
counter_culture

Turbo-charged counter caches for your Rails app. Huge improvements over the Rails standard counter caches:

- Updates counter cache when values change, not just when creating and destroying
- Supports counter caches through multiple levels of relations
- Supports dynamic column names, making it possible to split up the counter cache for different types of objects
- Can keep a running count, or a running total

Tested against Ruby 2.6, 2.7, 3.0, 3.1, 3.2 and 3.3 and against the latest patch releases of Rails 5.2, 6.0, 6.1, 7.0 and 7.1.

Please note that – unlike Rails’ built-in counter-caches – counter_culture does not currently change the behavior of the `.size` method on ActiveRecord associations. If you want to avoid a database query and read the cached value, please use the attribute name containing the counter cache directly.

```
1 product.categories.size # => will lead to a SELECT COUNT(*) query
2 product.categories_count # => will use counter cache without query
```

Installation

Add counter_culture to your Gemfile:

```
1 gem 'counter_culture', '~> 3.2'
```

Then run `bundle install`

Database Schema

You must create the necessary columns for all counter caches. You can use counter_culture’s generator to create a skeleton migration:

```
1 rails generate counter_culture Category products_count
```

Which will generate a migration with code like the following:

```
1 add_column :categories, :products_count, :integer, null: false, default
  : 0
```

Note that the column must be **NOT NULL** and have a default of zero for this gem to work correctly.

If you are adding counter caches to existing data, you must add code to manually populate their values to the generated migration.

Usage

Simple counter-cache

Has many association

```
1 class Product < ActiveRecord::Base
2   belongs_to :category
3   counter_cache :category
4 end
5
6 class Category < ActiveRecord::Base
7   has_many :products
8 end
```

Now, the `Category` model will keep an up-to-date counter-cache in the `products_count` column of the `categories` table.

Many to many association

```
1 class User < ActiveRecord::Base
2   has_many :group_memberships
3   has_many :groups, through: :group_memberships
4 end
5
6 class Group < ActiveRecord::Base
7   has_many :group_memberships
8   has_many :members, through: :group_memberships, class: "User"
9 end
10
11 class GroupMembership < ActiveRecord::Base
12   belongs_to :group
13   belongs_to :member, class: "User"
14   counter_cache :group, column_name: "members_count"
15   # If you'd like to also touch the group when `members_count` is
16   #   updated
17   # counter_cache :group, column_name: "members_count", touch: true
18 end
```

Now, the `Group` model will have an up to date count of its members in the `members_count` column

Multi-level counter-cache

```
1 class Product < ActiveRecord::Base
2   belongs_to :sub_category
3   counter_culture [:sub_category, :category]
4 end
5
6 class SubCategory < ActiveRecord::Base
7   has_many :products
8   belongs_to :category
9 end
10
11 class Category < ActiveRecord::Base
12   has_many :sub_categories
13 end
```

Now, the `Category` model will keep an up-to-date counter-cache in the `products_count` column of the `categories` table. This will work with any number of levels.

If you want to have a counter-cache for each level of your hierarchy, then you must add a separate counter cache for each level. In the above example, if you wanted a count of products for each category and sub_category you would change the `Product` class to:

```
1 class Product < ActiveRecord::Base
2   belongs_to :sub_category
3   counter_culture [:sub_category, :category]
4   counter_culture [:sub_category]
5 end
```

Customizing the column name

```
1 class Product < ActiveRecord::Base
2   belongs_to :category
3   counter_culture :category, column_name: "products_counter_cache"
4 end
5
6 class Category < ActiveRecord::Base
7   has_many :products
8 end
```

Now, the `Category` model will keep an up-to-date counter-cache in the `products_counter_cache` column of the `categories` table. This will also work with multi-level counter caches.

Dynamic column name

```
1 class Product < ActiveRecord::Base
2   belongs_to :category
```

```
3   counter_culture :category, column_name: proc {|model| "#{model.
    product_type}_count" }
4   # attribute product_type may be one of ['awesome', 'sucky']
5 end
6
7 class Category < ActiveRecord::Base
8   has_many :products
9 end
```

Delta Magnitude

```
1 class Product < ActiveRecord::Base
2   belongs_to :category
3   counter_culture :category, column_name: :weight, delta_magnitude:
    proc {|model| model.product_type == 'awesome' ? 2 : 1 }
4 end
5
6 class Category < ActiveRecord::Base
7   has_many :products
8 end
```

Now the `Category` model will keep the `weight` column up to date: `awesome` products will affect it by a magnitude of 2, others by a magnitude of 1.

