PostgreSQL as a secret weapon for high-performance Ruby on Rails applications

Gleb Arshinov, Alexander Dymo
PGCon 2010

www.acunote.com
Acunote www.acunote.com
Online project management and Scrum software

~7000 customers
Hosted on Own Servers
Hosted on Customer's Servers
nginx + mongrel
PostgreSQL 8.4
Web Development Pains

- Performance
- Data integrity
- Developer productivity
Types of Web Apps

- Web apps range from Digg to a custom accounting system
- Your app is somewhere in between
Outline

- Rails as ORM
- Optimizing Rails With PostgreSQL
- PostgreSQL Limitations
- PostgreSQL Approaches
- Optimizing Database
Myth: not close to SQL
Close to SQL:

```ruby
author = Author.find(:first)
  select * from authors limit 1;
articles = author.articles
  select * from articles where author_id = 1

author_name = "Orwell"
author = Author.find(:all, :conditions => ["name = ?", author_name])
  select * from authors where name = "Orwell"
```

Drop down to SQL easily:

```ruby
author = Author.find_by_sql("select * from authors where authors.birthday > now()")
```
The conventional Rails way is not so bad:

<table>
<thead>
<tr>
<th>Tasks</th>
<th>Tags</th>
<th>Tasks_Tags</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>id</td>
<td>tag_id</td>
</tr>
<tr>
<td>serial</td>
<td>serial</td>
<td>integer</td>
</tr>
<tr>
<td>name</td>
<td>name</td>
<td>varchar</td>
</tr>
</tbody>
</table>

tasks = Task.find(:all, :include => :tags)

select * from tasks
select * from tags inner join tasks_tags
  on tags.id = tasks_tags.tag_id
where tasks_tags.task_id in (1,2,3,..)
But classical ORM problem exists:

```ruby
create table Task ( id serial not null, parent_id integer )

class Task < ActiveRecord::Base
  acts_as_tree
end
```

Uses N+1 database queries to load N nodes from the tree:

```
(root) select * from tasks where parent_id = nil

- 1  select * from tasks where parent_id = 1
    - 11  select * from tasks where parent_id = 11
    - 111 select * from tasks where parent_id = 111
    - 112 select * from tasks where parent_id = 112

- 12  select * from tasks where parent_id = 12

- 2   select * from tasks where parent_id = 2
    - 21 select * from tasks where parent_id = 21
```
create table Task (  
id serial not null,  
parent_id integer
)

class Task < ActiveRecord::Base  
  acts_as_tree
end

Should rather be:

select * from tasks  
  left outer join  
    (select id as parents_id, parent_id as parents_parent_id from tasks)  
  as parents on (tasks.parent_id = parents_id)  
  left outer join  
    (select id as parents_parents_id from tasks)  
  as parents_parents on (parents_parent_id = parents_parents_id)
Rails as ORM > Generating Queries

1. Task tree (3 levels) (+2 joins & 1 subselect)
2. Task tags (+2 subselects)
3. Task property counters (+4 subselects)
4. Last "timecell" values (+4 joins to get group-wise maximum)
   etc... - 12 joins and subselects
All this done in **1 query** in **under 60ms** even on EeePC!

Equivalent Ruby code took up to **8 sec!**
rev 834: Show past and future sprints in the list

--- application_helper.rb
+++ application_helper.rb
@@ -456,8 +456,8 @@
sprints = []
sprints.concat current_project.sprints(:present)
+sprints.concat current_project.sprints(:past)
+sprints.concat current_project.sprints(:future)
rev 834: Show past and future sprints in the list

--- application_helper.rb
+++ application_helper.rb
@@ -456,8 +456,8 @@
sprints = []
sprints.concat current_project.sprints(:present)
+sprints.concat current_project.sprints(:past)
+sprints.concat current_project.sprints(:future)

Sprint 20 x (1+5) (C) | Before 0.87 ± 0.01 | After 0.88 ± 0.01
rev 834: Show past and future sprints in the list

--- application_helper.rb
+++ application_helper.rb
@@ -456,8 +456,8 @@
sprints = []
sprints.concat current_project.sprints(:present)
+sprints.concat current_project.sprints(:past)
+sprints.concat current_project.sprints(:future)

