Optimize Database Queries

Bogdan Gusiev

June 2017

Bogdan Gusiev:

- Working for startups for X years
- Contributing to Ruby on Rails X years

2011: Started

MassAssignmentSecurity: add ability to specify your own sanitizer

```
Added an ability to specify your own behavior on mass assingment
protection, controlled by option:
ActiveModel::MassAssignmentSecurity.mass_assignment_sanitizer
```

master v5.1.1 ... v3.2.0.rc1

bogdan committed on May 26, 2011 1 p

2017: Won an argument with DHH

dhh commented 15 days ago

Owner

I would bake #cache_version into ActiveModel and then perhaps provide a top-level setting for making it return nil for backwards compatibility.

Let's see how far we can get with all this. If we can arrive at a place where we have full backwards compatibility, full support for multiget etc, then maybe we won't need a new vcache method.

...

Why is SQL queries worth to be optimized?

Optimize Your App

- Web Server
- Application Container (unicorn etc)
- Rack/Rails
- Your code

90% it is all yours

Top of the mind

- Use a different database
- Buy a new hardware
- Use a magic setting
- Optimize Slow Parts

What takes time in the app

Category	Segment	% Time	Avg calls (per txn)	Avg time (ms)
Controller	ApiV2::OriginsController#create	51.9	1.0	29
Database	ActiveRecord other	20.2	2.02	11.3
Database	ActiveRecord Purchase find	8.0	4.66	4.45
Database	ActiveRecord Person find	3.3	1.73	1.83
Database	ActiveRecord EventCategory find	2.3	2.71	1.28
Database	ActiveRecord Account find	1.8	2.0	1.02
Middleware	ActionDispatch::Routing::RouteSet#call	1.6	1.0	0.887
Database	ActiveRecord Origin find	1.2	0.892	0.653

Show all segments →

What takes time in the app

Category	Segment	% Time	Avg calls (per txn)	Avg time (ms)
View	shared/_referrals_table.html.haml Partial	57.0	0.781	443
Database	ActiveRecord Referral find	10.9	5.77	84.6
Database	ActiveRecord Coupon find	3.8	47.2	29.1
Database	ActiveRecord CustomAttribute find	3.6	27.7	28.2
View	approvable_referrals/index.html.haml Template	3.1	0.987	23.8
Database	ActiveRecord User other	2.8	3.14	21.6
Database	ActiveRecord IncentiveOutcome find	2.2	26.7	16.8
Controller	CustomerService::ApprovableReferralsController#index	2.2	1.0	17.2
View	shared/_user_menu.html.haml Partial	1.6	0.987	12.6
Database	ActiveRecord Reward find	1.4	11.4	10.6
Database	ActiveRecord Visitor find	1.4	15.1	10.8
Custom	Authlogic/find	1.0	1.0	7.43

Optimize DB queries is important

Methods

- Avoid queries
- Make queries faster
- Merge queries (several simple into one complex)
- Use Cache

Types of queries

- Instant (< 10ms)
  - time spent mainly on networking IO and FS read
  - There are many of them
- Slow (> 50ms)
  - time spent on aggregating or searching the data
  - Slow time comes from joins or aggregation or dynamic conditions

Avoid Instant Queries

```
Campaign.has_many :offers
Offer.belongs_to :campaign
```

```
>> campaign.offers.first.campaign
SELECT  `offers`.* FROM `offers` WHERE `offers`.`campaign_id` =
SELECT  `campaigns`.* FROM `campaigns` WHERE `campaigns`.`id` =
```

Why does it happen?

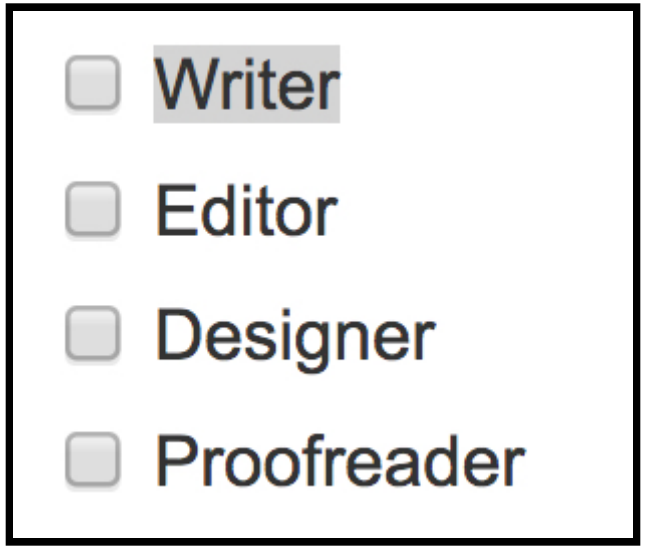
```
= render partial: "offers/offer", collection: campaign.offers
```

```
-# app/views/offers/_offer.html.haml
%a{href: campaign_offer_path(offer.campaign, offer)}
  = offer.name
```

