In een tijd waarin 'googelen' een werkwoord is, raken gebruikers steeds meer gewend aan ongestructureerd zoeken. De meeste software die wordt ontwikkeld, kan daar echter nog weinig mee. Dat is jammer want Google-achtige zoekfunctionaliteit is helemaal niet zo moeilijk. Dit artikel laat zien hoe dit kan op basis van Hibernate Search.

Ongestructureerd zoeken met Hibernate

Krachtige zoeken in iedere applicatie

Paul Bakker
is trainer/consultant bij Info Support.

Vrijwel iedere applicatie waar Java-ontwikkelaars aan werken maakt gebruik van een relationele database. Daar is niets mis mee, maar het legt wel een behoorlijke beperking op de mogelijkheden rondom zoeken. Als voorbeeld neem ik even een boekenwinkel. Om het voor de gebruiker eenvoudig te houden wil ik, net zoals bij Amazon, maar één zoekveld hebben. In dat zoekveld kan direct worden gezocht op (een combinatie van) titel, auteur en isbn-nummer. Het moet daarbij niet uitmaken in welke volgorde de zoekwoorden worden opgegeven. Een voorbeeld van zo'n query is: 'Dan Brown Lost Symbol' om te zoeken naar 'The Lost Symbol' van Dan Brown. Deze eenvoudige functionaliteit is direct een probleem in een relationele database. De enige manier om deze query uit te voeren is om op elk afzonderlijk woord in iedere relevante kolom een 'LIKE %woord%' query uit te voeren. Voor dit voorbeeld zijn dat 4 (woorden) * 3 (kolommen) = 12 queries die een volledige tablescan moeten uitvoeren (door de wildcards in de LIKE). Dat klinkt niet echt als een mechanisme dat heel goed gaat performen. En dan zijn we pas bij het begin. Wat als een hit op auteur belangrijker is dan een hit op titel? En wat als een gebruiker 'Don Brown' invoert in plaats van 'Dan Brown'? Dit soort functionaliteit is eigenlijk niet te implementeren met SQL-queries. Hoe krijgt Amazon dit dan toch voor elkaar?

Full-Text-Search

De oplossing is eigenlijk eenvoudig: je zoekt niet meer in de database met SQL-queries, maar gebruikt hier een 'Full Text Index' voor. In de basis is zo'n index een key/value structuur met daarin ieder woord dat is opgeslagen als key. Daarbij staan verwijzingen naar de rijen of documenten waar dat woord in voorkomt. Een index kan er bijvoorbeeld als volgt uitzien:

Dan	auteur	[1,2]
Brown	auteur	[1,2]
The	titel	[1,2]
Lost	titel	[2]
Symbol	titel	[2]
Da	titel	[1]
Vinci	titel	[1]
Code	titel	[1]

Dit voorbeeld gaat over een database waarin twee boeken van Dan Brown staan. Nu kan op ieder afzonderlijk woord gezocht worden, onafhankelijk van in welke kolom het woord voorkomt. Omdat een Full-Text-Index de data zal ordenen, gaat dat zoeken ook nog eens bijzonder efficiënt. De vraag is nu alleen hoe we aan zo'n index komen. Hier zijn verschillende mogelijkheden voor:

- Ingebouwd in het RDMS
- Als los server product
- Via een API / library

Alle drie de opties hebben hun eigen voor- en nadelen. Een in de database ingebouwde engine vergt het minste onderhoud. Na wat configuratie bouwt de database zelf de index op en beheert deze. Via een database specifieke API kun je vervolgens zoeken in de index. De opties van het indexeren zijn echter nogal beperkt en er is weinig integratie mogelijk direct vanuit de applicatie. Daarnaast is

dit een slecht schaalbare oplossing. Database servers hebben het vaak al zwaar en zijn vaak niet heel eenvoudig op te schalen, dus het is niet altijd handig om nog meer taken aan zo'n server te geven.