--- empty_controller_test.rb
+++ empty_controller_test.rb
@@ -79,11 +79,12 @@
   "Sprint Load",
+   "Sprint Load",
+   "Sprint Load",
   "common/_nav_dialog",
   "Project Load",
Query tests to make sure we don't fall into the multiplying queries trap

```ruby
def test_queries
  queries = track_queries do
    get :index
  end
  assert_equal queries, ["Task Load", "Tag Load", "Event Create", "SQL"]
end
```
module ActiveSupport

class BufferedLogger

    attr_reader :tracked_queries

    def tracking=(val)
        @tracked_queries = []
        @tracking = val
    end

    def add_with_tracking(severity, message = nil, progname = nil, &block)
        @tracked_queries << $1 if @tracking && message =~ /3\[56\];1m(.* (Load|Create|Update|Destroy))\)/
        @tracked_queries << $1 if @tracking && message =~ /3\[56\];1m(SQL)\)/
        add_without_tracking(severity, message, progname, &block)
    end

    alias_method_chain :add, :tracking

end
end

class ActiveSupport::TestCase
    def track_queries(&block)
        RAILS_DEFAULT_LOGGER.tracking = true
        yield
        result = RAILS_DEFAULT_LOGGER.tracked_queries
        RAILS_DEFAULT_LOGGER.tracking = false
        result
    end
end
Use SQL DDL not Rails DSL
(unless targeting multiple RDBMS)

Schema in SQL vs Rails parlance

**Migration in SQL**

```sql
execute "
create table Foo (
  id serial not null,
  name varchar(20),
  bar_id integer,
  primary key (id),
  foreign key (bar_id)
    references Bar (id)
)"
```

**Migration in Rails parlance**

```ruby
create_table :foo do |
  t|
  t.string :name, :limit => 20
  t.references :bar
end

execute "alter table foo add
  foreign key (bar_id)
    references Bar (id)"
```
Rails as ORM > Data Integrity

- Myth - rails does not support constraints
- Actually not possible to assure data integrity in Rails
- Use constraints, rules, triggers and other database magic to protect data integrity, not to implement business logic
- FK constraints -- everything should be RESTRICT ON X SET NULL and CASCADE is a problem
Outline

- Rails as ORM
- **Optimizing Rails with PostgreSQL**
- PostgreSQL Limitations
- PostgreSQL Approaches
- Optimizing Database
• Good language, bad implementation

• Slow

• Unreliable

• Deal with it!
Compare to the database:

**PostgreSQL:**
```
explain analyze select sin(2+2) as hard_stuff;
```

---

Result (cost=0.00..0.01 rows=1 width=0)
(actual time=0.001..0.002 rows=1 loops=1)
Total runtime: 0.012 ms

**Ruby:**
```
Benchmark.realtime{ sin(2+2) }*1000
> 0.027 ms
```

13x!
• Has a reputation of being slow
• Actually even slower
• Most of the time spent in GC
• CPU bound
• Doesn't parallelize
Keep a set of benchmarks for most frequent user requests.
For example:

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Benchmark Burndown 120</td>
<td>0.70 ± 0.00</td>
</tr>
<tr>
<td>Benchmark Inc. Burndown 120</td>
<td>0.92 ± 0.01</td>
</tr>
<tr>
<td>Benchmark Sprint 20 x (1+5) (C)</td>
<td>0.45 ± 0.00</td>
</tr>
<tr>
<td>Benchmark Issues 100 (C)</td>
<td>0.34 ± 0.00</td>
</tr>
<tr>
<td>Benchmark Prediction 120</td>
<td>0.56 ± 0.00</td>
</tr>
<tr>
<td>Benchmark Progress 120</td>
<td>0.23 ± 0.00</td>
</tr>
<tr>
<td>Benchmark Sprint 20 x (1+5)</td>
<td>0.93 ± 0.00</td>
</tr>
<tr>
<td>Benchmark Timeline 5x100</td>
<td>0.11 ± 0.00</td>
</tr>
<tr>
<td>Benchmark Signup</td>
<td>0.77 ± 0.00</td>
</tr>
<tr>
<td>Benchmark Export</td>
<td>0.20 ± 0.00</td>
</tr>
<tr>
<td>Benchmark Move Here 20/120</td>
<td>0.89 ± 0.00</td>
</tr>
<tr>
<td>Benchmark Order By User</td>
<td>0.98 ± 0.00</td>
</tr>
<tr>
<td>Benchmark Set Field (EP)</td>
<td>0.21 ± 0.00</td>
</tr>
<tr>
<td>Benchmark Task Create + Tag</td>
<td>0.23 ± 0.00</td>
</tr>
<tr>
<td>... 30 more ...</td>
<td></td>
</tr>
</tbody>
</table>
Benchmarks as a special kind of tests:

class RenderingTest < ActionController::IntegrationTest

  def test_sprint_rendering
    login_with users(:user), "user"

    benchmark :title => "Sprint 20 x (1+5) (C)",
              :route => "projects/1/sprints/3/show",
              :assert_template => "tasks/index"
  end

end

Benchmark Sprint 20 x (1+5) (C)        0.45 ± 0.00
Benchmarks as a special kind of tests:

```ruby
def benchmark(options = {})
  (0..100).each do |i|
    GC.start
    pid = fork do
      begin
        out = File.open("values", "a")
        ActiveRecord::Base.transaction do
          elapsed_time = Benchmark::realtime do
            send(request_method, options[:route])
          end
          out.puts elapsed_time if i > 0
          out.close
        end
        raise CustomTransactionError
      rescue CustomTransactionError
        exit
      end
    end
    Process::waitpid pid
    ActiveRecord::Base.connection.reconnect!
  end
end
```

```ruby
values = File.read("values")
print "#{mean(values).to_02f} ± #{sigma(values).to_02f}\n"
```
Scalability is not a substitute for performance
Delegate as much work as possible to...
Delegate as much work as possible to the database!
Rails Performance > Attribute Preloading

The conventional Rails way:

<table>
<thead>
<tr>
<th>Tasks</th>
<th>Tags</th>
<th>Tasks_Tags</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>id</td>
<td>tag_id integer</td>
</tr>
<tr>
<td>serial</td>
<td>serial</td>
<td></td>
</tr>
<tr>
<td>name</td>
<td>name</td>
<td>task_id integer</td>
</tr>
<tr>
<td>varchar</td>
<td>varchar</td>
<td></td>
</tr>
</tbody>
</table>

```
tasks = Task.find(:all, :include => :tags)
> 0.058 sec
```

2 SQL queries
```
select * from tasks
select * from tags inner join tasks_tags
    on tags.id = tasks_tags.tag_id
where tasks_tags.task_id in (1,2,3,..)
```

Rals creates an object for each tag, that's not fast and takes memory
Faster with Postgres arrays:

<table>
<thead>
<tr>
<th>Tasks</th>
<th>Tags</th>
<th>Tasks_Tags</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>id</td>
<td>tag_id</td>
</tr>
<tr>
<td>serial</td>
<td>serial</td>
<td>integer</td>
</tr>
<tr>
<td>name</td>
<td>name</td>
<td>task_id</td>
</tr>
<tr>
<td>varchar</td>
<td>varchar</td>
<td>integer</td>
</tr>
</tbody>
</table>

```
tasks = Task.find(:all, :select => "*,
  array(select tags.name from tags inner join tasks_tags
  on (tags.id = tasks_tags.tag_id)
  where tasks_tasks.task_id=tasks.id) as tag_names
")
> 0.018 sec
```

1 SQL query
Rails doesn't have to create objects
>3x faster:

(was 0.058 sec, now 0.018 sec)
Faster with Postgres arrays:

<table>
<thead>
<tr>
<th>Tasks</th>
<th>Tags</th>
<th>Tasks_Tags</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>serial</td>
<td>tag_id integer</td>
</tr>
<tr>
<td>name</td>
<td>varchar</td>
<td>task_id integer</td>
</tr>
</tbody>
</table>

```
tasks = Task.find(:all, :select => "*, array(select tags.name from tags inner join tasks_tags on (tags.id = tasks_tags.tag_id) where tasks_tasks.task_id=tasks.id) as tag_names")
puts tasks.first.tag_names
> "\{Foo,Bar,Zee\}"
```
Simplified model for user privilege management:

<table>
<thead>
<tr>
<th>Users</th>
<th>Role</th>
<th>Roles_Users</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>id serial</td>
<td>user_id integer</td>
</tr>
<tr>
<td>name</td>
<td>name varchar</td>
<td>role_id integer</td>
</tr>
</tbody>
</table>

- `user_id` integer
- `role_id` integer
- `privilege1` boolean
- `privilege2` boolean
- ...

```ruby
user = User.find(:first, :include => :roles)
can_do_1 = user.roles.any { |role| role.privilege1? }
```
Simplified model for user privilege management:

<table>
<thead>
<tr>
<th>Users</th>
<th>Role</th>
<th>Roles_Users</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>id serial</td>
<td>user_id integer</td>
</tr>
<tr>
<td>name</td>
<td>name varchar</td>
<td>role_id integer</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>privilege1 boolean</td>
<td></td>
</tr>
<tr>
<td></td>
<td>privilege2 boolean</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

user = User.find(:first, :include => :roles)

can_do_1 = user.roles.any { |role| role.privilege1? }

Where is the problem?
- 2 SQL queries
- Rails has to create objects for each role
- Ruby iterates over the roles array
Same in SQL:

<table>
<thead>
<tr>
<th>Users</th>
<th>Role</th>
<th>Roles_Users</th>
</tr>
</thead>
<tbody>
<tr>
<td>id serial</td>
<td>id serial</td>
<td>user_id integer</td>
</tr>
<tr>
<td>name varchar</td>
<td>name varchar</td>
<td>role_id integer</td>
</tr>
<tr>
<td></td>
<td>privilege1 boolean</td>
<td></td>
</tr>
<tr>
<td></td>
<td>privilege2 boolean</td>
<td></td>
</tr>
</tbody>
</table>

```ruby
user = User.find(:first, :select => "*", :joins => "
inner join
  (select user_id, bool_or(privilege1) as privilege1
  from roles_users
  inner join roles
  on (roles.id = roles_users.role_id)
  group by user_id)
as roles_users
  on (users.id = roles_users.user_id)
"
)

can_do_1 = ActiveRecord::ConnectionAdapters::Column.value_to_boolean(user.privilege1)
```
Rails Performance > Access Control

Optimization Effect:

can_do_1 = user.roles.any { |role| role.privilege1? }

> 2.1 sec

can_do_1 = ActiveRecord::ConnectionAdapters::Column.
  value_to_boolean(user.privilege1)

> 64 msec !!!
Perform calculations and aggregations on large datasets in SQL:

real life example:
600 000 data rows, 3-dimensional OLAP cube, slicing and aggregation:

Ruby: ~1 Gb RAM, ~90 sec

SQL: up to 5 sec
Outline

- Rails as ORM
- Rails Performance and PostgreSQL
- **PostgreSQL Experience**
- PostgreSQL Approaches
- Optimizing Database
Postgres Experience > Good Things

Good things about Postgres:

- SQL standard compliance (and useful non-standard addons)
- good documentation
- sustainable development
- good optimizer and EXPLAIN ANALYZE
- a lot of things can be expressed in pure SQLConstraints
- referential integrity
- deadlock detection
Good things that were introduced recently:

- replication (warm and hot standby, streaming replication)
- windowing functions
- recursive queries
- ordering for aggregates
And now... limitations
Pagination VS Subselects:

```
select *
    (select count(*) from attachments
        where issue_id = issues.id) as num_attachments
from issues
limit 100 offset 0;
```

Limit (cost=0.00..831.22 rows=100 width=143) (actual time=0.050..1.242 rows=100 loops=1)
  ->  Seq Scan on issues  (cost=0.00..2509172.92 rows=301866 width=143)
       (actual time=0.049..1.119 rows=100 loops=1)
SubPlan
  ->  Aggregate  (cost=8.27..8.28 rows=1 width=0)
       (actual time=0.006..0.006 rows=1 loops=100)
      ->  Index Scan using attachments_issue_id_idx on attachments
           (cost=0.00..8.27 rows=1 width=0) (actual time=0.004..0.004 rows=0 loops=100)
      Index Cond: (issue_id = $0)
Total runtime: 1.383 ms
Pagination VS Subselects:

```sql
select *,
    (select count(*) from attachments
        where issue_id = issues.id) as num_attachments
from issues
limit 100 offset 100;
```