You can also use a static multiplier as the `delta_magnitude`:

```
1 class Product < ActiveRecord::Base
2   belongs_to :category
3   counter_culture :category, column_name: :weight, delta_magnitude: 3
4 end
5
6 class Category < ActiveRecord::Base
7   has_many :products
8 end
```

Now adding a `Product` will increase the `weight` column in its `Category` by 3; deleting it will decrease it by 3.

Conditional counter cache

```
1 class Product < ActiveRecord::Base
2   belongs_to :category
3   counter_culture :category, column_name: proc {|model| model.special?
    ? 'special_count' : nil }
4 end
5
```

```
6 class Category < ActiveRecord::Base
7   has_many :products
8 end
```

Now, the `Category` model will keep the counter cache in `special_count` up-to-date. Only products where `special?` returns true will affect the `special_count`.

If you would like to use this with `counter_culture_fix_counts`, make sure to also provide the `column_names` configuration.

Temporarily skipping counter cache updates

If you would like to temporarily pause `counter_culture`, for example in a backfill script, you can do so as follows:

```
1 Review.skip_counter_culture_updates do
2   user.reviews.create!
3 end
4
5 user.reviews_count # => unchanged
```

Totaling instead of counting

Instead of keeping a running count, you may want to automatically track a running total. In that case, the target counter will change by the value in the totaled field instead of changing by exactly 1 each time. Use the `delta_column` option to specify that the counter should change by the value of a specific field in the counted object. For example, suppose the `Product` model table has a field named `weight_ounces`, and you want to keep a running total of the weight for all the products in the `Category` model's `product_weight_ounces` field:

```
1 class Product < ActiveRecord::Base
2   belongs_to :category
3   counter_culture :category, column_name: 'product_weight_ounces',
4     delta_column: 'weight_ounces'
5 end
6
7 class Category < ActiveRecord::Base
8   has_many :products
9 end
```

Now, the `Category` model will keep the counter cache in `product_weight_ounces` up-to-date. The value in the counter cache will be the sum of the `weight_ounces` values in each of the associated `Product` records.

The `:delta_column` option supports all numeric column types, not just `:integer`. Specifically, `:float` is supported and tested.

Dynamically over-writing affected foreign keys

```
1 class Product < ActiveRecord::Base
2   belongs_to :category
3   counter_culture :category, foreign_key_values:
4     proc {|category_id| [category_id, Category.find_by_id(category_id)
5       ].try(:parent_category).try(:id)] }
6 end
7 class Category < ActiveRecord::Base
8   belongs_to :parent_category, class_name: 'Category', foreign_key: '
9     parent_id'
10  has_many :children, class_name: 'Category', foreign_key: 'parent_id'
11  has_many :products
12 end
```

Now, the `Category` model will keep an up-to-date counter-cache in the `products_count` column of the `categories` table. Each product will affect the counts of both its immediate category and that category's parent. This will work with any number of levels.

Updating timestamps when counts change

By default, `counter_culture` does not update the timestamp of models when it updates their counter caches. If you would like every change in the counter cache column to result in an updated timestamp, simply set the `touch` option to `true`:

```
1 counter_culture :category, touch: true
```

This is useful when you require your caches to get invalidated when the counter cache changes.

Custom timestamp column

You may also specify a custom timestamp column that gets updated only when a particular counter cache changes:

```
1 counter_culture :category, touch: 'category_count_changed'
```

With this option, any time the `category_counter_cache` changes both the `category_count_changed` and `updated_at` columns will get updated.

Avoiding deadlocks / executing counter cache updates after commit

Some applications run into issues with deadlocks involving counter cache updates when using this gem. See #263 for information and helpful links on how to avoid this issue.

