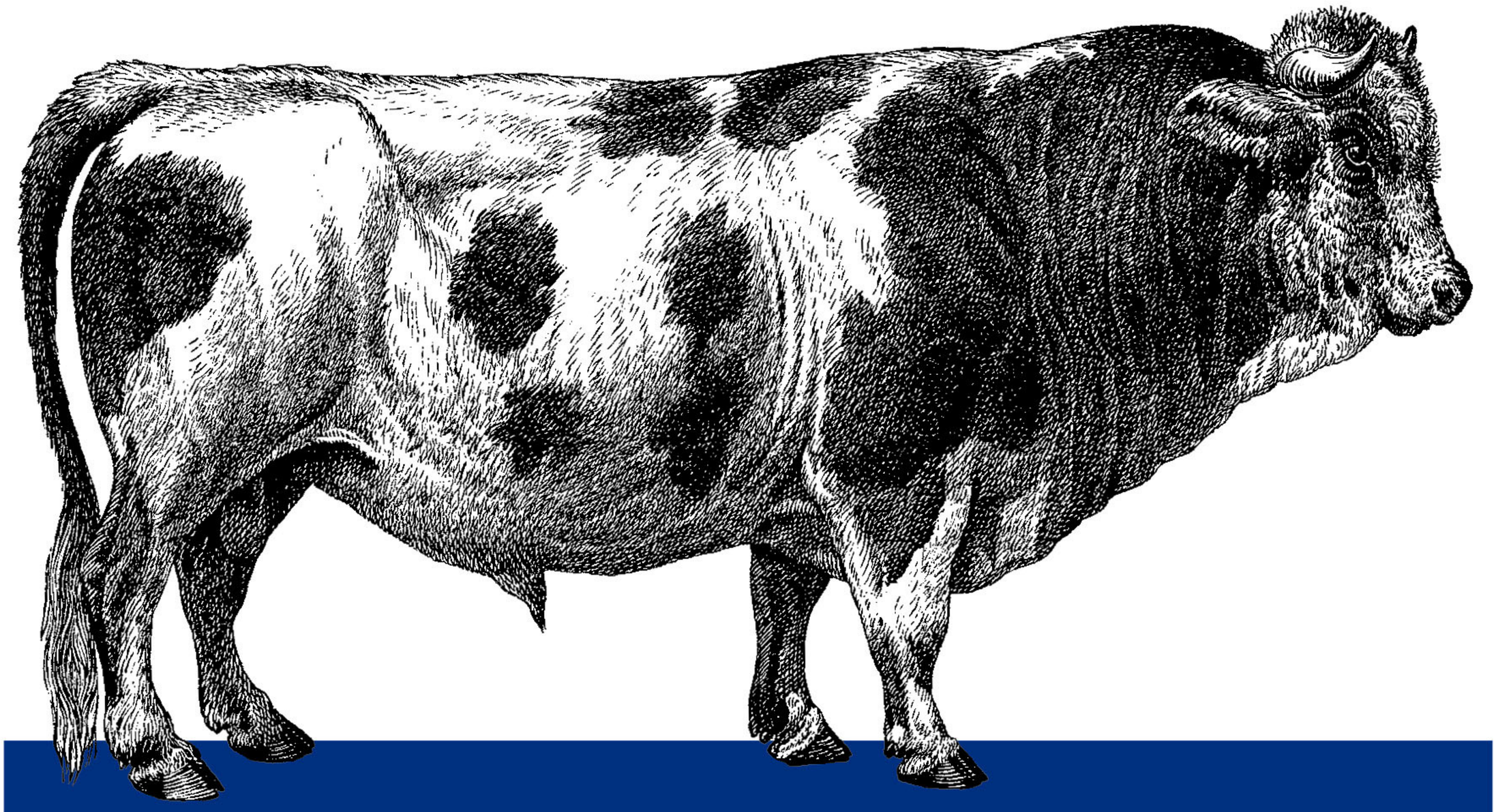
O'REILLY®



Einführung in Apache Solr

PRAXISEINSTIEG IN DIE INNOVATIVE SUCHTECHNOLOGIE

Markus Klose & Daniel Wrigley

Inhalt

KAPITEL 4

Was kann Solr out-of-the-box?

Solr bringt viele Funktionalitäten bereits in der *solrconfig.xml* vorkonfiguriert mit, doch ohne Konfiguration und Anpassung an das eigene Schema sind diese Funktionen nicht optimal nutzbar.

Dieses Kapitel gibt einen Überblick über Inhalte der *solrconfig.xml* und zeigt Ihnen, was Sie in dieser Konfigurationsdatei alles einstellen können. Dabei konzentriert sich dieses Kapitel auf die Funktionalitäten, die direkt auf die Suche Einfluss haben. Funktionalitäten zum Betrieb des Solr und zur Konfiguration skalierbarer Systeme werden in Kapitel 6, *Skalierung der Suche – die Solr-Architektur gestalten* separat beleuchtet.

Die Konfigurationsdatei solrconfig.xml

Jeder Solr-Core hat seine eigene *solrconfig.xml*. Sie dient als zentrale Anlaufstelle, um einen Solr-Core zu konfigurieren. Diese Datei werden Sie immer wieder anfassen müssen, wenn Sie neue Funktionalitäten entwickeln. Das gilt vor allem für den Anfang der Entwicklung einer Suchapplikation.

Folgende grundlegende Einstellungen können in der *solrconfig.xml* manipuliert werden:

- allgemeine Einstellungen
- Index-Einstellungen
- Query-Einstellungen
- Request Dispatcher-Einstellungen
- RequestHandler und SearchComponents
- UpdateHandler und UpdateRequestProcessorChain
- ResponseWriter
- Konfiguration der Admin-Oberfläche

Die *solrconfig.xml*, die mit der Solr-Distribution ausgeliefert wird, enthält viele Beispiele und Kommentare. Sie sollten diese Datei gleich zu Beginn aufräumen und alles, was Sie nicht brauchen, entfernen. Ein Feature wieder einzubauen, ist nicht schwer, aber aus Erfahrung wissen wir, dass die *solrconfig.xml* nur in seltenen Fällen hinterher aufgeräumt wird.

Nachfolgend werden die einzelnen Bereiche näher betrachtet und die wichtigsten Stellschrauben vorgestellt. Die RequestHandler, SearchComponents, UpdateRequestProcessorChains und ResponseWriter, die in der *solrconfig.xml* eingestellt werden können, wurden bereits vorgestellt (siehe Kapitel 2, Abschnitt »Der Grundaufbau von Solr« auf Seite 17) und fehlen daher in der folgenden Auflistung.

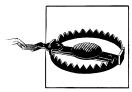
Allgemeine Einstellungen

Zu den allgemeinen Einstellungen gehört alles, was sich nicht direkt in die anderen Bereiche einordnen lässt.

Lucene-Version

Viele Funktionalitäten werden durch die Lucene-Version beeinflusst bzw. ermöglicht, die hinter dem Index stehen. Neuere Solr-Instanzen können mit älteren Lucene-Indexen arbeiten, da Solr abwärtskompatibel ist. Die Version des Index stellt man mit dem `luceneMatchVersion`-Element ein.

```
<luceneMatchVersion>LUCENE_43</luceneMatchVersion>
```



Es reicht nicht, die so spezifizierte Lucene-Version zu aktualisieren, um von den Verbesserungen neuerer Lucene-Versionen zu profitieren, da viele der Änderungen auch Auswirkung auf die physische Speicherung des Index haben. Daher muss man fast immer, wenn man Solr bzw. Lucene aktualisiert, den Index neu aufbauen.

Externe Bibliotheken

Solr ist Open Source, daher werden ständig neue Komponenten entwickelt. Um diese problemlos in Solr zu integrieren, sind Schnittstellen geschaffen worden, über die externe JAR-Dateien eingebunden werden. Eine dieser Schnittstellen ist in der *solrconfig.xml*. Mit dem `<lib>`-Element können fremde Bibliotheken referenziert werden, indem man entweder den Pfad zu einem Ordner angibt oder die JAR-Datei direkt referenziert.

Beispiele für das Referenzieren externer Bibliotheken in der *solrconfig.xml* sehen wie folgt aus:

```
<lib dir="XXX" regex=".*\.jar" />
<lib path="XXX.jar" />
```

Im ersten Beispiel steht das `XXX` für ein relativ oder absolut angegebenes Verzeichnis, in dem alle Dateien mit dem Suffix *.jar* eingebunden werden. Im zweiten Bei-

spiel wird die JAR-Datei *XXX.jar* direkt eingebunden. Auch hier kann die Datei relativ oder absolut angegeben werden.

Datenverzeichnis
: Per Default wird der Index im Verzeichnis *data* innerhalb des Solr-Core-Verzeichnisses angelegt. Mit dem `<dataDir>`-Element lässt sich alternativ ein anderes Verzeichnis definieren, in dem der Index abgelegt wird, um Daten und Konfiguration voneinander zu trennen.

```
<dataDir>c:\solr_data</dataDir>
```

Das Beispiel sorgt dafür, dass Solr die Daten im Verzeichnis *c:\solr_data* ablegt, egal wo das Solr-Home-Verzeichnis ist.

DirectoryFactory
: Per Default wird der Index physisch im Dateisystem abgelegt. Mit dem `<directoryFactory>`-Element kann man Solr zwingen, den Index an einem anderen Ort zu speichern, beispielsweise im Speicher oder auf einem Hadoop-Filesystem

```
<directoryFactory name="DirectoryFactory" class="solr.RAMDirectoryFactory"/>
```

Das obige Beispiel hält den Index im Speicher und nicht persistent im Dateisystem.

Index-Einstellungen

In der *solrconfig.xml* gibt es ein `<indexConfig>`-Element, in dem man den Indexierungsprozess und den Aufbau des Index in gewissem Rahmen beeinflussen. Auch wenn die Default-Einstellungen in den meisten Situationen ausreichen, werden die wichtigsten Parameter nachfolgend beschrieben.

maxTokenCount
: Dieses Element definiert, wie viele Token maximal je Feld gespeichert werden. Auch wenn die Analyse mehr Token generiert, werden die Token, die dieses Limit überschreiten, nicht im Index abgelegt.

maxIndexingThreads
: Dieses Element spezifiziert die Anzahl der Threads, die gleichzeitig Indexierung durchführen können. Je mehr Threads, desto schneller ist die Indexierung.

useCompoundFile
: Dieses Property steuert, ob ein Segment in mehreren Datei abgespeichert wird oder in einer einzelnen. Diesen Parameter auf `true` zu stellen, kann nützlich sein, wenn dem Betriebssystem die FileHandles ausgehen.

ramBufferSizeMB/maxBufferedDocs
: Mit diesen Elementen wird die maximale Speichergröße bzw. Dokumentenanzahl definiert, die während des Indexierungsvorgangs genutzt werden kann, bevor ein Flush ausgeführt wird. Je höher die Werte, desto schneller geht die Indexierung. Nach einem Flush sind die Dokumente nicht suchbar, erst nach einem Commit bzw. Optimize.

mergeFactor
: Bei Commits und Optimize-Operationen werden gleich große Segmente automatisch zusammengefasst. Dieses Mergen von Segmenten dauert eine gewisse Zeit, beeinflusst die Indexierungszeit negativ und kann mit diesem Property beeinflusst werden.

reopenReader
: Nach einem Commit wird ein neuer IndexSearcher instanziiert, der Lesezugriff auf die einzelnen Segmente hat. Alte Segmente können wiederverwendet werden, was die AutoWarming-Zeit reduziert.

Query-Einstellungen

Analog zu den Index-Einstellungen gibt ein `<query>`-Element, mit dem man die Suche beeinflussen kann. Folgende Properties können genutzt werden, um die Suche zu optimieren.

maxBooleanClauses
: Dieser Parameter spezifiziert, wie viele boolesche Teile eine Query haben darf. Range-Queries, die nicht auf dem Trie-FeldTyp basieren, können dieses Limit leicht sprengen.

filterCache
: Dieser Cache speichert die Ergebnisse des `fq`-Parameters, damit dieses Subset von Dokumenten bei Suchen mit gleicher FilterQuery wiederverwendet werden kann.

queryResultCache
: Dieser Cache speichert die IDs der gefundenen Dokumente. Wenn sich der Anwender mehrere Seiten einer Trefferliste anschaut, muss dank dieses Caches nicht jedes Mal eine Suche durchgeführt werden, sondern es reicht, die nächsten n Dokumente aus diesem Cache auszuliefern.

documentCache
: Dieser Cache speichert die Anzeigeinformationen (Stored Information) und beschleunigt daher den Aufbau der Trefferliste, wenn das gleiche Dokument mehrmals angezeigt wird.

FieldValueCache
: Dieser Cache speichert die Werte eines Felds und wird primär für die Sortierung und Facettierung genutzt.

enableLazyFieldLoading
: Wenn dieser Parameter auf `true` gesetzt wird, werden nur die Felder in den documentCache geladen, die für die Anzeige der Trefferliste benötigt werden.

queryResultMaxDocsCached
: Mit diesem Parameter spezifiziert man die Anzahl der Dokumente, die im queryResultCache gespeichert werden. Der Default von 200 ist recht hoch, denn bei den meisten Suchen werden höchstens die ersten drei Seiten angezeigt, bevor der Anwender eine neue Suche durchführt.

newSearcher/firstSearcher
: In diesen Elementen können statische Suchanfragen hinterlegt werden, die im Zuge des AutoWarming ausgeführt werden, um die Caches zu befüllen. Die Suchen des firstSearcher werden nur beim Start des Solr ausgeführt, die des newSearcher hingegen bei jedem Commit.

UseColdSearcher
: Setzt man dieses Property auf `true`, wird nicht gewartet, bis das AutoWarming fertig ist, sondern der neue IndexSearcher wird sofort für Suchanfragen freigegeben. Dies führt dazu, dass neue Dokumente schneller auffindbar sind, jedoch die ersten Suchanfragen länger dauern, da die Caches noch leer sind.

Request Dispatcher-Einstellungen

Das `<requestDispatcher>`-Element in der *solrconfig.xml* kontrolliert die grundlegende Verarbeitung von Requests. Dies betrifft vor allem die technische Verarbeitung des Requests und das HTTP-Caching.

Das `<requestParsers>`-Element innerhalb des Request Dispatcher hat folgende Attribute:

enableRemoteStreaming
: Ist dieser Wert auf `true` gesetzt, ist Remote Streaming erlaubt, d. h., bei der Indexierung von Daten können diese an den Solr-Server gestreamt werden.

multipartUploadLimitInKB
: Dieser Parameter gibt die Obergrenze in Kilobyte an, wie groß ein Dokument bei einem multi-part HTTP POST-Request sein darf.

formdataUploadLimitInKB
: Dieser Parameter gibt die Obergrenze in Kilobyte an, wie groß die Daten bei einem `x-www-form-urlencoded`-Request sein dürfen.

Das `<httpCaching>`-Element innerhalb des Request Dispatcher hat folgende Attribute, mit denen das Caching konfiguriert wird:

never304
: Wird dieses Attribut auf `true` gesetzt, wird ein GET-Request nie den Status 304 zurückliefern, sondern immer den angeforderten Body.

etagSeed
: Der Wert dieses Attributs wird als ETag im Header des Requests mitgesendet.

Konfiguration der Admin-Oberfläche

In dem `<admin>`-Element in der *solrconfig.xml* können einige Default-Einstellungen für die Admin-Oberfläche durchgeführt werden. Es kann beispielsweise der `q`-Parameter des Query-Elements in der Admin-Oberfläche eingestellt werden.

Des Weiteren kann man definieren, wie der Solr auf einen HealthCheck-Request eines LoadBalancer reagieren soll.

Such-Features out-of-the-box

Die *solrconfig.xml*, wie sie in der Solr-Distribution enthalten ist, ist prall gefüllt mit SearchComponents, RequestHandlern und anderen Elementen, die die Suche beeinflussen. Vieles von dem ist überflüssig oder einfach nur überladen. Auf den nächsten Seiten werden wir Licht ins Dunkel dieser Komponenten bringen und die wichtigsten Features, die Solr out-of-the-box anbietet, vorstellen.

Die einzelnen Funktionalitäten werden erst kurz vorgestellt und die Requests sowie Response-Informationen dargestellt. Hierfür werden die Beispieldaten, die in der Solr-Distribution enthalten sind, zugrunde gelegt.

Im Anschluss wird am Beispiel des Wikipedia-Index aufgezeigt, wie man diese Funktionalitäten in eine Benutzeroberfläche einbindet.

Velocity und der /browse-RequestHandler

Eine fertige Benutzeroberfläche bringt Solr bereits mit. Hierbei handelt es sich um ein User-Interface, das mit Velocity erstellt wird. Velocity ist ein Apache-Projekt mit dem Ziel, Template-basierte Webseiten zu erstellen. Dieses Velocity-Template eignet sich hervorragend für das Prototyping, da zum einen schon viele Funktionalitäten vorhanden sind, die man leicht adaptieren kann, zum anderen ist die Velocity Template Language (VTL) grundsätzlich leicht zu verstehen. Man muss sich nicht vollständig in Velocity auskennen, um Änderungen durchführen zu können.

Die Benutzeroberfläche wird über den `/browse`-RequestHandler aufgerufen. Folgender Request zeigt die initiale Ansicht, die nach und nach mit Leben gefüllt werden wird:

```
http://localhost:8983/solr/de_wikipedia/browse
```

Führt man den Request im Browser aus, bekommt man nicht wie üblich eine XML- oder JSON-Struktur zurück, sondern die fertige Website aus Abbildung 4-1.