Avoid Queries is Possible

```
Campaign.has_many :offers, inverse_of: :campaign
Offer.belongs_to :campaign, inverse_of: :offers
```

```ruby
class UserRole < AR::Base
  belongs_to :user
  validates_uniqueness_of :role_id, scope: [:user_id]
end
```



Use unique index instead

Rails built-in avoid queries methods:

- inverse_of
- counter_cache
- has_many :through
- preload/includes

Avoid Queries

```
Coupon.belongs_to :purchase, counter_cache: true
class Purhcase < AR::Base
  has_many :coupons, class_name: "DiscountCoupon"

  def coupon_codes
    coupons.map(&:code)
  end
end
```

```
def coupon_codes
  coupon_count > 0 ? coupons.map(&:code) : []
end
```

```
Comment.belongs_to :post, counter_cache: true
```

```
- if @post.comments.any?
  Comments (#{@post.comments.count}):
  - @post.comments.each do |comment|
    = render partial: "posts/comment", object: comment
- else
  No Comments Made yet
```

How many queries
in the worst case scenario?

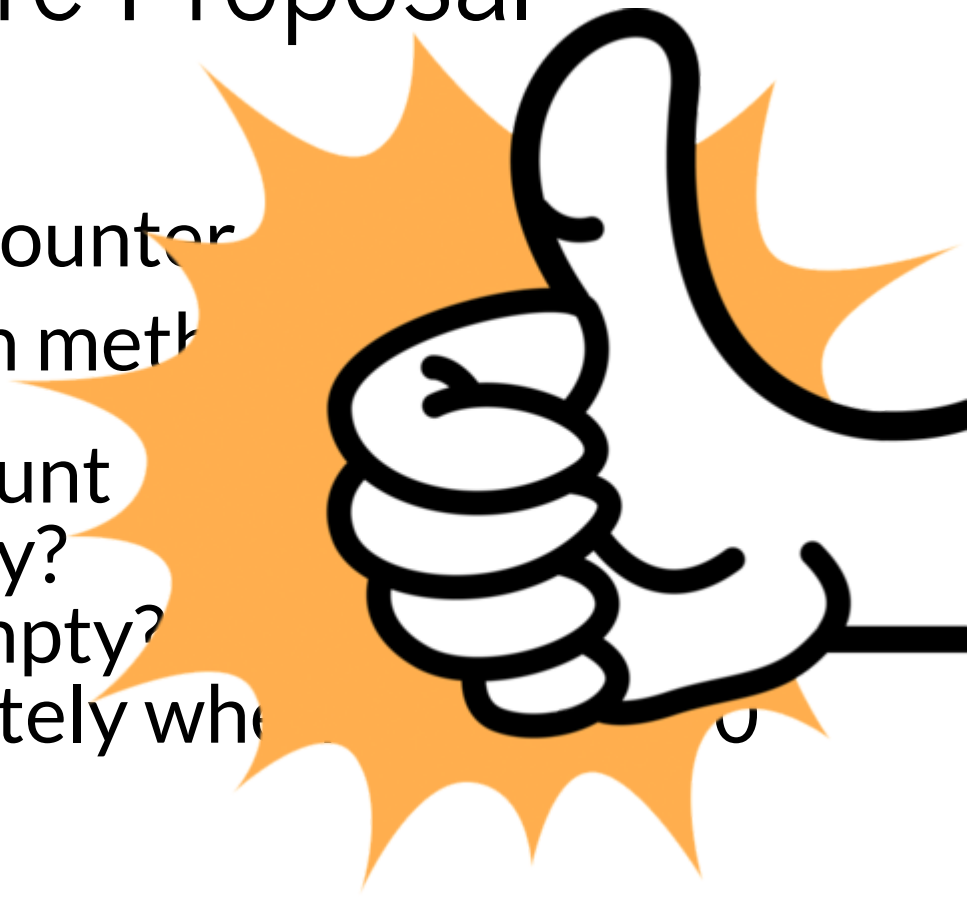
3 Queries

Rails Feature Proposal

Make use of `counter
in Association meth

- count
- any?
- empty?