Another option is to simply defer the update of counter caches to outside of the transaction. This gives up transactional guarantees for your counter cache updates but should resolve any deadlocks you experience. This behavior is disabled by default, enable it on each affected counter cache as follows:

```
1 counter_culture :category, execute_after_commit: true
```

[NOTE] You need to manually specify the `after_commit_action` as dependency in the Gemfile to use this feature

```
1 ...
2 gem "after_commit_action"
3 ...
```

You can also pass a `Proc` for dynamic control. This is useful for temporarily moving the counter cache update inside of the transaction:

```
1 counter_culture :category, execute_after_commit: proc { !Thread.
    current[:update_counter_cache_in_transaction] }
```

Manually populating counter cache values

You will sometimes want to populate counter-cache values from primary data. This is required when adding counter-caches to existing data. It is also recommended to run this regularly (at BestVendor, we run it once a week) to catch any incorrect values in the counter caches.

```
1 Product.counter_culture_fix_counts
2 # will automatically fix counts for all counter caches defined on
   Product
3
4 Product.counter_culture_fix_counts exclude: :category
5 # will automatically fix counts for all counter caches defined on
   Product, except for the :category relation
6
7 Product.counter_culture_fix_counts only: :category
8 # will automatically fix counts only on the :category relation on
   Product
9
10 # :exclude and :only also accept arrays of one level relations
11 # if you want to fix counts on a more than one level relation you need
   to use convention below:
```

```

12
13 Product.counter_culture_fix_counts only: [[:subcategory, :category]]
14 # will automatically fix counts only on the two-level [[:subcategory, :
    category] relation on Product
15
16 Product.counter_culture_fix_counts column_name: :reviews_count
17 # will automatically fix counts only on the :reviews_count column on
    Product
18 # This allows us to skip the columns that have already been processed.
    This is useful after running big DB changes that affect only one
    counter cache column.
19
20 # :except and :only also accept arrays
21
22 Product.counter_culture_fix_counts verbose: true
23 # prints some logs to STDOUT
24
25 Product.counter_culture_fix_counts only: :category, where: { categories
    : { id: 1 } }
26 # will automatically fix counts only on the :category with id 1
    relation on Product

```

The `counter_culture_fix_counts` counts method uses batch processing of records to keep the memory consumption low. The default batch size is 1000 but is configurable like so

```

1 # In an initializer
2 CounterCulture.config.batch_size = 100

```

or by passing the `:batch_size` option to the method call

```

1 Product.counter_culture_fix_counts batch_size: 100

```

`counter_culture_fix_counts` returns an array of hashes of all incorrect values for debugging purposes. The hashes have the following format:

```

1 { entity: which model the count was fixed on,
2   id: the id of the model that had the incorrect count,
3   what: which column contained the incorrect count,
4   wrong: the previously saved, incorrect count,
5   right: the newly fixed, correct count }

```

`counter_culture_fix_counts` is optimized to minimize the number of queries and runs very quickly.

Similarly to `counter_culture`, it is possible to update the records' timestamps, when fixing counts. If you would like to update the default timestamp field, pass `touch: true` option:

```

1 Product.counter_culture_fix_counts touch: true

```

If you have specified a custom timestamps column, pass its name as the value for the `touch` option:

```
1 Product.counter_culture_fix_counts touch: 'category_count_changed'
```

Parallelizing fix counter cache in multiple workers The options `start` and `finish` are especially useful if you want multiple workers dealing with the same processing queue. You can make worker 1 handle all the records between id 1 and 9999 and worker 2 handle from 10000 and beyond by setting the `:start` and `:finish` option on each worker.

! NOTE: the IDs we pass as `start` and `finish` here are in fact `Category` IDs, not `Product`!

```
1 Product.counter_culture_fix_counts start: 10_000
2 # will fix counts for all counter caches defined on Product from record
  10000 and onwards.
3
4 Product.counter_culture_fix_counts finish: 10_000
5 # let's process until 10000 records.
6
7 Product.counter_culture_fix_counts start: 1000, finish: 2000
8 # In worker 1, lets process from 1000 to 2000
9
10 Product.counter_culture_fix_counts start: 2001, finish: 3000
11 # In worker 2, lets process from 2001 to 3000
```

Fix counter cache using a replica database When fixing counter caches the number of reads usually vastly exceeds the number of writes. It can make sense to offload the read load to a replica database in this case. Rails 6 introduced native handling of multiple database connections. You can use this to send read traffic to a read-only replica using the option `db_connection_builder`:

```
1 Product.counter_culture_fix_counts db_connection_builder: proc{|reading
  , block|
2   if reading # Count calls will request a reading connection
3     Product.connected_to(role: :reading, &block)
4   else # Update all calls will request a non-reading connection
5     Product.connected_to(role: :writing, &block)
6   end
7 }
```