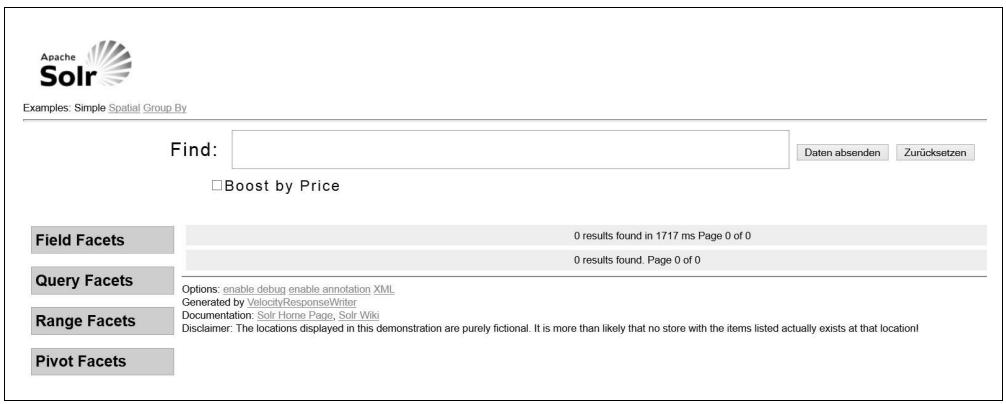


Abbildung 4-1: Leeres Browse-Interface

Auf den nächsten Seiten werden wir diese leere Suchmaske nach und nach mit Leben füllen. Da die einzelnen Funktionalitäten nicht zwingend aufeinander aufbauen, können Sie sich die für Sie interessanteren Features heraussuchen und weniger interessante überspringen.

Funktionalitäten wie Facetten, Highlighting oder AutoSuggest sind essentiell für eine gute Suche und sollten daher unbedingt getestet werden. Mit diesen drei können Sie eine Suche effizient gestalten, indem Sie dem Anwender schon mit einem guten AutoSuggest die passenden Suchbegriffe vorgeben. Sollte es dann trotzdem zu viele Treffer geben, kann der Anwender gezielt die Trefferliste mit den Facetten einschränken, und wenn er dann noch auf den ersten Blick das gesuchte Wort im Text des Treffer-Dokuments erkennt, ist das Sucherlebnis vollkommen.

Result Grouping oder Elevate passen jedoch nicht in jeden Kontext und sind daher nicht zwingend notwendig für das Gelingen einer Suche. Es gibt aber Anwendungsfälle, wo diese Funktionalitäten die Suche sehr unterstützen. Result Grouping wird beispielsweise gern genutzt, um unterschiedliche Produktgruppen in Webshops darzustellen.

Die Stats- und die TermVektor-Komponente, die am Ende des Kapitels beschrieben werden, werden in der Regel nicht direkt in eine Suchmaske integriert, sondern dienen eher der Analyse des Indexes oder werden vorbereitend für spezielle Suchanfragen genutzt.

Die einzelnen Funktionalitäten von Solr sind schon für die unterschiedlichsten Szenarien eingesetzt worden und besitzen daher oft eine Menge von Parametern, mit denen man die Funktionalität beeinflussen kann. Nehmen Sie sich daher Zeit: Versuchen Sie zunächst, die Funktionen mit einer Minimalkonfiguration umzusetzen, und erweitern Sie diese dann sukzessiv. So wird es leichter, die einzelnen Stellschrauben zu verstehen.

Das Browse-Interfaces mit allen in diesem Kapitel besprochenen Funktionalitäten könnte wie in Abbildung 4-2 aussehen.

Kommen wir zurück zum `/browse`-RequestHandler. Diese Oberfläche ist modular aufgebaut. Es gibt einzelne Templates für Facetten, die Trefferliste, das AutoSuggest etc., also für alle Features, die Solr mitbringt. Diese Modularität erlaubt zum einem, das Look-and-feel schnell anzupassen, aber auch Funktionalitäten einzubinden oder zu verändern.

Die einzelnen Velocity-Templates befinden sich im Verzeichnis *de_wikipedia\conf\velocity* innerhalb des Solr-Home-Verzeichnisses. Da das ganze Template auf die Beispieldaten ausgelegt ist, die Solr in der Distribution enthält, passt die Anzeige nicht zu den indexierten Daten der Wikipedia. Um dennoch eine schöne Anzeige der Wikipedia-Dokumente zu ermöglichen, müssen nur zwei kleine Änderungen bzw. Erweiterungen vorgenommen werden. Die erste Erweiterung ist die Erstellung eines eigenen Templates für ein Wikipedia-Dokument. In dem Template-Verzeichnis muss eine Datei mit dem Namen *wikipedia.vm* abgelegt werden, die folgenden Inhalt hat:

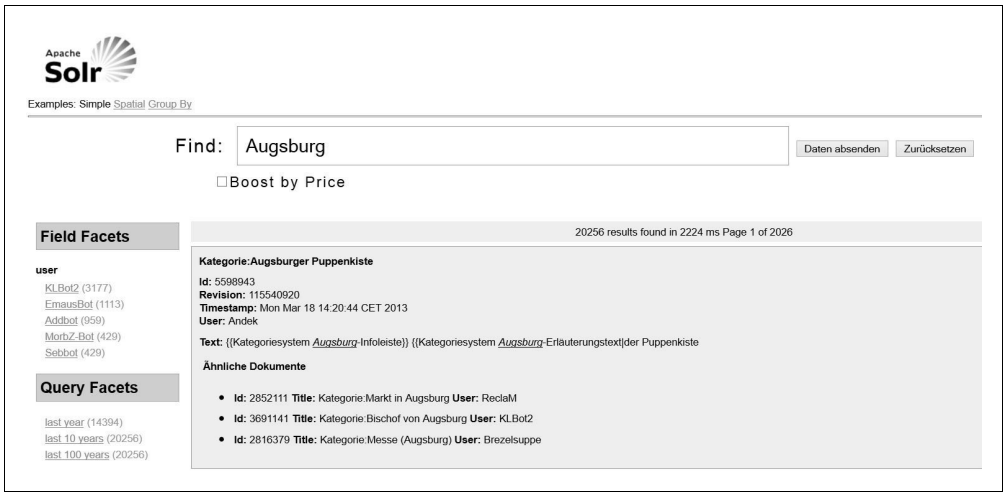


Abbildung 4-2: Das fertige Browse-Interface

```
<div class="result-title">
   <b>#field('titleText')</b>
</div>
<div>
   <b>Id:</b> #field('id')</br>
   <b>Revision:</b> #field('revision')</br>
   <b>Timestamp:</b> #field('timestamp')</br>
   <b>User:</b> #field('user')</br>
</div>
<div>
   <b>Text:</b> #field('text')
</div>
#parse('debug.vm')
```

Das Template für das Wikipedia-Dokument, wie es oben dargestellt wird, zeigt neben dem Titel des Dokuments auch weitere Informationen, wie ID oder User, an. Die Anzeige aller Treffer wird im Template *hit.vm* umgesetzt. Dort muss nun die *wikipedia.vm* wie folgt eingebunden werden:

```
#set($docId = $doc.getFieldValue('id'))

<div class="result-document">
   #parse("wikipedia.vm")
</div>
```

Aufseiten des Velocity-Templates ist nun alles vorkonfiguriert, aber auch der `/browse`-RequestHandler ist auf die Solr-Beispieldokumente ausgelegt und muss daher angepasst werden. Ein RequestHandler mit einer Basisfunktionalität sieht wie folgt aus:

```
<requestHandler name="/browse" class="solr.SearchHandler" startup="lazy">
   <lst name="defaults">
      <str name="df">text</str>
```

```
      <!-- velocity -->
      <str name="wt">velocity</str>
      <str name="v.template">browse</str>
      <str name="v.layout">layout</str>
      <str name="title">Solritas - Wikipedia</str>
    </lst>
  </requestHandler>
```

Damit sind nun alle Anpassungen an der Default-Konfiguration durchgeführt, und das Velocity kann mit dem Schema und den Daten der Wikipedia arbeiten. Beim Aufruf des RequestHandler kann man natürlich auch Parameter mitgeben. Folgender Request führt eine Suche nach dem Begriff »Augsburg« aus.

```
http://localhost:8983/solr/de_wikipedia/browse?q=Augsburg
```

Im Browse-Interface werden nun Treffer dargestellt. Weitere Funktionalitäten fehlen noch, wie man in Abbildung 4-3 sieht. Diese werden jetzt schrittweise hinzugefügt.

Abbildung 4-3: Browse-Interface mit Treffern der Wikipedia

Facetten – Suchergebnisse verfeinern

Facetten sind bei der Suche einer der am meisten genutzten Features, und das aus gutem Grund. Man kann sie beispielsweise sehr gut nutzen, um zu große Trefferlisten nach bestimmten Kriterien, wie beispielsweise Kategorien, Farben, Länder, Zeiträume, Preisbereiche etc., einzuschränken. In diesem Fall wird üblicherweise die gleiche Suche noch einmal ausgeführt, nur dass beim zweiten Mal eine zusätzliche FilterQuery mit der Einschränkung angehängt wird.

Ein anderer Weg der Nutzung ist die Seitennavigation. In diesem Fall wird üblicherweise auf eine Volltextsuche verzichtet, und man nutzt die Facetten, um eine Website wie zum Beispiel einen E-Commerce-Shop in Bereiche zu unterteilen, noch ehe der Anwender selbst eine Suche ausgelöst hat.

Technisch gesehen, ist eine Facette nichts anderes als eine »Gruppierung« der gefundenen Dokumente nach bestimmten Kriterien, kombiniert mit einer Zählung, wie viele Dokumente zu einem Kriterium insgesamt gehören. Bei dieser Gruppierung und Zählung werden alle Dokumente berücksichtigt, die als Treffer identifiziert worden sind, also nicht nur die ersten n Dokumente im angezeigten Ergebnis.

Die Facetten-Komponente ist eine von den sechs Standardkomponenten und muss daher nicht gesondert in den RequestHandlern referenziert werden. Man muss die Funktionalität nur aktivieren, entsprechend der Anforderung konfigurieren, und im Solr-Response wird ein neues Element `facets_counts` mit den Facetten ausgegeben.

Für die grundsätzliche Konfiguration der Facette gibt es folgende Parameter, die man entweder im RequestHandler hinterlegen oder als Aufrufparameter bei der Suche mitgeben kann:

facet
: Wird dieser Parameter auf `true` gesetzt, ist die Funktionalität aktiviert.

facet.mincount
: Über diesen Parameter spezifiziert man, wie oft ein Term vorkommen muss, damit ein Eintrag mit diesem Term in der Facette gebildet wird. Üblicherweise setzt man diesen Wert auf `1`, um auch dann ein Ergebnis zu bekommen, wenn man die Suche nach der Facette einschränkt.

facet.limit
: Mit diesem Parameter schränkt man ein, wie viele Einträge eine Facette haben soll. Es gibt Facetten mit zu vielen Einträgen, sodass man nur die ersten zehn Einträge anzeigt und später weiterer Einträge gegebenenfalls nachlädt.

facet.sort
: Mit diesem Parameter kann man die Sortierreihenfolge der Facette festlegen. `facet.sort=count` sortiert die Einträge so, dass der Term mit dem häufigsten Vorkommen oben in der Liste steht, alle weiteren Einträge folgen dann absteigend sortiert. `facet.sort=index` sorgt für eine alphabetische Sortierung der Facetten, wobei der erste Eintrag mit einem »a« beginnt, sofern Terme mit a vorhanden sind. Dieser Parameter erlaubt es nicht, die Richtung (ASC bzw. DESC) der Sortierung mitzugeben. Möchte man beispielsweise, dass der erste Eintrag mit »z« beginnt und der letzte mit »a«, muss man sich alle Einträge der Facette generieren lassen und die »verkehrte« Sortierung zur Anzeigezeit umsetzen.

facet.prefix
: Mit diesem Parameter kann man die Facetten-Komponente dazu zwingen, nur die Werte in die Facette aufzunehmen, die ein bestimmtes Präfix haben. Mit `facet.prefix` lässt sich somit ein einfacher AutoSuggest umsetzen.

Es gibt in Solr aktuell folgende Implementierungen für Facetten, die nachfolgend genauer betrachtet werden:

- Feld-Facetten
- Range-Facetten
- Date-Facetten
- Query-Facetten
- Pivot-Facetten

Feld-Facetten

Feld-Facetten basieren auf dem Inhalt eines Felds, dabei wird durch die Terme eines Felds iteriert und gezählt, in wie vielen Dokumenten dieser Term enthalten ist.

Um Feld-Facetten zu definieren, gibt es zusätzlich zu den allgemeinen Facetten-Parametern noch folgende:

facet.field
: Mit diesem Parameter gibt man das Feld an, über das die Facette generiert wird. Diesen Parameter kann man mehrmals angeben, um mehrere Feld-Facetten zu erhalten.

facet.method
: Mit diesem Parameter kann man kontrollieren, welcher Cache bei der Generierung der Facette genutzt wird.

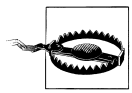
- `facet.method=emun` – Mit dieser Methode wird zum Iterieren und Speichern der Daten der FilterCache genutzt. Für jeden Term wird ein entsprechender Eintrag im FilterCache generiert, und anschließend werden alle Einträge zusammengefasst. Diese Methode lohnt sich nur für Felder mit wenig verschiedenen Termen. Ein Feld mit dem Namen der Bundesländer von Deutschland würde beispielsweise 16 Einträge im Filtercache generieren.
- `facet.method=fc` – Diese Methode nutzt den FieldCache bzw. bei Feldern mit `multiValued=true` den FieldValueCache, um die Daten zu speichern und zu akkumulieren. Diese Methode empfiehlt sich immer bei Feldern mit vielen unterschiedlichen Termen.
- `facet.method=fcs` – Diese Methode ist eine Sondervariante der `fc`-Methode und funktioniert aktuell nur mit singleValued-Feldern. Die Häufigkeitsinformationen werden hierbei pro Segment im Cache abgelegt, was die Suche um ein Vielfaches schneller macht und man flexibler bei Index-Updates ist.

Der Aufruf für eine Suche mit der Generierung von zwei auf Feldern basierten Facetten sieht dann wie folgt aus:

```
http://localhost:8983/solr/collection1/select?q=*:*&facet=true&facet.field=features&facet
.field=manu&facet.limit=5
```

Der nachfolgende Response zeigt nun für die Felder `features` und `manu` jeweils die ersten fünf Ergebnisse an.

```
<response>
   ...
   <lst name="facet_counts">
      <lst name="facet_queries"/>
      <lst name="facet_fields">
         <lst name="features">
            <int name="and">6</int>
            <int name="lcd">5</int>
            <int name="notes">5</int>
            <int name="2.0">4</int>
            <int name="coins">4</int>
         </lst>
         <lst name="manu">
            <int name="inc">8</int>
            <int name="apache">2</int>
            <int name="bank">2</int>
            <int name="belkin">2</int>
            <int name="canon">2</int>
         </lst>
      </lst>
      <lst name="facet_dates"/>
      <lst name="facet_ranges"/>
   </lst>
</response>
```



Die Werte, die von der Facette angezeigt werden, sind nicht die *stored*-, sondern die *indexed*-Werte, d. h. die von der Analyse veränderten Terme. Für Facetten sollte man daher immer per CopyField ein zweites Feld anlegen, das beispielsweise vom Typ String ist und daher nicht alles in Kleinbuchstaben schreibt oder in einzelne Wörter zerlegt. Der obige Response zeigt für das Feld `manu` sehr deutlich, was bei einem Feld mit einem textbasierten Feldtyp passieren kann. Nimmt man hingegen `manu_exact`, sieht das Ergebnis besser aus.

In diesem Beispiel hat der Parameter `facet.limit=5` Auswirkung auf beide Facetten. Es gibt Situationen, in denen man unterschiedliche Konfigurationen für verschiedene Facetten haben möchte, ohne mehrere Such-Requests abschicken zu müssen. Solr bietet hierfür die Möglichkeit, die oben beschriebenen Parameter für Mindestanzahl, Sortierung etc. für jedes Feld zu einzeln zu definieren. Dabei muss folgende Syntax eingehalten werden:

```
f.<fieldname>.facet.<parameter>
```

Möchten Sie nun für die Facette `manu` zehn Einträge bekommen und für die Facette `features` fünf, sieht die Query wie folgt aus:

```
http://localhost:8983/solr/collection1/select?q=*:*&facet=true&facet.field=features&facet
.field=manu&f.features.facet.limit=5&f.manu.facet.limit=10
```

Range-Facetten

Range-Facetten sind Facetten, die nur bei numerischen Feldern wie Preisen etc. funktionieren. Hierbei werden die Dokumente gleich großen Bereichen zugeordnet, beispielsweise kommen alle Dokumente mit einem Preis zwischen 0 _ und 10 _ in einen Bereich, alle Dokumente zwischen 10 _ und 20 _ in einen weiteren Bereich etc. Dies passiert dynamisch, d. h., Solr berechnet so viele Bereiche, wie es gibt.

Um Range-Facetten nutzen zu können, muss man einen Startwert, einen Endwert und die Größe des Bereichs definieren; den Rest macht Solr automatisch. Folgende Parameter werden hierfür genutzt:

facet.range
: Dieser Parameter spezifiziert das numerische Feld, für das die Facette berechnet werden soll.

facet.range.start
: Dieser Parameter ist der Startwert für die Facette.

facet.range.end
: Dieser Parameter ist der Endwert für die Facette.

facet.range.gap
: Dieser Parameter definiert die Größe der Bereiche, die von der Facetten-Komponente generiert werden soll.

facet.range.other
: Mit diesem Parameter kann man Solr dahin gehend konfigurieren, dass zusätzlich zu den berechneten Bereichen noch weitere Werte berechnet werden.

- `facet.range.other=before` – Alle Dokumente, die vor dem Startwert liegen, werden in einem Eintrag zusammengefasst.
- `facet.range.other=after` – Alle Dokumente, die nach dem Startwert liegen, werden in einem Bereich zusammengefasst.
- `facet.range.other=between` – Alle Dokumente zwischen Start- und Endwert werden in einem Bereich zusammengefasst.
- `facet.range.other=none` – Es werden keine Häufigkeiten berechnet.
- `facet.range.other=all` – Alle Dokumente (`before`, `after` und `between`) werden in einen Bereich zusammengefasst.