Limit (cost=831.22..1662.44 rows=100 width=143) (actual time=1.070..7.927 rows=100 loops=1)
   ->  Seq Scan on issues  (cost=0.00..2509172.92 rows=301866 width=143)
        (actual time=0.039..7.763 rows=200 loops=1)
          SubPlan
             ->  Aggregate  (cost=8.27..8.28 rows=1 width=0)
                   (actual time=0.034..0.034 rows=1 loops=200)
              ->  Index Scan using attachments_issue_id_idx on attachments
                   (cost=0.00..8.27 rows=1 width=0) (actual time=0.032..0.032 rows=0 loops=200)
                   Index Cond: (issue_id = $0)
Total runtime: 8.065 ms
Be careful with subselects: they are executed \texttt{limit} + \texttt{offset} times!

Use joins to overcome the limitation.
Use *any(array ())* instead of *in()*
to force subselect and avoid join

```
explain analyze select * from issues where id in (select issue_id from tags_issues);
```

**QUERY PLAN**

```
Merge IN Join  (actual time=0.096..576.704 rows=55363 loops=1)
  Merge Cond: (issues.id = tags_issues.issue_id)
  ->  Index Scan using issues_pkey on issues  (actual time=0.027..270.557 rows=229991 loops=1)
  ->  Index Scan using tags_issues_issue_id_key on tags_issues  (actual time=0.051..73.903 rows=70052 loops=1)
Total runtime: 605.274 ms
```

```
explain analyze select * from issues where id = any( array( (select issue_id from tags_issues) ) );
```

**QUERY PLAN**

```
Bitmap Heap Scan on issues  (actual time=247.358..297.932 rows=55363 loops=1)
  Recheck Cond: (id = ANY ($0))
InitPlan
  ->  Seq Scan on tags_issues  (actual time=0.017..51.291 rows=70052 loops=1)
  ->  Bitmap Index Scan on issues_pkey  (actual time=246.589..246.589 rows=70052 loops=1)
    Index Cond: (id = ANY ($0))
Total runtime: 325.205 ms
```

*2x!*
select * from
    (select *, (select min(split_date) from tasks
        where tasks.issue_id = issues.id) as split_date
        from issues where org_id = 2) as issues,
    (select generate_series(0,10) + date '2010-01-01' as date) as dates

QUERY PLAN
----------------------------------------------------------------------------------------
| Nested Loop  (actual time=2.581..2525.798 rows=149666 loops=1) |
| --> Result  (actual time=0.007..0.063 rows=11 loops=1) |
| --> Bitmap Heap Scan on issues  (actual time=2.697..47.756 rows=13606 loops=11) |
|    Recheck Cond: (public.issues.org_id = 2) |
|    --> Bitmap Index Scan on issues_org_id_idx (actual time=1.859..1.859 rows=13607 loops=11) |
    Index Cond: (public.issues.org_id = 2)

SubPlan 1
    --> Aggregate  (actual time=0.010..0.010 rows=1 loops=149666) |
    --> Index Scan using tasks_issue_id_key on tasks (actual time=0.006..0.008 rows=1 loops=149666) |
    Index Cond: (issue_id = $0)

Total runtime: 2608.891 ms
select * from
  (select * from issues
   left outer join (select issue_id, min(split_date) as split_date from tasks
     where org_id = 2 group by issue_id
   ) tasks
   on (tasks.issue_id = issues.id) where org_id = 2) as issues,
  (select generate_series(0,10) + date '2010-01-01' as date) as dates