Avoid loading completely whe



Cache Columns

as a way to avoid queries

Cache immutable data

```
create_table :comments do |t|
  t.integer :post_id
  t.integer :author_id
  t.text :body
end
```

#body is mutable but #post and #author are not

```
Comment.has_one :feed_activity, as: :target
FeedActivity.belongs_to :target, polymorphic: true
FeedActivity.before_validation do
  self.forum = comment.post.forum
  self.author = comment.author
}
```

Cache Columns

Pros

- Cheap to make
- Cheap to maintain for immutable data

Cons

- Expensive to maintain for mutable data
- Takes disc space

Speedup queries

```
add_index :users, :email, unique: true
add_index :products, :category_id
```

- Become actual after 100_000 records
- Indexes take disk space
- Foreign Key index is almost always a good idea

Index Usage Guide

Index by one column will help you make the related query fast

```
SELECT * FROM users WHERE email = ?
```

Multi-column index can be used to query by 1st column, 1st & 2nd column etc

```
add_index :users_roles, [:user_id, :role_id], unique: true
add_index :users_roles, :role_id
```

Some Math

If I have a table with N columns, how many indexes do I need to cover 100% of possible where conditions?

$$2^{n}$$

How about partial index usage optimization?

$$2^{n-1}$$

Indexes are not free

Talkable DB Space Usage in GB

Name	Data	Index	Index
origins	28.0	70.5	72%
short_urls	12.2	63.6	84%
activities	16.0	43.8	73%
visitors	21.3	25.4	54%
offers	9.4	24.7	72%
people	6.0	17.7	75%
items	16.0	10.8	40%
custom_attributes	6.9	9.6	58%
coupons	3.3	8.7	73%
customer_emails	3.6	6.2	63%
split_test_impressions	3.2	4.5	58%
visitor_offers	2.1	3.7	64%
previous_customers	5.2	3.4	40%
incentive_outcomes	1.2	2.8	71%
Total	**134.3**	**295.3**	**69%**

Index Usage Optimization

```
SELECT * FROM posts p
WHERE p.forum_id = ? AND p.user_id = ? AND p.created_at = ?
```

- Index A: forum_id, created_at
- Index B: user_id

But the DB may disagree

```
SELECT * FROM posts p USE INDEX(index_b)
WHERE p.forum_id = ? AND p.user_id = ? AND p.created_at = ?
```

WHERE conditions are applied one after another

Apply indexes to columns that would

cut the most records

from the result in where condition

Indexes are the best to speed up your queries

But they are not the only one

Cache Tables

```
Product.belongs_to :store
Product.belongs_to :category
Store.has_many :categories, through: :products
```

```
create_table :stores_categories do |t|
  t.integer :store_id
  t.integer :category_id
end
```

Faster join because of low number of records

Extra Conditions

```
SELECT * FROM offers
WHERE campaign_id = {campaign.id}
```

```
AND site_id = {campaign.site.id}
```

```
AND created_at
  BETWEEN {campaign.created_at} AND {campaign.deactivated_at}
```

Avoid Join

```
Purchase.has_many :discount_coupons
DiscountCoupon.belongs_to :purchase, counter_cache: true

Purchase.scope :shipped_for_free, -> {
 joins(:discount_coupons).
   where(discount_coupons: {free_shipping: true}).
}
```

```
where("discount_coupons.coupons_count > 0")
```

DB is smart on doing JOINS

```
Purchase.
  where(created_at: 30.days.ago..Time.now).
  shipped_for_free
```

```
SELECT * FROM purchases
USE INDEX (index_purchases_created_at)
INNER JOIN coupons ON purchases.coupon_id = coupons.id
WHERE
  purchases.coupons_count > 0 AND
  (purchases.created_at BETWEEN ? AND ?) AND
  coupons.free_shipping
```

created_at -> coupons_count ->
join coupons -> free_shipping

Query execution mystery

```
SELECT * FROM members LEFT JOIN projects ON
members.project_id = projects.id
WHERE projects.organization_id = ?

SELECT * FROM members WHERE project_id in (
  SELECT * FROM projects WHERE organization_id = ?)

-- pseudocode
ids = SELECT id FROM projects WHERE organization_id = ?
SELECT * FROM members WHERE project_id in (ids)
```