Een voorbeeld van een los serverproduct is Apache Solr. Je start de server van Solr op en vervolgens kun je via een HTTP API documenten in de index stoppen en in de index zoeken. Een document is hierbij een stuk XML in Solr specifiek formaat dat je vanuit een applicatie kunt genereren, maar ook bijvoorbeeld uit Word- en PDF-documenten kunt genereren. Dit maakt de index bruikbaar voor allerlei soorten data. Het nadeel is dat integratie met een applicatie of database volledig met de hand gemaakt moet worden.
De laatste variant is die waar de rest van dit artikel over gaat: een index die met een Java API in de applicatie te integreren is. Als je hiernaar gaat zoeken kom je al snel bij Apache Lucene uit: een low-level API om met een index te werken.

Apache Lucene

Lucene biedt indexing en zoektechnolgie via een Java API. Lucene is dan ook bedoeld om in een applicatie geïntegreerd te worden en is geen losstaand product. Het eerder besproken Apache Solr is bijvoorbeeld gebaseerd op Lucene. Omdat Lucene met een Java API werkt en daarmee volledige applicatie-integratie mogelijk maakt, lijkt dit de ideale oplossing om in eigen applicaties te gebruiken. Dat is het ook, maar er zijn nog wel een aantal problemen te overwinnen.
Ervan uitgaande dat de data uiteindelijk nog steeds gewoon in een relationele database wordt opgeslagen moet Lucene daarmee gaan integreren. Met Java is het gebruikelijk om voor de database toegang een ORM-framework zoals Hibernate te gebruiken, dus dat lijkt het logische punt om de integratie tot stand te brengen.

Hibernate en Lucene

Helaas is de koppeling tussen Lucene en Hibernate nog helemaal niet zo triviaal. Om te beginnen moet er worden gezorgd dat de index gevuld wordt, waarschijnlijk zodra er een entity via Hibernate wordt opgeslagen. Daarnaast moeten na een zoekactie in de index de resultaten hiervan worden vertaald naar entity-objecten. Om te indexeren moeten bepaalde waarden van een entity misschien wel worden getransformeerd. In een Full-Text -Index is namelijk alles tekst. Er zijn geen types voor bijvoorbeeld numerieke waarden of datums. Bij een zoekactie op producten met een prijs lager dan 10 worden dus de tekstuele representaties van de getallen gebruikt. De waarde 10 komt daardoor voor de waarde 2. Om dit op te lossen moeten getallen bijvoorbeeld met een 'zerofill' worden geprefixd. En hoe zit het met relaties tussen entities? Al met al een hoop werk om deze integratie met de hand te schrijven. Precies om deze reden is er nog een framework beschikbaar: 'Hibernate Search'.

Hibernate Search

Hiberate Search is een subproject van Hibernate, gestart door Emmanuel Bernard. Het doel van dit framework is 'out-of-the-box' Lucene te gebruiken in combinatie met Hibernate. Hibernate Search zorgt voor het bijwerken van de index als er entities worden opgeslagen. Daarnaast biedt Hibernate Search een uitbreiding op de standaard query API van Hibernate waarmee in de index kan worden gezocht via de bekende Hibernate API. Het framework zorgt voor conversie van types en voor het laden van entities na een query. Met Hibernate Search kan zonder al te veel inspanning Full-Text-Search-functionaliteit worden toegevoegd.

Na wat werk in de Hibernate-configuratiefiles kan Hibernate Search worden gebruikt. Het enige dat we nog moeten doen om een index op te bouwen is aangeven wat er geïndexeerd moet worden. Dit gebeurt door middel van annotaties op de entity klassen zoals in voorbeeld 1 staat opgenomen.

```
@Entity
@Indexed
public class Book {
    @Id
    @GeneratedValue
    @DocumentId
    long id;

    @Field
    String title;

    @Field
    String author;

    //Getters and setters
}
```

Codevoorbeeld 1: een basis Hibernate Search mapping.

