    1.95
 7:   12.31767  6.46900   81.2    154.6   12.32    6.47    1.90
 8:   11.72433  6.27067   85.3    159.5   11.72    6.27    1.87

GC:    c1 real  c2 real c1 #gc   c2 #gc  c1 gc%   c2 gc%   c1/c2
       6.48867  3.16600   43.0     23.0   11.38     8.76    1.87
```

- the relation c1/c2 depends on the hardware used. This seems to indicate that the quality of the Ruby implementation / OS memory management has a significant influence on relative performance.
- you probably shouldn't manually code every `link_to`. just use this method on pages with a large number of links. (Or wait for my template optimzer :-)




Measuring GC Performance Using `railsbench`

`perf_run_gc` *n* `"-bm=`*benchmark* `..."` [*data_file*]

runs named benchmark, producing a raw data file

`perf_times_gc` *data_file*

prints a summary for data in raw data file