Performance Rails

Writing Rails applications with sustainable performance
Ruby en Rails 2006

Stefan Kaes

www.railsexpress.de

skaes@gmx.net

The most boring talk, ever!

No DHTML visual effects!

No fancy AJAX programming!

No Java bashing!

No funny stories!

I won't even mention George!

What you will get

Lots of numbers!

Shameless promotion of my `railsbench` package!

Programming/engineering advice you might not want to hear

And I will even tell you to use Windows for Rails performance work!

How I got started on Rails

Cooking is one of my hobbies.

- wrote Perl screen scraper to collect recipes
- produced lots of static HTML and a PDF book
- $\#recipes > 1.000 \Longrightarrow$ unmanagable!

Needed: search, comments, favorites, menues, sharing with friends, access control $\Longrightarrow$ *a web application*

But: no interest (and justification) in writing another boring old web app using boring (PHP) or complicated (Java) web technology, so . . . project put to rest.

Enter: *Hackers and Painters*, **/.**, Rails movies, Ruby.

refreshing, interesting	$\Rightarrow$	Fun!
learn something new (Ruby)	$\Rightarrow$	Justification!




Knuzzlipuzzli Demo




Focus of this Talk

Rails performance benchmarking and tuning

- session container performance
- caching
- writing efficient Ruby/Rails code
- benchmarking
- tuning Ruby's garbage collector




Scaling Rails Apps

DHH says:

> ”Scaling Rails has been solved”

Don’t get fooled by this statement.

David likes to make provoking statements ;-)

It only means, ”it can be done” because of Rails’

Shared Nothing Architecture

In practice, scaling is a very complicated issue.

Rails is only a small part of the scaling problem.

I suggest to read Jason Hoffman’s slides on scaling Rails:

http://www.scalewithrails.com/downloads/ScaleWithRails-April2006.pdf

Or attend the workshop in Frankfurt (25.10/26.10)

A Scaling Strategy

1. Start simple!
2. 1 box, running everything, 1 FCGI listener.
3. Measure.
4. Adjust configuration, adding caching components, more listeners, more machines, etc.
5. Goto Step 3

This is a never ending, tedious process.

If you want to become the next Google ;-)




Rails Mailing List Quotes w.r.t. Performance Questions

- Don't worry, it's fast enough for us, so it will be fast enough for you.
- Look at our successful applications in production, with thousands of users.
- Premature optimization is evil.
- Just throw hardware at it.

These are not my answers!

- If your app can only handle 5 requests per second, you've done something wrong, which can't be rectified by JTHAI.
- And you probably want to know how to improve it.

On Performance Tuning

- Trying to improve performance without measuring is foolish.
- Trying to improve performance 1 week before you go live won't work.
- 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.
  (if you have only ten visitors per hour, performance is probably not a problem for you)
- 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.
- `railsbench` can help you with performance measuring and regression testing.

Performance Parameters

Latency
How fast can you answer a request?

Throughput
How many requests can you process per second?

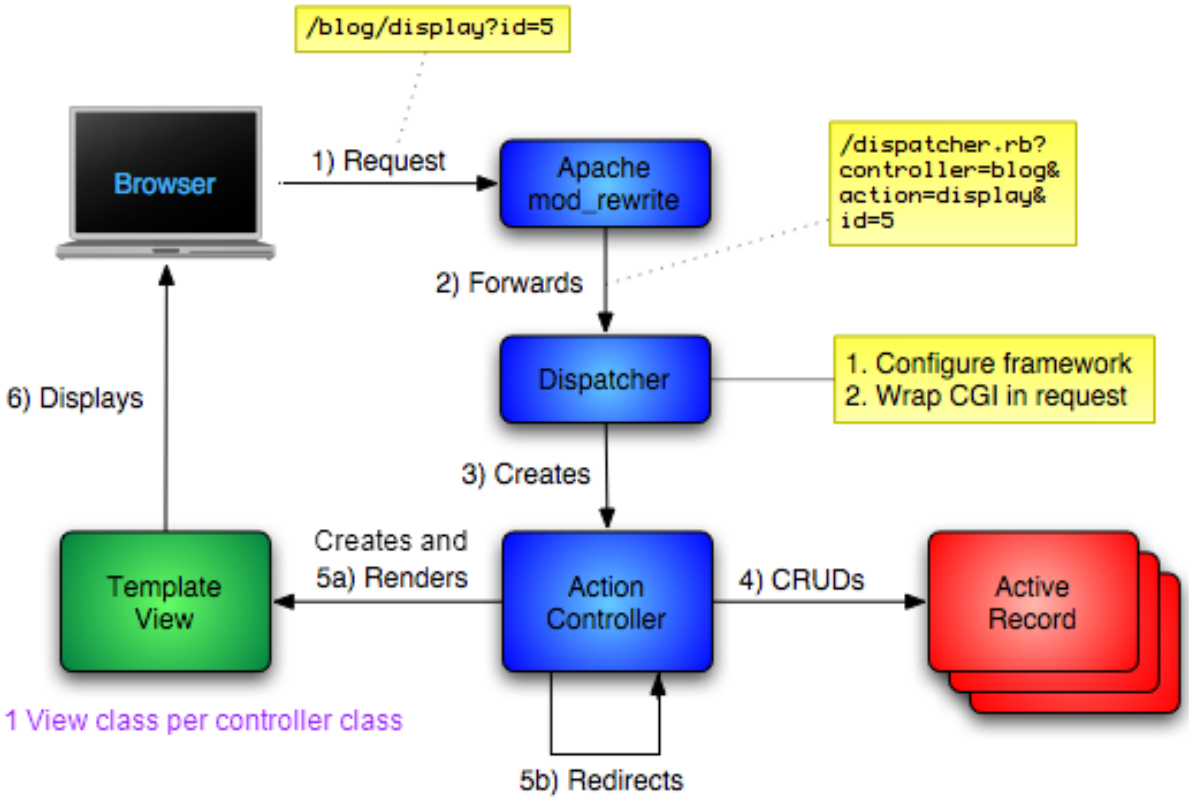
Utilization
Are your servers/components idle most of the time?

