Zutaten für performante Rails-Anwendungen

Lass knacken, Alter

Phillip Ghadir und Christoph Wiemers

„Rails ist langsam und inperformant!“, sagen Rails-Gegner. Ruby ist um ein Vielfaches langsamer als beispielsweise Java oder Python. Dennoch gibt es Rails-Anwendungen, die trotz Ruby schnell genug sind. In diesem Artikel stellen wir einige Rezepte zur Realisierung performanter Rails-Anwendungen vor.

Grundlagen performanter Webanwendungen

Webanwendungen sind Request/Response-basiert: Der Server verarbeitet Clientanfragen und sendet anschließend eine Antwort an den Client zurück. Eine Webanwendung ist dann performant, wenn sie für die Verarbeitung einer Anfrage sparsam mit den Ressourcen (CPU-Zeit und I/O) umgeht. Der Ressourcenverbrauch bei der Verarbeitung einer Anfrage hängt zwar auch von der Effizienz der Implementierung ab, aber noch viel stärker von der Effektivität der Verarbeitung. Die Effektivität wiederum ergibt sich aus der Art der Anfragen, den möglichen Antworten und der Größe der Umgebung (Datenvolumen, eingebundene Systeme, räumliche Distanz).

Unserer Ansicht nach ist die Programmiersprache für die Performance von Webanwendungen weit weniger wichtig als allgemein angenommen. In diesem Artikel präsentieren wir Ideen, Überlegungen und Faktoren, mit anderen Worten: Zutaten, die in die Konstruktion von performanten Rails-Anwendungen maßgeblich einfließen sollten, um so für die Verarbeitung von Requests möglichst wenig Ressourcen (CPU-Zyklen, I/O, Netzwerk, Datenbankzugriffe) zu verbrauchen. Ähnlich wie beim Backen bringen die Zutaten allein aber noch keinen leckeren Kuchen hervor – wir müssen sie noch im richtigen Verhältnis zusammenmischen.

Anatomie einer Rails-Anwendung

Abbildung 1 stellt schematisch dar, wie eine Rails-Anwendung betrieben werden kann. Typischerweise hat man einen Webserver, der als Verteiler für die nachgelagerten Rails-Server dient.

kurz und bündig

Inhalt

Reduktion der Komplexität der Datenbankanfragen.

Quellcode mit angeliefert

Nein

Diese teilen sich als gemeinsames Backend eine Datenbank. In der Abbildung ist farbig unterlegt, zu welchen Themen wir uns Gedanken machen müssen, um für dieses vereinfachte Deployment-Szenario eine performante Rails-Anwendung entwerfen zu können. Im Folgenden greifen wir einige Aspekte heraus.

Das Programmiermodell

Sehen wir uns zunächst anhand von zwei Beispielen das Zusammenspiel zwischen Clientanfragen, Verarbeitung in den Rails-Servern und Datenbankabfragen an. Dazu starten wir in der Mitte von Abbildung 1, ausgehend vom Programmiermodell. Hier bietet sich für uns Entwickler die Chance, die Grundlagen für eine performante Anwendung zu legen.

Ein Kardinalfehler beim Entwickeln von Rails-Anwendungen ist die Auswahl und Bereitstellung von Funktionalität im Web, die beliebig große Datenmengen verarbeiten, annehmen oder zurückliefern muss: Nimmt man das Scaffolding als Vorgabe, schleicht sich ein, dass beispielsweise die Liste aller Kundenobjekte über den URI */customers/* dargestellt wird. Ressourcenschonender wäre es, unter selbiger URL die Liste der ersten zehn Kunden zurückzugeben. Statt also in den *Controller#index*-Methoden

```
<Entitätsklasse>.find(:all)
```

zu schreiben, sollte man hier eine Paging-Funktionalität einbauen:

```
<Entitätsklasse>.find(:all,
    :limit => 10, :offset => 10 * (params[:page] || 0) )
```

Es bleibt jedem Leser selbst überlassen, sich dazu ein passendes Gem oder Plug-in auszusuchen. Wir haben gute Erfahrungen mit dem *gem paginator* gemacht [5]. Für Anwendungsfälle, in denen man begrenzt große Ausschnitte aus Treffermengen benötigt, ist diese Strategie praktikabel. Fälle, in denen man den kompletten Datenbestand durchsuchen bzw. laden muss, z. B. bei Auswertungen oder Massendatenverarbeitung, können eine Implementierung am Framework vorbei notwendig machen. Dies kann für kleinere Datenmengen unter direkter Verwendung der Datenbanktreiber erfolgen oder aber durch Rails-externe Prozesse. In jedem Fall sollten Operationen zur Massendatenverarbeitung nicht innerhalb von Onlinetransaktionen ausgeführt werden. Denkbar wären z. B. die zyklische Aufbereitung der Daten außerhalb der Rails-Anwendung oder