Queries can be equivalent for your data

but they are not equivalent for any data

Know your data is a key to performance

OR operation

```
Product.where(
  category_id: Category.where(archived: false).ids
)
```

```
SELECT id FROM categories WHERE archived = false
SELECT * FROM product WHERE category_id in (1,2,3,4,5 ....)
```

```
SELECT * FROM product WHERE category_id = 1 UNION
SELECT * FROM product WHERE category_id = 2 UNION
SELECT * FROM product WHERE category_id = 3 UNION ...
```

"OR" operator is bad for your performance

Beware of "OR" in the hidden form

”Nested queries are evil”

(c) My University Professor 2006

Nested queries are good when

- They is no closures
  - Result is the same for each query row
- The result set has not more than 100 records
- They let you define a stricter query execution plan

In most cases, two simple queries will be faster than one complex

if nested query didn’t work

Query Execution Plan

- PostgreSQL `explain` statement is amazing
- MySQL `explain` statement is useless

PostgreSQL Explain

```
tpcc=# explain select (
  select count(*) from item where i_id = empty.i_id
) from empty;

 QUERY PLAN
-----------------------------------------------------------------
Seq Scan on empty (cost=0.00..19933.22 rows=2400 width=4)
  SubPlan
    -> Aggregate (cost=8.28..8.29 rows=1 width=0)
      -> Index Scan using pk_item on item (cost=0.00..8.28 rows=1
          Index Cond: (i_id = $0)
```

wiki.postgresql.org/wiki/File:Explaining_EXPLAIN.pdf

Real Example:

Leaderboard

Rank	Name	Comments
1st	Jonh	97
2nd	Rick	95
3rd	Aaron	60
95th	You	20

Manually controlling the query execution plan

```
find_person_by_id(add_ranking(get_names_and_counts), ?)
```

```
SELECT people.*, leaderboard.* FROM (
  SELECT leaders.*, CAST(@rnk:=@rnk+1 AS UNSIGNED) leaderboard_ra
  FROM (
    SELECT  `comments`.`person_id` person_id,
      COUNT(*) AS `leaderboard_count`
    FROM `comments`
    WHERE `comments`.`created_at` BETWEEN ? AND ?
    GROUP BY `comments`.`person_id`
    ORDER BY `leaderboard_count` DESC
    LIMIT ?
  ) AS leaders, (SELECT @rnk:=0) AS ranking
) AS leaderboard left join people on people.id = leaderboard.pers
WHERE people.id = ?
ORDER BY leaderboard_rank
```

Supercomplex SQL query can be fast if you control the execution plan

```
SELECT people.*, leaderboard.* FROM (
  SELECT leaders.*, CAST(@rnk:=@rnk+1 AS UNSIGNED) leaderboard_ra
    SELECT `offers`.`person_id` person_id, COUNT(*) AS `leaderboa
      INNER JOIN `offers` ON `offers`.`id` = `referrals`.`offer_i
      INNER JOIN `origins` ON `origins`.`id` = `referrals`.`refer
      WHERE `referrals`.`site_id` = %SITE_ID% AND `referrals`.`st
        (`offers`.`person_id` IS NOT NULL) AND `origins`.`event_c
        (`referrals`.`status` != 'blocked') AND `referrals`.`camp
        (`referrals`.`created_at` BETWEEN %STARTED_AT% AND %FINIS
      GROUP BY `offers`.`person_id`
      ORDER BY `leaderboard_count` DESC, `leaderboard_subtotal` D
  ) AS leaders, (SELECT @rnk:=0) AS ranking
) AS leaderboard
LEFT JOIN people on people.id = leaderboard.person_id
WHERE `people`.`site_id` = %SITE_ID% AND `people`.`id` = {person_
ORDER BY leaderboard.rank
```

Use Cache

- Key-Value DB
- HTTP ETAG

Key-Value DB

```
class Forum
  def comments_leaderboard(period)
    Rails.cache.fetch(cache_key(period)) do
      all_that_heavy_sql
    end
  end
end
```

```
def cache_key(period)
  ["comments_leaderboard", period,
    comments.where(created_at: period).
      order(:created_at).last.id]
end
```

~~Expiration by key~~

Rails 5.2

Expiration by version

PR #29092:

Recyclable Cache Keys

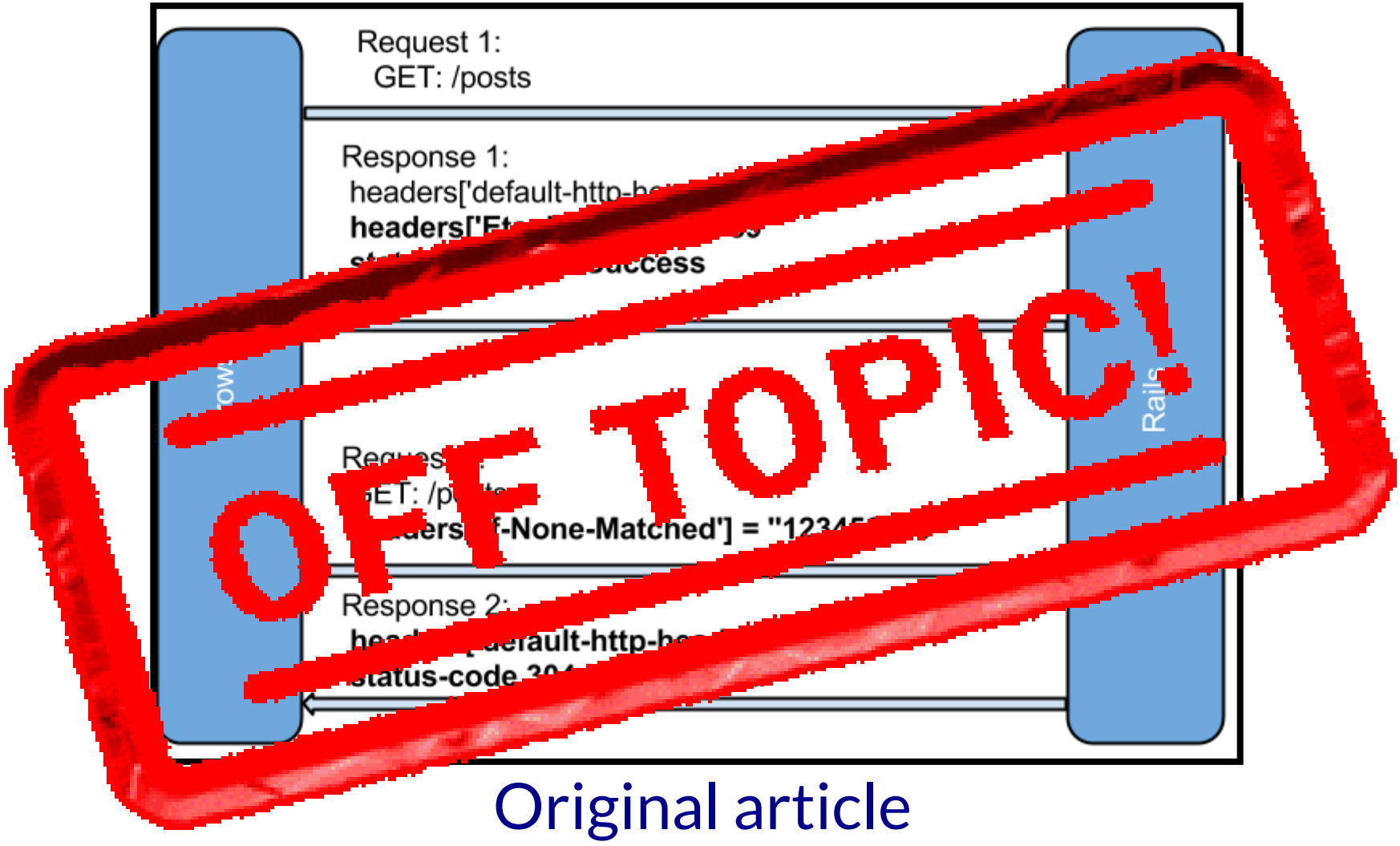
Expiration Version Feature

```
Rails.cache.fetch(["post_preview", post]) do
  heavy_lifting
end
```

```
class Post
  def cache_key
    "post/#{id}"
  end
  def cac
    _at
  d
```

Etag Caching

304 Not Modified



Original article

Can’t optimize the query?

Cache at higher level!

High Level Caching is tough

- Cache Hit
- Invalidation
- Invalidation Maintenance
- Invalidation Bugs
  - Description: "Sometimes..."

Optimizing Steps

1. Many Queries -> Merge or Avoid Queries
2. Slow Queries -> Optimize Queries
    1. Figure Out Domain Specifics
    2. Use Indexes Effectively
    3. Control Disk Space
    4. Control Query Execution Plan
3. Use High Level Cache

Thank You

: