УДК 004.93 **Информационные технологии**

Никишанин Роман Олегович, *магистр, Национальный исследовательский Мордовский государственный университет им. Н.П. Огарева*
Ямашкин Станислав Анатольевич, *кандидат технических наук, доцент кафедры автоматизированных систем обработки информации и управления, Национальный исследовательский Мордовский государственный университет им. Н. П. Огарева*

ОПТИМИЗАЦИЯ SQL-ЗАПРОСОВ В ВЕБ-ПРИЛОЖЕНИИ, НАПИСАННОМ НА RUBY ON RAILS

Аннотация: В статье описаны основные проблемы, с которыми сталкивается Ruby on Rails разработчик при написании программного кода, в котором происходит обращение к базе данных через SQL-запросы; описаны основные подходы, которые используются в Ruby on Rails для доступа к ассоциативным данным и ситуации, когда следует применять каждый из подходов.

Ключевые слова: Ruby on Rails, SQL, база данных, Active Model, Active Record, проблема N+1 запросов, lazy load, eager load.

Abstract: The article describes the main problems that a Ruby on Rails developer faces when writing program code that accesses the database through SQL queries; describes the main approaches that are used in Ruby on Rails to access associative data and situations when each of the approaches should be applied.

Keywords: Ruby on Rails, SQL, database, Active Model, Active Record, N+1 query problem, lazy load, eager load.

Введение

В настоящее время нередко возникает потребность в создании

высоконагруженных систем массового обслуживания, развернуты в сети Интернет. Это могут быть сервисы по продаже авиабилетов, торговые площадки, биржи и т.д. В таких системах необходимо оперировать большими объемами данных с довольно сложной структурой [1]. Если в качестве базы данных планируется использовать один из реляционных вариантов (Postgresql, Mysql), то она неизбежно обрастет множеством таблицы с различными связями. На этапе планирования очень часто в качестве инструмента для реализации подобных веб-сервисов выбирается веб-фреймворк Ruby on Rails [6].

RoR предоставляет множество инструментов для быстрой и эффективной разработки. Одними из таких инструментов являются библиотеки Active Model и Active Record, которые обеспечивают представление базы данных приложения в виде классов-моделей с определенной бизнес логикой, а также удобную работу с содержимым базы данных с помощью универсального ORM интерфейса [2; 3].

Однако на практике в процессе разработки реальных веб-приложений нередко возникают ситуации, когда код, в котором выполняется доступ к базе данных, выполняется достаточно медленно, что негативно сказывается на общем времени выполнения запроса и ожидания пользователем ответа с сервера. При масштабировании приложения это проблема может оказаться критической, поэтому важно писать оптимизированные запросы в базу данных. Зачастую это может быть связано с тем, что разработчик при написании программного кода использовал не подходящие под конкретную ситуацию ORM - методы для выполнения SQL-запросов и получения дочерних таблиц (моделей на языке ORM) [7].

В данной статье будут рассмотрены проблемы, с которыми сталкивается RoR - разработчик при написании кода, отвечающего за доступ к базе данных, а также основные способы решения этих проблем.

1. Проблема N+1 запросов

“Проблема N + 1 запросов” - это проблема при работе с ассоциациями (таким образом в Active Model реализуются табличные отношения “один к

одному”, “один ко многим” и “многие ко многим”) [8]. В данном случае N - это количество записей, у которых берутся ассоциации. Например, у нас есть две модели User и Comment, которые связаны отношением “один к одному”:

```
class User < ApplicationRecord
  has_many :comments
end

class Comment < ApplicationRecord
  belongs_to :user
end
```

В коде нам необходимо получить массив записей пользователей и для каждого получить связанные с ним комментарии следующим образом:

```
users = User.all
users.each do |user|
  comments = user.comments
  # выполнить необходимые действия с комментариями
end
```

В данной ситуации запросы в базу данных будут выглядеть следующим образом:

```
User Load (0.8ms)  SELECT "users".* FROM "users" LIMIT $1  [["LIMIT",
11]]
Comment Load (0.2ms)  SELECT "comments".* FROM "comments" WHERE
"comments"."user_id" = $1  [["user_id", 1]]
Comment Load (0.2ms)  SELECT "comments".* FROM "comments" WHERE
"comments"."user_id" = $1  [["user_id", 2]]
Comment Load (0.2ms)  SELECT "comments".* FROM "comments" WHERE
"comments"."user_id" = $1  [["user_id", 3]]
Comment Load (0.2ms)  SELECT "comments".* FROM "comments" WHERE
"comments"."user_id" = $1  [["user_id", 4]]
Comment Load (0.1ms)  SELECT "comments".* FROM "comments" WHERE
"comments"."user_id" = $1  [["user_id", 5]]
```

В результаты выполнения программного кода было выполнено 5 + 1 (количество пользователей в данном случае 5) запросов в базу данных. Если число пользователей возрастет, это может привести к серьезному падению производительности. Для решения данной проблемы в Active Record существуют специальные методы.

