# 4 Alternative Komponenten

- Wie kann ich Benutzern eine Suche anbieten?
- Was steckt hinter NoSQL?
- Harmoniert Ruby on Rails auch mit nichtrelationalen Datenbanken?
- Welche alternativen NoSQL-Datenbanken gibt es?

Im zurückliegenden Kapitel sind wir durch nahezu sämtliche Komponenten und Schichten von Rails gegangen und haben eine solide Applikation erstellt. Doch was ist, wenn das nicht ausreicht?

Webapplikationen mit einer großen Anzahl an Besuchern wird im Blick auf gute Performanz vieles abverlangt. Da gerade Festplatten- und Datenbankzugriffe viel Zeit benötigen, ist dies oftmals die erste Stelle, an der Optimierungen erforderlich werden.

Um Sie in die Lage zu versetzen, skalierbare Anwendungen erstellen zu können, die auch hohen Anforderungen gerecht werden, zeigen wir in diesem Kapitel, wie Ruby on Rails mit hochperformanten Such- und Datenbanklösungen eingesetzt wird.

## 4.1 Suchen mit Apache Solr

Apache Solr ist ein HTTP-Frontend für die Java-Volltextsuche Lucene. Der Server läuft als Java-Applikation in einem Servlet-Container, wie zum Beispiel Tomcat, und wird über eine REST-ähnliche Schnittstelle angesprochen.

Solr bietet neben einer obligatorischen Volltextsuche sowie guter Performanz weitere nützliche Features: Facetten, Highlighting der Suchergebnisse, Replikation und viele mehr.

### 4.1.1 Solr in das Beispielprojekt einbinden

Im kommenden Abschnitt erweitern wir die Beispielanwendung um eine Suche, mit der Bookmarks gefunden werden können. Dafür verwenden wir das RubyGem sunspot. Diese Bibliothek erlaubt Models für Solr zu konfigurieren.

Stefan Sprenger / Kieran Hayes, Ruby on Rails 3.1 Expertenwissen, dpunkt.verlag, ISBN 978-3-89864-697-0

### Installation

Als Erstes müssen wir Sunspot und eine Erweiterung von Kaminari dem `Gemfile` hinzufügen:

*Listing 4.1 Einfügen in das Gemfile*

```
gem "sunspot_rails", "~> 1.2.1"
gem "sunspot_with_kaminari", "~> 0.1.0"
```

Nach der Installation mit

```
bundle install
```

bleiben wir auf der Kommandozeile und rufen

```
bundle exec rails generate
```

auf. Dadurch werden alle verfügbaren Generatoren aufgelistet, so auch der neu hinzugekommene Generator `sunspot_rails:install`. Diesen stoßen wir an:

```
bundle exec rails generate sunspot_rails:install
```

Der Generator legt im Verzeichnis `config` die Konfigurationsdatei `sunspot.yml` an.

#### Solr-Server

Sunspot legt im Root-Verzeichnis der Rails-Anwendung den Ordner `solr` an, in dem sich unter anderem die Daten und Konfigurationen von Solr befinden. Das Gem selbst hat Solr inklusive des Servers Jetty integriert, sodass für die lokale Entwicklung zunächst kein weiterer Installationsaufwand entsteht – alles kann out-of-the-box verwendet werden.

Will man später einen eigenen Server für Solr bereitstellen, muss man nur noch in der Konfigurationsdatei von Sunspot die Serverdaten hinterlegen sowie die Indexierung neu anstoßen.

In dieser Konfigurationsdatei können die Verbindungsdaten zu Solr, abhängig von der Rails-Umgebung, hinterlegt werden:

*Listing 4.2 Konfigurationsdatei von sunspot*

```
production:
  solr:
    hostname: localhost
    port: 8983
    log_level: WARNING

development:
  solr:
    hostname: localhost
    port: 8982
    log_level: INFO
```

```
test:
  solr:
    hostname: localhost
    port: 8981
    log_level: WARNING
```

Sunspot stellt eine Reihe an Rake-Tasks zur Verfügung, um den Solr-Server zu verwalten:

- Durch Aufruf von `rake sunspot:reindex` können die verwalteten Models neu indexiert werden.
- `rake sunspot:solr:run` startet den Solr-Server im Vordergrund.
- Möchte man den Server hingegen im Hintergrund starten, ist der Rake-Task `rake sunspot:solr:start` zu verwenden.
- Am Ende wird der Server mithilfe von `rake sunspot:solr:stop` gestoppt.

Wir starten den mitgelieferten Solr-Server mittels

```
bundle exec rake sunspot:solr:start
```

Die Logdatei des Dienstes befindet sich unter `log/sunspot-solr-development.log`.

### Model-Konfiguration

Da in den Suchergebnissen die gespeicherten Links ausgegeben werden sollen, konfigurieren wir das Link-Model für die Suche. Dazu öffnen wir das Model und spezifizieren innerhalb der Klassendefinition den Block `searchable`, der die zu indexierenden Model-Attribute enthält:

*Listing 4.3 Indexieren der Links*

```
class Link < ActiveRecord::Base
  ...

  searchable do
    text :url, :boost => 2.0

    text :bookmark_titles do
      bookmarks.map { |bookmark| bookmark.title }
    end

    text :bookmark_notes do
      bookmarks.map { |bookmark| bookmark.notes }
    end
```

Stefan Sprenger / Kieran Hayes, Ruby on Rails 3.1 Expertenwissen, dpunkt.verlag, ISBN 978-3-89864-697-0

```
    text :tags do
      bookmarks.map do |bookmark|
        bookmark.tags.map { |tag| tag.name }
      end.flatten.uniq
    end
  end
end
```

In der ersten Zeile des Blocks wird das Attribut `url` als Textfeld spezifiziert. Der Parameter `:boost => 2.0` gibt diesem Feld eine zweimal so hohe Bedeutung im Vergleich zu den anderen Feldern.

Da das Link-Model selbst nur die URL und die Anzahl der Bookmarks speichert, müssen Titel, Beschreibung und Tags über die Beziehung ausgelesen werden. Wichtig ist, dass die Felder als Array übergeben werden. Glücklicherweise ist das dank Ruby kein Problem und wir können beim verschachtelten Block des `tags`-Attributs mittels der Methoden `flatten` und `uniq` sicherstellen, dass Solr die Daten im korrekten Format erhält.

Nachdem das Model für die Suche konfiguriert wurde, können wir uns mit der Ausgabe der Suchergebnisse beschäftigen.

### 4.1.2 Suchen im Controller

Dazu öffnen wir den Controller PagesController und fügen die Action `search` hinzu:

***Listing 4.4***
*Suche im Controller*

```
class PagesController < ApplicationController
  ...

  def search
    if params[:search].present? && params[:search][:q].present?
      @query = params[:search][:q]
      @search = Sunspot.search(Link) do
        fulltext(params[:search][:q])
        paginate(:page => params[:page], :per_page => 10)
      end
    else
      redirect_to root_url
    end
  end
end
```

Dort überprüfen wir, ob der Parameter `[:search][:q]` gesetzt ist. Ist das der Fall, wird die Suche angestoßen, andernfalls wird der Benutzer zur Startseite weitergeleitet.