QUERY PLAN

---------------------------------------------------------------
| Nested Loop (actual time=174.706..831.263 rows=149666 loops=1) |
|   -> Result (actual time=0.006..0.055 rows=11 loops=1) |
|     -> Merge Left Join (actual time=15.885..60.496 rows=13606 loops=11) |
|       Merge Cond: (public.issues.id = public.tasks.issue_id) |
|         -> Sort (actual time=8.048..18.068 rows=13606 loops=11) |
|           -> Bitmap Heap Scan on issues (actual time=2.7..55 rows=13606 loops=1) |
|             Recheck Cond: (org_id = 2) |
|               -> Bitmap Index Scan on issues_org_id_idx (actual time=1.912..1.) |
|                 Index Cond: (org_id = 2) |
|     -> Sort (actual time=7.834..15.519 rows=13202 loops=11) |
|       -> HashAggregate (actual time=62.150..71.767 rows=13202 loops=1) |
|         -> Bitmap Heap Scan on tasks (actual time=3.177..41.700 rows=18) |
|           Recheck Cond: (org_id = 2) |
|             -> Bitmap Index Scan on tasks_org_id_idx (actual time=2.50) |
|               Index Cond: (org_id = 2) |

Total runtime: 906.146 ms
Outline

• Rails as ORM
• Rails Performance and PostgreSQL
• PostgreSQL Limitations
• PostgreSQL Approaches
• Optimizing Database
Postgres Approaches

- Benchmarking/performance
- Distrust vendors
- Sane appreciation of commodity hardware
- Culture of operations
- Release management
Outline

- Rails as ORM
- Rails Performance and PostgreSQL
- PostgreSQL Experience
- PostgreSQL Approaches
- **Optimizing Database**
How to optimize PostgreSQL:

explain analyze
explain analyze
explain analyze
...

Optimize Database > Basics
EXPLAIN ANALYZE explains everything, but...

... run it also for the "cold" database state!

Example: complex query which works on 230 000 rows and does 9 subselects / joins:

cold state: 28 sec, hot state: 2.42 sec

Database server restart doesn't help

Need to clear disk cache:

```
sudo echo 3 | sudo tee /proc/sys/vm/drop_caches
```

(Linux)
You're competing for memory cache on a shared server:

1. two databases with equal load share the cache
You're competing for memory cache on a shared server:

2. one of the databases gets more load and wins the cache
As a result, your database can always be in a "cold" state and you **read data from disk, not from memory!**

complex query which works on 230 000 rows and does 9 subselects / joins:
from disk: **28** sec, from memory: **2.42** sec

**Solutions:**
- optimize for IO/cold state
  - `sudo echo 3 | sudo tee /proc/sys/vm/drop_caches`
- push down SQL conditions
# How much memory we have to cache the database, RAM_FOR_DATABASE * 3/4

effective_cache_size = <%= ram_for_database.to_i * 3/4 %> MB

# Shared memory to hold data in RAM, RAM_FOR_DATABASE/4

shared_buffers = <%= ram_for_database.to_i / 3 %> MB

# Work memory for queries (RAM_FOR_DATABASE/max_connections) ROUND DOWN 2^x

work_mem = <%= 2**(Math.log(ram_for_database.to_i / expected_max_active_connections.to_i)/Math.log(2)).floor %> MB

# Memory for vacuum, autovacuum, index creation, RAM/16 ROUND DOWN 2^x

maintenance_work_mem = <%= 2**(Math.log(ram_for_database.to_i / 16)/Math.log(2)).floor %> MB

# To ensure that we don't lose data, always fsync after commit

synchronous_commit = on

# Size of WAL on disk, recommended setting: 16

checkpoint_segments = 16

# WAL memory buffer

wal_buffers = 8 MB

# Ensure autovacuum is always turned on

autovacuum = on

# Set the number of concurrent disk I/O operations that PostgreSQL expects can be executed simultaneously.

effective_io_concurrency = 4
Effect from better configuration:

<table>
<thead>
<tr>
<th>Query</th>
<th>Default Settings</th>
<th>Custom Settings</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aggregation on a large dataset</td>
<td>8205 ms</td>
<td>7685 ms</td>
<td>6%</td>
</tr>
<tr>
<td>Query with complex joins and subselects</td>
<td>229 ms</td>
<td>143 ms</td>
<td>38%</td>
</tr>
</tbody>
</table>
Thanks!

Rails performance articles and more:
http://blog.pluron.com

Gleb Arshinov
CEO
gleb@pluron.com

Alexander Dymo
Director of Engineering
adymo@pluron.com