Zoals te zien is in het voorbeeld worden standaard JPA/Hibernate annotaties gecombineerd met Hibernate Search annotaties zoals @Indexed, @DocumentId en @Field. In tegenstelling tot JPA moet op ieder veld dat geïndexeerd moet worden, expliciet de @Field-annotatie worden opgegeven. Dit heeft een goede reden, want lang niet ieder veld van een entity kan zinvol worden gebruikt om op te zoeken. Alleen velden waarop eindgebruikers kunnen zoeken, moeten in de index worden opgenomen. Hoe velden precies geïndexeerd worden, kan geconfigureerd worden met analyzers.

Analyzers

Het ophakken van teksten (bijvoorbeeld een titel) in afzonderlijke woorden is de taak van een Analyzer. De StandardAnalyzer splitst zinnen op basis van (onder andere) spaties en leestekens, waardoor de meeste Europese talen hiermee aardig worden ondersteund. In veel teksten staan echter woor-

De koppeling tussen Hibernate en Lucene is helaas niet zo triviaal.

den die weinig zinvol zijn om te indexeren. Denk bijvoorbeeld aan lidwoorden. Omdat deze woorden in bijna iedere tekst voorkomen vervuilen ze slechts de index zonder op een zinvolle manier de zoekresultaten te beïnvloeden. Dit soort woorden kan dan ook beter weggelaten worden. Hiervoor kunnen filters gebruikt worden. Filters kunnen de stroom van tokens (meestal woorden) filteren en daar tokens uit weghalen, eraan toevoegen of wijzigen. De StandardAnalyzer heeft onder andere een LowerCaseFilter om alle woorden naar kleine letters te converteren en een StopFilter waarmee veelvoorkomende woorden weggefilterd kunnen worden. Dit kunnen taalspecifieke woorden zijn (de, het, en, een etc. in het Nederlands), maar het zouden ook domeinspecifieke termen kunnen zijn die je zelf in een tekstfile opgeeft. Analyzers worden meestal per entity met annotaties geconfigureerd, maar het is mogelijk voor verschillende velden verschillende analyzers te gebruiken. In codevoorbeeld 2 wordt een StandardAnalyzer geconfigureerd.

```
@Entity
@Indexed
@AnalyzerDef(name = “applicationanalyzer”, tokenizer =
@TokenizerDef(factory = StandardTokenizerFactory.class),
         filters = {
 @TokenFilterDef(factory = StandardFilterFactory.class),
@TokenFilterDef(factory = LowerCaseFilterFactory.class),
     @TokenFilterDef(factory = StopFilterFactory.class)
        })
public class Book {
```

Codevoorbeeld 2: het configureren van een StandardAnalyzer.

Zoeken in de index is erg snel, dus er kunnen meerdere queries naast elkaar bestaan.

Om een voorbeeld te geven van het effect van deze configuratie kijken we naar de volgende voorbeeldzin:
“Het is lekker weer: de zon schijnt, maar het is niet te warm.”

StandardTokenizer (verwijdert leestekens)
Het | is | lekker | weer | de | zon | schijnt | maar | het | is | niet | te | warm

StandardFilter (geen effect)
Het | is | lekker | weer | de | zon | schijnt | maar | het | is | niet | te | warm

LowerCaseFilter (vervangt hoofdletters)
het | is | lekker | weer | de | zon | schijnt | maar | het | is | niet | te | warm

StopFilter (verwijdert veelvoorkomende woorden)
lekker | weer | zon | schijnt | maar | niet | te | warm

Spelfouten

Als je op Amazon of Google een spelfout maakt in het woord waarnaar je op zoek bent, krijg je vaak toch de resultaten waarnaar je zocht. Ook wordt er vaak een hint gegeven dat er een fout in je zoekopdracht zit. Dit is met weinig werk ook met Hibernate Search voor elkaar te krijgen. Er zijn twee manieren hoe je dit kunt aanpakken: tijdens het indexeren rekening houden met spelfouten of tijdens het zoeken woorden bij benadering zoeken. We kijken naar de (meest krachtige) aanpak waarbij tijdens het indexeren al rekening wordt gehouden met fouten in zoekopdrachten. Hiervoor worden weer analyzers en filters gebruikt. Een eerste filter dat gebruikt kan worden, is een n-gram filter. Een n-gram filter hakt ieder woord op in stukjes van bijvoorbeeld drie letters. ‘hibernate’ wordt bijvoorbeeld:
‘hib ibe ber ern nat ate’. Ieder afzonderlijk stukje komt in de index. Bij het zoeken moet de zoekopdracht met hetzelfde filter worden opgesplitst. Bij spelfouten kan dan nog steeds het juiste resultaat worden gevonden.