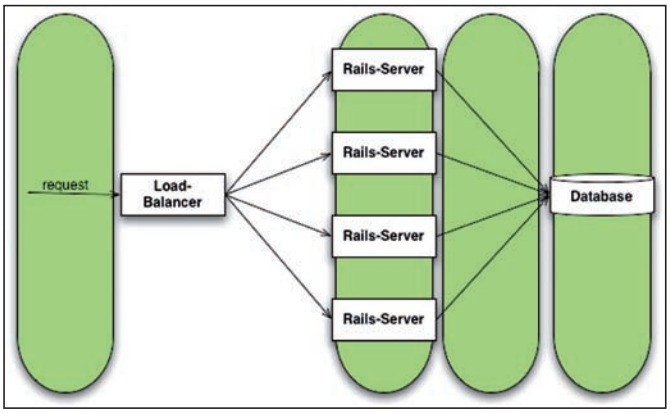

Abb. 1: Typische Webserverlandschaft

eine asynchrone Verarbeitung, die durch eine Onlinetransaktion ausgelöst werden kann.

Klassische Implementierungsmuster

Aber nicht nur das Anbieten der falschen Funktionalität kann zu Problemen führen, auch das Missachten des Programmiermodells kann sich verheerend auf die Performance auswirken. Am Beispiel von Plausibilitätsprüfungen skizzieren wir nun Überlegungen, die die Entwurfsentscheidungen für die Realisierung performanter Rails-Anwendungen beeinflussen sollten. Grundsätzlich sollten Plausibilitätsprüfungen im Modell hinterlegt werden. Dadurch kann man plausible Objektzustände garantieren, sofern der Datenbestand jeweils nur mithilfe des Modells geändert wird.

Was tun mit Daten, die durchaus in unplausiblen Zuständen vorliegen können, aber nur in plausiblen verarbeitet werden dürfen? Abhängig vom Aufwand der Plausibilitätsprüfung sollte deren Ergebnis immer dann am Hauptobjekt in einem Attribut gespeichert werden, wenn sie das Laden unterschiedlicher Objekte erfordert. In Fällen, in denen sich die Prüfung nur auf geladene Daten (also die eigenen Attribute) bezieht und effizient berechnet werden kann, ist das Speichern des Plausibilitätsstatus in einem Attribut nicht nötig.

Aber was hat das mit performanten Rails-Anwendungen zu tun? Prüfungen, die das Laden größerer Objektnetze verursachen und während des Besuchs einer Seite mehrfach geprüft werden müssen, können unnötig viele (Server-)Ressourcen binden.

Unit of Work/Request

Ein weiteres potenzielles Performanceproblem lauert in der eigentlichen Applikationslogik. Als Entwickler sollte man sich unbedingt die Frage stellen, wie viele Operationen durch einen Request ausgelöst werden können. Dies ist insbesondere dann wichtig, wenn diese Operationen jeweils Latenz und Verarbeitungszeit verursachen. Ein Trivialbeispiel haben wir ja bereits mit dem *find(:all)* in Controller-Methoden angerissen. Dabei werden – abhängig vom Datenbankinhalt – beliebig viele Objekte instanziiert. Um ein Vielfaches schlimmer wird es, wenn man für jedes Objekt aus der Treffermenge weitere Datenbankabfragen und Logik aufrufen müsste. Michael Nygard nennt dies das 1+N-Problem [7]. ActiveRecord unterstützt bei der Lösung dieses Problems beispielsweise durch die *:include*-Option, die bei *find*-Aufrufen übergeben werden kann:

```
Customer.find(:all, :limit => 10, :include => [ :orders ])
```

Ein Blick in das Log-File zeigt die beiden Datenbankabfragen, die nun das Laden von Kunden und Bestellungen durchführen:

```
SELECT * FROM 'customers' LIMIT 10
SELECT 'orders'.* FROM 'orders' WHERE ('orders'.customer_id IN (<ids...>))
```

Aber halt! Haben wir hier nicht den gleichen Fehler wie ursprünglich gemacht? Die Anweisung würde jetzt nur die ersten zehn Kunden laden, allerdings alle Bestellungen dieser Kunden. Um jetzt auch sicherzustellen, dass man nicht beliebig viele Bestellungen lädt, kann man die Beziehung gleichermaßen attributieren wie zuvor die Abfrage. Am Beispiel von *Customer has_many :orders* sähe das folgendermaßen aus:

```
class Customer < ActiveRecord::Base
  has_many :recent_orders, :order => "order_date DESC", :limit => 3
end
```