2. Lazy Loading и Eager Loading

В Ruby on Rails существует два варианта загрузки ассоциаций: Lazy Loading и Eager Loading. Lazy Loading - это подход, при котором ассоциации записей загружаются в память не сразу, а только в момент, когда они необходимы в коде и вызываются с помощью соответствующих методов. В Ruby on Rails он применяется по умолчанию при работе с ассоциациями. Lazy Loading позволяет сократить количество выделяемой памяти и SQL-запросов при получении исходных записей. Однако, если после этого мы будем использовать ассоциации каждой полученной записи в программном коде, это приведет к вызову дополнительных SQL-запросов, что было продемонстрировано ранее. Lazy Loading рекомендуется использовать только в том случае, если ассоциации не будут запрашиваться явно в программном коде или если данные из ассоциаций необходимы в SQL-запросе для исходных записей [4]. Для Active Record предоставляет специальный метод joins. Пример его использования:

```
users = User.all.joins(:comments).where("comments.body='test'")
  User Load (3.4ms)  SELECT "users".* FROM "users" INNER JOIN
"comments" ON "comments"."user_id" = "users"."id" WHERE
(comments.body='test') LIMIT $1
```

Eager Loading - подход, при котором ассоциации заранее подгружаются в память и запрашиваются вместе с исходными записями или одним отдельным запросом в зависимости от используемых методов [5]. Данный подход увеличивает расход памяти, однако позволяет сократить количество SQL-запросов в тех случаях, когда ассоциации непосредственно запрашиваются в коде, что обеспечивает хорошую производительность при большом количестве

записей. В Active Record существует несколько методов для запроса ассоциаций в формате Eager Loading: preload, eager_load и includes. Последний метод является наиболее универсальным и чаще всего применятся на практике. Добавим Eager Loading в приведенный выше программный код и посмотри на SQL-запросы:

```
users = User.all.includes(:comments)
users.each do |user|
  comments = user.comments
  # выполнить необходимые действия с комментариями
end
```

```
User Load (0.2ms)  SELECT "users".* FROM "users" LIMIT $1  [["LIMIT", 11]]
Comment Load (0.3ms)  SELECT "comments".* FROM "comments" WHERE "comments"."user_id" IN ($1, $2, $3, $4, $5)  [["user_id", 1], ["user_id", 2], ["user_id", 3], ["user_id", 4], ["user_id", 5]]
```

Количество запросов в базу данных значительно сократилось, и при увеличении кол-ва пользователей разница будет существенно возрастать.

Участки программного кода, где осуществляется взаимодействие с базой данных, являются одной из наиболее частых причин низкой производительности. Поэтому в приложениях, написанных на Ruby on Rails, очень важно уметь правильно использовать методы для SQL-запросов в тех ситуациях, где они будут уместны и наиболее эффективны для предотвращения проблем в будущем с ростом содержимого базы данных.

Выводы

Таким образом, в данной статье была рассмотрена проблема ‘N+1 запросов’, которая является одной из основных причин низкой производительности программного кода в современных Ruby on Rails приложениях. Она возникает из-за неправильного использования способов подгруздки ассоциативных данных (eager loading и lazy loading). Методы,

реализующие eager loading, следует использовать в тех случаях, когда мы явно запрашиваем доступ к ассоциативным данным в программном коде. Использование данного подхода позволяет решить проблему ‘N+1 запросов’. В свою очередь методы, реализующие lazy loading, должны применяться для формирования сложных SQL-запросов с использованием ассоциативных данных.

Библиографический список:

1. Афонин В. В. Основы анализа систем массового обслуживания / В. В. Афонин, С. М. Мурюмин, С. А. Федосин / — Текст: непосредственный // Научная электронная библиотека ELibrary.ru — 2003. — С. 234. — Режим доступа: https://www.elibrary.ru/item.asp?id=19581660 (дата обращения: 28.11.2022).

2. Документация Ruby [Электронный ресурс]: ruby-doc.org – Режим доступа: https://ruby-doc.org/ (дата обращения: 25.10.2022).

3. Официальная документация API Ruby on Rails [Электронный ресурс]: api.rubyonrails.org – Режим доступа: https://api.rubyonrails.org/ (дата обращения: 25.05.2022).

4. Preload, Eagerload, Includes and Joins [Электронный ресурс]: www.bigbinary.com – Режим доступа: https://www.bigbinary.com/blog/preload-vs-eager-load-vs-joins-vs-includes (дата обращения: 15.11.2022).

5. Различные методы загрузки ассоциаций в Ruby on Rails [Электроный ресурс]: habr.com – Режим доступа: https://habr.com/ru/post/191762/ (дата обращения: 15.11.2022).

6. Ruby on Rails по-русски [Электронный ресурс]: rusrails.ru – Режим доступа: http://rusrails.ru/ (дата обращения: 28.11.2022).

7. Rails N+1 queries and eager loading [Электронный ресурс]: dev.to – Режим доступа: https://dev.to/junko911/rails-n-1-queries-and-eager-loading-10eh (дата обращения: 18.11.2022).

8. Проблема N+1 [Электронный ресурс]: medium.com. – Режим доступа:

https://medium.com/@croatech/rails-является-одним-из-самых-популярных-фреймворков-для-созданиямvp-minimum-viable-products-49a00a63137c (дата обращения: 18.11.2022).