Ein möglicher Request für eine Range-Facette sieht wie folgt aus:

```
http://localhost:8983/solr/collection1/select?q=*:*&facet=true&facet.range=price&facet.
range.start=0&facet.range.end=100&facet.range.gap=10&&facet.range.other=after
```

Basierend auf diesem Request, wird Solr folgende Antwort generieren:

```
<response>
   ...
   <lst name="facet_counts">
```

```
        <lst name="facet_queries"/>
        <lst name="facet_fields"/>
        <lst name="facet_dates"/>
        <lst name="facet_ranges">
           <lst name="price">
              <lst name="counts">
                 <int name="0.0">3</int>
                 <int name="10.0">2</int>
                 <int name="20.0">0</int>
                 <int name="30.0">0</int>
                 <int name="40.0">0</int>
                 <int name="50.0">0</int>
                 <int name="60.0">0</int>
                 <int name="70.0">1</int>
                 <int name="80.0">0</int>
                 <int name="90.0">1</int>
              </lst>
              <float name="gap">10.0</float>
              <float name="start">0.0</float>
              <float name="end">100.0</float>
              <int name="after">9</int>
           </lst>
        </lst>
     </lst>
  </response>
```

Durch die Range-Facette des Felds Preis sind automatisch 10 Bereiche generiert worden, die jeweils 10 (Euro) groß sind. Alle Dokumente (9), die mehr als 100 (Euro) kosten, wurden separat zusammengefasst.

Ein paar der automatisch berechneten Bereiche haben 0 Dokumente zugeordnet bekommen. Wenn man auf diese Einträge verzichten möchte, kann man `facet.mincount=1` bzw. `f.price.facet.mincount=1` als zusätzlichen Parameter angeben.

Date-Facetten

Technisch gesehen, sind Date-Facetten das Gleiche wie Range-Facetten, jedoch mit dem einen Unterschied, dass sich die Facette auf Felder des Typs Date (siehe Kapitel 3, Abschnitt »Solr-Feldtypen« auf Seite 91) beziehen. Die Date-Facette unterstützt natürlich sowohl `NOW` als auch die gesamte Datumsarithmetik. Somit können beispielsweise Tages-Facetten generiert werden, aber auch Facetten, bei denen der Gap eine Stunde oder zehn Jahre beträgt.

Da beide Facetten auf der gleichen technischen Grundlage basieren, ist die Konfiguration (`facet.range.start`, `facet.range.end`, `facet.range.gap` etc.) identisch, und der Request für die Date-Facette ist genauso aufgebaut wie der Request für die Range-Facette.

```
http://localhost:8983/solr/collection1/select?q=*:*&facet=true&rows=0&facet.range=
manufacturedate_dt&facet.range.start=NOW/YEAR-10YEARS&facet.range.end=
NOW/YEAR&facet.range.gap=+1YEARS
```

Der obige Request beschreibt eine Date-Facette, die alle Dokumente jahresweise zusammenfasst und für den Zeitraum von 2003 bis 2013 Einträge generiert.

Solr liefert einen ähnlichen Response für diese Facette aus, wie er bei der Range-Facette zu finden ist. Auch hier gibt es bei den Solr-Beispieldokumenten Einträge, denen kein Dokument zugeordnet worden ist.

```
<response>
   ...
   <lst name="facet_counts">
      <lst name="facet_queries"/>
      <lst name="facet_fields"/>
      <lst name="facet_dates"/>
      <lst name="facet_ranges">
         <lst name="manufacturedate_dt">
            <lst name="counts">
              <int name="2003-01-01T00:00:00Z">0</int>
               <int name="2004-01-01T00:00:00Z">2</int>
               <int name="2005-01-01T00:00:00Z">9</int>
               <int name="2006-01-01T00:00:00Z">0</int>
               <int name="2007-01-01T00:00:00Z">0</int>
               <int name="2008-01-01T00:00:00Z">0</int>
               <int name="2009-01-01T00:00:00Z">0</int>
               <int name="2010-01-01T00:00:00Z">0</int>
               <int name="2011-01-01T00:00:00Z">0</int>
               <int name="2012-01-01T00:00:00Z">0</int>
            </lst>
            <str name="gap">+1YEARS</str>
            <date name="start">2003-01-01T00:00:00Z</date>
            <date name="end">2013-01-01T00:00:00Z</date>
         </lst>
      </lst>
   </lst>
</response>
```

Query-Facetten

Mit den Query-Facetten kann man sehr spezielle Facetten generieren, die sich nicht mit Feld-, Range- oder Date-Facetten umsetzen lassen. Um Query-Facetten zu nutzen, muss man nur den Parameter `facet.query` so oft spezifizieren, wie man Einträge in seiner Facette haben möchte. Als Wert für diesen Parameter kann jede Query in Lucene-Syntax verwendet werden.

Sehr häufig wird diese Art der Facette für »Range-Facetten« genutzt, bei denen der Gap nicht immer die gleiche Größe haben soll. Das folgende Beispiel zeigt den Request für eine Query-Facette, die drei Einträge haben soll. Der erste Eintrag soll alle die Dokumente enthalten, die im vergangenen Jahr erstellt worden sind. Der zweite Eintrag enthält die Dokumente der letzten 10 Jahre und der dritte Eintrag die Dokumente der letzten 100 Jahre.

```
http://localhost:8983/solr/collection1/select?
q=*:*&facet=true&facet.query=manufacturedate_dt:[NOW-1YEARS TO NOW]&facet.query=
manufacturedate_dt:[NOW-10YEARS TO NOW-1YEARS]&facet.query=manufacturedate_dt:
[NOW-100YEARS TO NOW-10YEARS]
```

Eine Query-Facette muss nicht zwingend über die Syntax der Range-Suche (`feld:[xxx TO yyy]`) umgesetzt werden. Es sind auch Szenarien denkbar wie `facet.query categorie:Sport AND jahreszeit:Winter` oder `facet.query=wintersport`.

Der Solr-Response für die Query-Facette ist ähnlich aufgebaut wie bei den anderen Facetten. Es werden für das obige Beispiel genau drei Einträge in der Facette generiert.

```
<response>
   ...
   <lst name="facet_counts">
      <lst name="facet_queries">
         <int name="manufacturedate_dt:[NOW-1YEARS TO NOW]">0</int>
         <int name="manufacturedate_dt:[NOW-10YEARS TO NOW-1YEARS]">11</int>
         <int name="manufacturedate_dt:[NOW-100YEARS TO NOW-10YEARS]">0</int>
      </lst>
      <lst name="facet_fields"/>
      <lst name="facet_dates"/>
      <lst name="facet_ranges"/>
   </lst>
</response>
```

Im Gegensatz zu den Feld-, Range- und Date-Facetten können bei der Query-Facette nicht mehrere Facetten generiert werden. Möchte man mehrere von diesen »Custom«-Facetten haben, muss man aktuell einen Umweg machen. Entweder man führt mehrere Requests mit unterschiedlichen `facet.query`-Parametern aus, oder man definiert alle `facet.query`-Parameter in einer Suche und zerteilt im Anschluss das Ergebnis selbst in die gewünschten Einheiten.

Pivot-Facetten

Pivot-Facetten werden oft auch hierarchische Facetten oder Entscheidungsbäume genannt, denn mit diesen Pivot-Facetten können die Daten des Index dynamisch in einer hierarchischen Struktur aufbereitet werden. Wie bei allen anderen Facetten wird gezählt, wie viele Dokumente der entsprechenden Hierarchieebene zugeordnet sind.

Beispiele finden sich häufig bei Onlineshops, die ihre Waren in verschiedene Kategorien einteilen. Eine Art der Kategorisierung könnte die Einteilung in »Damen«, »Herren« und »Kinder« sein, eine zweite in »T-Shirts«, »Hosen«, »Schuhe« etc. Man könnte natürlich bei der Suche zwei separate Facetten anzeigen, jedoch ist es für die Nutzerfreundlichkeit oft besser, eine Pivot-Facette zu nutzen, sodass man sehr leicht die »Herrenschuhe« herausfiltern kann.

Um Pivot-Facetten zu definieren, benötigt man nur den Parameter `facet.pivot`. Als Wert bekommt dieser Parameter die Namen der Felder, die eine hierarchische Struktur bilden sollen.

Bei den Beispieldaten der Solr-Distribution könnte man beispielsweise den Hersteller und die Kategorien in Relation setzen. Wenn man also in einer Facette alle Kategorien jedes Herstellers haben möchte, muss der Request wie folgt aussehen:

```
http://localhost:8983/solr/collection1/select?q=*:*&facet=true&facet.pivot=manu_exact,cat
```

Auch bei den Pivot-Facetten werden im Response die *indexed*-Werte angezeigt, daher ist bewusst im obigen Beispiel das Feld `manu_exact` genutzt worden, das vom Typ String ist.

Aufgrund der hierarchischen Struktur ist der Response komplexer. Im Grunde erhält man immer den Wert eines Felds mit der Anzahl der Dokumente – und das für alle Ebenen.

```
<response>
   ...
   <lst name="facet_counts">
      <lst name="facet_queries"/>
      <lst name="facet_fields"/>
      <lst name="facet_dates"/>
      <lst name="facet_ranges"/>
      <lst name="facet_pivot">
         <arr name="manu_exact,cat">
            <lst>
               <str name="field">manu_exact</str>
               <str name="value">Apache Software Foundation</str>
               <int name="count">2</int>
               <arr name="pivot">
                  <lst>
                     <str name="field">cat</str>
                     <str name="value">search</str>
                     <int name="count">2</int>
                  </lst>
                  <lst>
                     <str name="field">cat</str>
                     <str name="value">software</str>
                     <int name="count">2</int>
                  </lst>
               </arr>
            </lst>
            <lst>
               <str name="field">manu_exact</str>
               <str name="value">Belkin</str>
               ...
            </lst>
         </arr>
      </lst>
   </lst>
</response>
```

Im obigen Response sieht man, dass für den ersten Hersteller »Apache Software Foundation« zwei Dokumente indexiert worden sind. Die Dokumente sind mit den Kategorien `search` und `software` verschlagwortet worden. Dass jeder der Kategorien zwei Dokumente zugeordnet sind, liegt daran, dass die in den »exampledocs« zwei XML-Dateien (*solr.xml* und *utf8-example*) enthalten sind, die die gleichen Werte bei `manu_exact` und `cat` haben.

LocalParams

Die Facetten-Komponente kann die Syntax der LocalParams verarbeiten, um so Zusatzinformationen mitgegeben zu können, die die Facetten aufwerten.

Ein einfaches, aber effektives Beispiel hierfür ist die Änderung der Ausgabe mittels LocalParams. Die Query-Facette zum Beispiel liefert als Anzeigewert die Query an sich zurück. Dies ist selten für die Anzeige geeignet. Mit folgender Syntax kann man einen beliebigen Text anstatt der Query zurückgeben lassen:

```
facet.query={!key='mein text'}manufacturedate_dt:[NOW-1YEARS TO NOW]
```

Im Solr-Response wird dann im Attribut `name` der definierte Key zurückgegeben.

```
<int name="mein Text">0</int>
```

Dies vereinfacht die Verarbeitungslogik zur Anzeigezeit, denn man muss nun kein Mapping mehr im User-Interface durchführen.

Facetten in der Wikipedia

Das Velocity-Template, das mittels des `/browse`-RequestHandler angesprochen wird, ist auf die unterschiedlichsten Facetten-Arten vorbereitet. Auch der Index der Wikipedia beinhaltet auf den ersten Blick genug Daten für Facetten. Es bietet sich beispielsweise an, das Feld `user` als Feld-Facette zu definieren und mit dem Feld `timestamp` zeitbasierte Facetten zu erstellen.

Damit die Facetten in der Browse-UI angezeigt werden, muss der `/browse`-RequestHandler wie folgt angepasst werden:

```
<requestHandler name="/browse" class="solr.SearchHandler" startup="lazy">
   <lst name="defaults">
      ...
      <!-- faceting defaults -->
      <str name="facet">true</str>
      <str name="facet.mincount">1</str>
      <str name="facet.limit">5</str>
      <!-- field facet -->
      <str name="facet.field">user</str>
      <!-- query facet -->
      <str name="facet.query">{!key='last year'}timestamp:[NOW-1YEARS TO NOW]</str>
      <str name="facet.query">{!key='last 10 years'}timestamp:[NOW-10YEARS TO NOW]</str>
      <str name="facet.query">{!key='last 100 years'}timestamp:[NOW-100YEARS TO NOW]
      </str>
      <!-- range facet -->
      <str name="facet.range">timestamp</str>
      <str name="facet.range.start">NOW/YEAR-10YEARS</str>
      <str name="facet.range.end">NOW</str>
      <str name="facet.range.gap">+1YEAR</str>
      <str name="facet.range.other">before</str>
      <str name="facet.range.other">after</str>
      <!-- pivot facet -->
```

```
      <str name="facet.pivot">user,titleText</str>
    </lst>
  </requestHandler>
```

Nach der Änderung des RequestHandler muss der Solr-Core neu geladen werden, damit die Änderung greift. Ruft man nun das Browse-Interface auf und sucht etwas, findet man auf der linken Seite die einzelnen Facetten. In Abbildung 4-4 sieht man die Feld-Facette für das Feld *user* und die zeit-basierten Query- bzw. Date-Facetten. Bei der Query-Facette wurde noch mit den LocalParams die Ausgabe formatiert, sodass diese leserlich ist.

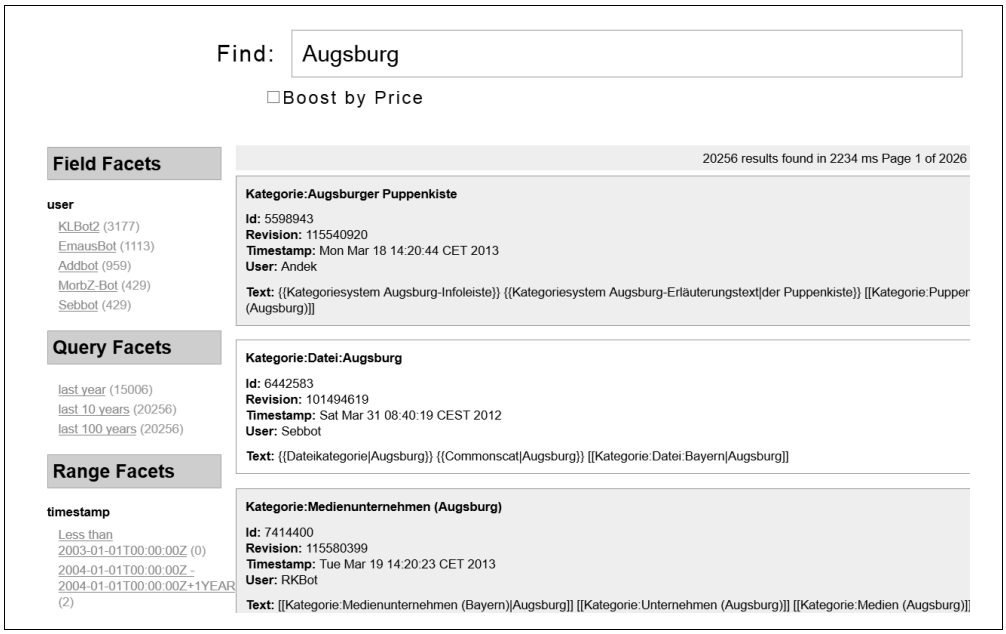


Abbildung 4-4: Trefferliste mit Facetten – Teil 1

Weiter unten auf der Trefferseite findet sich dann auch die Pivot-Facette, basierend auf den Feldern *user* und *title* (siehe Abbildung 4-5).

AutoSuggest – Suchbegriffe vorschlagen

Das Feature AutoSuggest ist häufig das erste, mit dem der Anwender in Berührung kommt. Üblicherweise werden dem Anwender, sobald er anfängt, eine Eingabe zu machen, Wörter oder auch Phrasen vorgeschlagen. Meist geschieht das als Drop-down-Box unterhalb des Eingabefelds. Was dabei vorgeschlagen wird, hängt ganz von den Daten bzw. der Intention des AutoSuggest ab. Ein E-Shop wie Amazon beispielsweise schlägt mit seinem AutoSuggest vorwiegend ganze Titel der Produkte vor und nicht etwa Wörter aus der Produktbeschreibung. Dies geht relativ gut, da die Dokumente mehr oder minder homogen sind. Bei Google ist dies anders. Google schlägt keine Titel vor, sondern Wörter bzw. Wortkombinationen, die im Dokument vorkommen.

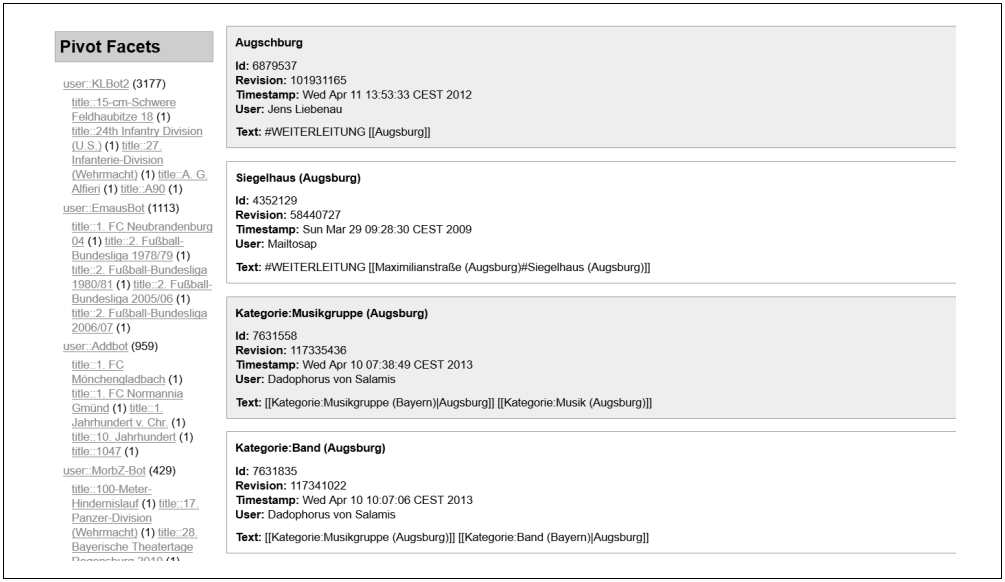


Abbildung 4-5: Trefferliste mit Facetten – Teil 2

Die Abbildungen 4-6 und 4-7 zeigen, wie unterschiedlich man mit dem AutoSuggest umgehen kann.