Leider liefert der Aufruf des *Customer#find* mit *:include* dieser Beziehung trotzdem alle *Orders*, da an die Datenbank eine Abfrage wie folgt geschickt wird:

```
SELECT * FROM 'customers' WHERE (id IN (<ids...>)) LIMIT 10
SELECT 'orders'.* FROM 'orders' WHERE ('orders'.customer_id IN (<ids...>)) ORDER BY
order_date DESC
```

Wenn man für ein bereits geladenes *Customer*-Objekt die *recent_orders* aufruft, wird korrekt an die Datenbank geschickt:

```
SELECT * FROM `orders` WHERE (`orders`.customer_id = <id>) ORDER BY order_date DESC
LIMIT 3
```

Eine gelungene Darstellung von ActiveRecord-Performance-Tipps bietet [1]. Eine perfekte Lösung gibt es also nicht. Man muss sich daher sehr genau überlegen, an welchen Stellen die Laufzeit für das Instanziieren von Model-Klassen besser ist als die für eine limitierte Anzahl von Datenbankabfragen, die ebenfalls limitierte Ergebnismengen zurück liefern. Dies führt zum nächsten wesentlichen Punkt.

Die Verwaltung des Informationslebenszyklus

Eine performante Webanwendung sollte so wenig CPU-Zyklen wie möglich für die Beantwortung einer Anfrage verbrauchen. Man kann hier auf performantere VMs setzen oder bei der Implementierung darauf achten, dass man Sprachkonstrukte und Framework-Funktionen verwendet, die möglichst wenige CPU-Zyklen benötigen. Einige nicht mehr ganz aktuelle Hilfestellungen dazu können [2] bzw. [3] bieten. Viel wichtiger als die Optimierung von Details ist jedoch das Verwalten des Lebenszyklus von Daten.

Beim Entwurf eines Datenmodells, das die fachliche Domäne repräsentiert, wird häufig von Bearbeitungsschritten abstrahiert, die von Anwendungssystemen (hier: unserer Rails-Anwendung) realisiert werden. Oft führt dies dazu, dass für Anwendungsfälle, die nur schlecht vom Datenmodell unterstützt werden, komplexe Abfragen erforderlich sind. Betrachten wir als Beispiel folgende Fragestellungen:

- Finde Tupel aus folgenden, in Beziehung stehenden Entitäten.
- Finde Elemente, die aktuell (bzw. noch schlimmer: zum Zeitpunkt t) gültig sind.
- Finde Elemente, die noch nicht gelöscht sind.
- Finde Elemente, die man ausnahmsweise doch sehen darf, obwohl man im Regelfall keinen Zugriff darauf hat.

Diese Fragestellungen erfordern meist aufwändige Datenbankabfragen. Insbesondere die Kombination solcher Fragestellungen schließt schon bei nicht mehr ganz kleinen Datenvolumen die Verwendung innerhalb von Onlinetransaktionen im Internet aus. Einen signifikanten Performancegewinn erreicht man einerseits durch direkte Zugriffe auf relevante, aufbereitete Daten (Abb. 2) und andererseits durch eine Reduktion des Datenvolumens.

Große Datenmengen mancher Anwendungen kann man durch Segmentierung in den Griff bekommen. Vielfach wird dies auf Basis von Attributbelegungen gemacht, ohne dabei den Informationskontext zu betrachten. Dabei bringt eine Unterscheidung von regelmäßig benötigten Daten, häufig aktualisierten Daten, langfristig aber selten benötigten Daten und historischen Daten eine vollkommen andere Perspektive ins Spiel. Eine solche Datenmodell-Segmentierung lässt sich einerseits einfach im Datenmodell abbilden und gibt andererseits klare Ansätze, um „teure" Wiederbeschaffung historischer Daten außerhalb des Rails-Servers zu bedienen.

Für alle Daten, die in die Hoheit der Rails-Anwendung fallen, ist wichtig, eine vollständige Auszeichnung des Verarbeitungsstatus, der Zusammenhänge, erkannter Fehlerkonstellationen und fehlender Verbindungen zu speichern. Das Model muss also solche Informationen direkt repräsentieren und deren Konsistenz sicherstellen. So können mit einfachen Anfragen Informationen bereitgestellt und über die Webanwendung angeboten werden.

Strategien für die Ressourcenschonung

Die bislang diskutierten Überlegungen ranken sich rund um das Thema Wie-muss-die-Anwendung-mit-den-Daten-umgehen? Aus Betriebsicht können Ressourcen geschont werden, indem Antworten auf Anfragen nicht mehrfach berechnet werden müssen oder noch besser, wenn eine Reihe von Anfragen nicht mehr gestellt werden muss. Es ist gängige Praxis, statische Inhalte nicht von einem Rails-Server, sondern von anderen Servern ausliefern zu lassen. Für größere Systeme bietet sich sogar an, hier auf so genannte Content Delivery Networks zu setzen [6].

