Rails: Pagination e Caching

Progetto di Sistemi WEB

Anno Accademico 2025/2026

Filippo Poltronieri

filippo.poltronieri@unife.it

Pagination in Rails 8 – Panoramica

La **pagination** divide dataset ampi in pagine gestibili, migliorando performance (riduce query DB) e UX (contenuti digeribili).

Rails 8 non ha pagination built-in, ma integra sia *storicamente* che facilmente con la gemma **Pagy** per efficienza e compatibilità con Hotwire. Esistono anche altre alternative che non andremo a trattare.

Gemma	Descrizione	Vantaggi
Pagy	Leggera, veloce (no overhead)	Default per Rails 8; integra Turbo Streams
Kaminari	Flessibile per API	Buona per JSON responses
will_paginate	Legacy	Evita; meno performante

Perché la Pagination?

1. **Performance**
   → Limita record caricati (es. `limit 20, offset 0`).
2. **User Experience**
   → Navigazione intuitiva (prev/next, numeri pagine).
3. **Scalabilità**
   → Evita timeout su tabelle grandi (>10k record).
4. **Integrazione Hotwire**
   → Aggiornamenti DOM dinamici via Turbo Streams.

> **Nota importante**:
> Senza pagination, `User.all` carica tutto in memoria – rischio OOM.

Pagy

- Vediamo come utilizzare **Pag y** in Rails 8. Come le altre gemme, *pagy* va aggiunta al `Gemfile`:

```
gem 'pagy' # controllare sempre la versione corrente
```

```
bundle install
```

- Poi includiamo il modulo Pagy nel nostro `ApplicationController`:

```
class ApplicationController < ActionController::Base
  include Pagy::Mehtod
end
```

> **Vantaggio**: Pagy usa <1MB RAM per 1M pagine; zero SQL extra.

Pagination Base con Pagy

Controller

- Torniamo al nostro blog e proviamo ad applicare la paginazione alla lista dei post.
- Possiamo definire uno scope per la ricerca per titolo nel modello Post:

```
# app/models/post.rb
class Post < ApplicationRecord
  scope :search_by_title, ->(q) { where('title LIKE ?', "%#{q}%") }
 ####
 end
```

Controller (2)

```ruby
# app/controllers/posts_controller.rb
# --
  def index
    @q = params[:q]  # Per search
    scope = Post.all
    scope = scope.search_by_title(@q) if @q.present?
    @pagy, @posts = pagy(scope, items: 10)
  end
end
```

- `pagy(scope)` calcola pagine, limiti.
- Ritorna `@pagy` (metadata) e `@posts` (record paginati).

View

- Pagy fornisce anche un helper per la navigazione, che possiamo usare nella view:

```
<!-- app/views/posts/index.html.erb -->
<h1>Posts</h1>
<div id="posts">
  <%= render @posts %>
</div>
<%== @pagy.series_nav if @pagy.pages > 1 %>
<%== @pagy.info_tag %>
<%= link_to "New post", new_post_path %>
```

- `pagy_nav` : Link prev/next/num.
- `pagy_info` : "Mostra 1-10 di 100".

Pagination Avanzata: Turbo Streams

Possiamo integrare *pagy* con *Hotwire* per paginazione con `link` o `infinite scroll`.

Caso - Load More

- Link di richiesta nuova pagina via Turbo Stream.

Di cosa ho bisgono?

1. **Route** per caricamento pagine:

```
#config/routes.rb
resources :posts do
  get :load_more, on: :collection
end
```

Caso - Load More (2)

2. Dobbiamo modificare `PostsController` per gestire richieste Turbo Stream:

```
def load_more
  @pagy, @posts = pagy(Post.all, items: 10, page: params[:page])
  respond_to do |format|
    format.turbo_stream { render turbo_stream: turbo_stream.append(:posts, partial: "posts/post", collection: @posts) }
  end
end
```

3. Dobbiamo aggiungere un trigger alla View per caricare più post:

   Per esempio, un link "Load More" in `app/views/posts/index.html.erb`:

```
# versione con link
<%= link_to "Load More", load_more_posts_path(page: @pagy.next),
    data: { turbo_stream: true } %>
```

Caso - Infinite Scroll (1)

- Caricamento automatico al raggiungimento del fondo pagina.
- Anche qui, dobbiamo modificare `controller` e `view` e aggiungere una `route` per infinite scroll.
- Partendo dalla route, possiamo aggiungere `page` :

```
resources :posts do
  collection do
  get 'page' # → /posts/page?page=3&format=turbo_stream
  end
end
```

Caso - Infinite Scroll (2)

- Modifichiamo il controller di **Post** per rispondere a richieste Turbo Stream:

```
class PostsController < ApplicationController
  def page
    @pagy, @posts = pagy(Post.all, items: 10, page: params[:page])
    respond_to do |format|
      format.turbo_stream
      format.html
    end
  end
end
```

- Dopo aver modficato il controller per usare pagy, dobbiamo occuparci di inserire i turbo frame nella view `app/views/posts/index.html.erb` e creare una view apposita per turbo `app/views/posts/index.turbo_stream.erb`.