Een andere optie is om te zoeken op woorden die gelijk klinken bij het uitspreken. Dit mechanisme is geïmplementeerd voor een groot aantal talen. Ook woorden die volledig verkeerd gespeld zijn, maar wel ongeveer dezelfde uitspraak hebben kunnen zoals eerder gezegd op deze manier worden gevonden. Het is wel weer van belang dezelfde analyzer te gebruiken bij zowel het indexeren als bij het zoeken.

Synoniemen

Naast het opvangen van spelfouten kan het ook zinvol zijn te kunnen zoeken op synoniemen. Bij het zoeken naar DVD’s van live-optredens zijn ‘optreden’ en ‘concert’ bijvoorbeeld aan elkaar gelijk. Dit kan weer opgelost worden met een analyzer die ervoor zorgt dat bij een zin waar ‘optreden’ in voorkomt, ook ‘concert’ wordt opgeslagen in de index. De index wordt hierdoor weer groter, maar dat is over het algemeen geen enkel probleem.
Een vergelijkbaar mechanisme heeft te maken met werkwoorden. Als er gezocht wordt naar ‘zoeken’ zijn documenten met ‘zoek’ of ‘zoekt’ waarschijnlijk ook relevant. Dit kan door werkwoorden zowel bij het indexeren als zoeken naar hun stam terug te brengen.

Zoekt en gij zult vinden

Tot dusver hebben we alleen gezien hoe de index met allerlei opties kan worden opgebouwd. Dat is natuurlijk pas zinvol als je hier ook in kunt zoeken. Hibernate Search maakt het mogelijk de standaard Hibernate API met wat uitbreidingen te gebruiken voor het zoeken in de index. In plaats van queries geschreven in JPQL / HQL schrijf je nu echter Lucene-queries. Deze queries kunnen zelf worden geschreven of met een API worden gegenereerd. Hibernate Search zorgt ervoor dat de resultaten van een query uiteindelijk weer entities opleveren, net zoals een gewone HQL query dat zou doen. In codevoorbeeld 3 wordt dit geïllustreerd. Bij het maken

van de query moet altijd de analyzer gebruikt worden die ook bij het indexeren is gebruikt.

```
QueryParser parser =
        new MultiFieldQueryParser(new String[]{"title",
 "author"},
                  new StandardAnalyzer());

Query luceneQuery = parser.parse(query);
FullTextQuery fullTextQuery =
         fullTextSession.createFullTextQuery
(luceneQuery, Book.class);
```

Codevoorbeeld 3: Zoeken over meerdere velden.

Zoeken in stappen

Zoals je hebt gezien, zijn er vele manieren om een woord in de index op te slaan met behulp van de verschillende analyzers. De vraag is nu hoe je de verschillende analyzers kunt combineren. Het is gebruikelijk ieder woord door verschillende analyzers te laten indexeren. Dat betekent dat ieder woord ook in een net iets andere vorm, verschillende keren in de index voorkomt. De index wordt daardoor groter, maar dat is meestal geen enkel probleem. Vervolgens kan er worden gezocht in meerdere stappen. Als eerste wordt er bijvoorbeeld een query gedaan die geanalyseerd wordt met de StandardAnalyzer. Spelfouten, synoniemen en stamvormen zullen daar niet mee gevonden worden, maar gelijke woorden wel. Als deze query te weinig resultaten oplevert, kan er een tweede query worden uitgevoerd die gebruik maakt van bijvoorbeeld een n-gram filter. Je kunt op die manier een lijstje analyzers afwerken totdat er genoeg resultaten zijn om aan de gebruiker te tonen. Omdat het zoeken in de index erg snel is (zolang er nog geen entities uit de database worden gehaald), is het ook geen enkel probleem meer dan één query uit te voeren.