Innerhalb des Blocks `Sunspot.search` werden die von Sunspot auszuführenden Aktionen definiert. In diesem ersten Beispiel wird mittels `fulltext` eine Volltextsuche auf dem Suchwort durchgeführt. Die Methode `paginate` stellt den Suchergebnissen die von Kaminari gewohnten Hilfsmethoden zur Verfügung.

Zuletzt müssen wir noch einen Eintrag im Routing vornehmen:

*Listing 4.5*
*Routing für die Suche*

```
Beispielanwendung::Application.routes.draw do
  ...

  match "/search", :to => "pages#search"
end
```

### 4.1.3 Das Suchformular

In der Beispielanwendung soll das Suchformular auf jeder Seite verfügbar sein. Dazu öffnen wir das Layout, zu finden unter `app/views/layouts/application.html.erb`, und fügen im oberen Teil der Sidebar das Formular ein:

*Listing 4.6*
*Suchformular*

```
<div id="sidebar">
  <div id="search">
    <%= form_tag(search_path, :method => :get) do %>
      <%= text_field :search, :q, :value => @query %>
      <%= submit_tag "Search" %>
    <% end -%>
    <br class="clear" />
  </div>

  ...
</div>
```

Aber wie werden eigentlich die Suchergebnisse angezeigt? Hierfür erstellen wir ein View-Template, das in der Datei `app/views/pages/search.html.erb` Platz findet und folgenden Inhalt enthält:

*Listing 4.7*
*View-Template der Suchergebnisse*

```
<h2>Search</h2>
<% @search.each_hit_with_result do |hit, link| %>
  <div class="bookmark">
    <div class="hit-score">
      <%= hit.score %>
    </div>
    <%=link_to link.url, link.url, :target => :blank %>
    <span class="host">
      <%= URI.parse(link.url).host %>
    </span>
```

Stefan Sprenger / Kieran Hayes, Ruby on Rails 3.1 Expertenwissen, dpunkt.verlag, ISBN 978-3-89864-697-0

```
      <% link.bookmarks.limit(3).each do |bookmark| %>
        <div class="saved-by">
        <h4><%= bookmark.title %></h4>
          saved
          <%= distance_of_time_in_words link.↩
                  created_at, Time.now %>
          ago by <%= link_to bookmark.user.↩
                  login, user_path(bookmark.user) %>
        </div>
      <% end %>
    </div>
    <br class="clear" />
  <% end %>
  <%= paginate @search %>
```

Über den von Sunspot bereitgestellten Enumerator `each_hit_with_result` können wir an die Suchergebnisse gelangen. Das Objekt `hit` gibt dabei über `hit.score` die Relevanz des Suchresultats zurück.

Die Paginierung funktioniert übrigens wie bei gewöhnlichen ActiveRecord-Objekten.

### 4.1.4 Indexierung von Daten

Da wir im Link-Model die Methode `searchable` hinzugefügt haben, sorgt Sunspot dafür, dass Änderungen an Models automatisch vom Solr-Server bemerkt werden. Da in unserem Fall die Daten aber hauptsächlich aus den Bookmarks stammen, müssen wir dieses Model mithilfe eines Observers überwachen, um bei Änderungen den Suchindex aktuell zu halten. Dazu rufen wir auf der Konsole

```
bundle exec rails generate observer Bookmark
```

auf. Wir öffnen den generierten Observer und ergänzen ihn um die Methode `after_save`. Damit stoßen wir die Indexierung des assoziierten Link-Models an, sobald ein Bookmark gespeichert wurde:

***Listing 4.8*** *Observer des Bookmark-Models*

```
class BookmarkObserver < ActiveRecord::Observer
  def after_save(bookmark)
    Sunspot.index!(bookmark.link.reload)
  end
end
```

Um den Observer zu aktivieren, sollten wir nicht vergessen, ihn der Anwendungskonfiguration, zu finden in `config/application.rb`, hinzuzufügen:

*Listing 4.9*
*Observer aktivieren*

```
module Beispielanwendung
  class Application < Rails::Application
    ...

    config.active_record.observers = :bookmark_observer
  end
end
```

Anschließend muss der Server neu gestartet werden, damit die Änderungen aktiv werden.

**Wichtig: Neu indexieren nicht vergessen**

Nach Änderungen an der Suchkonfiguration sollte man nicht vergessen, die Daten neu zu indexieren, da es ansonsten zu einem falschen Suchergebnis kommen kann. Dazu wird auf der Konsole der Rake-Task

```
bundle exec rake sunspot:solr:reindex
```

ausgeführt.

### 4.1.5 Highlighting der Suchergebnisse

Als Nächstes heben wir die mit dem Suchbegriff übereinstimmenden URLs hervor. Dazu müssen wir zunächst in der Suchkonfiguration des Link-Models die Konfiguration für das `url`-Attribut anpassen:

*Listing 4.10*
*Attribut komplett in Solr speichern*

```
class Link < ActiveRecord::Base
  ...

  searchable do
    text :url, :boost => 2.0, :stored => true

    ...
  end
end
```

Die Option `stored` stellt sicher, dass das gesamte Attribut im Solr-Index gespeichert wird. Das ist Voraussetzung, damit das Highlighting korrekt funktioniert.

Zusätzlich muss Highlighting aktiviert werden. Dazu passen wir die Action `search` des PagesControllers an:

Stefan Sprenger / Kieran Hayes, Ruby on Rails 3.1 Expertenwissen, dpunkt.verlag, ISBN 978-3-89864-697-0

*Listing 4.11 Highlighting in Abfrage aktivieren*

```
class PagesController < ApplicationController
  ...

  def search
    if params[:search].present? && params[:search][:q].present?
      @query = params[:search][:q]
      @search = Sunspot.search(Link) do
        fulltext(params[:search][:q]) do
          highlight :url
        end
        paginate(:page => params[:page], :per_page => 10)
      end
    else
      redirect_to root_url
    end
  end
end
```

Da wir Änderungen an der Konfiguration vorgenommen haben, müssen wir die Indexierung anstoßen:

```
bundle exec rake sunspot:solr:reindex
```

Innerhalb der View stehen die Hervorhebungen über das Objekt `hit` zur Verfügung:

*Listing 4.12 Highlighting in der View*

```
<% if hit.highlights(:url).present? %>
   <%= link_to raw(highlight_url(hit.highlights(:url).first)),↩
                hit.stored(:url).first,↩
                :target => :blank %>
<% else %>
  <%= link_to hit.stored(:url).first,↩
               hit.stored(:url).first,↩
               :target => :blank %>
<% end -%>
```

Dafür muss auch eine View-Helper-Methode erstellt werden. Wir öffnen den PagesHelper unter `app/helpers/pages_helper.rb` und fügen die Methode `highlight_url` hinzu:

*Listing 4.13 PagesHelper*

```
module PagesHelper
  def highlight_url(url)
    url.format do |word|
      "<span class='highlight'>#{word}</span>"
    end
  end
end
```

Zu guter Letzt sollte man nicht vergessen, entsprechende Styles für die CSS-Klasse `highlight` zu erstellen, um die Übereinstimmungen auch optisch hervorzuheben.