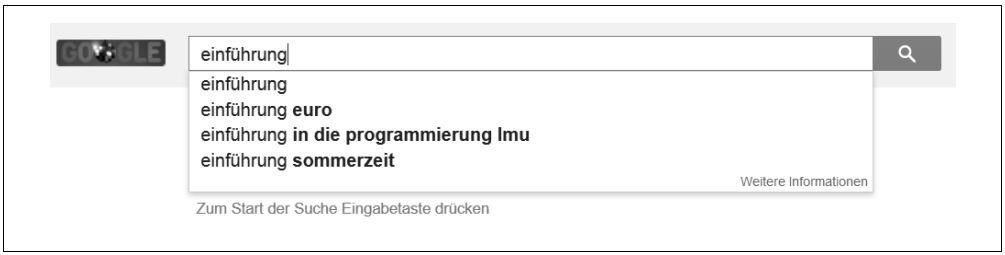


Abbildung 4-6: AutoSuggest von Google (Abbildung von http://www.google.de)

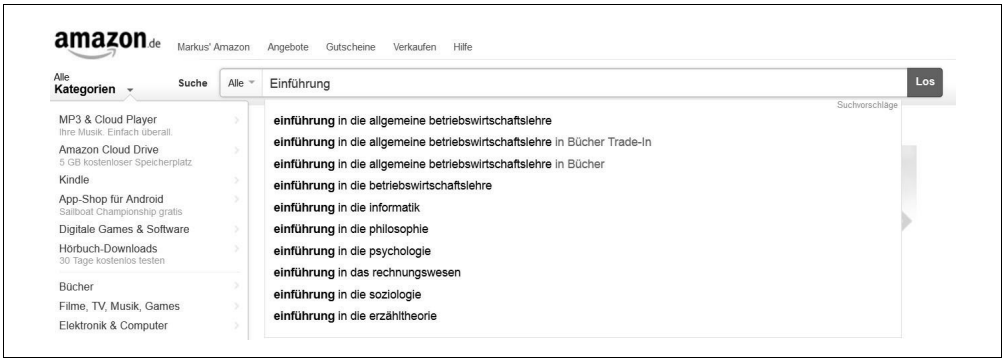


Abbildung 4-7: AutoSuggest von Amazon (Abbildung von http://www.amazon.de)

Ein gutes AutoSuggest vermindert drastisch die Fehlerquote bei der Rechtschreibung. Dadurch kommt es seltener zu Suchergebnissen mit 0 Treffern. Trefferlisten ohne Treffer beeinflussen das gefühlte Erlebnis des Anwenders meist negativ und sollten daher vermieden werden.

Solr bietet für das AutoSuggest unterschiedliche Lösungen an, die sich auf unterschiedliche SearchComponents beziehen. Die meisten AutoSuggests werden mit den folgenden Implementierungen umgesetzt:

- Spellcheck
- Facetten
- Terms

AutoSuggest mit Spellcheck

Die erste Variante wird Suggester genannt und basiert auf der Spellcheck-SearchComponent. Die Spellcheck-Komponente kann entweder mit einem Dictionary, einer Textdatei mit Vorschlägen oder mit den Daten des Index arbeiten.

Es empfiehlt sich eigentlich immer, mit den Daten des Index zu arbeiten. Ein Dictionary zu pflegen, bedeutet zusätzlichen Aufwand, und es besteht die Gefahr, dass Vorschläge des Dictionary nicht im Index enthalten sind und es so zu keinen Treffern führen würde.

Für das AutoSuggest mit dem Suggester muss die SearchComponent in der *solrconfig.xml* angelegt und im entsprechenden RequestHandler registriert werden. Für das AutoSuggest nutzt man optimalerweise einen eigenen RequestHandler, der nur die ersten fünf oder zehn Vorschläge zurückgibt, jedoch keine weiteren Informationen wie Trefferliste, Facetten etc. Das schont Ressourcen und garantiert, dass die AutoSuggest-Vorschläge in wenigen Millisekunden geliefert werden können.

Die SearchComponent muss nur wenig konfiguriert werden, um ein einfaches AutoSuggest zu ermöglichen. Es reicht, die Anzahl der Ergebnisse zu bestimmen und das Feld festzulegen, aus dem der AutoSuggest seine Vorschläge bezieht. Auf die wichtigsten Parameter der Spellcheck-Komponente gehen wir bei dem Feature »Meinten Sie ...« ein.

Der folgende Ausschnitt aus der *solrconfig.xml* zeigt einen `/suggest`-RequestHandler mit der konfigurierten SearchComponent.

```
<searchComponent class="solr.SpellCheckComponent" name="suggest">
   <lst name="spellchecker">
      <str name="name">default</str>
      <str name="classname">org.apache.solr.spelling.suggest.Suggester</str>
      <str name="lookupImpl">org.apache.solr.spelling.suggest.tst.TSTLookup</str>
      <str name="field">text</str>
   </lst>
</searchComponent>

<requestHandler class="org.apache.solr.handler.component.SearchHandler" name="/suggest">
```

```
    <lst name="defaults">
      <str name="spellcheck">true</str>
      <str name="spellcheck.onlyMorePopular">true</str>
      <str name="spellcheck.count">3</str>
    </lst>
    <arr name="components">
      <str>suggest</str>
    </arr>
  </requestHandler>
```

Der AutoSuggest-Request ist sehr simpel aufgebaut. Es wird nur ein Parameter (`q`) übergeben, der sich mit jedem getippten Zeichen ändert. Der folgende Request wird ausgeführt, wenn der Anwender »augs« in das Eingabefeld getippt hat.

```
http://localhost:8983/solr/de_wikipedia/suggest?q=augs
```

Als Ergebnis liefert Solr folgenden Response mit den drei besten Vorschlägen, basierend auf den Wikipedia-Daten.

```
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">106347</int>
  </lst>
  <lst name="spellcheck">
    <lst name="suggestions">
      <lst name="augs">
        <int name="numFound">3</int>
        <int name="startOffset">0</int>
        <int name="endOffset">4</int>
        <arr name="suggestion">
          <str>augsburg</str>
          <str>augsburger</str>
          <str>augsburg.de</str>
        </arr>
      </lst>
    </lst>
  </lst>
</response>
```

Möchte man – wie bei Amazon – Phrasen vorgeschlagen haben, muss man bei der Konfiguration der Spellcheck-SearchComponent ein Feld angeben, das nicht tokenized worden ist, d. h. ein Feld vom Typ String bzw. eines mit einem KeywordTokenizer.

AutoSuggest mit Facetten

Wie bereits im Abschnitt über die Facetten erwähnt, lässt sich mithilfe des Parameters `facet.prefix` ebenfalls ein AutoSuggest umsetzen. Eine zusätzliche Konfiguration ist dafür nicht notwendig. Es lohnt sich aber, auch hierfür einen eigenen RequestHandler zu definieren, der 0 Dokumente zurückgibt, denn diese Daten werden für das AutoSuggest nicht benötigt. Es geht jedoch auch, wie im folgenden Request gezeigt wird, den Parameter `rows` auf `0` zu setzen.

```
http://localhost:8983/solr/de_wikipedia/select?q=*:*&rows=0&facet=true&facet.field=title&
facet.prefix=Augs
```

Die Antwort von Solr sieht wie folgt aus:

```
<response>
   ...
   <lst name="facet_counts">
      ...
      <lst name="facet_fields">
         <lst name="title">
            <int name="Augsberg">1</int>
            <int name="Augsburg">1</int>
            <int name="Augsburg (1916)">1</int>
            <int name="Augsburg (1931)">1</int>
            <int name="Augsburg (2008)">1</int>
            <int name="Augsburg (Arkansas)">1</int>
            ...
         </lst>
      </lst>
      ...
   </lst>
</response>
```

Die Einträge aus der Facette dienen nun dem Anwender als Vorschlag, den er nutzen kann, um die eigentliche Suche durchzuführen.

Im obigen Beispiel wurde für das Feld `title` genutzt, das vom Typ String und somit case-sensitiv ist, weswegen hier bei `facet.prefix` mit einem Großbuchstaben angefangen werden muss. Natürlich kann man auch andere Felder und Feldtypen nutzen.

Das AutoSuggest des E-Shops »reuter.de« nutzt beispielsweise mehrere Facetten für das AutoSuggest und kann so Vorschläge aus mehreren Bereichen generieren (Abbildung 4-8).

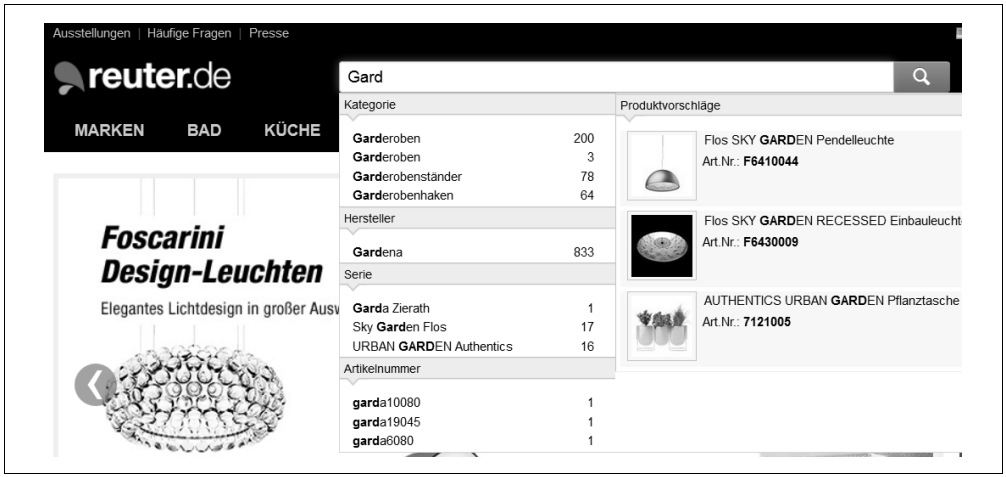


Abbildung 4-8: AutoSuggest mit Facetten bei reuter.de (Abbildung von http://www.reuter.de)

AutoSuggest mit der Terms-Komponente

Die Terms-Komponente wird später gesondert betrachtet (siehe Abschnitt »Terms-Komponente – Solr-Felder auslesen« auf Seite 178); dann wird auch die Möglichkeit des AutoSuggest mit der Terms-Komponente beschrieben.

AutoSuggest in der Wikipedia

Das Browse-Interface ist in ihrer Standardkonfiguration darauf ausgelegt, das AutoSuggest mit der Terms-Komponente zu realisieren. Bei der Beschreibung der Terms-Komponente wird auch gezeigt, wie man dies in das Velocity-Template einfügt.

Highlighting – Suchbegriffe im Treffer hervorheben

Highlighting ist wichtig für den Anwender, damit er schnell erkennt, warum ein Treffer ein Treffer ist. Wenn ich nach einem Begriff suche, möchte ich, dass dieser aus der Trefferliste heraussticht. Dies übernimmt in Solr die Highlighting-Komponente.

Um das Highlighting nutzbar zu machen, muss man das Feature nur konfigurieren und aktivieren. Ein separates Registrieren im RequestHandler ist nicht notwendig, da Highlighting eine der sechs Standardkomponenten ist. Mit folgenden wichtigen Parametern kann das Highlighting kontrolliert werden:

hl
: Wird dieser Parameter auf `true` gesetzt, ist das Highlighting aktiviert.

hl.q
: Per Default wird nach den Suchbegriffen aus dem `q`-Parameter gesucht, damit diese durch den Highlighter markiert werden können. Mit `hl.q` kann ein alternativer Text definiert werden, der den `q`-Parameter für das Highlighting überschreibt.

hl.fl
: Mit diesem Parameter werden die Felder definiert, in denen der Suchbegriff markiert werden soll. Es muss sich hier nicht zwingend um die gleichen Felder handeln, die auch für die Suche genutzt werden.

hl.snippets
: Mit diesem Parameter kann man die Anzahl der dargestellten Fundstellen (Snippets) beeinflussen. Per Default wird nur ein einziges Snippet generiert.

hl.fragsize
: Mit diesem Parameter kann man die Größe der Snippets beeinflussen. Per Default sind 100 Zeichen für ein Snippet eingestellt.

hl.alternateField
: Mit diesem Parameter wird ein zweites Feld angegeben, dessen Inhalt angezeigt wird, sollte Solr für dieses Dokument kein Snippet generieren können. Somit ist sichergestellt, dass die Highlighting-Komponente immer etwas zurückliefert.

hl.maxAlternatFieldLength
: Mit diesem Parameter wird konfiguriert, wie viele Zeichen des Alternativfelds angezeigt werden. Das ist vor allem dann sehr nützlich, wenn man als Backup das Volltextfeld angegeben hat. Dies ist somit eine Möglichkeit, sich einen Teaser zur Laufzeit generieren zu lassen. Wird dieser Parameter nicht spezifiziert, wird das gesamte Feld ausgegeben.

hl.formatter
: Mit diesem Parameter kann man die Formatter-Implementierung spezifizieren, die die gefundenen Stellen der Suchbegriffe markiert. Der Standard-Formatter fügt vor und nach der Fundstelle frei definierbaren Text ein.

hl.simple.pre
: Dieser Parameter spezifiziert, was vor der Fundstelle eingefügt wird. Default ist `<em>`.

hl.simple.post
: Dieser Parameter spezifiziert, was nach der Fundstelle eingefügt wird. Default ist `</em>`.

hl.fragmenter
: Mit diesem Parameter kann man die Implementierung des Fragmenters bestimmen, der die einzelnen Fundstellen generiert. Per Default werden Fragmente mit fester Anzahl an Buchstaben generiert.

hl.useFastVectorHighlighting
: Wird dieser Parameter auf `true` gesetzt, wird der FastVectorHighlighter verwendet. Dieser ist schneller, was die Ausführungszeit betrifft, jedoch ist für ihn notwendig, dass in den Feldern in der *schema.xml* die Attribute `termOffsets`, `termPositions` und `termVectors` auf `true` gesetzt sind.

Die vollständige Liste der Parameter finden Sie im Solr-Wiki:

http://wiki.apache.org/solr/HighlightingParameters

Die Alternativen für `hl.fragmenter`, `hl.formatter` etc. finden sich bereits in der *solrconfig.xml*. In der Highlight-SearchComponent ist unter anderem der ScoreOrderFragmentBuilder enthalten, der die einzelnen Fragmente nach einem internen Score sortiert. Je mehr Suchbegriffe im Fragment, desto weiter oben kommt das Fragment. Ein Experimentieren mit diesen Alternativen lohnt sich allemal.

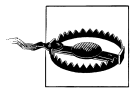
Die minimale Konfiguration für das Highlighting sind die Parameter `hl` und `hl.fl`.

```
http://localhost:8983/solr/collection1/select?q=apache&hl=true&hl.fl=manu
```

Ist die Highlighting-Komponente aktiviert, wird vom Solr ein zusätzlicher Block `highlighting` erstellt. Dieser Block enthält für die Dokumente, die im `response`-Block enthalten sind, die Highlighting-Fragmente.

```
<response>
   ...
   <result name="response" numFound="2" start="0">
      <doc>
         <str name="id">SOLR1000</str>
         ...
      </doc>
      <doc>
         <str name="id">UTF8TEST</str>
         ...
      </doc>
   </result>
   <lst name="highlighting">
      <lst name="SOLR1000">
         <arr name="manu">
            <str><em>Apache</em> Software Foundation</str>
         </arr>
      </lst>
      <lst name="UTF8TEST">
         <arr name="manu">
            <str><em>Apache</em> Software Foundation</str>
         </arr>
      </lst>
   </lst>
</response>
```

Für die Ausgabe im User-Interface muss man lediglich die Verbindung zwischen dem Response und dem Highlighting über den Unique Key herstellen.



Auch beim Highlighting gilt, dass die Komponente von der Analyse des Feldtyps beeinflusst wird. Werden Stemmer für das Highlighting-Feld genutzt, kann unter Umständen ein falsches Wort markiert werden, das jedoch den gleichen Wortstamm hat. Wird ein Feld vom Typ String genutzt, ist die Wahrscheinlichkeit hoch, dass nichts markiert wird, da der Highlighter die einzelnen Term in diesem Feld nicht finden kann.

Highlighting in der Wikipedia

Wie man in Abbildung 4-9 sieht, wird das Trefferdokument im Browse-Interface eigentlich bereits gut dargestellt. Die Treffer sind klar voneinander abgegrenzt, der Titel des Dokuments wird fett angezeigt und ist prominent platziert. Was jedoch noch fehlt, ist der Zusammenhang zwischen eingegebenem Suchbegriff und dem Treffer bzw. der Fundstelle.

Auch wenn in der aktuellen Konfiguration des `/browse`-RequestHandler nur das Feld `text` durchsucht wird, kann man für das Highlighting jedes beliebige Feld nutzen. Basierend auf dem obigen Screenshot, sind sowohl das Feld `title` als auch das Feld `text` gute Kandidaten für das Highlighting. Um beide Felder zu highlighten, indem der Suchbegriff unterstrichen und kursiv dargestellt werden soll, muss die Konfiguration des `/browse`-RequestHandler wie folgt erweitert werden:

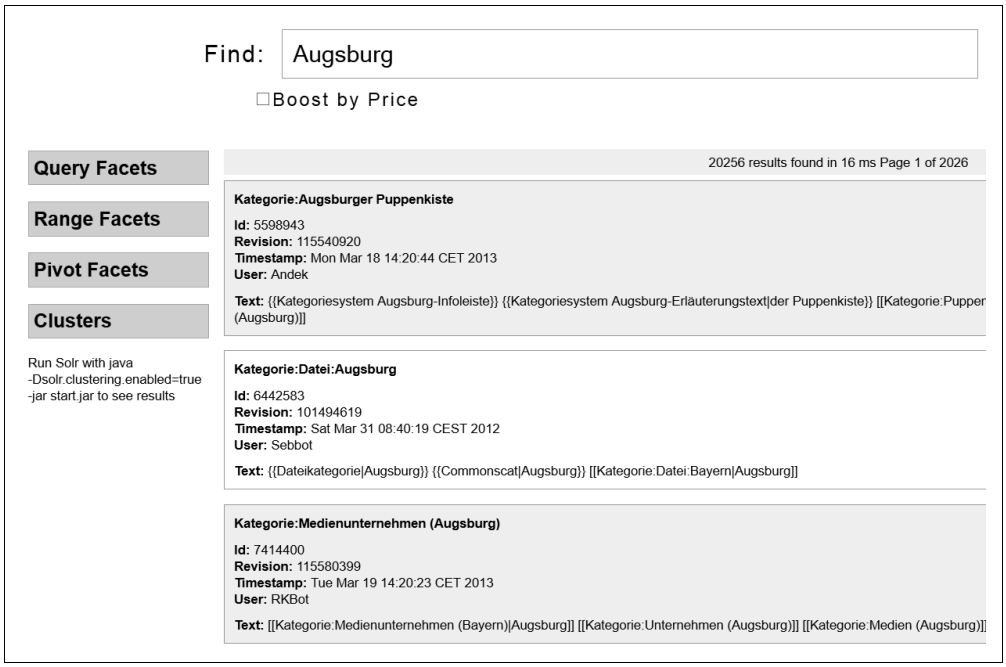


Abbildung 4-9: Trefferliste ohne Highlighting

```
<requestHandler name="/browse" class="solr.SearchHandler" startup="lazy">
   <lst name="defaults">
      ...
      <!-- highlighting -->
      <str name="hl">false</str>
      <str name="hl.fl">titleText text</str>
      <str name="hl.simple.pre">&lt;u&gt;&lt;i&gt;</str>
      <str name="hl.simple.post">&lt;/i&gt;&lt;/u&gt;</str>
   </lst>
</requestHandler>
```

Wie man in der obigen Konfiguration sieht, müssen die HTML-Tags durch ihrer Entities dargestellt werden, da es ansonsten zu einem Fehler beim Parsen der *solrconfig.xml* kommt.