Relaties

Een Full-Text-index ondersteunt geen relaties tussen documenten. Voor ieder geïndexeerd entitytype wordt zelfs een aparte index gebruikt. Dat is een probleem, omdat de meeste entitymodellen die in Hibernate worden gebruikt, wel relationeel zijn (er liggen relaties tussen entities). Hibernate Search heeft hier gelukkig een elegante oplossing voor. Op een relatie kun je aangeven dat de relatie als veld in de index moet komen. Voor onderstaande relatie wordt voor iedere auteur een authors.name-veld opgenomen in de index bij een boek. In de Author-klasse moet vervolgens wel op de relevante velden een @Field annotatie worden geplaatst.

```
@ManyToMany
@IndexEmbedded
Set<Author> authors;
```

Relevantie

Omdat met behulp van analyzers entities kunnen worden gevonden die slechts gedeeltelijk aan de query voldoen, is niet ieder resultaat even relevant. Resultaten die exact aan alle eisen van de zoekopdracht voldoen, zijn waarschijnlijk interessanter voor een gebruiker dan resultaten die slechts gedeeltelijk of na het uitvoeren van allerlei extra analyzers gevonden worden. Lucene zorgt direct al voor de eerste stap hierin. Voor ieder resultaat wordt een score berekend. Aan de hand van deze score is te zien hoe relevant het resultaat is. De best-scorende resultaten komen automatisch al bovenaan te staan.

Daarnaast kunnen bepaalde velden in een entity functioneel belangrijker zijn dan andere velden. Titel en de naam van de auteur zijn waarschijnlijk belangrijker dan de omschrijving van een boek. Op die manier wordt een boek met exact de titel die in de zoekopdracht wordt beschreven als eerste resultaat opgeleverd. Een ISBN nummer is daarnaast een nog exactere omschrijving. Dit soort functionele keuzes kunnen worden opgenomen in een query met een boost-factor. Aan velden kan een boost worden gegeven die zorgt dat bij het berekenen van de score van een resultaat het veld zwaarder of juist minder zwaar meeweegt. In codevoorbeeld 4 wordt een query getoond met een boost-factor voor de titel en het ISBN-nummer. Hieruit kan simpelweg worden gelezen dat een ISBN-nummer tien keer zo belangrijk is als een ander veld en een titel drie keer zo belangrijk als andere velden.

```
Map boosts = new HashMap();
boosts.put("title", 3f);
boosts.put("ISBN", 10f);

QueryParser parser =
        new MultiFieldQueryParser(new String[]{"title",
"authors.name", "ISBN"},
                 new StandardAnalyzer(), boosts);
```

Codevoorbeeld 4: Het zoeken met boostfactoren.

Tools

Hibernate Search maakt het bijna triviaal krachtige zoekmogelijkheden aan een Hibernate-applicatie toe te voegen. Bij het ontwikkelen van een uitgebreid zoeksysteem hoort echter ook heel veel uitproberen. Omdat iedere applicatie andere data bevat en de gebruikers van ieder systeem anders zoeken, moet ook iedere zoekoplossing hierop worden afgesteld. Dit afstellen betekent analyseren van zoekopdrachten en bijbehorende resultaten en op basis daarvan de index verder optimaliseren. Wat is bijvoorbeeld de juiste verhouding van boost-factoren? Welke stopwoorden zorgen voor ruis in de resultaten? Een onmisbare tool hierbij is Apache Luke. Met deze tool kun je direct in een index kijken, en daar queries op uitvoeren. Dit geeft inzicht in de index en dat is het startpunt om verder te gaan optimaliseren. «

Met een analyzer kan ook worden gezocht op basis van synoniemen.

Referenties

- Hibernate training: http://www.infosupport.com/Training/CursusInfo/JAHIBER
- Presentatie van Emmanuel Bernard over Hibernate Search op Parleys: http://parleys.com/#st=5&id=1639&sl=1
- Boek: Hibernate Search in Action: ISBN 1933988649