Caso - Infinite Scroll (3)

```
<!-- app/views/posts/index.html.erb -->
<div id="posts">
  <%= render @posts %>
</div>

  <% if @pagy.next %>
    <%= turbo_frame_tag "pagination",
    src: posts_path(page: @pagy.next, format: :turbo_stream),
    loading: :lazy %>
  <% end %>
```

Caso - Infinite Scroll (4)

```
<!-- app/views/posts/index.turbo_stream.erb -->
<%= turbo_stream.append "posts" do %>
  <%= render @posts %>
<% end %>

<%= turbo_stream.replace "pagination" do %>
  <% if @pagy.next %>
    <%= turbo_frame_tag "pagination",
    src: posts_path(page: @pagy.next, format: :turbo_stream),
    loading: :lazy %>
  <% end %>
<% end %>
```

Configurazione e Personalizzazione

In `config/initializers/pagy.rb`:

```
Pagy.option[:limit] = 20
Pagy.option[:max_pages] = 5  # Limita nav
```

Per API JSON:

```
def index
  @pagy, @posts = pagy(Post.all)
  render json: { posts: @posts, pagy: pagy_url_for(@pagy, 'json') }
end
```

> **Sicurezza**: Pagy valida `page` per evitare offset negativi.

Quando Usare Pagination

Scenario	Tipo	Esempio
Liste Utente	Base	`/posts` con pagy_nav
Search/API	Avanzata	JSON con page param
Infinite Scroll	Hotwire	Social feed con Turbo

Rails: Caching

Progetto di Sistemi WEB

Anno Accademico 2025/2026

Filippo Poltronieri

filippo.poltronieri@unife.it

Perché il Caching?

Il **caching** è una tecnica per memorizzare contenuti generati durante il ciclo request-response, riutilizzandoli per richieste simili.
Permette di:

1. **Migliorare le performance**
   → Riduce il carico su database e CPU, gestendo migliaia di utenti con infrastruttura modesta.
2. **Ridurre i tempi di risposta**
   → Contenuti serviti dalla cache sono istantanei.

Perché il Caching?

3. **Scalare l'applicazione**
   → Evita rigenerazioni inutili di viste o query costose.
4. **Integrarsi con Rails**
   → Funzionalità built-in per caching a vari livelli.

> **Nota importante**:
> In Rails 8+, il caching è abilitato solo in produzione per default. Usa `bin/rails dev:cache` per testare localmente.
> Si può abilitare settando a true l'opzione `config.action_controller.perform_caching` in `config/environments/development.rb` o con il comando `bin/rails dev:cache.`

Caching in Rails 8 – Panoramica

Rails 8 offre un **sistema di caching completo e moderno**, con focus su store persistenti come Solid Cache.

Tipo di Caching	Cosa memorizza	Quando usarlo
Fragment	Parti di viste (es. partials)	Viste complesse con dati statici
Low-Level	Valori serializzabili arbitrari	Logica di business o API esterne
SQL	Risultati di query in memoria	Query ripetute in una request
Page/Action	Intere pagine o azioni	Pagine statiche (ereditato)
Collection	Liste di partials	Rendering di collezioni

> **Novità Rails 8+**:
> Solid Cache è lo store di default, basato su database per SSD veloci.

Fragment Caching

Memorizza **parti specifiche di una vista**, permettendo invalidazione indipendente.

Esempio Base

```
<% @products.each do |product| %>
  <% cache product do %>
    <%= render product %>
  <% end %>
<% end %>
```

- **Chiave cache**: `views/products/index:.../products/1` (include digest template e versione record).
- Usa `updated_at` per invalidazione automatica.

Condizionale

- Per fare il caching di un fragment solamente in certe condizioni, si può utilizzare `cache_if` o `chache_unless`. Ad esempio, per fare il caching di un prodotto solo se l'utente è un admin:

```
<% cache_if admin?, product do %>
  <%= render product %>
<% end %>
```

Collection Caching

Memorizza **multipli partials da una collezione** in un colpo solo.

```
<%= render partial: 'products/product',
           collection: @products,
           cached: true %>
```

- **Chiave personalizzata** con lambda:

```
<%= render partial: 'products/product',
           collection: @products,
           cached: ->(product) { [I18n.locale, product] } %>
```

> **Best practice**: Usa per liste grandi, riducendo chiamate DB.