### 4.1.6 Suche verfeinern mit Facetten

Eines der spannendsten Features von Apache Solr sind dynamische Facetten. Bei herkömmlichen Suchmaschinen gibt es oft ein Formular, in das detaillierte Suchkriterien eingegeben werden können. Das Problem bei diesem Ansatz liegt in der Tatsache, dass man als Benutzer nicht weiß, ob die übergebenen Parameter zum Erfolg führen.

Facetten hingegen funktionieren umgekehrt. Sie erscheinen nur, wenn es auch Ergebnisse für den Suchbegriff gibt. Gerade bei Onlineshops ist das ein komfortabler Weg, den Benutzer gezielt zum gewünschten Produkt zu führen. Man kann so zum Beispiel nach dem Wort `Fernseher` suchen und bekommt als Facetten `Bildschirmgröße` und `Stromverbrauch` angeboten. Wählt man nun eine der Facetten aus, werden nur noch Suchergebnisse angezeigt, die die Facette erfüllen. Natürlich können auch mehrere Facetten gleichzeitig angewendet werden.

Facetten werden bei Sunspot innerhalb des Controllers ausgewertet. In der Beispielanwendung soll eine Facette eingeführt werden, mit der ausschließlich Links einer bestimmten Domain, beispielsweise `rubyonrails.org`, angezeigt werden können. Da wir bis jetzt noch gar nicht die Domain eines Links speichern, müssen wir hierfür zunächst das Model anpassen. Dazu fügen wir das Feld `domain` innerhalb der Sunspot-Konfiguration hinzu, das wir durch Auswertung des Links füllen:

***Listing 4.14***
*Domain als zusätzliches Suchfeld*

```
class Link < ActiveRecord::Base
  ...

  searchable do
    string :domain do
      URI.parse(url).host
    end

    ...
  end
end
```

Es können bei Solr auch Felder indexiert werden, die nicht Attribut des Objekts sind. Wichtig ist dabei nur, dass bei Feldänderungen nicht vergessen wird, den Index von Solr neu aufzubauen.

Stefan Sprenger / Kieran Hayes, Ruby on Rails 3.1 Expertenwissen, dpunkt.verlag, ISBN 978-3-89864-697-0

Um Facetten im Controller zu aktivieren, muss dieser lediglich um zwei Zeilen ergänzt werden:

*Listing 4.15* *Facette im Controller*

```
class PagesController < ApplicationController
  ...

  def search
    if params[:search].present? && params[:search][:q].present?
      @query = params[:search][:q]
      @search = Sunspot.search(Link) do
        fulltext(params[:search][:q]) do
          highlight :url
        end
        facet :domain, :minimum_count => 1
        with :domain, params[:search][:domain]↩
            if params[:search][:domain].present?
        paginate(:page => params[:page], :per_page => 10)
      end
    else
      redirect_to root_url
    end
  end
end
```

Der Methode `facet` übergeben wir den Feldnamen als Ruby-Symbol: `:domain`. Mit dem Parameter `:minimum_count` wird festgelegt, dass die Facette nur angezeigt werden soll, wenn sie auf mindestens ein Suchergebnis zutrifft.

Die eigentliche Suche steuern wir über die Methode `with`. Ihr übergeben wir ebenfalls den Feldnamen, aber nur falls die Facette aktiv ist und der GET-Parameter existiert.

Um die Realisierung der Facette abzuschließen, müssen wir sie noch der View `app/views/pages/search.html.erb` hinzufügen:

*Listing 4.16* *Facette domain in der Sidebar*

```
<%= content_for :sidebar do %>
  <% if @search.present? &&↩
     @search.facet(:domain).present? &&↩
     @search.facet(:domain).rows.count > 0 ||↩
     params[:domain].present? %>
  <h2>Domain</h2>
    <ul class="facets">
    <% @search.facet(:domain).rows.each do |result| %>
      <li>
        <%= link_to "#{result.value} (#{result.count})",
            search_path({:search => {:domain => result.value, ↩
                :q => @query}}) %>
```

```
        <%= link_to "-remove",
            search_path({:search => {:domain => nil, ↩
                :q => @query}}),
            :class => "remove" if params[:search][:domain].↩
                present? %>
      </li>
      <% end %>
    </ul>
  <% end %>
<% end %>
```

Wichtig ist, dass wir im Link die Facette integrieren. Der zweite Link hebt im Übrigen eine aktivierte Facette wieder auf, sodass der Benutzer hierdurch zur Ausgangssituation zurückkehrt.

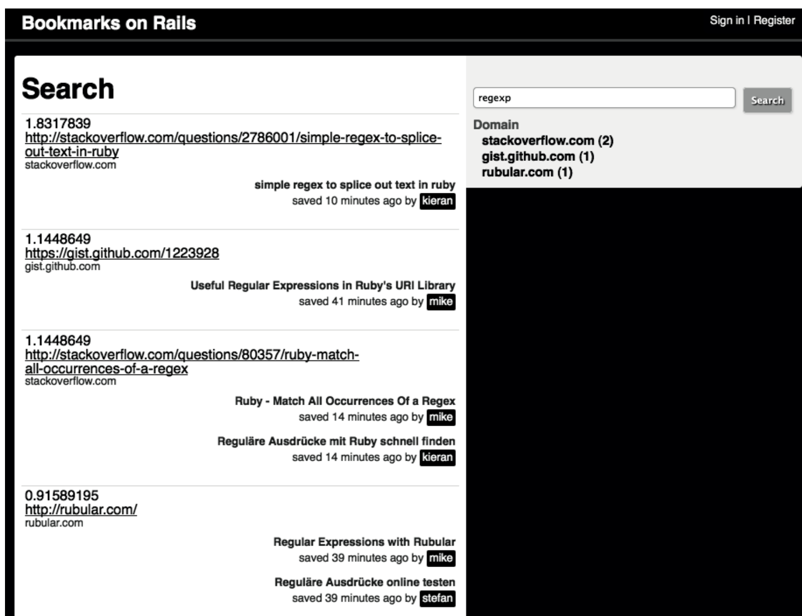


***Abbildung 4-1*** *Suchergebnisse mit Facetten*

## 4.1.7 Fazit

Wer hätte gedacht, dass sich so leicht eine skalierende und bewährte Suchlösung in das Beispielprojekt integrieren lässt? Einen großen Anteil daran hat sicherlich Sunspot: Hiermit lässt sich die Indexierung auf Model-Ebene konfigurieren und steuern.

Wir mussten noch nicht einmal einen Server installieren, da das Gem alle für die Entwicklung benötigten Werkzeuge und Abhängigkeiten mitbringt.

Stefan Sprenger / Kieran Hayes, Ruby on Rails 3.1 Expertenwissen, dpunkt.verlag, ISBN 978-3-89864-697-0

Mit dem Einsatz eines Observers ist der Suchindex stets auf dem aktuellsten Stand, mehr zu diesem Feature findet sich übrigens im Kapitel 7.1.4.

Im nächsten Kapitel widmen wir uns ganz der Datenhaltung und steigen auf eine moderne, für Webanwendungen optimierte Datenbank um. Die Suche sowie alle anderen implementierten Funktionalitäten funktionieren dank der Modularität von Ruby on Rails natürlich auch weiterhin wie gewohnt und harmonieren mit den neuen Komponenten.