Handling dynamic column names Manually populating counter caches with dynamic column names requires additional configuration:

```
1 class Product < ActiveRecord::Base
2   belongs_to :category
```

```

3   counter_culture :category,
4     column_name: proc {|model| "#{model.product_type}_count" },
5     column_names: {
6       ["products.product_type = ?", 'awesome'] => 'awesome_count',
7       ["products.product_type = ?", 'sucky'] => 'sucky_count'
8     }
9   # attribute product_type may be one of ['awesome', 'sucky']
10 end

```

You can specify a scope instead of a where condition string for `column_names`. We recommend providing a Proc that returns a hash instead of directly providing a hash: If you were to directly provide a scope this would load your schema cache on startup which will break things like `rake db:migrate`.

```

1  class Product < ActiveRecord::Base
2    belongs_to :category
3    scope :awesomes, ->{ where "products.product_type = ?", 'awesome' }
4    scope :suckys, ->{ where "products.product_type = ?", 'sucky' }
5
6    counter_culture :category,
7      column_name: proc {|model| "#{model.product_type}_count" },
8      column_names: -> { {
9        Product.awesomes => :awesome_count,
10       Product.suckys => :sucky_count
11     } }
12 end

```

If you would like to avoid this configuration and simply skip counter caches with dynamic column names, while still fixing those counters on the model that are not dynamic, you can pass `skip_unsupported`:

```

1  Product.counter_culture_fix_counts skip_unsupported: true

```

You can also use context within the block that was provided with the `column_names` method:

```

1  class Product < ActiveRecord::Base
2    belongs_to :category
3    scope :awesomes, -> (ids) { where(ids: ids, product_type: 'awesome')
4      }
5
6    counter_culture :category,
7      column_name: 'awesome_count'
8      column_names: -> (context) {
9        { Product.awesomes(context[:ids]) => :awesome_count }
10     }
11 end
12 Product.counter_culture_fix_counts(context: { ids: [1, 2] })

```

Handling over-written, dynamic foreign keys Manually populating counter caches with dynamically over-written foreign keys (`:foreign_key_values` option) is not supported. You will have to write code to handle this case yourself.

Soft-deletes with paranoia or discard

This gem will keep counters correctly updated in Rails 4.2 or later when using paranoia or discard for soft-delete support. However, to ensure that counts are incremented after a restore you have to make sure to set up soft deletion (via `acts_as_paranoid` or `include Discard::Model`) before the call to `counter_culture` in your model:

Paranoia

```
1 class SoftDelete < ActiveRecord::Base
2   acts_as_paranoid
3
4   belongs_to :company
5   counter_culture :company
6 end
```

Discard

```
1 class SoftDelete < ActiveRecord::Base
2   include Discard::Model
3
4   belongs_to :company
5   counter_culture :company
6 end
```

PaperTrail integration

If you are using the `paper_trail` gem and would like new versions to be created when the counter cache columns are changed by `counter_culture`, you can set the `with_papertrail` option:

```
1 class Review < ActiveRecord::Base
2   counter_culture :product, with_papertrail: true
3 end
4
5 class Product < ActiveRecord::Base
6   has_paper_trail
7 end
```

Polymorphic associations `counter_culture` now supports polymorphic associations of one level only.

To discover which models need to be updated via `counter_culture_fix_counts`, `counter_culture` performs a `DISTINCT` query on the polymorphic relationship. This query can be expensive so we therefore offer the option (`polymorphic_classes`) to specify the models' counts that should be corrected:

```
1 Image.counter_culture_fix_counts(polymorphic_classes: Product)
2 # or
3 Image.counter_culture_fix_counts(polymorphic_classes: [Product,
    Employee])
```

Contributing to counter_culture

- Check out the latest master to make sure the feature hasn't been implemented or the bug hasn't been fixed yet.
- Check out the issue tracker to make sure someone already hasn't requested it and/or contributed it.
- Fork the project.
- Start a feature/bugfix branch.
- Commit and push until you are happy with your contribution.
- Make sure to add tests for it. This is important so I don't break it in a future version unintentionally.
- Please try not to mess with the Rakefile, version, or history. If you want to have your own version, or is otherwise necessary, that is fine, but please isolate to its own commit so I can cherry-pick around it.

Copyright

Copyright (c) 2012-2021 BestVendor, Magnus von Koeller. See LICENSE.txt for further details.