Russian Doll Caching

Caching nidificati: frammenti interni invalidano quelli esterni. Per esempio, prodotti con giochi:

```
<% cache product do %>
  <%= render product.games %>
<% end %>
```

```
<% cache game do %>
  <%= render game %>
<% end %>
```

- Richiede `touch: true` nelle associazioni:

```
class Game < ApplicationRecord
  belongs_to :product, touch: true
end
```

Low-Level Caching

Usa `Rails.cache.fetch` per **memorizzare valori serializzabili**.

```
Rails.cache.fetch("greeting") { "Hello, world!" }
```

Esempio con Modello

```
def competing_price
  Rails.cache.fetch("#{cache_key_with_version}/competing_price", expires_in: 12.hours) do
    Competitor::API.find_price(id)
  end
end
```

- `cache_key_with_version` genera una chiave unica basata su `id` e `updated_at`. Questa opzione aiuta a invalidare **automaticamente la cache quando il record viene aggiornato**.
- **Evita Active Record**: Cache ID o primitivi.

SQL Caching

Memorizza **risultati di query** in memoria durante una singola request.

Esempio

```
def index
        @products = Product.all  # Prima query: DB
        @products = Product.all  # Seconda: usa la cache
end
```

- Attivo dall'inizio all'end dell'azione.
- Non persistente tra request.

> **Vantaggio**: Veloce per query ripetute in una pagina.

Configurazione del Caching

Abilita caching nel controller:

```
config.action_controller.perform_caching = true  # In config/environments/production.rb
```

Store di Cache

Configura in `config/environments/*.rb`:

```
config.cache_store = :memory_store, { size: 64.megabytes }
```

Store di Cache

Store	Descrizione	Uso
:solid_cache_store	Default Rails 8+, DB-backed su SSD	Produzione scalabile
:memory_store	In memoria, per process	Sviluppo locale
:file_store	File system	Condiviso via FS
:redis_cache_store	Redis	Alta performance, eviction
:null_store	No-op	Test

Solid Cache

Store basato su database per SSD, default in Rails 8.

In `config/database.yml`:

```
production:
  cache:
    database: storage/production_cache.sqlite3
    migrations_paths: db/cache_migrate
```

Solid Cache

In `config/application.rb`:

```
config.cache_store = :solid_cache_store
```

- **Caratteristiche**:
  - Eviction FIFO.
  - Configurabile via `config/cache.yml` (max_age, max_size).
  - Supporta sharding, encryption.
  - Background expiry al 50% di capacità.

> **Installazione**: `rails db:create:cache` per creare il DB.

Chiavi Cache e Scadenza

Chiavi

- Stringhe, array o oggetti con `cache_key`.
- Active Record: `cache_key_with_version` (include `updated_at`).

```
Rails.cache.read(site: "mysite", owners: [owner_1, owner_2])
```

Scadenza

- **Automatica**: Via versione record.
- **Manuale**: `Rails.cache.delete(key)`.
- **Tempo-based**: `expires_in` in `fetch`.
- Solid Cache: Task background per cleanup.

> **Dipendenze esplicite** in template:

```
<%# Template Dependency: todolists/todolist %>
```

Conditional GET

- Conditional GET è una feature di HTTP che permette al server di rispondere con un codice 304 Not Modified se la risorsa richiesta non è cambiata dall'ultima volta che il client l'ha richiesta.
- Funziona tramite header HTTP come `If-Modified-Since` e `If-None-Match`.
- Rails da supporto a Conditional GET con i metodi `fresh_when` e `stale?` nei controller.

```ruby
class ProductsController < ApplicationController
  def show
    @product = Product.find(params[:id])
    # If the request is stale according to the given timestamp and etag value
    if stale?(last_modified: @product.updated_at.utc, etag: @product.cache_key_with_version)
      respond_to do |wants|
      end
    end
    # If the request is fresh (i.e. it's not modified) then don't do anything.
  end
end
```

Conditional GET (2)

`fresh_when` imposta automaticamente gli header di risposta basati su timestamp e etag.

```
class ProductsController < ApplicationController
  def show
    @product = Product.find(params[:id])
    fresh_when last_modified: @product.updated_at.utc, etag: @product
    # Normal response processing
  end
end
```

- Il codice manda una risposta :not_modified se il client ha la versione più recente.
- Se invece è stale, il server risponde con il contenuto aggiornato.

Best Practices

Regola	Spiegazione
Usa fragment per viste costose	Cache partials indipendenti
Non cache Active Record	Usa ID o dati primitivi
Russian Doll con touch: true	Per nidificati efficienti
Store dedicato in produzione	Redis o Solid Cache
GET condizionale	`fresh_when` per browser cache
Testa il caching	Abilita in dev con `bin/rails dev:cache`
Evita chiavi complesse	Mantieni semplici per debugging

Quando Usare Ogni Tipo di Caching

Tipo	Priorità	Quando usarlo
Fragment	Alta	Parti dinamiche di viste
Low-Level	Alta	Computi costosi o API
SQL	Media	Query ripetute in request
Collection	Media	Liste grandi
Russian Doll	Bassa	Gerarchie nidificate