#### Weiterführende Informationen

- Projektwebseite von Apache Solr: `http://lucene.apache.org/solr/`
- Projektwebseite von Sunspot: `http://outoftime.github.com/sunspot/`
- Facetten beim Heise-Preisvergleich: `http://www.heise.de/preisvergleich/?cat=tvlcd`

## 4.2 Rails 3 und NoSQL

*Wird meistens als Not Only SQL verwendet.*

Seit einigen Jahren erlebt die Webentwicklungsszene eine Revolution in Sachen Datenbanken. Aufgrund zahlreicher Probleme und Unzufriedenheiten mit relationalen Datenbanken entstand eine Bewegung, die unter dem Namen NoSQL bekannt ist. NoSQL-Datenbanken versuchen die Bereiche zu besetzen, in denen relationale Datenbanken – allen voran MySQL – bisher schlecht abschnitten. Dabei gleicht keine NoSQL-Datenbank der anderen, jede versucht sich auf bestimmte Anwendungsfälle zu spezialisieren und gewissermaßen eine Nische zu besetzen.

Je nach Datenbank können die Vorteile wie folgt definiert werden:

- Die meisten NoSQL-Datenbanken wurden auf eine sehr gute Skalierbarkeit optimiert. Auf veränderte Anforderungen hinsichtlich Datenaufkommen oder Anzahl der Zugriffe kann leicht durch Hinzufügen oder Entfernen von Hardware reagiert werden.
- Durch ein flexibles, schemaloses Datenmodell sind Datensätze gleichen Typs nicht an die starren Restriktionen des relationalen Modells gebunden.
- Beziehungen zwischen Datensätzen müssen in relationalen Datenbanken mühsam über Fremdschlüssel verwaltet werden. Fast alle NoSQL-Datenbanken stellen hierfür performante und intuitiv benutzbare Lösungen zur Verfügung.

- Oftmals werden Datenbanktransaktionen, die von relationalen Datenbanken implementiert werden, in der Praxis nicht benötigt. NoSQL-Datenbanken setzen hier an und erzielen durch das Weglassen von Transaktionen enorme Geschwindigkeitsvorteile.
- Der Cluster-Betrieb von relationalen Datenbanken wird üblicherweise im Master-Slave-Modus durchgeführt. Neben einer schlechten Skalierbarkeit ergibt sich daraus auch ein `Single Point of Failure`: Fällt der Master-Server aus, ist die gesamte Datenbank nicht funktionsfähig. Einige NoSQL-Datenbanken setzen hier von Grund auf mit einer auf den verteilten Betrieb ausgerichteten Architektur an.

ActiveRecord, das ORM-Framework von Ruby on Rails, wurde für den Einsatz mit relationalen Datenbanken entwickelt, weshalb es keine passenden Adapter für NoSQL-Datenbanken gibt. Seit Rails 3.0 existiert allerdings ActiveModel, das als Basis für viele NoSQL-Mapper-Frameworks verwendet wird, wodurch eine einheitliche Schnittstelle über Frameworkgrenzen hinweg existiert.

**NoSQL und der Begriff ORM**

Im NoSQL-Kontext macht der Begriff ORM keinen Sinn, da es sich nicht um relationale Datenbanken handelt. Aus diesem Grund verwenden wir in den meisten Fällen Objekt-Mapper oder Objekt-Dokument-Mapper, wenn wir über das Pendant zu ActiveRecord für eine nichtrelationale Datenbank sprechen.

Auch wenn sich NoSQL-Datenbank nicht miteinander vergleichen lassen, so können doch vier Gruppen gebildet werden, mit denen sie kategorisiert werden können.

### Document-Stores

Document-Stores erlauben die Verwaltung von schemalosen, textbasierten Dokumenten. Üblicherweise wird innerhalb der Dokumente JSON für die Datenrepräsentation verwendet. Sie gelten als Allrounder in der NoSQL-Szene und eignen sich gut für das Speichern von gewöhnlichen Model-Datensätzen, wie beispielsweise Benutzer einer Applikation. Bekannte Vertreter dieser Kategorie sind MongoDB und CouchDB.

Stefan Sprenger / Kieran  Hayes, Ruby on Rails 3.1 Expertenwissen, dpunkt.verlag, ISBN 978-3-89864-697-0

#### Key-Value-Stores

Häufig benötigen Programme nur einen sehr simplen Zugriff auf Daten: Einem Datensatz (Value) ist ein Schlüssel (Key) zugeordnet, über den er identifiziert wird. Einfach gesagt implementieren Key-Value-Stores das Prinzip von assoziativen Arrays als Datenbanksystem.

Sie eignen sich für flache Datenstrukturen, wie beispielsweise Caches. Sowohl Redis als auch Riak gehören zu dieser Gruppe.

#### Wide-Column-Stores

Bei Wide-Column-Stores werden, im Gegensatz zu relationalen Datenbanken, Elemente einer Tabelle nicht untereinander (zeilenorientiert), sondern nebeneinander (spaltenorientiert) gespeichert. Daraus ergeben sich, je nach Anwendungsfall, Vorteile für das Caching und Abfragen der verwalteten Daten. Auch bei dieser Kategorie ist das Datenmodell an kein festes Schema gebunden und kann während des Betriebs geändert werden.

Googles Datenbank BigTable war Vorreiter in dieser Gruppe und beeinflusste viele Wide-Column-Stores: unter anderem die Open-Source-Implementierungen Cassandra und HBase.

#### Graph-Databases

Daten einer Graph-Database werden in Graphen- oder Baumstrukturen verwaltet. In den 80er-Jahren gab es bereits erste Vorläufer, die zur Verwaltung von Netzen eingesetzt wurden. Durch die Entstehung des semantischen Web und dem Fortschreiten von sozialen Netzwerken, die oftmals komplizierte Freundschaftsbeziehungen mit sich bringen, ist der Bedarf an Graphenstrukturen in Datenbanken gewachsen.

Auch andere Anwendungen, wie beispielsweise Suchmachinen, die auf Basis von komplizierten Algorithmen Dokumente bewerten müssen, benötigen diesen Datenbanktyp. Bekannte Vertreter aus dem Open-Source-Bereich sind Neo4j und sones.

#### Weitere Informationen

- Vergleich gängiger NoSQL-Datenbanken:
  `http://kkovacs.eu/cassandra-vs-mongodb-vs-couchdb-vs-redis`

### 4.2.1 Wahl der NoSQL-Datenbank

Für die Beispielanwendung haben wir uns für den Document-Store MongoDB als Datenbank entschieden. Dokumente werden bei Mon-

goDB im BSON-Format, einer binären Erweiterung von JSON, verwaltet. Die Struktur der Dokumente ist an kein festes Schema gebunden, somit können zwei Dokumente desselben Typs unterschiedliche Attribute besitzen.

Durch eine intuitive Benutzung sowie eine gute Integration in Ruby on Rails bietet sich MongoDB auch für Entwickler an, die bisher keine Erfahrungen mit nichtrelationalen Datenbanken sammeln konnten. Nicht umsonst scheint es die beliebteste NoSQL-Datenbank für Rails-Anwendungen zu sein.