Cost Efficiency
Performance per unit cost

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

Standard deviation will tell you how reliable your data is.

Rails Request Cycle



Top Rails Performance Problems

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

- slow helper methods
- complicated routes
- using associations when you don't have to
- retrieving too much from DB
- slow session storage

DB performance is usually not a bottleneck!

Processing ActiveRecord objects after retrieval is the more expensive part.

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).




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

Demo

- Downloadable from http://rubyforge.org/projects/railsbench
- I recommend the README file. Web doc is somewhat out of date.

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 FCGI 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.




Use Strings as Fragment Cache Keys

Route generation can be excruciatingly slow.

Avoid using URL hashes as cache keys.

```
1 <% cache :action => "my_action", :user => session[:user] do %>
2  ...
3 <% end %>
```

This is *much* faster:

```
1 <% cache "#{@controller}/my_action/#{session[:user]}" do %>
2  ...
3 <% end %>
```

Also gives you more control over efficient expiry using regular expressions.




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

Don't use too many of them.

If you can combine several related filters into one, do it.

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).




Internal Render API

At one point in time DHH decided he liked hashes and symbols soo much, that he redesigned the render API.

```
1 render_text "Hello world!"
2 render_action "special"
```

became

```
1 render :text => "Hello world!"
2 render :action => "special"
```

In the process, rendering performance was impacted (esp. for partials).

Thanks to my intervention, the old methods are still available.

You can still use them.




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')
```

Consider:

```
1 <%= end_form_tag %>
```

vs.

```
1 </form>
```

How's the first one better?

textilize

Really, really, slow.

If you're textilizing database fields, consider caching them, some place.

- 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.

Same trick can be applied to all kinds of formatting jobs!

As pointed out by a member of the audience, there is/was a plugin which you could use.




link_to and url_for

Lo and behold: the highly touted superstars of Rails template helpers are the bad guys (w.r.t. performance)

```
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 ⇒ every single route specified in `config/routes.rb` must be examined.

A much more efficient way to write this is:

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

How much more efficient?




The Difference link_to makes

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

Notes:

- 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 code every `link_to` as above. just use this method on pages with a large number of links.




Ruby Performance Validator

Demo

- will tell you where the time is spent in your app
- railsbench package supports running defined benchmarks in RPVL
- need to buy a licence if you want to use it (30 day trial version available)
- more info: Software Verifaction Ltd.

ActiveRecord Issues

Accessing fields via association proxies is slow.

- use piggy backing for `has_one` relations.

Field values are retrieved from the DB as strings.

Type conversion happens on each access.

- cache converted values, if you use them several times during one request.

Sometimes you need only partial objects:

- use the `:select` parameter of `find` to retrieve ony fields used for display.




Ruby's interpreter is sloooow!

Ruby's Interpreter

Deeply rooted in 60's technology:

- no byte code, no JIT, interprets ASTs directly
- doesn't perform any code optimization at compile time
- GC performance is erratic (more on that later)

But it doesn't matter that much, because

> Rails ***scales easily*** (in principle, and because David said so ;-)),

The interesting questions are:

- how well can you make it scale?
- how hard is it?
- how much hardware do you need?




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




Warning: Don't run through your code and apply the optimizations to follow everywhere.

Remember: this is all about hot spots!

Micro Optimization

Avoid testing for `nil` using `.nil?`

```
1  logger.info "Prepare to take off" unless logger.nil?
```

vs.

```
1  logger.info "Prepare to take off" if logger
```

- Faster, smaller code, smaller AST, fewer method calls, less GC.
- Can't be applied if logger can take on value `false` (see here)

Don't use return unless you need to abort the current method or block

```
1    return expr
2  end
```

can be simplified to

```
1    expr
2  end
```

Note: methods and blocks implicitely return the value of the last evaluated expr.