Sitzungen und transiente, anfragenübergreifende Speicherung von Daten

Transiente Daten im Hauptspeicher zu halten, hat Vor- und Nachteile. Auf der einen Seite führt die Praxis, transiente Daten mit einem Sitzungsschlüssel zu verknüpfen, zu einer Laufzeitabhängigkeit des Clients zu einer Serverinstanz. Auf der anderen Seite lässt sich so die Zahl der Datenbankabfragen reduzieren. Aber auch das geschieht nicht ohne Trade-off – die Daten belegen natürlich Platz im Hauptspeicher. Dieses Problem kann wiederum durch Mechanismen gelöst werden, die sitzungsbezogene Daten ins Dateisystem (oder in die Datenbank oder in einen externen Cache) auslagern.

Fazit

Mit Ruby und Ruby on Rails kann man performante Webanwendungen realisieren. Prinzipiell gibt es viele Faktoren, die ein System inperformant machen. Die Wahl der Programmiersprache trägt dazu neben anderen Faktoren bei, spielt aber sicherlich nicht die Hauptrolle. In Rails ist es besonders einfach, für jede Information im System eine eindeutige Quelle zu definieren. Das Model-View-Controller-Pattern hilft dabei, Anwendungslogik je nach Aufgabenbereich zu realisieren und dabei auch fortgeschrittene Aspekte wie die Verwaltung von Informationslebenszyklen im Modell passend zu repräsentieren.

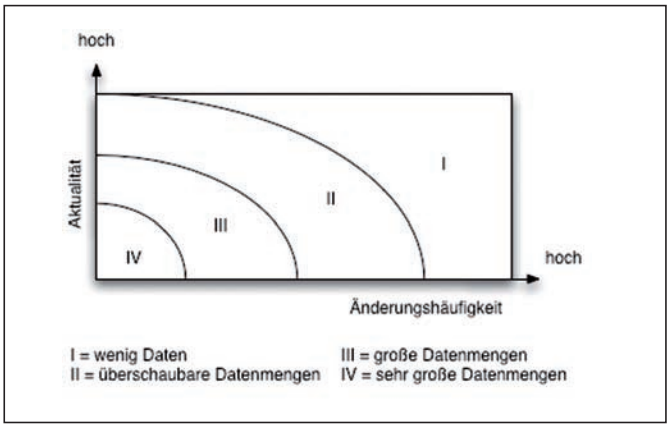

Abb. 2: Grenzen des Datenvolumens abhängig von Aktualität und Volatilität

Ruby ist langsamer als andere Sprachen, allerdings ist dies für den Kontext einer Webanwendung nicht kriegsentscheidend, sofern man einige Grundwahrheiten der Informatik beherzigt. Effiziente Algorithmen lassen sich auch mit Ruby genügend schnell beantworten. Erst bei großen Arbeitseinheiten pro Request hat die Ruby-VM einen spürbar negativen Einfluss auf das Antwortzeitverhalten bzw. den erreichbaren Durchsatz.

Dennoch: Die niedrigen Anforderungen von Ruby on Rails an die Betriebsumgebung ermöglichen hier bei geeigneter Produktionsumgebung sowohl Scale-Up (Aufrüsten der Maschine) als auch ein Scale-Out (Hinzufügen von Maschinen). Insbesondere beim Scale-Out ist der ressourcenschonende Gebrauch von gemeinsam genutzten Ressourcen wie Datenbank, Dateisystem und anderen Diensten essenziell, da diese leicht zum Flaschenhals werden können.

Links & Literatur

[1] Envy Casts/Advanced Active Record: http://envycasts.com/products/advancedactiverecord

[2] Top 10 Ruby on Rails performance tips, 2007: http://antoniocangiano.com/2007/02/10/top-10-ruby-on-rails-performance-tips/

[3] A Look at Common Performance Problems in Rails, infoQ, 2006: http://www.infoq.com/articles/Rails-Performance

[4] Passenger Architectural Overview: http://www.modrails.com/documentation/Architectural%20overview.html

[5] Gem Paginator: http://rubyforge.org/projects/paginator/

[6] Stefan Tilkov, RESTful Web Services mit Rails, S. 67

[7] MiNy2007: Release it!

Phillip Ghadir ist Chief Technology Officer bei der innoQ Deutschland GmbH. Sein Interesse gilt Konzepten und Architekturen für skalierbare, verteilte Systeme.

Christoph Wiemers beschäftigt sich bei innoQ mit Mustern für skalierbare Ruby-on-Rails-Anwendungen.