Eine performante Abfragesprache erlaubt eine Verwendung ähnlich wie MySQL, weshalb nur wenige Änderungen am Datenmodell notwendig sind. Darüber hinaus ist Sharding fest integriert, wodurch die Leistungsfähigkeit der Datenbank bei Bedarf durch das Hinzufügen weiterer Serverinstanzen linear verbessert werden kann. Das ist nützlich, wenn der Benutzeransturm auf eine Applikation höher ausfällt als erwartet.

Ferner können durch die Definition von Sekundärindizes Datenbank-Anfragen an eigene Anforderungen angepasst werden.

**Horizontale Skalierung**

Der Begriff horizontale Skalierung beschreibt die Eigenschaft eines verteilten Systems, beispielsweise einer Datenbank, wenn durch das Hinzufügen von weiteren Hardwareressourcen die Leistungsfähigkeit linear verbessert wird.

Natürlich kann durch das Entfernen von Ressourcen auch der entgegengesetzte Effekt erreicht werden.

Neben den angesprochenen Funktionalitäten stellt MongoDB weitere bereit, die zwar nicht für die Beispielapplikation benötigt werden, aber in der Zukunft von Nutzen sein können:

- Mit GridFS steht eine performante Möglichkeit bereit, größere Dateien samt Metadaten in der Datenbank zu verwalten.
- MongoDB unterstützt den Map-Reduce-Ansatz, mit dem große Datenmengen parallel verarbeitet und aggregiert werden können.
- Analog zu Stored Procedures erlaubt MongoDB Stored JavaScript, um JavaScript-Abfragen auf dem Server abzulegen.

MongoDB unterstützt alle gängigen Betriebssysteme, beispielsweise Mac OS X, Linux oder Windows, wodurch es eine hohe Akzeptanz erreicht. Auch viele Cloud-Hosting-Betreiber, unter anderem das in der Ruby-Szene beliebte Heroku, bieten es an.

Stefan Sprenger / Kieran Hayes, Ruby on Rails 3.1 Expertenwissen, dpunkt.verlag, ISBN 978-3-89864-697-0

Eine Installationsanleitung für die drei großen Betriebssysteme findet sich online auf der offiziellen Webseite von MongoDB unter `http://www.mongodb.org/display/DOCS/Quickstart`. Nach der erfolgreichen Installation kann MongoDB, wie in der Anleitung angegeben, gestartet werden.

#### Weitere Informationen

- Projektwebseite: `http://www.mongodb.org`
- Heroku: `http://heroku.com`

### 4.2.2 MongoDB – Aufbau und Prinzipien

Die Basiseinheit, in der ein Datensatz in MongoDB abgebildet wird, ist das Dokument. Es wird im BSON-Format verwaltet und besteht aus einer Reihe von Key-Value-Paaren. Diese Paare repräsentieren zum einen selbst definierte Attribute und zum anderen von MongoDB intern verwendete Werte. Ein Beispiel hierfür ist die `id` eines Dokuments, mit der es systemweit eindeutig identifiziert wird. Solche von MongoDB vorbelegte Keys sind mit einem vorangestellten _ markiert und sollten nicht durch den Benutzer verändert werden.

***Listing 4.17*** *Beispieldokument in MongoDB*

```
{
  "_id" : ObjectId("4dc0fa30019633226c297d01"),
  "name" : "Ruby on Rails 3.1 Expertenwissen",
  "authors" : ["Stefan Sprenger", "Kieran Hayes"],
  "price" : 34.95,
  "year" : 2011
}
```

Für Attributwerte stehen nahezu alle auch in Ruby vorhandenen Datentypen zur Verfügung: von Integers über Gleitkommazahlen bis hin zu Strings und Arrays.

Will man einen Vergleich zum relationalen Modell ziehen, lassen sich MongoDBs Dokumente mit Zeilen von Tabellen vergleichen.

Eine Sammlung solcher Dokumente gleichen Typs würde einer Tabelle entsprechen und ist auch bekannt unter dem Begriff Collection. Es gibt jedoch auch einen wichtigen Unterschied: Collections haben im Gegensatz zu Tabellen kein festes Schema. Somit können Dokumente, die unterschiedliche Attribute besitzen, trotzdem der gleichen Collection angehören. Dadurch wird eine extrem große Flexibilität erreicht. Die größte Übereinstimmung zeigt sich im Begriff der Datenbank: Sowohl bei MongoDB als auch bei relationalen Datenbanken enthält eine Da-

tenbank mehrere Collections beziehungsweise Tabellen und dient somit als zusätzlicher Namensraum, um Daten einer Applikation zu isolieren.

Die Tabelle 4-1 stellt die gängigsten Begriffe von relationalen Datenbanken den Begriffen von MongoDB gegenüber, auch wenn diese natürlich nicht identisch verwendet werden können.

*Tabelle 4-1* *Gegenüberstellung der Begriffe von relationalen Datenbanken und MongoDB*

| Relationale Datenbank | MongoDB |
|---|---|
| Datenbank | Datenbank |
| Tabelle | Collection |
| Zeile einer Tabelle | Dokument |
| Spalte einer Tabelle | Key eines Dokuments |

MongoDB bietet ein wesentlich natürlicheres Datenmodell, das aufgrund seiner Schemafreiheit gerade für Webanwendungen geeignet ist.

### 4.2.3 Der Objekt-Dokument-Mapper Mongoid

Der Objekt-Dokument-Mapper Mongoid ist die zurzeit wohl verbreitetste Integration von MongoDB in Ruby-Anwendungen, wobei es gleichzeitig auf den Einsatz mit Rails optimiert ist. Mongoid wurde 2009 zum ersten Mal der Öffentlichkeit vorgestellt und ist mittlerweile in Version 2.2 erhältlich. Dabei stand das Ziel, einen ähnlichen Funktionsumfang wie ActiveRecord anzubieten, während der Entwicklung stets an oberster Stelle.

Mongoid verwendet ActiveModel und besitzt somit eine perfekte Integration in Rails 3. Auch andere Bibliotheken, wie die beim Beispielprojekt verwendete Authentifizierungslösung devise, harmonieren mit dem Dokument-Mapper auf allen Ebenen.

Die Installation von Mongoid verläuft wie gewohnt einfach, es müssen lediglich zwei neue Abhängigkeiten in das `Gemfile` der Anwendung aufgenommen werden:

*Listing 4.18* *Einbinden von Mongoid*

```
source "http://rubygems.org"

...

gem "bson_ext", "~> 1.3.1"
gem "mongoid", "~> 2.2.0"
```

Bei `bson_ext` handelt es sich um eine C-Implementierung des BSON-Formats, die Performanzgewinne gegenüber der mitgelieferten Ruby-Variante erreicht.

Stefan Sprenger / Kieran Hayes, Ruby on Rails 3.1 Expertenwissen, dpunkt.verlag, ISBN 978-3-89864-697-0

Für das Abschließen der Installation muss nur noch

```
bundle install
```

auf der Kommandozeile aufgerufen werden.