Nach dem obligatorischen Reload des Solr-Core und dem erneuten Ausführen der Suche ist das Highlighting aktiv, wie man in Abbildung 4-10 sehen kann.

Gesucht wurde »Augsburg«, und auch nur die Fundstellen dieses Worts werden entsprechend formatiert. Möchten man nun zusätzlich ähnliche Schreibweisen des Worts, wie beispielsweise »Augsburger«, ebenfalls markiert haben, muss der Feldtyp in der *schema.xml* um Tokenizer, z. B. Stemmer, erweitert werden.

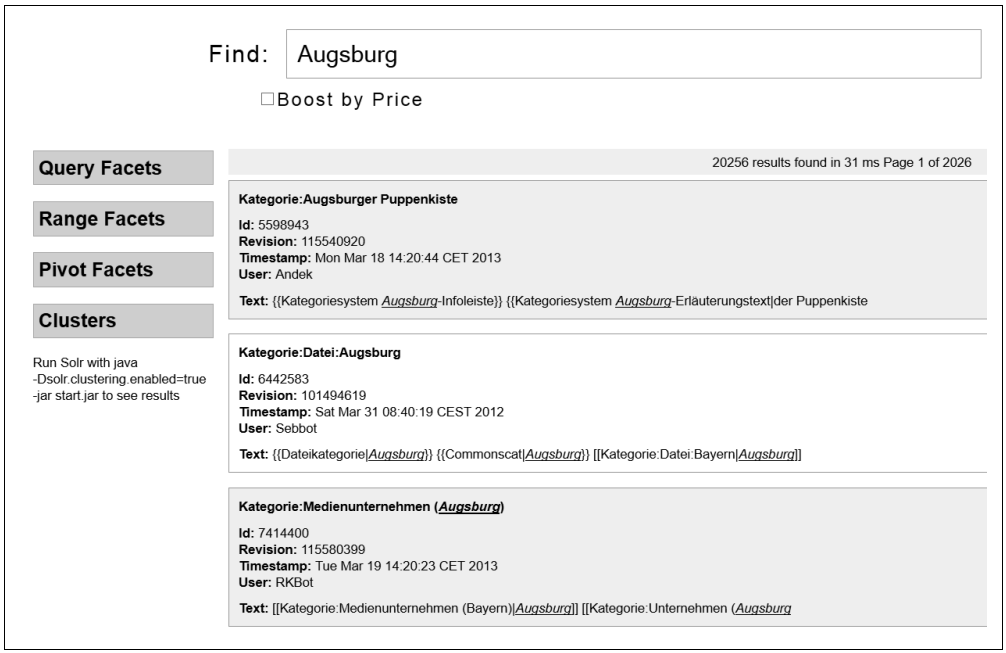


Abbildung 4-10: Trefferliste mit Highlighting

Result Grouping – ähnliche Dokumente gruppieren

Beim Result Grouping handelt es sich um eine alternative Darstellung der Trefferliste. Traditionell werden die Dokumente nacheinander dargestellt, wobei sie nach ihrem Score-Wert sortiert worden sind. Es besteht bei dieser Darstellung keinerlei Zusammenhang zwischen diesen Dokumenten. Es gibt jedoch immer Zusammenhänge, beispielsweise lassen sich die Dokumente in Kategorien einteilen, die man in der Trefferliste auch darstellen möchte.

Das Result Grouping bildet basierend auf den Inhalten eines Felds Gruppen und sortiert die Dokumente in diese Gruppen ein. Somit verhält sich das Result Grouping wie die Feld-Facette, eben nur in der Trefferliste.

Das Feature Result Grouping wird auch Field Collapsing genannt, da man in vielen Suchapplikationen die Dokumentliste auf- bzw. zuklappen kann. Google nutzt Result Grouping beispielsweise bei der Suche nach News, um gleiche Neuigkeiten zu gruppieren, wobei initial nur ein Dokument angezeigt wird und weitere Quellen aufgeklappt werden können (siehe Abbildung 4-11).

Das Result Grouping zu aktivieren, geht einfach. Man muss jedoch bedenken, dass, sobald man das Grouping-Feature aktiviert, die normale Trefferliste – das Response-Element – nicht mehr von Solr ausgeliefert wird. Mit folgenden Parametern kann man das Grouping konfigurieren:

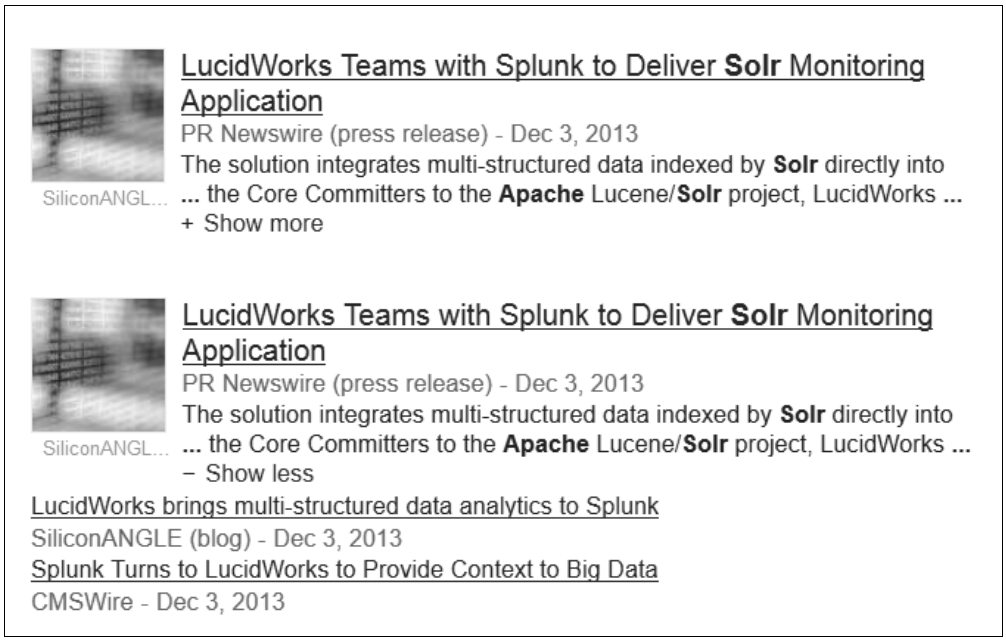


Abbildung 4-11: Result Grouping bei Google (Abbildung von http://www.google.de)

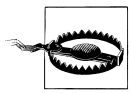
group
: Mit diesem Parameter aktiviert man das Grouping.

group.field
: Dieser Parameter bestimmt das Feld, nach dem die Gruppen gebildet werden.

group.limit
: Dieser Parameter bestimmt, wie viele Dokumente pro Gruppe angezeigt werden. Per Default wird nur ein Dokument zurückgeliefert.

Die komplette Liste der Parameter finden Sie im Solr-Wiki:

http://wiki.apache.org/solr/FieldCollapsing



Die allgemeinen Request-Parameter zum Paging (`rows` und `start`) beziehen sich beim Result Grouping auf die Gruppen, d. h., wie viele Gruppen angezeigt werden etc. `rows` hat keinen Einfluss darauf, wie viele Dokumente je Gruppe angezeigt werden, dies wird über `group.limit` konfiguriert.

Folgender Request aktiviert das Grouping auf dem Feld `manu_exact`:

```
http://localhost:8983/solr/collection1/select?q=*:*&group=true&group.field=manu_exact
```

Der Solr-Response enthält nun ein neues Element `grouped`, in dem die Dokumente nach ihren Herstellern, beispielsweise »Samsung Electronics Co. Ltd.« einsortiert sind.

```
<response>
   ...
   <lst name="grouped">
      <lst name="manu_exact">
         <int name="matches">32</int>
         <arr name="groups">
            ...
            <lst>
               <str name="groupValue">Samsung Electronics Co. Ltd.</str>
               <result name="doclist" numFound="1" start="0">
                  <doc>
                     <str name="id">SP2514N</str>
                     ...
                  </doc>
               </result>
            </lst>
            ...
         </arr>
      </lst>
   </lst>
</response>
```

Grouping in der Wikipedia

Das Browse-Interface ist auf Grouping vorbereitet. Im Kopfbereich befindet sich der Link *Group By*. Aktiviert man diesen, kommt man zur Ansicht mit den gruppierten Treffern. Diese Ansicht wird jedoch nicht über einen RequestHandler definiert, sondern ist im Velocity-Template selbst versteckt.

Es bietet sich bei dem Wikipedia-Index an, die Dokumente nach ihren Autoren zu gruppieren, d. h., alle Dokumente eines Autors bilden eine Gruppe.

Wir müssen nur wenige Dateien anpassen, um das Grouping für den Wikipedia-Index zu aktivieren. Zum einem muss die *tabs.vm* angepasst werden, die den initialen Request mit den Grouping-Parametern enthält. Dort muss bei dem Parameter `group.field` das Feld `user` eingetragen werden. Die komplette *tabs.vm* sieht dann wie folgt aus:

```
##TODO: Make some nice tabs here
#set($queryOpts = $params.get("queryOpts"))
<span #annTitle("Click the link to demonstrate various Solr capabilities")><span>Examples
: </span><span class="tab">#if($queryOpts && $queryOpts != "")<a href="#url_for_home/?
#debug#annotate">Simple</a>#{else}Simple#end</span>
<span class="tab">#if($queryOpts == "spatial")Spatial#else<a href="#url_for_home?&query
Opts=spatial#debug#annotate">Spatial</a>#end</span>
<span class="tab">#if($queryOpts == "group")Group By#else<a href="#url_for_home?#debug#
annotate&queryOpts=group&group=true&group.field=user">Group By</a>#end</span></span>
<hr/>
```

Als Nächstes muss die *queryGroup.vm* angepasst werden. In dieser Datei ist das Dropdown enthalten, das sich auf der Grouping-Seite befindet. Die *queryGroup.vm* muss wie folgt angepasst werden:

```
#set($queryOpts = $params.get("queryOpts"))
#if($queryOpts == "group")
<div>
        #set($groupF = $request.params.get('group.field'))
        <label #annTitle("Add the &group.field parameter. Multiselect is supported")>
Group By:
          <select id="group" name="group.field" multiple="true">
            ##TODO: Handle multiple selects correctly
            <option value="none"
            #if($groupF == '')selected="true"#end>No Group</option>
            <option value="user"
            #if($groupF == 'user')selected="true"#end>User</option>

          </select>
        </label>
<input type="hidden" name="group" value="true"/>
</div>

#end
```

Zu guter Letzt muss nur noch die Trefferdarstellung an den Wikipedia-Index angepasst werden. Hierfür wird die *hitGrouped.vm* wie folgt angepasst, sodass unsere Konfiguration der Trefferdarstellung eingebunden wird:

```
<div class="result-document">
  <div class="result-title"><b>$grouping.key</b></div>
  <div>Total Matches in Group: $grouping.value.matches</div>
  <div>#foreach ($group in $grouping.value.groups)
    <div class="group-value">$group.groupValue <span #annTitle("The count of the number
    of documents in this group")>($group.doclist.numFound)</span></div>
    <div class="group-doclist" #annTitle("Contains the top scoring documents in the
    group")>
      #foreach ($doc in $group.doclist)
        #set($docId = $doc.getFieldValue('id'))
          #parse("wikipedia.vm")
      #end
    </div>
    #end</div>
  </div>
  #if($params.getBool("debugQuery",false))
    <a href="#" onclick='jQuery(this).siblings("pre").toggle(); return false;'>toggle
    explain</a>
    <pre style="display:none">$response.getExplainMap().get($doc.getFirstValue('id'))
    </pre>
  #end
</div>
```

Nach dem Neuladen des Solr-Core sieht man im Browse-Interface, wenn man nach »Augsburg« sucht und sich die Gruppierung anzeigen lässt, das Resultat aus Abbildung 4-12.

Hier sehen Sie die ersten Dokumente der User *Sebbot* und *RKBot*, die 429 bzw. 5 Dokumente in unseren Beispielindex haben.

Abbildung 4-12: Grouping-Resultat

Meinten Sie … – Tippfehler ausbessern

Oft kommt es vor, dass man die genaue Schreibweise eines Begriffs oder eines Namens nicht kennt oder sich einfach nur vertippt. Sucht man nun dennoch, kann es daher sehr leicht passieren, dass man eine leere Trefferliste bekommt oder ein völlig falsches Ergebnis. Für den Suchenden ist eine leere Trefferliste fast das Schlimmste, was passieren kann. Tritt dies öfter auf, ist es durchaus möglich, dass der Suchende das Vertrauen in die Suche verliert und zu einem anderen Produkt bzw. Anbieter wechselt.

Um das zu vermeiden, gibt es die »Meinten Sie ...«-Funktionalität, die auch DidYouMean genannt wird. Diese Funktionalität nutzt die Spellcheck-Komponente, die ja auch von AutoSuggest verwendet wird, um diesmal nachgelagert alternative Suchbegriffe vorzuschlagen. Diese Vorschläge werden von Solr in den Response der Suche mit aufgenommen und können dann auf der Nutzeroberfläche angezeigt werden.

Manche Anbieter von Suchen, wie beispielsweise Google, gehen sogar einen Schritt weiter. Anstatt die vermeintlich falsche und womöglich leere Trefferliste anzuzeigen, wird direkt eine zweite Suche mit der besten Alternative durchgeführt. In Abbildung 4-13 wird von Google direkt nach dem richtig geschriebenen Begriff »Augsburg« gesucht.

Um die »Meinten Sie ...«-Funktionalität zu nutzen, muss eine SearchComponent erstellt und im entsprechenden RequestHandler definiert werden. Bei der Spellcheck-Komponente gibt es mehrere unterschiedliche Implementierungen, die es beispielsweise erlauben, Vorschläge aus verschiedenen Quellen generieren zu lassen.

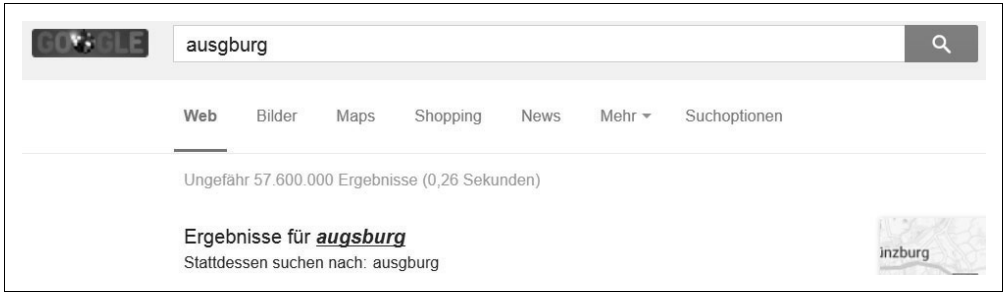


Abbildung 4-13: »Meinten Sie …« bei Google (Abbildung von http://www.google.de)

Suggester

Die Suggester-Spellcheck-Variante, die bereits beim AutoSuggest beschrieben wurde, kann natürlich auch auch genutzt werden, um für die »Meinten Sie ...«-Funktionalität Vorschläge zu generieren (siehe Abschnitt »AutoSuggest mit Spellcheck« auf Seite 153).

IndexBasedSpellChecker

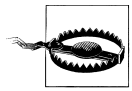
Der IndexBasedSpellChecker baut aus den indexierten Dokumenten einen separaten Solr-Index und nutzt diesen, um die Vorschläge zu generieren. Hierfür muss in der *schema.xml* des Original-Solr-Core ein Feld definiert werden, das als Datenlieferant für den »Spellcheck-Index« dient. Es ist üblich, hier per CopyField-Statements alle relevanten Felder in das Feld zu kopieren.

Die SearchComponent in der *solrconfig.xml* selbst benötigt nur wenig Konfiguration.

spellcheckIndexDir
: Dieser Parameter gibt den Namen des Solr-Core an, in dem die Spellcheck-Vorschläge gespeichert werden sollen. Da man dies für jeden SearchComponent einzeln definieren muss, kann man somit auch unterschiedliche Spellcheck-Indexe für unterschiedliche Szenarien definieren.

field
: Mit diesem Parameter definiert man das Feld mit den Quelldaten für den Spellcheck-Index.

buildOnCommit
: Mit diesem Parameter wird festgelegt, ob der Index bei jedem Commit neu aufgebaut werden soll oder nicht. Ist dieser Parameter nicht oder auf `false` gesetzt, sind der Original-Index und der Spellcheck-Index nicht synchron, was dazu führen kann, dass Wörter neuer Dokumente nicht vorgeschlagen werden oder Vorschläge generiert werden, die ins »Leere« führen, da das entsprechende Dokument gelöscht worden ist.