Avoiding Repeated Hash Access

```
1  def submit_to_remote(name, value, options = {})
2    options[:with] ||= 'Form.serialize(this.form)'
3    options[:html] ||= {}
4    options[:html][:type] = 'button'
5    options[:html][:onclick] = "#{remote_function(options)}; return false;"
6    options[:html][:name] = name
7    options[:html][:value] = value
8    tag("input", options[:html], false)
9  end
```

This code is both simpler and faster:

```
1  def submit_to_remote(name, value, options = {})
2    options[:with] ||= 'Form.serialize(this.form)'
3    html = (options[:html] ||= {})
4    html[:type] = 'button'
5    html[:onclick] = "#{remote_function(options)}; return false;"
6    html[:name] = name
7    html[:value] = value
8    tag("input", html, false)
9  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 (i.e. doesn't slow down GC a lot) 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
```

Note: the cached value could be a query from the database

Example: guest user.

Coding Variable Caching Efficiently

Bad:

```
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
```

Better:

```
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:

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

Better:

```
1  VALID_FIND_OPTIONS = [
2      :conditions, :include, :joins, :limit,
3      :offset, :order, :select, :readonly, :group, :from ]
4
5  def validate_find_options (options)
6    options.assert_valid_keys(VALID_FIND_OPTIONS)
7  end
```

Faster and much easier to customize.




Using Local Variables Effectively

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]) && (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`

Don't be Stupid w.r.t. Logging

Don't forget to 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?
```





Ruby's GC collects and is garbage!




Back
Close

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)




Back
Close

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!

Note: improving GC performance increases *throughput*, not *latency*!

(unless you have a collection on each request)

Improving GC Performance

As a first attempt, I caused the addition of the possibilty to 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

Recommendation:

Patch Ruby's garbage collector!

Patching Ruby's Garbage Collector

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

You can then influence GC behavior by setting environment variables:

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

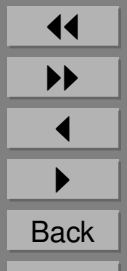
RUBY_HEAP_FREE_MIN
: number of free heap slots that should be available after GC (default 4096)

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



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




Which Database Package?

Rails uses an Application Database.

Contrast this with an Integration Database.

Choice of DB vendor is largely a metter of taste. Or external restrictions imposed on your project :-(

But: Mysql and Postgresql have best support in the Rails community. Core team uses both.




Database Performance

Mysql outperforms Postgres (even without query caching)

```
page   c1 real   c2 real c1 r/s c2 r/s c1 ms/r c2 ms/r c1/c2
 1:    2.51567  1.14067  397.5  876.7    2.52    1.14  2.21
 2:    3.33300  1.35933  300.0  735.7    3.33    1.36  2.45
 3:    2.78600  1.88567  358.9  530.3    2.79    1.89  1.48
 4:    4.27167  2.67200  234.1  374.3    4.27    2.67  1.60
 5:   11.91667  7.45300   83.9  134.2   11.92    7.45  1.60
 6:   23.40567 15.07300   42.7   66.3   23.41   15.07  1.55
 7:   22.91667 15.54667   43.6   64.3   22.92   15.55  1.47
 8:   10.68733  6.79167   93.6  147.2   10.69    6.79  1.57

GC:    c1 real  c2 real c1 #gc c2 #gc  c1 gc%  c2 gc% c1/c2
       2.44833  2.60367   14.0   18.0    2.99    5.01  0.78
```

Don't use Postgres for session storage!

Use memcached or a separate Mysql session DB using MyISAM tables:

No need for transaction support on session data!

Mysql Query Caching

Greatly speeds up performance of complex queries, if

- there's no index to use
- query involves complex joins

Mysql 5.0 implements views using query caching, so you'll get it anyway.

Query caching will slow down session retrieval slightly.

But the majority of web apps read more than write (to the DB).

For apps of this type, I recommend turning it on.




End

Thanks very much for your attention.

If you appreciated this session, you might consider buying my book, available around November 2006 from Addison Wesley, as part of the soon to be announced ”Professional Ruby” series.

If you’re doing commercial Rails apps, I’m also available for consulting.

Questions?




Shared Nothing Architecture

App servers rely on a (centralized) external resource to store application state.

The state is retrieved from and stored back to the external resource per request.

J2EE parlance: Stateless Session Bean




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&`

Compile Time Optimizations

These are usually taken for granted in modern interpreters:

- method inlining
- strength reduction
- constant propagation
- common subexpression elimination
- loop invariant detection
- loop unrolling

You don't have any of these in current Ruby interpreter.

$\Longrightarrow$

Performance aware programming can increase performance significantly!




Booleans and Conditionals

```
1  nil      || v   ⇝ v
2  false    || v   ⇝ v
3  other    || v   ⇝ other
4
5  nil      && v   ⇝ false
6  false    && v   ⇝ false
7  other    && v   ⇝ v
8
9  nil . nil ?     ⇝ true
10 other. nil ?    ⇝ false
11
12 if nil   then e1 else e2   ⇝ e2
13 if false then e1 else e2   ⇝ e2
14 if other then e1 else e2   ⇝ e1
```





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
```