Mongoid wird über eine eigene Konfigurationsdatei an individuelle Anforderungen angepasst. Diese lässt sich mithilfe von

```
bundle exec rails generate mongoid:config
```

generieren und ist in `config/mongoid.yml` verfügbar. Hier können Verbindungsdaten zum MongoDB-Server im YAML-Format eingepflegt werden, für die lokale Entwicklung sollte aber in den meisten Fällen die mitgelieferte Konfiguration ausreichen.

Um MongoDB als Datenbankserver für die Beispielanwendung zu verwenden, bedarf es nur weniger Änderungen. Den ersten Schritt, die Installation von Mongoid, haben wir bereits erfolgreich gemeistert. Im nächsten Schritt müssen die Models an den neuen Objekt-Dokument-Mapper angepasst werden.

#### Weitere Informationen

- Projektwebseite: `http://mongoid.org`
- Dokumentation: `http://mongoid.org/docs.html`

### 4.2.4 Ein bestehendes Model mit Mongoid verwalten

In diesem Abschnitt zeigen wir anhand des Bookmark-Models, wie der Umstieg von ActiveRecord zu Mongoid durchgeführt wird.

MongoDBs Dokumente besitzen kein festes Schema, weshalb Migrationen überflüssig werden. Stattdessen werden Attribute innerhalb des Models spezifiziert.

Zunächst öffnen wir das Model `app/models/bookmark.rb` und entfernen die Abhängigkeit zu ActiveRecord. Anstelle der Vererbungsbeziehung zur Klasse `ActiveRecord::Base` binden wir das Modul `Mongoid::Document` ein:

```
class Bookmark
  include Mongoid::Document

  ...
end
```

Da Mongoid als Modul eingebunden wird, besteht im Gegensatz zu ActiveRecord die Möglichkeit, von einer Klasse zu erben.

#### Attribute

Nun geht es zu den Attributen: Im aktuellen Datenbankschema des Models, das zum Beispiel in der Datei `db/schema.rb` eingesehen werden kann, ist ersichtlich, dass das Bookmark-Model folgende Attribute besitzt:

- user_id
- link_id
- title
- notes
- created_at
- updated_at

Die Attribute `user_id` und `link_id` werden für die Abbildung der Beziehungen genutzt. `created_at` und `updated_at` sind interne Zeitstempel, die den Zeitpunkt des Erstellens und der letzten Änderung speichern. Somit sind die eigentlichen, direkt verwendeten Attribute lediglich `title` und `notes`.

Beschäftigen wir uns zunächst mit den letzten beiden. Für beide Attribute wird String als Datentyp verwendet, das soll auch weiterhin beibehalten werden. Für die Definition der Model-Attribute liefert Mongoid die Methode `field` mit. Ihr wird der Name und Datentyp übergeben:

```
class Bookmark
  include Mongoid::Document

  field :title, :type => String
  field :notes, :type => String

  ...
end
```

***Listing 4.19*** *Definition von Attributen*

Neben Strings unterstützt Mongoid auch weitere Datentypen:

- Array
- BigDecimal
- Boolean
- Date
- DateTime
- Float
- Hash
- Integer
- Symbol
- Time

Stefan Sprenger / Kieran Hayes, Ruby on Rails 3.1 Expertenwissen, dpunkt.verlag, ISBN 978-3-89864-697-0

### Beziehungen zwischen Models

Mongoid unterscheidet im Gegensatz zu ActiveRecord zwischen zwei Arten von Beziehungen: referenzierte und eingebettete Beziehungen.

Referenzierte Beziehungen funktionieren wie gehabt: Die Mitglieder der Beziehung werden in zwei unterschiedlichen Dokumenten in MongoDB gespeichert.

Zur Spezifizierung innerhalb der Models stehen die Methoden `has_one`, `has_many`, `belongs_to` und `has_many_and_belongs_to` zur Verfügung. Übrigens werden die für referenzierte Beziehungen benötigten Felder automatisch von Mongoid angelegt.

Davon unterscheiden sich eingebettete Beziehungen: Wie der Name schon sagt, wird bei dieser Art ein Model in ein anderes eingebettet. Für die Speicherung in MongoDB wird somit auch nur ein Dokument verwendet. Dieser Beziehungstyp wird mithilfe der Methoden `embeds_many` (1-n) und `embeds_one` (1-1) auf Seite des einbindenden Models und mit `embedded_in` auf Seite des eingebetteten Models aufgebaut. Die Verwendung von eingebetteten Beziehungen macht immer dann Sinn, wenn ein Mitglied der Beziehung nur über das andere verwendet wird und nicht über einen eigenen Identifikator oder außerhalb der Beziehung verfügbar sein muss.

Aufgrund MongoDBs Datenmodell ist das Tagging-Model, das wir als Hilfe zur Modellierung der Beziehung verwendet haben, nicht mehr sinnvoll. Wir löschen die Datei `app/models/tagging.rb` und ersetzen in `app/models/bookmark.rb`

```
has_many :taggings, :dependent => :destroy
has_many :tags, :through => :taggings
```

mit

```
has_and_belongs_to_many :tags
```

und im Tag-Model, zu finden unter `app/models/tags.rb`,

```
has_many :taggings, :dependent => :destroy
has_many :bookmarks, :through => :taggings
```

mit

```
has_and_belongs_to_many :bookmarks
```

Bei Beziehungen unterstützt Mongoid die Option `counter_cache` nicht. Wir entfernen sie aus dem Bookmark-Model:

```
class Bookmark
  ...

  belongs_to :link
end
```

Natürlich sollte auch das Tag-Model auf den Betrieb mit Mongoid umgestellt werden.

#### Zeitstempel

Auch Mongoid unterstützt die automatische Erstellung und Pflege von Zeitstempeln beim Anlegen und Aktualisieren von Datensätzen. Hierfür muss das Modul `Mongoid::Timestamps` eingebunden werden:

***Listing 4.20*** *Automatische Zeitstempel*

```
class Bookmark
  include Mongoid::Document
  include Mongoid::Timestamps

  field :title, :type => String
  field :notes, :type => String

  ...
end
```

#### Validierung

Für die Validierung setzt Mongoid auf die Implementierung von ActiveModel, die auch von ActiveRecord verwendet wird. Dadurch müssen bestehende Validierungen nicht verändert werden, das Erlernen einer neuen Syntax ist überflüssig.

Nach dieser kurzen Einführung können auch die restlichen Models, außer das User-Model, auf den Einsatz mit Mongoid umgestellt werden. Es ist empfehlenswert, die Umstellung bei allen Models durchzuführen, da es ansonsten zu Problemen bei Beziehungen zwischen ActiveRecord- und Mongoid-Models kommt. Die Benutzerauthentifizierung weicht von dieser Vorgehensweise etwas ab, worauf im folgenden Abschnitt eingegangen wird.