Setzt man den Parameter *buildOnCommit* auf `true` und hat eine hohe Frequenz an Updates, kann dies zu einer schlechten Indexierungsperformance führen. In diesem Fall sollte man lieber auf die DirectSolrSpellCheck-Variante wechseln.

Folgendes Beispiel zeigt eine Konfiguration des IndexBasedSpellChecker in der *solrconfig.xml*:

```
<searchComponent name="spellcheck" class="solr.SpellCheckComponent">
   <lst name="spellchecker">
      <str name="classname">solr.IndexBasedSpellChecker</str>
      <str name="spellcheckIndexDir">./spellcheck</str>
      <str name="field">text</str>
      <str name="buildOnCommit">true</str>
   </lst>
</searchComponent>
```

Der IndexBasedSpellChecker lohnt sich vor allem dann, wenn man mehrere verschiedene Indexe in seiner Solr-Instanz hat, die dann auf einen zentralen Spellcheck-Index zugreifen können.

DirectSolrSpellChecker

Der DirectSolrSpellChecker funktioniert im Grunde so wie der IndexBasedSpellChecker, nur mit dem Unterschied, dass Solr hier keinen parallelen Index aufbaut, der die Vorschläge enthält. Aber genau wie in der anderen Variante muss man hier ein Feld definieren, aus dem die Vorschläge generiert werden, und auch bei dieser Variante kopiert man sich alle relevanten Felder per CopyField-Statement zusammen.

Die DirectSolrSpellChecker-Implementierung bietet noch ein paar Parameter, mit denen die Qualität und Quantität der generierten Vorschläge eingestellt werden können.

maxEdits
: Dieser Parameter spezifiziert, wie viele Änderungen gemacht werden dürfen, um vom Suchbegriff zum Vorschlag zu kommen.

minPrefix
: Dieser Parameter definiert die minimale Anzahl an Buchstaben, die sowohl der Vorschlag als auch der Suchbegriff gemeinsam haben müssen.

minQueryLength
: Dieser Parameter definiert die minimale Anzahl an Buchstaben im Suchbegriff, ab der Vorschläge generiert werden.

maxQueryFrequency
: Dieser Parameter definiert, wie oft ein Term im Dokumentenbestand enthalten sein muss, damit er als Vorschlag infrage kommen darf.

Die folgende Konfiguration der SearchComponent ist ein Beispiel dafür, wie man sie in der *solrconfig.xml* hinterlegen kann.

```
<searchComponent name="spellcheck" class="solr.SpellCheckComponent">
   <lst name="spellchecker">
      <str name="name">default</str>
      <str name="field">name</str>
      <str name="classname">solr.DirectSolrSpellChecker</str>
      <str name="distanceMeasure">internal</str>
      <float name="accuracy">0.5</float>
      <int name="maxEdits">2</int>
      <int name="minPrefix">1</int>
      <int name="maxInspections">5</int>
      <int name="minQueryLength">4</int>
      <float name="maxQueryFrequency">0.01</float>
      <float name="thresholdTokenFrequency">.01</float>
   </lst>
</searchComponent>
```

FileBasedSpellChecker

Der FileBasedSpellchecker nutzt ein Dictionary, um die Vorschläge zu generieren. Dies ist vor allem dann sehr vorteilhaft, wenn man mit dem »Meinten Sie ...«-Feature beispielsweise nur Eigennamen »korrigieren« möchte. Diese können dann in einer Datei gepflegt und in die Spellcheck-Konfiguration aufgenommen werden.

Dieses Dictionary bedarf jedoch wiederum manueller Pflege. Somit sind die generierten Vorschläge nur so gut wie das Dictionary.

Für diese Implementierung müssen folgende Parameter konfiguriert werden:

sourceLocation
: Dieser Parameter spezifiziert die Datei, die als Grundlage für die generierten Vorschläge dienen soll.

characterEncoding
: Dieser Parameter spezifiziert das Encoding, in dem das Dictionary vorliegt.

Mit diesen Parametern sieht eine entsprechende Konfiguration der SearchComponent wie folgt aus:

```
<searchComponent name="spellcheck" class="solr.SpellCheckComponent">
      <lst name="spellchecker">
      <str name="classname">solr.FileBasedSpellChecker</str>
      <str name="name">file</str>
      <str name="sourceLocation">spellings.txt</str>
      <str name="characterEncoding">UTF-8</str>
      <str name="spellcheckIndexDir">./spellcheckerFile</str>
   </lst>
</searchComponent>
```

Zusätzlich zur Konfiguration der SearchComponent kann man im RequestHandler das Feature »Meinten Sie ...« mit folgenden Parametern tunen bzw. konfigurieren:

spellcheck
: Mit diesem Parameter wird das »Meinten Sie ...«-Feature aktiviert.

spellcheck.collate
: Wenn mehrere Begriffe gesucht werden, generiert Solr für jeden Begriff einzeln Alternativen. Wird `spellcheck.collate` auf `true` gesetzt, wird zusätzlich die Kombination aus allen Topalternativen als separater Vorschlag generiert.

spellcheck.count
: Dieser Parameter spezifiziert, wie viele Alternativvorschläge pro Suchbegriff generiert werden sollen.

spellcheck.onlyMorePopular
: Wird dieser Parameter auf `true` gesetzt, generiert Solr nur Vorschläge, die *mehr* Treffer ermitteln würden als der aktuelle Begriff.

Eine vollständige Liste der Parameter finden Sie im Solr-Wiki:

http://wiki.apache.org/solr/SpellCheckComponent

»Meinten Sie ...« in der Wikipedia

Wie die meisten Features ist auch die »Meinten Sie ...«-Funktionalität bereits im Velocity-Template enthalten und muss nur noch entsprechend konfiguriert werden. Es müssen lediglich ein paar Einträge im `/browse`-Handler erstellt werden, um das Spellcheck einzubinden und zu konfigurieren.

```
<requestHandler name="/browse" class="solr.SearchHandler" startup="lazy">
   <lst name="defaults">
      ...
      <!-- did you mean -->
      <str name="spellcheck">true</str>
      <str name="spellcheck.onlyMorePopular">true</str>
      <str name="spellcheck.count">3</str>
      <str name="spellcheck.collate">true</str>
   </lst>
   <arr name="last-components">
      <str>suggest</str>
   </arr>
</requestHandler>
```

Zusätzlich muss eine Search-Komponente mit dem Namen `suggest` eingebaut werden. Für dieses Beispiel kann einfach auf die Suggester-Variante zurückgegriffen werden.

```
<searchComponent class="solr.SpellCheckComponent" name="suggest">
   <lst name="spellchecker">
      <str name="name">default</str>
      <str name="classname">org.apache.solr.spelling.suggest.Suggester</str>
      <str name="lookupImpl">org.apache.solr.spelling.suggest.tst.TSTLookup</str>
```

```
      <str name="field">text</str>
    </lst>
  </searchComponent>
```

Für einen deutschen Text bei der Anzeige von »Meinten Sie ...« kann man die *did_you_mean.vm* wie folgt anpassen:

```
#set($dym = $response.response.spellcheck.suggestions.collation)
#if($dym)
  Meinten Sie ...
<a href="#{url_for_home}#{lensNoQ}&q=$esc.url($dym)">$esc.html($dym)</a>?
#end
```

Nach dem Neuladen des Solr-Core steht die Funktionalität im Browse-Interface zur Verfügung. Wenn man nun nach »aug« sucht, wird, wie in Abbildung 4-14 zu sehen ist, ein alternativer Vorschlag angezeigt.

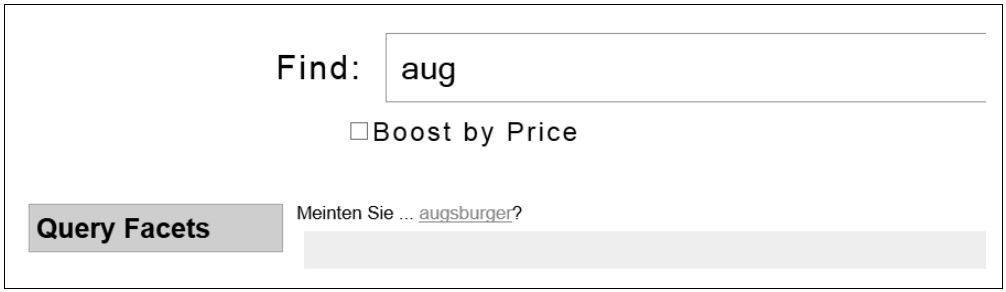


Abbildung 4-14: Meinten Sie ...

Der hier angezeigte Vorschlag basiert auf dem durch `spellcheck.collate` generierten, d. h., es wird aktuell immer nur ein alternativer Vorschlag angezeigt, unabhängig davon, wie viele Wörter man eingibt und wie `spellcheck.count` konfiguriert ist.

MoreLikeThis – ähnliche Dokumente finden

Das MoreLikeThis-Feature ermittelt zu einem Dokument ähnliche Dokumente, wobei die Ähnlichkeit durch die Daten und den Use Case definiert wird. Ähnlich bedeutet nicht zwingend, dass es inhaltlich um das Gleiche gehen muss. Viele E-Shops nutzen das MoreLikeThis-Feature, um Zubehör zu einem Produkt zu finden.

Der klassische Use Case für das MoreLikeThis-Feature ist aber dennoch, thematisch relevante Dokumente zu identifizieren und diese dem Suchenden zu präsentieren. Sehr präsent sind solche Vorschläge in E-Shops, die analysieren, was eine Person gesucht oder welche Produkte man sich angeschaut hat. Basierend auf diesen Informationen, können die E-Shops dann gezielt Vorschläge für andere Produkte generieren, wie in Abbildung 4-15 zu sehen ist.

Newsportale nutzen das MoreLikeThis-Feature, um zu einer Meldung andere Meldungen anzuzeigen, die im gleichen geografischen Gebiet anzusiedeln sind, im gleichen Zeit-

Abbildung 4-15: MoreLikeThis bei Amazon (Abbildung von http://www.amazon.de)

raum stattfinden oder sich um die gleiche Person drehen etc. Dabei werden diese Verwandtschaftsbeziehungen nicht manuell gepflegt, sondern während der Suchzeit berechnet.

Technisch gesehen, werden dabei die TermVectoren[1] von Dokumenten miteinander verglichen. Dokumente, die ähnliche TermVectoren haben, zählen dann als ähnlich.

Um in Solr das MoreLikeThis-Feature nutzen zu können, muss daher für das Feld, das analysiert werden soll, das Property `termVectors` auf `true` gesetzt werden. Diese Einstellung wird in der *schema.xml* bei der Definition des entsprechenden Felds durchgeführt:

```
<field name="category" ... termVectors="true" />
```

Im Solr gibt es für die Nutzung der MoreLikeThis-Funktionalität zwei generelle Ansätze. Bei dem ersten Ansatz wird während der Suche zu jedem Dokument der Trefferliste nach ähnlichen Dokumenten gesucht. Beim zweiten Ansatz wird ein eigener RequestHandler für MoreLikeThis erstellt, der nur bei Bedarf ähnliche Dokumente ermittelt. Dieser RequestHandler sucht dann zu einem definierten Input, wie beispielsweise einem Suchwort oder einer Dokument-ID, entsprechend ähnliche Dokumente.

Bei beiden Ansätzen kann die MoreLikeThis-Komponente durch folgende Parameter konfiguriert und beeinflusst werden:

mlt
: Mit diesem Parameter aktiviert man die Komponente.

mlt.count
: Mit diesem Parameter spezifiziert man, wie viele ähnliche Dokumente man vorgeschlagen bekommen möchte.

mlt.fl
: Mit diesem Parameter wird die Liste der Felder definiert, die für das MoreLikeThis-Feature analysiert werden.

1 TermVectoren beinhalten Informationen über ein Feld. Es ist in den TermVectoren gespeichert, welche Terme enthalten sind und wie oft sie vorkommen.

mlt.mintf
: Dieser Parameter gibt an, wie oft ein Term innerhalb eines Dokuments vorkommen muss, damit er vom MoreLikeThis für den Vergleich verwendet wird.

mlt.mindf
: Dieser Parameter gibt an, in wie vielen Dokumenten ein Term enthalten sein muss, damit er vom MoreLikeThis für den Vergleich verwendet wird.

Eine vollständige der Liste der Parameter finden Sie im Solr-Wiki:

http://wiki.apache.org/solr/MoreLikeThis

Nachfolgend werden wir die unterschiedlichen Nutzungsweisen des MoreLikeThis-Features aufzeigen.

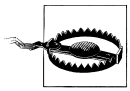
Variante MoreLikeThis-Component

Die Variante, die MoreLikeThis-Funktionalität als Search-Komponente zu nutzen, ist die am meisten genutzte, da sie den geringsten Konfigurationsaufwand bedeutet, man keine separate Suche ausführen muss und direkt in der Trefferliste für alle Dokumente ähnliche Dokumente vorschlagen kann.

Um diese Variante zu nutzen, muss man – bis auf die Schema-Anpassungen – nur die Parameter `mlt` und `mlt.fl` konfigurieren, und Solr generiert ein weiteres Element `moreLikeThis` im Response. Analog zum Highlighting wird hier die Verbindung vom Trefferdokument zu den Vorschlägen über den Unique Key des Dokuments durchgeführt.

```
<response>
   ...
   <result name="response" numFound="20256" start="0">
      <doc>
         <str name="id">6442583</str>
         <str name="titleText">Kategorie:Datei:Augsburg</str>
      </doc>
   </result>
   <lst name="moreLikeThis">
      <result name="6442583" numFound="20255" start="0">
         <doc>
            <str name="id">7414400</str>
            <str name="titleText">Kategorie:Medienunternehmen (Augsburg)</str>
         </doc>
         ...
      </result>
   </lst>
</response>
```

Das obige Beispiel eines solchen Response enthält ein Trefferdokument, zu dem ein Vorschlag generiert wurde.



Diese Variante geht zulasten der Performance. Für jedes Dokument werden ähnliche Dokumente ermittelt, und es ist nicht sicher, ob der Anwender auch das verlinkte Dokument suchen wird. Einen besseren Kosten-Nutzen-Aspekt bietet die Variante mit einem eigenständigen Request-Handler.

Variante MoreLikeThis-Handler

Die Variante MoreLikeThis als eigenen RequestHandler umzusetzen, ist ressourcenschonender, da nur dann die Vorschläge generiert werden müssen, wenn der Anwender bzw. die Applikation es unbedingt benötigt.

Die Konfiguration in der *solrconfig.xml* ist einfach, da es bereits eine Implementierung als RequestHandler gibt. Natürlich können hier ebenfalls die allgemeinen MoreLikeThis-Parameter, wie `mlt.count` oder `mlt.mintf`, als Default bzw. Invariants definiert werden.

```
<requestHandler name="/mlt" class="solr.MoreLikeThisHandler">
</requestHandler>
```

Die Bedingungen an das Schema gelten für diese Variante ebenfalls.

Um nun mittels des MoreLikeThis-RequestHandler ähnliche Dokumente zu finden, können unterschiedliche Requests ausgeführt werden.

Oft wird in der Trefferliste bei jedem Dokument ein Link oder Ähnliches angeboten, der nach den verwandten Dokumenten sucht. Der Request hierfür sieht dann wie folgt aus:

```
http://localhost:8983/solr/de_wikipedia/mlt?q=id:6442583&mlt.fl=text
```

In dem obigen Beispiel wird mittels des `q`-Parameters spezifiziert, für welches Dokument (`id:6442583`) Vorschläge generiert werden sollen. Für das so spezifizierte Dokument wird nun das Feld `text` (`mlt.fl=text`) ausgewertet. Im `q`-Parameter kann selbstverständlich jede valide Lucene-Syntax genutzt werden.

Eine weitere Möglichkeit ist die direkte Angabe von Termen bzw. Begriffen. Dies wird beispielsweise genutzt, um Vorschläge basierend auf dem Suchverhalten eines Anwenders zu generieren. Als Input für den RequestHandler dienen dann die früheren Suchbegriffe des Anwenders:

```
http://localhost:8983/solr/de_wikipedia/mlt?stream.body=augsburg&mlt.fl=text&mlt.mintf=1
```

Im obigen Beispiel wird basierend auf der Eingabe `augsburg` nach ähnlichen Dokumenten gesucht.

Für beide Request-Varianten wird folgender Solr-Response generiert, der im Response-Element die Vorschläge enthält:

```
<response>
   ...
   <result name="response" numFound="20255" start="0">
      <doc>
```

```
        <str name="id">7414400</str>
        ...
      </doc>
      ...
   </result>
</response>
```

MoreLikeThis in der Wikipedia

Das Velocity-Template ist von vornherein auf MoreLikeThis ausgelegt. Sobald die Funktionalität im `/browse`-RequestHandler aktiviert worden ist, wird zu jedem Dokument eine Liste von ähnlichen Dokumenten angezeigt.

Um das nun auch für das Wikipedia-Beispiel zu nutzen, muss der RequestHandler wie folgt erweitert werden:

```
<requestHandler name="/browse" class="solr.SearchHandler" startup="lazy">
   <lst name="defaults">
      ...
      <!-- more like this -->
      <str name="mlt">titleText,true</str>
      <str name="mlt.fl">user</str>
      <int name="mlt.count">3</int>
   </lst>
   ...
   </requestHandler>
```

Des Weiteren muss das Template (*wikipedia.vm*) zur Anzeige eines Wikipedia-Treffers ebenfalls erweitert werden. Die kann nach der Anpassung wie folgt aussehen:

```
<div class="result-title">
   <b>#field('titleText')</b>
</div>
<div>
   <b>Id:</b> #field('id')</br>
   <b>Revision:</b> #field('revision')</br>
   <b>Timestamp:</b> #field('timestamp')</br>
   <b>User:</b> #field('user')</br>
</div>
<div>
   <b>Text:</b> #field('text')
</div>

<div class="mlt">
   #set($mlt = $mltResults.get($docId))
   #set($mltOn = $params.getBool('mlt'))
   #if($mltOn == true)<div class="field-name">Ähnliche Dokumente</div>#end
   #if ($mltOn && $mlt && $mlt.size() > 0)
   <ul>
      #foreach($mltHit in $mlt)
      #set($mltId = $mltHit.getFieldValue('id'))
      <li>
         <div>
```

```
            <span class="field-name">Id:</span> $mltId</a>
            <span class="field-name">Title:</span> $mltHit.getFieldValue('titleText')
            <span class="field-name">User:</span> $mltHit.getFieldValue('user')
          </div>
        </li>
        #end
      </ul>
      #elseif($mltOn && $mlt.size() == 0)
        <div>No Similar Items Found</div>
      #end
    </div>
    #parse('debug.vm')
```

Nach dem Reload des Solr-Core steht dann MoreLikeThis zu Verfügung und sieht im Browse-Interface aus, wie in Abbildung 4-16 gezeigt:

Abbildung 4-16: MoreLikeThis

Elevate – Top-Treffer definieren

Es gibt Situationen, in denen trotz ausgefeilten Boostings und QueryCookings die erwarteten Dokumente nicht als Erstes in der Trefferliste erscheinen. Oft liegt das an den Daten, die im Zusammenspiel mit dem Relevanz-Algorithmus dafür sorgen, dass das entsprechende Dokument keinen hohen Score-Wert bekommt. Gibt es solche Situationen in der Suche oft, muss man weiter am Boosting und Scoring tunen. Tritt dies jedoch nur in wenigen Fällen auf, kann man auf das Elevate zurückgreifen.

