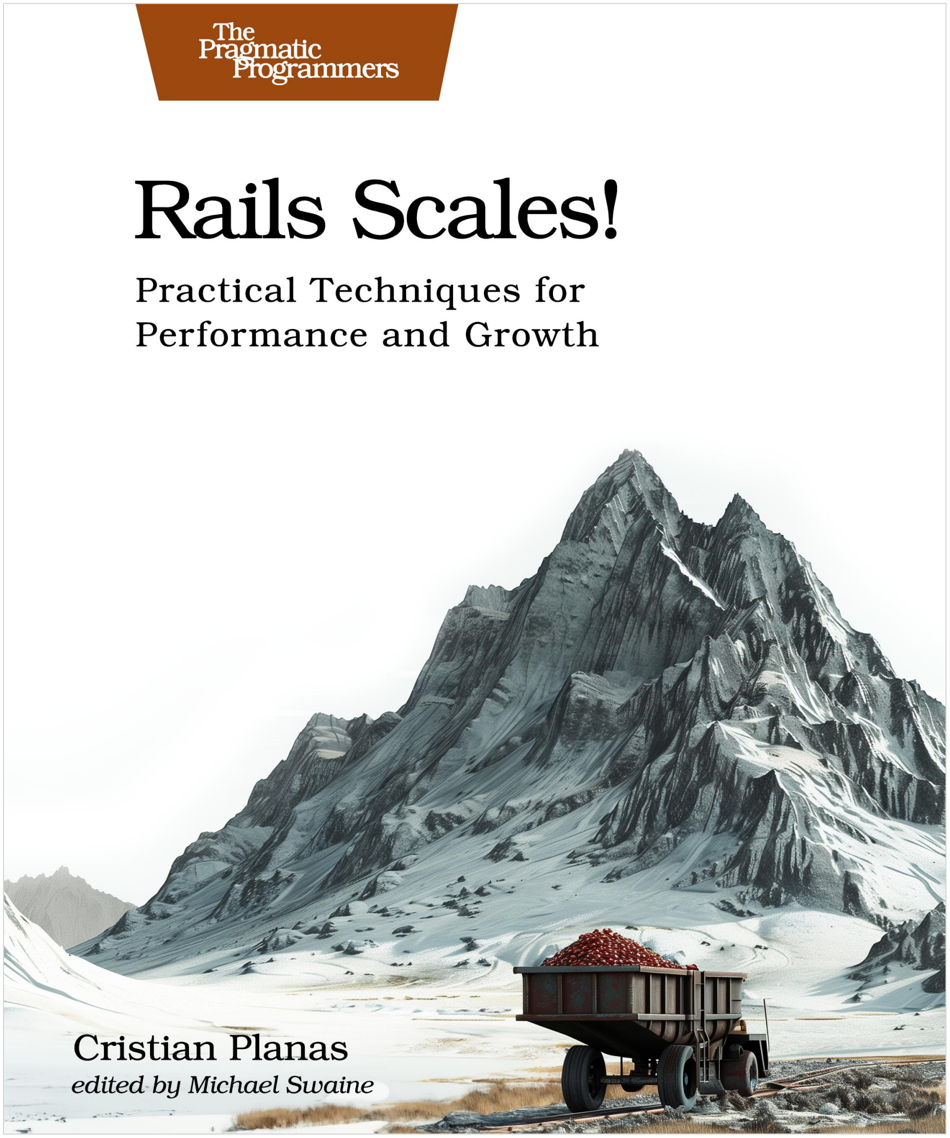


The Pragmatic Programmers

Rails Scales!

Practical Techniques for Performance and Growth

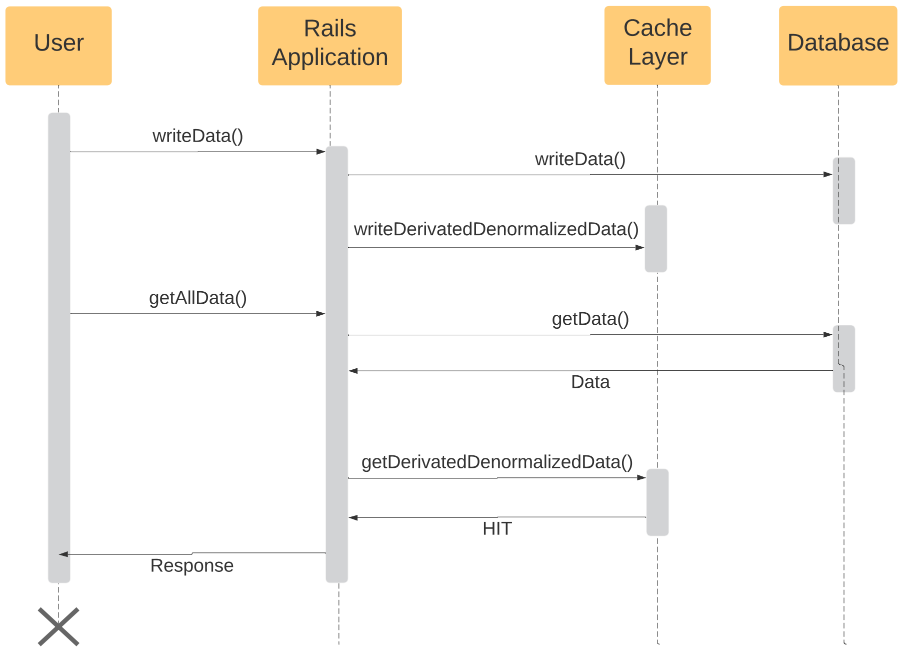
Cristian Planas

edited by Michael Swaine

Using Write-through Caching

One of the most important mottos of this book, one that you will read again and again repeated on these pages, is that performance improvements many times are a matter of trade-offs. The most common trade-off that performance engineers make on web applications is writing time vs. reading time; in other words, make writing slower so reading is faster. This *typically* makes sense in most web applications, because reading is way more common than writing.

The following image summarizes write-through caching:



You can implement write-through caching in our application easily. In this case, the ActiveModel callbacks will be extremely useful, and in particular, you can make use of after_save. Go and modify the Film model so that every time a film is modified, the application updates the cache at the same time. Something like the following:

```
class Film < ApplicationRecord
  after_save :write_cache

  [...]

  private

  def write_cache
```

```
    # Updates the cache
    Api::V1::FilmPresenter.new(self).to_json
  end
end
```

The call to Api::V1::FilmPresenter.new(film).to_json works because, as you remember, the to_json method renews the cache if it's expired.

This change has renewed the "smaller doll" in our Russian doll caching. Now change the code so the application also updates the cache for the whole /api/v1/films response. You can do that by modifying the ActiveModel callback you just created.

```
class Film < ApplicationRecord

  [...]

  def write_cache
    Rails.cache.write("/api/v1/films",
      Film.all.map { |film| Api::V1::FilmPresenter.new(film).to_json }.append(
        expiration_key: "#{Film.count}-#{updated_at}"
      )
    )

    true
  end
end
```

The previous code replaces the individual update of the cache of one particular for a full refresh of the cache of the endpoint. With this, any access to api/v1/films will *always* result in a cache hit, with obvious performance advantages. The next and final step would be to remove the check of the expiration key: it is not needed anymore. With write-through caching, reading can be an O(1) operation.

Still, write-through caching as implemented here has some significant drawbacks. Let's look at them.

First of all, updating a record has become significantly more expensive computationally, and this means it's slower. One way to improve this is to make the call to write_cache asynchronous. Nevertheless, taking this step implies a new set of trade-offs that we will explore together later in the book.

The fact that write-through caching slows down the write is an important issue, but in my opinion, the main drawback of this technique is its impact on the complexity of the application. As a matter of fact, you just *significantly* increased the complexity of the application. The Film model is now entangled with our implementation of the response for a particular endpoint (api/v1/films). Of course, there are ways to make this cleaner: the example you just coded is basically a proof of concept to show the power of write-through caching. All this can refactored quite deeply: even from a design perspective, one can separate the updating of the database from the writing of the cache by -again- making it asynchronous. Nevertheless, complexity has increased, and this will just get worse as our application gets more endpoints.

Still, there is more to caching and to write-through caching in particular than a sheer storing of an API response. Next, you will explore a different way of using it: fan-out writing.

Implementing Fan-out Writing

Yes, our application may sound outdated, being a video store management system, but don't think we are stuck in the 90s. We have implemented a social network on top of it! Our customers can follow each other, Twitter-style. But wait, it gets even cooler than that: customers have a public timeline that shows the latest films rented by the people they follow! You can see it calling api/v1/customers/#{id}/timeline.

Unfortunately, the queries that our application needs to run to fetch the timeline aren't pretty. Let's call the endpoint and check the logs. We will take a look at Matz timeline: appropriately for the creator of Ruby, he is the customer with id 1. Hit api/v1/customers/1/timeline and you will see this DB query:

```
Rental Load (43.6ms)  SELECT `rentals`.* FROM `rentals`
  WHERE `rentals`.`customer_id` IN (2, 3, 4, 5, 6, 7, 8,
    9, 10, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
    16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30,
    31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45,
    46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60,
    ...)
    ORDER BY `rentals`.`created_at` DESC LIMIT 10
```

The thing is that Matz is nice, so he is following thousands of users in our application! This makes for a pretty complex query. To build the timeline, the application needs to order the rentals by created_at but the issue here is that the customer_id value is arbitrary. Our database will need to pick all the rentals of the customers that Matz is following and then order them. Even if the number of rentals finally fetched is limited, the database still needs to operate with thousands of rentals to select the latest ones. One quick look at the controller action will show that the ActiveRecord query seems written well enough:

```
class Api::V1::CustomersController < ApplicationController
  [...]
  def timeline
    customer = Customer.find(params[:customer_id])
    rentals = Rental
      .where(customer_id: customer.followings.pluck(:followed_id))
      .order(created_at: :desc).limit(10)
      .includes(inventory: :film)

    render json: rentals.map do |rental|
      Api::V1::RentalPresenter.new(rental).to_json
    end
  end
end
```

Not much to optimize here. There is a way to eliminate the Customer.find call (can you implement it?), but that's small potatoes compared to the crucial method here: the Rental.where. In summary, not much you can do here to make this faster refining the way the application accesses the database.

The suggested solution here is not changing how the query runs but rather using write-through caching instead. However, in this particular caching the full response would be ill-advised. Imagine what would happen if one of the customers was very popular, being followed by millions of other customers. The application would cache exactly the same JSON Rental objects millions of times. Such a waste! The alternative is for each customer to store the ids of the rentals of the customers she is following. It would be a bit like having a mailbox: every time that a new Rental object is created, the application would store the id of this new rental as the "mailbox" of all the customers following the creator of that rental. Implementing this is very straightforward:

```
class Rental < ApplicationRecord
  after_create :cache_for_followers

  [...]

  private

  def cache_for_followers
    customer.followers.each do |follower|
      timeline = Rails.cache.read(follower.timeline_cache_key) || []

      Rails.cache.write(
        follower.timeline_cache_key, timeline.unshift(id)[0..9]
      )
    end
  end
end

class Customer < ApplicationRecord
  [...]

  def timeline_cache_key
    @timeline_cache_key ||= "rental-timeline-#{id}"
  end
end
```

This implementation seems sane, but it's not thread-safe: if two Rails processes would concurrently try to add two different rentals to a user's timeline, chances are that one of them would be accidentally wiped out. To fix this, you will need a storage system that would allow us to append values to an array in an atomic way. Fortunately, you have quite a few options, including Redis, one of the most popular key-value stores in the market. We will comment on this further later in this chapter, in the section dedicated to choosing a storage system to use for caching purposes.

Thanks to caching on the write, now the query needed to fetch the timeline is *extremely* simple. You just need to get the ids of the Rentals that need to be presented from our cached timeline and load them from the database using the primary key, the id. Fast and simple!

```
class Api::V1::CustomersController < ApplicationController
  [...]

  def timeline
    customer = Customer.find(params[:customer_id])
    rentals = Rental.where(id: Rails.cache.read(customer.timeline_cache_key))
    render json: rentals.map do |rental|
      Api::V1::RentalPresenter.new(rental).to_json
    end
  end
end
```

This example, though simplified, illustrates how real systems scale. For instance, when I was the CTO (aka solo engineer) of Playfulbet, my first start-up, we had to scale so the application worked for hundreds of thousands of users, and with very few resources. The application had a timeline, and I used fan-out writing to make it scale. It's also how Twitter[6] worked, at least until 2012, when the company still used Rails in its stack. Currently, at my job at Zendesk (another big company using Rails!) a similar technique powers our views, one critical feature of our Support product. Fan-out writing is a very useful design pattern that can power many complex operations.

Still, fanout writing has its own drawbacks. The most obvious is yet another increase in the complexity of the application. A most subtle and dangerous is that the increased computational cost of the write may be too expensive in certain specific cases. For example, some social networks[7] have what is called

6. https://www.infoq.com/presentations/Twitter-Timeline-Scalability/
7. https://www.wired.com/2015/11/how-instagram-solved-its-justin-bieber-problem/

the “Justin Bieber problem”: each post by a user with many followers could become a problem for the infrastructure. At a given point in 2010, Justin Bieber was using 3% of Twitter resources at any moment; with the rumor being that there were full servers dedicated to his account.[8] A way around this would be to introduce mixed fanout, so the vast majority of tweets use fanout writing, but keep the ones written by celebrities exempt. The timeline of a user would be constructed by using her timeline “bucket” created by fanout writing and mixing it with the tweets of the celebrities she follows, which will be fetched directly from the database -this is called fanout reading-. Nevertheless, despite these inconveniences, fanout writing can considerably speed up applications that operate at scale.

8. https://gizmodo.com/justin-bieber-has-dedicated-servers-at-twitter-5632095