### 4.2.5 Benutzerverwaltung auf MongoDB umstellen

Um devise mitzuteilen, dass wir Mongoid als Framework für die Verwaltung der Models verwenden, öffnen wir die Konfiguration unter `config/initializers/devise.rb`. Hier ist die Zeile

Stefan Sprenger / Kieran Hayes, Ruby on Rails 3.1 Expertenwissen, dpunkt.verlag, ISBN 978-3-89864-697-0

```
require "devise/orm/active_record"
```

durch

```
require "devise/orm/mongoid"
```

zu ersetzen.
Anschließend muss nur noch das User-Model an Mongoid angepasst werden. Dazu ist die Vererbungsbeziehung, wie auch schon bei den anderen Models, zu entfernen und das Modul `Mongoid::Document` einzubinden.
Die Attribute, die devise zur Speicherung der Benutzer benötigt, erstellt es selbst. Wir müssen nur `email` und `login` definieren:

```
class User
  include Mongoid::Document
  include Mongoid::Timestamps

  field :email, :type => String
  field :login, :type => String

  ...
end
```

Da wir zuvor die Funktionalität von devise erweitert haben, sodass sich Benutzer auch mit ihrem Nicknamen authentifizieren können, müssen wir diese Stelle im User-Model anpassen:

```
class User
  ...

  def self.find_for_database_authentication(conditions)
    login = conditions.delete(:login)
    self.any_of({ :login => login }, { :email => login }).first
  end
end
```

In der Lokalisierungsdatei `config/locales/en.yml` muss noch der Key `activerecord` durch `mongoid` ersetzt werden, damit das Label für das Login-Feld korrekt gesetzt wird.

### 4.2.6 Abfragen

Im Grunde ähneln sich die Schnittstellen von AREL und Mongoid. Bei beiden Bibliotheken werden Models um die Methode `where` erweitert, mit der Objekte nach ihren Attributen gefiltert werden können. Bei Mongoid wird die Abfrage allerdings nicht sofort ausgeführt, man erhält zunächst ein Objekt vom Typ `Mongoid::Criteria`. Dieses Objekt

besitzt unter anderem die Methoden `entries`, `first` und `last`, mit denen auf alle, das erste oder das letzte Ergebnis zugegriffen werden kann.

Zunächst öffnen wir `app/controllers/tags_controller.rb` und passen die Action `show` an:

*Listing 4.21 Anpassen der show-Action im TagsController*

```
class TagsController < ApplicationController
  ...

  def show
    @tag = Tag.where(:name => params[:id]).first
    @bookmarks = Bookmark.where(:tag_ids => @tag.id).↩
                    page(params[:page]).per(5)
  end
end
```

Mongoid unterstützt die Sortierung von Datensätzen über die Methode `order_by`. Im PagesController muss deshalb die Abfrage abgeändert werden:

*Listing 4.22 Anpassen der Abfrage im PagesController*

```
class PagesController < ApplicationController
  def index
    @bookmarks = Bookmark.order_by(:created_at.desc).↩
                   page(params[:page]).per(5)
  end

  ...
end
```

Analog dazu passen wir den UsersController an:

*Listing 4.23 Anpassen der Abfrage im UsersController*

```
class UsersController < ApplicationController
  def show
    @user = User.find(params[:id])
    @bookmarks = @user.bookmarks.order_by(:created_at.desc).↩
                   page(params[:page]).per(5)
  end
end
```

### 4.2.7 Observer

Im Kapitel 4.1 haben wir einen Observer für die Überwachung der Bookmarks erstellt. Da er für den Einsatz mit ActiveRecord generiert wurde, passen wir ihn an Mongoid an:

Stefan Sprenger / Kieran Hayes, Ruby on Rails 3.1 Expertenwissen, dpunkt.verlag, ISBN 978-3-89864-697-0

```
class BookmarkObserver < Mongoid::Observer
  def after_save(bookmark)
    Sunspot.index!(bookmark.link.reload)
  end
end
```

Die Konfigurationsoption in `config/application.rb` ändern wir zu

```
config.mongoid.observers = :bookmark_observer
```

ab.

### 4.2.8 Mongoid und Sunspot

Mit dem Gem `sunspot_mongoid` können Mongoid und Sunspot gemeinsam verwendet werden. Wir binden es in das `Gemfile` ein:

```
gem "sunspot_mongoid", "~> 0.4.1"
```

und installieren es.

Anschließend binden wir das Modul `Sunspot::Mongoid` in das Link-Model ein:

```
class Link
  include Mongoid::Document
  include Mongoid::Timestamps
  include Sunspot::Mongoid

  ...
end
```

Da der Solr-Index noch ActiveRecord-Objekte beinhaltet, leeren wir ihn auf der Rails-Konsole und indexieren anschließend alle Mongoid-Objekte:

```
Link.solr_remove_all_from_index!
Link.all.each { |link| link.index! }
```

### 4.2.9 Besonderheiten von Mongoid

Auf wenige Besonderheiten muss trotzdem noch geachtet werden. Da MongoDB keine relationale Datenbank ist, steht auch kein Join-Operator für die Verknüpfung von zwei Tabellen in Abfragen zur Verfügung. Stellen, an denen bisher auf diese Funktionalität gesetzt wird, müssen angepasst werden. In unserer Beispielanwendung betrifft das das User-Model und den BookmarksController. Öffnen wir zunächst `app/models/user.rb` und passen die Methode `saved_bookmark?` an:

*Listing 4.24*
*Anpassen des User-Models*

```
class User
  ...

  def saved_bookmark?(bookmark)
    if bookmark.present? && bookmark[:url].present?
      link = Link.where(:url => bookmark[:url]).first
      link.present? && bookmarks.where(:link_id => link.id).↩
                        first.present?
    end
  end
end
```

Anschließend öffnen wir `app/controllers/bookmarks_controller.rb` und ersetzen auch hier die Join-Abfrage innerhalb des Before-Filters `check_url`:

*Listing 4.25*
*Anpassen des BookmarksControllers*

```
class BookmarksController < ApplicationController
  ...

  def check_url
    if current_user.saved_bookmark?(params[:bookmark])
        link = Link.where(:url => params[:bookmark][:url]).first
        bookmark = current_user.bookmarks.↩
                    where(:link_id => link.id).first
        redirect_to edit_user_bookmark_url(:id => bookmark.id,
                :bookmark => params[:bookmark]),
                :flash => { :notice => "You already ↩
                            saved this URL." }
    end
  end
end
```

Auch der LIKE-Operator steht bei MongoDB nicht zur Verfügung. Die gleiche Funktionalität kann allerdings mit regulären Ausdrücken implementiert werden. Wir öffnen den TagsController und modifizieren die Action `index`:

*Listing 4.26*
*Anpassen des TagsControllers*

```
class TagsController < ApplicationController
  ...

  def index
    @tags = Tag.where(:name => /#{params[:term]}/).limit(10)

    ...
  end
end
```

Stefan Sprenger / Kieran Hayes, Ruby on Rails 3.1 Expertenwissen, dpunkt.verlag, ISBN 978-3-89864-697-0

Zudem unterscheidet sich die Syntax des ActiveRecord-Features `find_or_create_by_attribute`. Mongoid unterstützt die gleiche Funktionalität über die Methode `find_or_create_by`, der der Attributsname als Parameter übergeben wird. Wir öffnen `app/models/bookmark.rb` und passen die Methoden `assign_tags` und `set_link` an:

```
class Bookmark
  ...

  def assign_tags
    if @tag_list
      self.tags = @tag_list.gsub(/\s+/,"").split(/,/).↩
               uniq.map do |name|
        Tag.find_or_create_by(:name => name.strip)
      end
    end
  end

  def set_link
    self.link = Link.find_or_create_by(:url => @url)
  end
end
```

Jetzt sind wir auch schon fertig: In wenigen Schritten konnten wir die relationale Datenbank durch MongoDB, einen NoSQL-Vertreter, ersetzen. Um eine saubere Applikationsstruktur beizubehalten, ist es empfehlenswert, die verbleibenden Dateien von und Abhängigkeiten zu ActiveRecord zu entfernen.