Das Elevate-Feature ist im Allgemeinen auch unter den Begriffen »Sponsored Links« und »Top-Treffer« bekannt. Gemeint ist damit das gezielte Manipulieren der Trefferliste, indem explizit definierte Dokumente ganz am Anfang der Trefferliste erscheinen. Dabei ist es unerheblich, ob das »elevated« Dokument auch ein regulärer Treffer ist oder nicht.

Das Elevate-Feature ist dafür gedacht, in einzelnen Situationen das technische Scoring auszuschalten und einzelnen Dokumenten den Vorrang zu geben. Elevate ist jedoch kein ganzheitlicher Ansatz, um das Scoring zu manipulieren.

Beim Elevate werden diese Situationen redaktionell gepflegt, d. h., für jeden Suchbegriff wird definiert, welches Dokument am Anfang der Trefferliste erscheint. Ist dieses Dokument ein regulärer Treffer, wird es nur in der Trefferliste »verschoben«. Hat die Suche das Dokument nicht als Treffer identifiziert, wird es künstlich in die Trefferliste eingefügt.

Per Default greift das Elevate nur bei nicht sortierten Trefferlisten. Ist eine Sortierung, beispielsweise nach Datum, angegeben, wird das von der Elevate-Komponente berücksichtigt, und die »elevated« Dokumente reihen sich entsprechend ein. Möchte man trotz Sortierung, dass die »elevated« Dokumente zu Beginn der Trefferliste auftauchen, muss dies entsprechend konfiguriert werden.

Trotz Elevate liefert Solr immer nur die definierte Anzahl von Dokumenten zurück. Ist der Parameter `rows` auf `10` eingestellt, liefert Solr genau zehn Dokumente aus. Die Dokumente, die durch die Elevate-Konfiguration verdrängt worden sind, erscheinen auf der »nächsten Seite« der Trefferliste.

Die Konfiguration dieses Features muss an zwei Stellen durchgeführt werden. Zum einen muss in der *solrconfig.xml* die entsprechende SearchComponent spezifiziert und im RequestHandler registriert werden, und zum anderen muss eine XML-Datei erstellt werden, die spezifiziert, welche Dokumente bei welchen Suchbegriffen am Anfang der Trefferliste erscheinen sollen.

Die QueryElevationComponent ist relativ einfach konfiguriert. Es gibt nur ein paar wenige Parameter, die gepflegt werden müssen:

queryFieldType
: Mit diesem Parameter spezifiziert man den Feldtyp, mit dem der User-Input in der Elevate-Komponente analysiert werden soll. Üblicherweise nimmt man hier einen Feldtyp, der die einzelnen Wörter in kleinen Buchstaben im Index ablegt, d. h. Classic-/Standard-/Whitspace-Tokenizer kombiniert mit einem LowerCaseFilter.

config-file
: Mit diesem Parameter definiert man die XML-Datei, in der die Manipulation der Trefferliste beschrieben ist. Diese XML-Datei muss sich im *conf*-Verzeichnis des Solr-Cores befinden.

forceElevation
: Wird dieser Parameter auf `true` gesetzt, werden »elevated« Dokumente auch bei einer expliziten Sortierung an den Anfang der Trefferliste gesetzt.

QueryElevation ist keine der Standardkomponenten und muss daher in jedem RequestHandler, der Elevate beherrschen soll, hinzugefügt werden. Dies wird üblicherweise über die `last-components` umgesetzt, wie im folgenden Beispiel zu sehen ist:

```
<searchComponent name="elevator" class="solr.QueryElevationComponent" >
   <str name="queryFieldType">text</str>
   <str name="config-file">elevate.xml</str>
</searchComponent>

<requestHandler name="/wikipedia" class="solr.SearchHandler" startup="lazy">
...
   <arr name="last-components">
      <str>elevator</str>
   </arr>
</requestHandler>
```

Da man SearchComponent und RequestHandler explizit miteinander »verbinden« muss, kann man auch unterschiedliche Elevate-Konfigurationen für unterschiedliche Suchsituationen erstellen, indem man mehrere SearchComponents definiert, die auf jeweils eine andere XML-Datei (`config-file`) verweisen.

Die XML-Dateien zur Manipulation der Trefferliste sind ebenfalls sehr einfach aufgebaut. Innerhalb von `<elevate>`-Tags können mehrere `<query>`-Tags definiert werden. Jedes `<query>`-Tag repräsentiert dabei eine mögliche Eingabe eines Anwenders. Innerhalb des `<query>`-Tags werden die Dokumente (`<doc>`-Tag) über die `id` angegeben, die am Anfang der Trefferliste erscheinen sollen.

Es können optional auch Dokumente aus der Trefferliste mittels Elevate entfernt werden, indem man im `<doc>`-Tag das Attribut `exclude="true"` setzt. Dies kann in manchen Situationen sehr hilfreich sein, um »falsche« Dokumente zu entfernen, ohne das Scoring oder den Index komplett zu ändern.

```
<elevate>
   <query text="augsburg">
      <doc id="1" />
      <doc id="99" exclude="true" />
   </query>
</elevate>
```

Im obigen Beispiel wird bei der Eingabe des Suchbegriffs `augsburg` das Dokument mit der ID `1` als Erstes angezeigt, wobei das Dokument mit der ID `99` aus der Trefferliste entfernt wird.

Das Beispiel wird aber nicht funktionieren, wenn man »augsburger puppenkiste« sucht, da diese Kombination der Sucheingabe nicht in der *elevate.xml* enthalten ist. Konfiguriert man ein entsprechendes `<quer>`-Tag, würde dieser Fall ebenso funktionieren.

Man sieht hier schon die große Einschränkung der Funktionalität. Das Elevate-Feature ist nicht flexibel genug, um beispielsweise die Reihenfolge von Suchbegriffen zu ändern oder zu prüfen, ob ein Teil der eingegebenen Wörter konfiguriert worden ist. Man muss wirklich für jede mögliche Eingabekombination ein eigenes Element in der *elavate.xml* definieren, was zu einem riesigen Pflegeaufwand führen kann.

Die Dokumente, die mittels des Elevate-Features am Anfang der regulären Trefferliste eingefügt werden, kann man nach der Suche nicht mehr von »regulären« Treffern unterscheiden. Möchte man solche »Top-Treffer« auch gezielt als solche ausweisen, muss man dies über eine separate Suche machen. Bei der ersten Suche sollte das Elevate nicht enthalten sein, und bei der zweiten Suche sollte dann zusätzlich zu den oben beschriebenen Parametern noch `exclusive=true` gesetzt sein. Dies zwingt Solr dazu, nur die »elevated« Dokumente auszugeben. Somit hat man zwei getrennte Trefferlisten, die man unterschiedlich darstellen kann.

Elevate in der Wikipedia

Mit der oben beschriebenen Ausgangskonfiguration wird im `/browse`-Handler im Feld `text` gesucht. In der indexierten Wikipedia gibt es nun Dokumente mit wirklichem Inhalt, und es gibt Übersichtsdokumente, die nur auf die entsprechenden Dokumente verweisen. Dokumente, die lediglich einen Verweis beinhalten, sind oft kürzer und kommen daher weiter nach vorn in der Trefferliste (siehe Kapitel 5, Abschnitt »Der TF-IDF-Algorithmus« auf Seite 194). In Abbildung 4-17 sieht man ein solches Übersichtsdokument als ersten Treffer, wenn man nach der Augsburger Puppenkiste sucht.

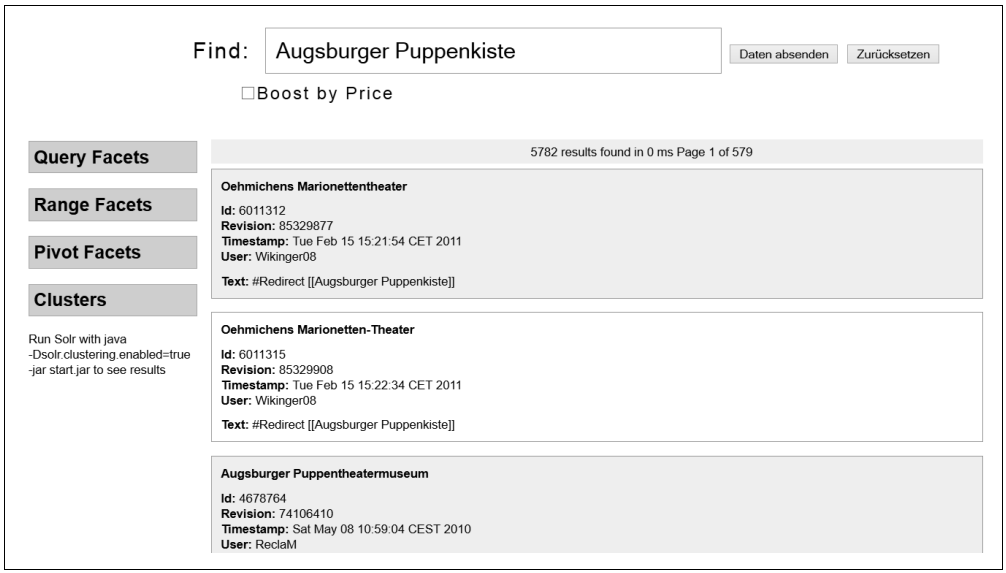


Abbildung 4-17: Trefferliste ohne Elevate

Als ersten Treffer erwarte ich hier jedoch das Dokument, das das Theater genauer beschreibt. Im Wikipedia-Index hat dieses Dokument die ID 14601 und kommt bei der obigen Suche erst an zehnter Stelle. Mit dem Elevate-Feature kann man das nun ändern.

Hierfür muss in der *solrconfig.xml* die Elevate-SearchComponent definiert und im `/browse`-Handler registriert werden. Als Nächstes wird die *elevate.xml* wie folgt angepasst:

```
<elevate>
   <query text="augsburger puppenkiste">
      <doc id="14601" />
   </query>
</elevate>
```

Nun wird nur noch der Solr-Core neu geladen, und bei gleicher Suchanfrage wird die Trefferliste aus Abbildung 4-18 generiert.

Abbildung 4-18: Trefferliste mit Elevate

Terms-Komponente – Solr-Felder auslesen

Die TermsComponent erlaubt einen direkten Zugriff auf den indexierten Inhalt – die Terme – eines Felds, ohne eine Volltextsuche durchzuführen. Sie nutzt dabei direkt Lucenes TermEnum-Objekt, um durch das Dictionary zu iterieren. Damit ist sie um ein Vielfaches schneller als die »normale« Suche.

Die Terms-Komponente ist daher sehr nützlich, wenn man beispielsweise eine feldbasierte Facette oder ein einfaches AutoSuggest umsetzen möchte. Es gibt jedoch ein paar Haken bei diesem Feature. Beispielsweise wird der `fq`-Parameter von der TermsComponent nicht unterstützt, d. h., wenn man zum Beispiel in einem Intranet die nutzerspezifischen Rechte über FilterQueries umgesetzt hat, kann man die Terms-Komponente nicht nutzen. Des Weiteren werden von dieser Komponente Dokumente, die als gelöscht markiert worden sind, nicht herausgefiltert und landen mit im Ergebnis.

Wenn man immer einen optimierten Index hat und keine FilterQueries nutzt, sollte man jedoch auf die Terms-Komponente zurückgreifen, und zwar so oft es geht, um von den Performancevorteilen zu profitieren.

Die Terms-Komponente wird in der *solrconfig.xml* als SearchComponent konfiguriert und muss wieder gesondert in den RequestHandlern eingebunden werden, da sie ebenfalls keine der sechs Standardkomponenten ist.

Bei der Konfiguration der SearchComponent gibt es nichts Spezielles zu beachten. Nur die Einbindung in einen RequestHandler kann durch folgende Parameter beeinflusst werden.

terms
: Wird dieser Parameter auf `true` gesetzt, wird die TermsComponent aktiviert.

terms.fl
: Mit diesem Parameter wird das Feld spezifiziert, das durch die TermsComponent ausgewertet werden soll.

terms.mincount/terms.maxcount
: Mit diesen Parametern lässt sich festlegen, wie oft mindestens oder maximal ein Term in dem Feld enthalten sein muss, damit er im Ergebnis enthalten ist.

terms.limit
: Mit diesem Parameter kann festgelegt werden, wie viele Ergebnisse angezeigt werden. Er funktioniert analog zu *facet.limit*.

terms.prefix
: Mit diesem Parameter spezifiziert man, womit der Term anfangen muss, damit er im Ergebnis angezeigt wird.

Einige dieser Parameter, wie beispielsweise *terms.prefix*, werden je nach Verwendungszweck erst zur Suchzeit spezifiziert.

Die vollständige Liste mit allen Parametern finden Sie im Solr-Wiki unter folgender URL:

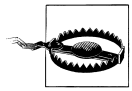
http://wiki.apache.org/solr/TermsComponent

Das nachfolgende Beispiel zeigt die Default-Konfiguration, die in der *solrconfig.xml* mitgeliefert wird. Besonders ist hier, dass im `/terms`-RequestHandler die SearchComponent als Component registriert wird. Damit werden alle anderen Funktionalitäten, wie Suche, Facetten oder Debug-Ausgaben, deaktiviert.

```
<searchComponent name="terms" class="solr.TermsComponent"/>

<requestHandler name="/terms" class="solr.SearchHandler" startup="lazy">
   <lst name="defaults">
      <bool name="terms">true</bool>
   </lst>
   <arr name="components">
      <str>terms</str>
   </arr>
</requestHandler>
```

Ein solcher RequestHandler wie im obigen Beispiel ist sowohl für Facettierung als auch für AutoSuggest flexibel einsetzbar.



Da der Output der Terms-Komponente die indexierten Terme sind, gibt es je nach Feldanalyse unterschiedliche Ergebnisse. Wurde bei der Felddefinition beispielsweise der NGramFilter verwendet, erhält man einzelne Buchstaben, Buchstabentupel, -tripel etc. Möchte man die Terms-Komponente für Facetten oder AutoSuggest nutzen, sollte man entweder beim Typ String bleiben oder einen nutzen, der nur sparsam mit TokenFiltern arbeitet.

Terms-Komponente in der Wikipedia

Die beiden Nutzungsmöglichkeiten (Facetten und AutoSuggest) der Terms-Komponente, die oben angedeutet wurden, können auch für den Wikipedia-Index umgesetzt werden.

Terms-Facetten

Um Facetten zu generieren, müssen sowohl die SearchComponent als auch der `/terms`-RequestHandler eins zu eins in die *solrconfig.xml* übernommen werden. Das Einzige, was nun noch übrig bleibt, ist, einen entsprechenden Aufruf an den RequestHandler zu definieren.

Mit dem folgenden Request lassen wir uns von Solr die ersten zehn Einträge des Felds `user` ausgeben:

```
http://localhost:8983/solr/de_wikipedia/terms?terms.fl=user&terms.limit=10
```

Das Ergebnis von Solr sieht dann wie folgt aus:

```
<response>
   <lst name="responseHeader">
   ...
   </lst>
   <lst name="terms">
      <lst name="user">
         <int name="KLBot2">525638</int>
         <int name="EmausBot">220794</int>
         <int name="Addbot">76838</int>
         <int name="Aka">67887</int>
         <int name="Sebbot">32051</int>
         <int name="Xqbot">31021</int>
         <int name="MorbZ-Bot">29327</int>
         <int name="Wikinger08">20089</int>
         <int name="Akkakk">17573</int>
         <int name="ChristophDemmer">17087</int>
      </lst>
   </lst>
</response>
```

Das Ergebnis ist ähnlich wie bei den Facetten. Man erhält eine Liste mit Einträgen, bestehend aus einem Key/Value-Paar, wobei der Key der Term ist und der Value die Häufigkeit seines Vorkommens im Dokumentenbestand.

Mit der Terms-Komponente können natürlich gleichzeitig auch mehrere feldbasierte Facetten generiert werden. Hierzu muss man den Parameter `terms.fl` nur so oft spezifizieren, wie man Facetten haben möchte. Die Einstellungen, wie `terms.limit`, gelten dann für alle Felder.

Auf eine Umsetzung im Template wird an dieser Stelle verzichtet, da Facetten bereits beschrieben wurden.

Terms-AutoSuggest

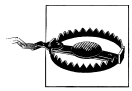
Ähnlich wie der Request für die Facette, funktioniert die Nutzung der Terms-Komponente für das AutoSuggest. Man benötigt nur zusätzlich den Parameter `terms.prefix`, und schon bekommen man die top zehn Ergebnisse, die mit einer bestimmten Zeichenkette anfangen. Folgender Request ermittelt die Einträge, die mit »Chr« beginnen:

```
http://localhost:8983/solr/de_wikipedia/terms?terms.fl=user&terms.prefix=Chr
```

Der Solr-Response ist analog zu dem der Facette, nur dass für dieses Szenario die Häufigkeitsangaben nicht ausgewertet werden.

```
<response>
   <lst name="responseHeader">
   ...
   </lst>
   <lst name="terms">
      <lst name="user">
         <int name="ChristophDemmer">17087</int>
         <int name="ChristianBier">1596</int>
         <int name="Christian1985">1397</int>
         <int name="Christian140">461</int>
         <int name="Chricho">408</int>
         <int name="Chris828">276</int>
         <int name="ChrisiPK">252</int>
         <int name="Chron-Paul">217</int>
         <int name="Christian2003">201</int>
         <int name="Chris06">142</int>
      </lst>
   </lst>
</response>
```

Diese Umsetzung des AutoSuggest ist im Vergleich zu der Facetten-Variante um ein Vielfaches schneller in der Ausführung, jedoch mit den oben beschriebenen Einschränkungen.