### 4.2.10 Andere NoSQL-Datenbanken

Neben MongoDB gibt es einige weitere interessante Vertreter aus dem NoSQL-Spektrum, die ähnlich gut mit Ruby on Rails zusammenarbeiten und für manche Anwendungsfälle sogar besser geeignet sind. Wir stellen die Datenbanken Riak und Redis vor und zeigen, welche Vorteile sie besitzen.

#### Riak

Riak ist ein von Basho Inc. entwickelter Key-Value-Store, implementiert die Prinzipien von Amazon Dynamo und ist speziell auf den verteilten Einsatz in großen Datenbank-Clustern optimiert.

Zur Datenabfrage bietet Riak den Map-Reduce-Ansatz an, um große Datenmengen parallel verarbeiten zu können. Die Map-Reduce-Funktionen können in den Programmiersprachen JavaScript oder Erlang implementiert werden. Im Gegensatz zu MongoDB verfügt Riak

über ein REST-Interface, das intuitiv verwendet werden kann. Somit kann mit der Datenbank sogar auf Kommandozeilenebene kommuniziert werden:

```
curl -H "Accept: text/plain" http://127.0.0.1:8091/stats
```

***Listing 4.27*** *Abfrage der Datenbankstatistiken*

In Riak werden binäre Dateien gespeichert, sodass beispielsweise auch Bilder problemlos in der Datenbank abgelegt werden können. Üblicherweise werden aber JSON-Dokumente verwaltet, was Riak gewissermaßen zum Document-Store werden lässt.

Für nahezu alle Programmiersprachen existieren Bibliotheken, um mit dem Key-Value-Store zu arbeiten, so auch für Ruby. Will man Riak in einer Rails-Anwendung verwenden, kann auf Ripple zurückgegriffen werden. Das ist ein Objekt-Dokument-Mapper, der wie Mongoid vollständig auf ActiveModel basiert und ähnlich benutzt wird. Somit ist ein Umstieg ohne Probleme zu meistern.

Nun wissen wir zwar eine Menge über Riak, aber bei welchen Anwendungsfällen sollte man unbedingt auf den Key-Value-Store setzen?

Dank der Implementierung von Dynamos Prinzipien können sehr gute Skalierungseigenschaften vorgefunden werden. Dadurch kann leicht auf wechselnde Anforderungen reagiert werden: Startet man beispielsweise den Produktionsbetrieb mit einem Datenbank-Cluster von zehn Instanzen und stellt später fest, dass das nicht ausreicht, können problemlos weitere Instanzen hinzugefügt werden.

Ist man hingegen nicht auf die Skalierungseigenschaften von Riak angewiesen, sondern benötigt eine leicht zu verwendende Abfragesprache, sollte man lieber zu MongoDB greifen, da Riak ausschließlich Map-Reduce unterstützt und selbst simple `findByAttribute`-Abfragen mit einer eigenen Funktion implementiert werden müssen.

Ein ganz anderes Ziel verfolgt die nächste Datenbank.

### Redis

Redis ist ein klassischer Key-Value-Store, wurde von Salvatore Sanfilippo entwickelt und erstmals 2009 der Öffentlichkeit vorgestellt. Es ist auf das Speichern von folgenden Datentypen optimiert: Strings, Listen, Hashs sowie sortierte und unsortierte Sets. Redis verwaltet die Daten im Hauptspeicher, lediglich von Zeit zu Zeit wird ein aktueller Stand auf die Festplatte geschrieben. Dadurch wird eine sehr gute Performanz erreicht.

Es gibt atomare Operationen, die auf der Datenbank ausgeführt werden können. So erlaubt Redis beispielsweise die Sortierung von Listen und Sets. Auch ganze Datenbanken können von einem Server auf einen anderen repliziert werden.

Stefan Sprenger / Kieran Hayes, Ruby on Rails 3.1 Expertenwissen, dpunkt.verlag, ISBN 978-3-89864-697-0

Für die Verwendung mit Ruby existiert das Gem redis, das alle verfügbaren Operationen bereitstellt.

Im folgenden Beispiel werden die Erfinder von Programmiersprachen in Redis abgelegt und anschließend wieder ausgelesen:

***Listing 4.28*** *Kommunikation mit Redis in Ruby*

```
require "redis"

# Verbindung zum Server herstellen
redis = Redis.new(:host => "localhost", :port => 6379)

# Erstellen von Key-Value-Paaren
redis.set("PHP", "Rasmus Lerdorf")
redis.set("Python", "Guido van Rossum")
redis.set("Ruby", "Yukihiro Matsumoto")

# Auslesen des Value-Wertes eines Keys
redis.get("Ruby")
```

Aufgrund der simplen Datenstruktur eignet sich Redis besonders als Speicherort für Caches. Für Rails-Anwendungen existiert zu diesem Zweck sogar ein eigenes Gem, redis-store, mit dem Redis innerhalb des Webframeworks verwendet werden kann. Nach dem Hinzufügen des Gems in das `Gemfile` und der Installation, muss lediglich der zu verwendende Cache-Store in der Umgebungskonfiguration, beispielsweise in `config/environments/production.rb` für die Produktionsumgebung, hinterlegt werden:

```
config.cache_store = :redis_store
```

Auch andere Daten, die kein kompliziertes Datenmodell aufweisen, können problemlos mit dem Key-Value-Store verwaltet werden. Für das Speichern von Models sollte aber besser auf MongoDB oder Riak zurückgegriffen werden.

### Weitere Informationen

- Riak-Webseite: `http://basho.com/products/riak-overview/`
- Amazon Dynamo: `http://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf`
- Riaks Map-Reduce: `http://wiki.basho.com/MapReduce.html`
- Ripple: `https://github.com/seancribbs/ripple`
- Redis-Webseite: `http://redis.io/`
- Redis-Operationen: `http://redis.io/commands`
- Redis-Store für Rails: `https://github.com/jodosha/redis-store`

### 4.2.11 Fazit

Die Umstellung auf MongoDB verlief dank der guten Integration von Mongoid in Rails 3 problemlos. Nahezu alle Funktionalitäten von ActiveRecord können auch weiterhin unter Benutzung der gleichen API verwendet werden, weshalb wenig Neuland betreten werden muss.

Auch im Einsatz mit anderen NoSQL-Datenbanken, wie Riak oder Redis, überzeugt Ruby on Rails. Je nach Anwendungsfall und Anforderung kann so die geeignete Datenbank verwendet werden.

Stefan Sprenger / Kieran Hayes, Ruby on Rails 3.1 Expertenwissen, dpunkt.verlag, ISBN 978-3-89864-697-0