Genau wie bei den Facetten ist der Parameter `terms.prefix` case-sensitiv, das bedeutet für das obige Beispiel, dass `terms.prefix=chr` kein Ergebnis liefert. Man muss hier also beachten, welcher Feldtyp im Schema definiert ist.

Das AutoSuggest im Browse-Interface wird über die Terms-Komponente umgesetzt. Der entsprechende Request wird dynamisch über eine Skript zusammengesetzt, das in der *head.vm* definiert ist. In diesem Template muss das Feld angepasst werden, aus dem die Vorschläge generiert werden. Die *head.vm* sieht nach der Anpassung wie folgt aus:

```
  <title>#param('title')</title>
  <meta http-equiv="content-type" content="text/html; charset=UTF-8"/>

<script type="text/javascript" src="#{url_root}/js/lib/jquery-1.7.2.min.js"></script>
<link rel="stylesheet" type="text/css" href="#{url_for_solr}/admin/file?
     file=/velocity/main.css&contentType=text/css"/>
<link rel="stylesheet" href="#{url_for_solr}/admin/file?
     file=/velocity/jquery.autocomplete.css&contentType=text/css" type="text/css" />
<script type="text/javascript" src="#{url_for_solr}/admin/file?
     file=/velocity/jquery.autocomplete.js&contentType=text/javascript"></script>

  <script>
  $(document).ready(function(){
    $("\#q").autocomplete('#{url_for_solr}/terms', {  ## backslash escaped
                          #q as that is a macro defined in VM_global_library.vm
         extraParams:{
           'terms.prefix': function() { return $("\#q").val();},
           'terms.sort': 'count',
           'terms.fl': 'titleText',
           'wt': 'velocity',
           'v.template': 'suggest'
         }
       }
    ).keydown(function(e){
      if (e.keyCode === 13){
        $("#query-form").trigger('submit');
      }
    });
  });

  </script>
```

Zusätzlich muss in der *suggest.vm*, die für die Generierung des Drop-Down verantwortlich ist, ebenfalls eine Anpassung durchgeführt werden:

```
#foreach($t in $response.response.terms.titleText)
$t.key
#end
```

Nach dem Reload des Solr-Core kann der AutoSuggest direkt genutzt werden und sieht dann im Browse-Interface wie in Abbildung 4-19 zu sehen aus.

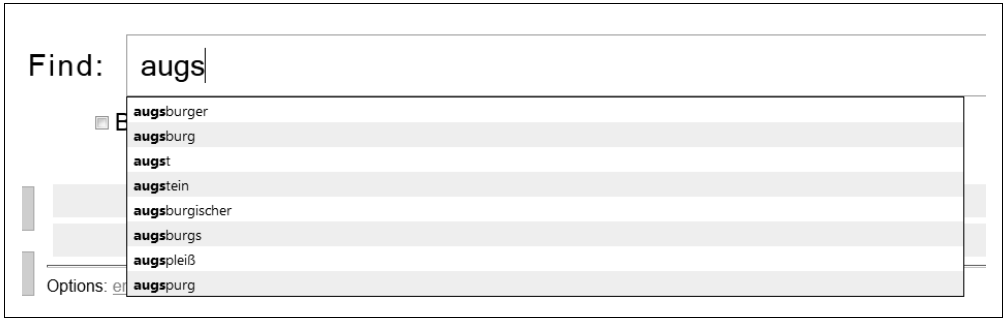


Abbildung 4-19: AutoSuggest mit der Terms-Komponente

TermVector-Komponente – Term-Informationen auswerten

TermVectoren sind dokumentenspezifische Informationen, die beispielsweise die Häufigkeit, die Position sowie Offset-Informationen beinhalten. Per Default speichert Solr diese Informationen nicht mit im Index ab, um die Größe des Index klein zu halten. Wenn man die TermVector-Komponente nutzen möchte, muss man für die entsprechenden Felder in der *schema.xml* folgende Attribute konfigurieren:

- termVectors
- termPositions
- termOffsets

Es müssen nur die Attribute auf `true` gesetzt werden, für die man mit der TermVector-Komponente Ausgaben erhalten möchte. Eine Felddefinition in der *schema.xml* sieht dann wie folgt aus, wenn alle Optionen aktiviert sind:

```
<field name="termInfos" type="text" indexed="true" stored="true" multiValued="true" termV
ectors="true" termPositions="true" termOffsets="true"/>
```

Die TermVector-Komponente trägt nicht direkt zur Suche oder zur Darstellung von Trefferlisten bei, kann aber genutzt werden, um beispielsweise ein eigenes Highlighting basierend auf den Offset-Informationen zu realisieren oder mit den TermVector-Informationen, wie TF und IDF, ein eigenes Scoring umzusetzen.

Es gibt ein paar wenige Parameter, mit denen man die Ausgabe der TermVector-Komponente beeinflussen kann:

tv
: Wird dieser Parameter auf `true` gesetzt, erscheint im Solr-Response der Block `termVectors` mit den Ausgaben der Komponente.

tv.fl
: In diesem Parameter kann eine Liste von Feldern definiert werden, die von der TermVector-Komponente ausgewertet werden soll. Ist dieser Parameter nicht gesetzt, greift die Komponente auf den Parameter `fl` zurück.

tv.docIds
: Mit diesem Parameter spezifiziert man die Dokumente basierend auf der internen Lucene-Dokument-ID, die von der TermVector-Komponente ausgewertet werden sollen.

tv.offset
: Wird dieser Parameter auf `true` gesetzt, werden die Offset-Informationen der einzelnen Terme ausgegeben.

tv.positions
: Wird dieser Parameter auf `true` gesetzt, werden die Positionsinformationen der einzelnen Terme zurückgegeben.

tv.df
: Wird dieser Parameter auf `true` gesetzt, wird die Häufigkeit des Terms im Dokumentenbestand zurückgegeben. Da dies zur Laufzeit ermittelt wird, kann es dazu führen, dass die Response-Zeit hoch wird.

tv.tf
: Wird dieser Parameter auf `true` gesetzt, wird die Häufigkeit des Terms im Dokument zurückgegeben.

tv.tf_idf
: Wird dieser Parameter auf `true` gesetzt, wird für jeden Term das Produkt von TF und IDF berechnet. Hierfür müssen `tv.df` und `tv.tf` auf `true` gesetzt sein. Auch dies kann große Performanceverluste nach sich ziehen, da die Berechnung ebenfalls zur Laufzeit durchgeführt wird.

tv.all
: Dieser Parameter ist die Kurzschreibweise für die Parameter `tv.df`, `tv.offset`, `tv.position`, `tv.tf`, `tv.tf_idf`. Wird dieser Parameter auf `true` gesetzt, werden alle oben beschriebenen Informationen ausgegeben.

Auch bei dieser Komponente ist die Konfiguration in der *solrconfig.xml* recht einfach. Sie muss als SearchComponent definiert und in einem RequestHandler registriert werden. Das folgende Beispiel zeigt die Default-Implementierung, wie sie mit dem Solr-Paket mitgeliefert wird:

```
<searchComponent name="tvComponent"
class="org.apache.solr.handler.component.TermVectorComponent"/>

<requestHandler name="/tvrh" class="org.apache.solr.handler.component.SearchHandler">
   <lst name="defaults">
      <bool name="tv">true</bool>
   </lst>
   <arr name="last-components">
      <str>tvComponent</str>
   </arr>
</requestHandler>
```

Da in unserem Beispiel-Schema für die Wikipedia bewusst auf TermVector-Informationen verzichtet wurde, basiert das folgende Beispiel auf der Default-Konfiguration der `collection1`, wie man es nach dem Download von Solr in seinem Home-Verzeichnis vorfindet.

In der *solrconfig.xml* findet sich schon ein vorbereiteter RequestHandler `/tvrh`, bei dem die TermVectorComponent bereits registriert worden ist. Man muss nur noch die Felder spezifizieren, die ausgewertet werden sollen. Da es in der *schema.xml* nur ein Feld (`includes`) mit der passenden Konfiguration gibt, sieht der Aufruf wie folgt aus:

```
http://localhost:8983/solr/collection1/tvrh?q=*:*&tv.fl=includes&tv.all=true
```

Der Solr-Response beinhaltet auch alle mit *:* gefundenen Dokumente. Für die Auswertung der TermVector-Komponente kann man dies jedoch vernachlässigen. Der nachfolgende XML-Schnipsel zeigt Teile des Outputs der TermVector-Komponente.

```
<response>
   ...
   <lst name="termVectors">
      <str name="uniqueKeyFieldName">id</str>
      ...
      <lst name="MA147LL/A">
         <str name="uniqueKey">MA147LL/A</str>
         <lst name="includes">
            <lst name="cable">
               <int name="tf">1</int>
               <lst name="positions">
                  <int name="position">3</int>
               </lst>
               <lst name="offsets">
                  <int name="start">23</int>
                  <int name="end">28</int>
               </lst>
               <int name="df">3</int>
               <double name="tf-idf">0.3333333333333333</double>
            </lst>
            <lst name="earbud">
               ...
            </lst>
            ...
         </lst>
      </lst>
   </lst>
</response>
```

Als Erstes sieht man im obigen Beispiel, dass das Feld `id` als Unique Key definiert worden ist.

Im Anschluss daran werden für jedes Dokument, das ebenfalls im Response-Block enthalten ist, die Informationen dargestellt. Im Beispiel sieht man das Dokument mit der ID `MA147LL/A`. Für dieses Dokument wird nun das Feld `includes` ausgewertet, und für alle Terme, beispielsweise `cable` oder `earbud`, werden Positionen, Offsets etc. angezeigt.

Stats-Komponente – statistische Auswertung

Die Stats-Komponente ist eine reine Auswertungskomponente von Solr. Sie dient nicht direkt der Suche oder Indexierung, kann aber indirekt dazu genutzt werden, indem man die Informationen auswertet, die von der Stats-Komponente zurückgegeben werden.

Die Stats-Komponente ist für numerische Feldtypen ausgelegt, es wird aber auch ein Ergebnis zurückgeliefert, wenn man Felder von anderen Feldtypen angibt.

Folgende Informationen werden von der Stats-Komponente zurückgeliefert, jedoch werden nicht alle von nicht numerischen Feldtypen unterstützt:

min
: Hier ist der kleinste Wert des Felds enthalten.

max
: Hier ist der größte Wert des Felds enthalten.

sum
: Hier ist die Summe aller Werte des Felds enthalten.

count
: Hier ist die Anzahl der Dokumente enthalten, die in diesem Feld mindestens einen Wert haben.

missing
: Hier ist die Anzahl der Dokumente enthalten, die keinen Wert in diesem Feld haben.

sumOfSquares
: Hier ist die Summe der Quadrate aller Werte des Felds enthalten.

mean
: Hier ist der Durchschnittswert enthalten.

stddev
: Hier ist die Standardabweichung enthalten.

Die Stats-Komponente ist eine der Standard-Komponenten und muss daher nicht gesondert in einem RequestHandler registriert werden. Sie ist durch folgende Parameter konfigurierbar:

stats
: Dieser Parameter aktiviert die Komponente.

stats.field
: Dieser Parameter definiert das Feld, für das die oben beschriebenen Informationen ermittelt werden sollen. Dieser Parameter kann auch mehrfach definiert werden, dann gibt es die Ausgabe für mehrere Felder.

stats.facet
: Dieser Parameter generiert mehrere Sets der Stats-Informationen, basierend auf der definierten Facette.

Terms-Komponente in der Wikipedia

Zugegebenermaßen geben die aktuell indexierten Daten unseres Wikipedia-Beispiels nicht genug her, um alle oben beschriebenen Outputs der Stats-Komponente sinnvoll auszuwerten. Es gibt in unserem Schema nur die drei Felder `revision`, `userId` und `timestamp`, für die alle Werte von der Stats-Komponente ermittelt werden können.

Dennoch lässt sich beispielsweise zumindest feststellen, ob für alle Dokumente Inhalte in den Feldern enthalten sind, wenn man die Werte `count` und `missing` des Response auswertet. Folgender Request ermittelt die dazu notwendigen Informationen für die Felder `timestamp` und »user:

```
http://localhost:8983/solr/de_wikipedia/select?q=*:*&rows=0&stats=true&stats.field=timest
amp&stats.field=user
```

Das Ergebnis, das Solr zurückliefert, sieht wie folgt aus:

```
<response>
   ...
   <lst name="stats">
      <lst name="stats_fields">
         <lst name="timestamp">
            <date name="min">2002-08-28T08:50:42Z</date>
             <date name="max">2013-05-10T06:56:11Z</date>
             <long name="count">2627825</long>
             <long name="missing">0</long>
            <date name="sum">11064999O-09-26T03:57:39.999Z</date>
            <date name="mean">2012-02-09T00:21:23.531Z</date>
            <double name="sumOfSquares">4.087751846806247E22</double>
            <double name="stddev">1.2472230380915794E8</double>
            <lst name="facets"/>
         </lst>
         <lst name="user">
            <str name="min">"Cardinalen"</str>
            <str name="max">ÀÌ¹Ý</str>
            <long name="count">2529976</long>
             <long name="missing">97849</long>
            <lst name="facets"/>
         </lst>
      </lst>
   </lst>
</response>
```

Da es sich bei den beiden Feldern um Date- und String-Typen handelt, liefert Solr einige unsinnige Werte zurück. Dennoch lässt sich hieraus beispielsweise wunderbar ablesen, dass 97.849 Dokumente im Index keinen `user` haben. Diese Information kann man nun etwa nutzen, um die Daten in der Wikipedia nachzupflegen. Man kann auch ablesen, dass der erste Artikel im August 2008 in die Wikipedia eingepflegt worden war.

/browse-RequestHandler für die Wikipedia

Wenn Sie nun alle Features dieses Kapitels umgesetzt haben, sollte die *solrconfig.xml* stark gewachsen sein, und der /browse-RequestHandler sieht in etwa wie folgt aus:

```
<requestHandler name="/browse" class="solr.SearchHandler" startup="lazy">
   <lst name="defaults">
      <str name="df">text</str>

      <!-- velocity -->
      <str name="wt">velocity</str>
      <str name="v.template">browse</str>
      <str name="v.layout">layout</str>
      <str name="title">Solritas - Wikipedia</str>

      <!-- faceting defaults -->
      <str name="facet">true</str>
      <str name="facet.mincount">1</str>
      <str name="facet.limit">5</str>
      <!-- field facet -->
      <str name="facet.field">user</str>
      <!-- query facet -->
      <str name="facet.query">{!key='last year'}timestamp:[NOW-1YEARS TO NOW]</str>
      <str name="facet.query">{!key='last 10 years'}timestamp:[NOW-10YEARS TO NOW]</str>
      <str name="facet.query">{!key='last 100 years'}timestamp:[NOW-100YEARS TO NOW]
      </str>
      <!-- range facet -->
      <str name="facet.range">timestamp</str>
      <str name="facet.range.start">NOW/YEAR-10YEARS</str>
      <str name="facet.range.end">NOW</str>
      <str name="facet.range.gap">+1YEAR</str>
      <str name="facet.range.other">before</str>
      <str name="facet.range.other">after</str>
      <!-- pivot facet -->
      <str name="facet.pivot">user,title</str>

      <!-- highlighting -->
      <str name="hl">true</str>
      <str name="hl.fl">titleText text</str>
      <str name="hl.simple.pre">&lt;u&gt;&lt;i&gt;</str>
      <str name="hl.simple.post">&lt;/i&gt;&lt;/u&gt;</str>

      <!-- did you mean -->
      <str name="spellcheck">true</str>
      <str name="spellcheck.onlyMorePopular">true</str>
      <str name="spellcheck.count">3</str>
      <str name="spellcheck.collate">true</str>

      <!-- more like this -->
      <str name="mlt">true</str>
      <str name="mlt.fl">text</str>
      <int name="mlt.count">3</int>
   </lst>
   <arr name="last-components">
```

```
        <str>suggest</str>
        <str>elevator</str>
      </arr>
    </requestHandler>
```

Eine damit durchgeführte Suche führt dann zu dem Ergebnis aus Abbildung 4-20.

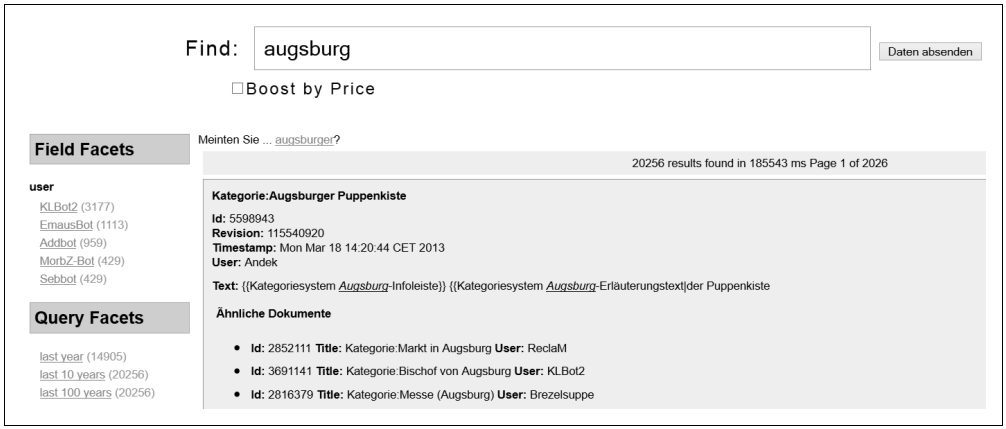


Abbildung 4-20: Browse-Interface mit Such-Features